PROGRAM ANALYSIS USING BINARY DECISION DIAGRAMS

by

Ondřej Lhoták

School of Computer Science
McGill University, Montreal

January 2006

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Doctor of Philosophy

Abstract

A fundamental problem in interprocedural program analyses is the need to represent and manipulate collections of large sets. Binary Decision Diagrams (BDDs) are a data structure widely used in model checking to compactly encode large state sets. In this dissertation, we develop new techniques and frameworks for applying BDDs to program analysis, and use our BDD-based analyses to gain new insight into factors influencing analysis precision.

To make it feasible to express complicated, interrelated analyses using BDDs, we first present the design and implementation of JEDD, a Java language extension which adds relations implemented with BDDs as a datatype, and makes it possible to express BDD-based algorithms at a higher level than existing BDD libraries.

Using JEDD, we develop PADDLE, a framework of context-sensitive points-to and call graph analyses for Java, as well as client analyses that make use of their results. PADDLE supports several variations of context-sensitive analyses, including the use of call site strings and abstract receiver object strings as abstractions of context.

We use the PADDLE framework to perform an in-depth empirical study of the effect of context-sensitivity variations on the precision of interprocedural program analyses. The use of BDDs enables us to compare context-sensitive analyses on much larger, more realistic benchmarks than has been possible with traditional analysis implementations.

Finally, based on the call graph computed by PADDLE, we implement, using JEDD, a novel static analysis of the cflow construct in the aspect-oriented language AspectJ. Thanks to the JEDD high-level representation, the implementation of the analysis closely mirrors its specification.

Résumé

Un problème fondamental en analyse interprocédurale des programmes est le besoin de représenter et manipuler des collections de grands ensembles. Les diagrammes de décision binaires (DDB) sont une structure de données largement utilisée dans la vérification de modèles pour coder de grands ensembles d'états. Dans cette thèse, nous développons de nouvelles techniques pour appliquer les DDB à l'analyse des programmes, et nous utilisons nos analyses basées sur les DDB pour acquérir des connaissance sur les facteurs qui influencent la précision des analyses.

Pour qu'il soit faisable d'exprimer des analyses compliquées et interdépendantes en utilisant les DDB, nous présentons d'abord JEDD, une extension du langage Java qui ajoute des relations implantées avec des DDB comme un type de données, et permet l'expression des algorithmes basés sur les DDB à un niveau plus haut qu'avec les bibliothèques de DDB existantes.

En utilisant JEDD, nous développons PADDLE, un système d'analyses de pointeur et de graphe d'appel sensibles au contexte pour Java, ainsi que des analyses client qui exploitent leurs résultats. PADDLE comprend plusieurs variantes d'analyses sensibles au contexte, y compris des analyses qui utilisent des chaînes de sites d'appel et des chaînes d'objets récepteurs abstraits en tant qu'abstractions de contexte.

Nous utilisons le système PADDLE pour effectuer une étude expérimentale de l'effet de la sensibilité au contexte sur la précision des analyses interprocédurales des programmes. L'utilisation des DDB nous permet de comparer des analyses sensibles au contexte sur des programmes plus grands et plus réalistes que ce qui a été possible avec les implantations traditionnelles des analyses.

Finalement, utilisant le graphe d'appel calculé par PADDLE, nous développons, en utilisant JEDD, une analyse statique originale de la construction cflow dans le langage orienté-aspect AspectJ. Grâce à la représentation JEDD de haut niveau, l'implantation de l'analyse suit directement sa spécification.

Acknowledgements

First, I would like to thank my advisor, Laurie Hendren. Throughout my time at McGill, her constant encouragement kept me going. This work benefited significantly from her thoughtful suggestions for improvement. Useful suggestions for the final version were also provided by the examination committee, particularly the external examiner Nelson Amaral.

The seed that eventually grew into this dissertation, the idea of using BDDs for points-to analysis, originated from a blackboard discussion between Feng Qian, Marc Berndl, and me. I thank them and the rest of the Sable group, as well as Oege de Moor and the `abc` team at Oxford and Århus, for all the productive discussions and co-operation. I am particularly grateful to Bruno Dufour for his help proofreading the French translation of the abstract. Thank you also to Gordon Cormack for his guidance and support, and to the WatForm group for showing me BDDs from a verification perspective.

The software developed as part of this work builds on the work of others. The Jedd framework is built on the excellent Polyglot Java front-end by Andrew Myers, Nathaniel Nystrom, Stephen Chong, and others, and can make use of several BDD libraries, notably BuDDy by Jørn Lind-Nielsen, and SAT solvers, particularly zChaff from Princeton University. Paddle builds on Soot, started by Raja Vallée-Rai and developed by the Sable group. The AspectJ part of this work builds on the `abc` compiler built by the `abc` team, spread between McGill, Oxford, and Århus. The dynamic call graphs for the empirical study were collected using Bruno Dufour's excellent *J tool. I am grateful to Manu Sridharan, the first external user of Jedd and Paddle, for his bug reports, fixes, and suggestions.

This work was supported financially by NSERC, by an IBM Ph.D. fellowship, and by a Richard H. Tomlinson fellowship.

A big thank you to Jennifer for sticking it out with me in Montreal, and for her constant love, help, encouragement, and patience.

Contents

List of Figures

List of Tables

List of Abbreviations

AST abstract syntax tree

BDD binary decision diagram [Bry92]

CFA control flow analysis [Shi88]

CGI common gateway interface

CHA class hierarchy analysis [DGC95]

CNF conjunctive normal form

DAG directed acyclic graph

DFS depth-first search

DNF disjunctive normal form

JNI Java native interface

RTA rapid type analysis [BS96]

SAT boolean satisfiability problem

SQL structured query language

SSA static single assignment [AWZ88]

VM virtual machine

VTA variable type analysis [SHR+00]

XML extensible markup language

ZCWL Zhu/Calman/Whaley/Lam algorithm [ZC04, WL04]

Chapter 1
Introduction

1.1 Motivation

Existing and new applications of program analysis demand increasingly precise, yet efficient, interprocedural analyses. A program analysis conservatively estimates the possible run-time behaviour of a program by analyzing the program without executing it. Traditionally, results of program analyses have been used to justify compiler optimizations. More recently, program analysis has found important applications in software engineering tools which help developers understand, maintain, and verify programs. These applications depend on the availability of precise, efficient program analyses. Imprecision in analysis results restricts the code optimizations that can be safely performed by compilers, and reduces the amount of information available to software engineering tools. The popularity of object-oriented languages has increased the importance of *interprocedural* program analysis in particular. The context of our work is the effort to improve the precision of interprocedural program analyses while making them efficient enough to be practical.

A fundamental challenge in the design of precise interprocedural program analyses is the need to represent and manipulate collections of large sets. In some program analyses, much of the complexity stems not from the analysis itself, but from data structures carefully customized to be efficient enough for the particular analysis. Therefore, a general data structure which would make it easier to write reasonably

efficient analyses would be very useful for experimenting with new, more precise program analyses.

BDDs [Bry92] have been found to be a very effective representation of state sets in the area of model checking, where they have made it feasible to exhaustively check large state spaces. Could BDDs also be useful in the area of program analysis? The present dissertation develops the thesis that **BDDs are an effective representation of collections of large sets in interprocedural program analyses, and their use facilitates the development of and experimentation with new, precise, efficient analyses.**

Context-sensitive analyses are widely believed to significantly improve program analysis precision, particularly when analyzing object-oriented programs. However, until now, detailed empirical evidence for this belief has been scarce, because context-sensitive analyses have so far been too expensive to be feasible for programs of reasonable size. We show that the use of BDDs can make context-sensitive analyses efficient enough to be feasible for realistic programs.

New programming paradigms, such as Aspect-Oriented Programming, require the design of new interprocedural program analyses. BDDs enable analysis designers to build prototypes of such analyses quickly, without requiring them to devise clever data structures to make the prototypes efficient enough to experiment with. We demonstrate this with a BDD-based implementation of a static analysis of the *cflow* construct in the aspect-oriented programming language AspectJ.

1.2 Challenges

Our work shows how to overcome the following challenges inherent in the use of BDDs for interprocedural program analysis.

Program analyses and model checkers differ significantly in the forms of data that they manipulate. A model checker uses BDDs to explore the reachable state space of a finite state machine, storing sets of states in the BDD. Program analyses manipulate a much wider variety of data, and it is not obvious how to encode them

or manipulate them in BDDs. We suggest relations as an abstraction over raw BDDs, and demonstrate how to express program analyses in terms of relations.

Program analyses are often interdependent, and the low-level nature of current BDD libraries makes it very difficult to manage a large code base of multiple interdependent analyses. In our experience, implementing program analyses directly using a BDD library such as BuDDy [LN] or CUDD [Som] is both tedious and error-prone, to the point that it is not feasible to implement analyses consisting of more than about 30 BDD operations. Higher-level tools for writing analyses with BDDs are therefore necessary.

Details of how data is encoded in BDDs can affect analysis cost by orders of magnitude, so support for careful tuning of the encoding is crucial. The two factors affecting performance the most are the BDD variable ordering and the assignment of attributes to BDD variables. Finding an optimal variable ordering even for a single BDD is already an NP-complete problem, and we need orderings that are simultaneously good for the many BDDs in a system of interrelated analyses. Effective heuristics are known for some applications, but they have yet to be developed for program analyses. Therefore, tools are required to enable programmers to easily experiment with these design variations and to provide detailed feedback about their actual effect on the BDDs.

1.3 Contributions

This work contributes to the development of BDD-based program analysis in four ways. Figure 1.1 summarizes how these four contributions build on each other.

First, we have developed Jedd, a language extension to Java which makes it feasible to write complicated, interrelated, BDD-based program analyses. In Jedd, BDDs are abstracted as relations. Jedd code is written at a high level in terms of relations, and the Jedd compiler translates it to low-level Java code with calls into a BDD library to implement the BDD operations. Since design of Jedd is guided by the need to experiment with the encoding of relations in BDDs, Jedd provides ways

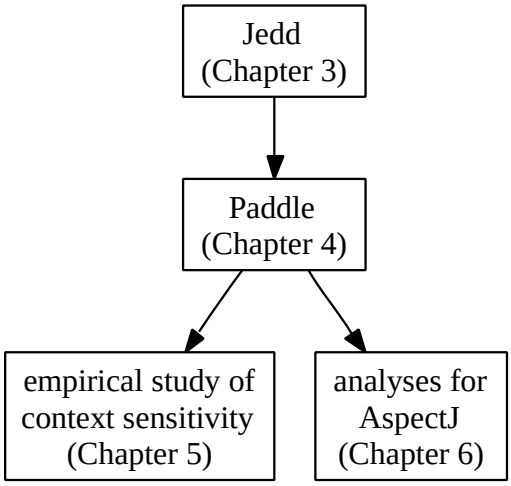


Figure 1.1: Summary of contributions

for the programmer to try different encodings and observe the resulting BDDs. We describe JEDD in detail in Chapter 3.

Second, we have used JEDD to implement PADDLE, a flexible framework of BDD-based call graph and points-to analyses and the various prerequisite analyses needed to compute them. PADDLE supports several variations of context sensitive analysis, including using strings of call sites [Shi88] and of abstract receiver objects [MRR02] as the context abstraction. While traditional implementations of these context-sensitive analyses generally do not scale beyond very small programs, our BDD-based implementation successfully analyzes significant Java applications in conjunction with the large Java class library. We present our analyses in more detail in Chapter 4.

Third, we have used PADDLE to perform an empirical study of the effects of different variations of context sensitivity on the precision of call graph and points-to analysis, and of client analyses that depend on their results. To the best of our knowledge, this is the first comprehensive comparison of these context sensitivity variations on Java programs of this size. We present our empirical study of context sensitivity in Chapter 5.

Fourth, we have developed a novel interprocedural analysis of the *cflow* construct in the aspect-oriented language AspectJ. The analysis is implemented in the JEDD

language and uses the call graph constructed by PADDLE. The BDD-based implementation of the analysis follows its specification almost exactly, without any additional implementation-specific clutter. The use of BDDs and the high-level JEDD language made it easy to experiment with the analyses without having to spend much time on tuning implementation details of each variation of the analysis. We describe the *cflow* analysis in Chapter 6.

1.4 Organization

The remainder of this dissertation is organized as follows. We begin by providing background information about BDDs in the next chapter. The following four chapters describe in detail each of the four contributions listed above. The JEDD language and translator are presented in Chapter 3. The PADDLE interprocedural analysis framework is described in Chapter 4. In Chapter 5, we report the results of our empirical study of variations of context sensitivity and their effect on analysis precision. The *cflow* analysis for AspectJ is presented and evaluated in Chapter 6. Finally, in Chapter 7, we conclude and suggest directions for further research.

This thesis makes contributions to three areas of knowledge: Chapter 3 on JEDD contributes to the application of BDDs to program analysis, Chapters 4 and 5 on PADDLE and context sensitivity contribute to the design and implementation of precise interprocedural analyses for Java, and Chapter 6 contributes to analysis of AspectJ. Therefore, we have included a section on related work for each contribution within the corresponding chapter (Sections 3.8, 4.1, 5.7, and 6.4).

Chapter 2
Background: BDDs and Points-to Analysis

In this chapter, we provide the background information about BDDs [Bry92] that will be necessary to understand the remainder of this thesis. Since the topic of this thesis is the use of BDDs to implement set-based interprocedural program analyses, we will use one such analysis, subset-based points-to analysis [EGH94, And94], and our BDD-based implementation of it [BLQ+03], as an example to illustrate the BDD concepts.

In Section 2.1, we give a brief introduction to subset-based points-to analysis. In Section 2.2, we introduce BDDs, and describe how BDD operations are used in implementing a subset-based points-to analysis. In Section 2.3, we show the complete BDD-based analysis that we developed in [BLQ+03], and briefly comment on the tuning that was required to make it efficient.

2.1 Subset-based Points-to Analysis

Analyses of programs with pointers to memory must estimate the effects of operations performed through pointers. A points-to analysis approximates, for each pointer in the program, the set of objects to which the pointer may point. In our example points-to analysis, we represent each object by the allocation site at which it is allocated. The analysis tracks the flow of objects from their allocation sites along pointer assignments in the program. For each pointer p, the analysis computes a *points-to set* of the

allocation sites whose objects may flow to p. Therefore, if the program contains an allocation site S of the form `p := new O()`, then the pointer `p` may point to the objects allocated at site S, so the analysis generates the constraint $S \in \textit{points-to}(\texttt{p})$. The analysis is *subset-based* in that it models data flow between pointers using subset constraints between their points-to sets. Suppose that `p` and `q` are pointers, and the assignment `p := q` appears in the program. Since `q` is assigned to `p`, after the assignment, `p` may point to any object to which `q` was pointing. This is modelled in the analysis with the subset constraint $\textit{points-to}(\texttt{q}) \subseteq \textit{points-to}(\texttt{p})$.

```
X: a = new O();
Y: b = new O();
Z: c = new O();
   a = b;
   b = a;
   c = b;
```

Figure 2.1: Example pointer propagation statements

To use a concrete example, consider the program statements shown in Figure 2.1. The first three statements are allocation statements, which would cause the analysis to initialize the points-to sets of `a`, `b`, and `c` to $\{\texttt{X}\}$, $\{\texttt{Y}\}$, and $\{\texttt{Z}\}$, respectively. The fourth line would be modelled by the subset constraint $\textit{points-to}(\texttt{b}) \subseteq \textit{points-to}(\texttt{a})$, which would be processed by propagating the points-to set of `b` into the points-to set of `a`, making $\textit{points-to}(\texttt{a}) = \{\texttt{X}, \texttt{Y}\}$. The fifth line would be processed by propagating $\textit{points-to}(\texttt{a})$ into $\textit{points-to}(\texttt{b})$, making $\textit{points-to}(\texttt{b})$ also $\{\texttt{X}, \texttt{Y}\}$. Finally, the sixth line would cause $\textit{points-to}(\texttt{b})$ to be propagated into $\textit{points-to}(\texttt{c})$, making $\textit{points-to}(\texttt{c}) = \{\texttt{X}, \texttt{Y}, \texttt{Z}\}$. The final points-to sets for the example would be

$$\begin{aligned} \textit{points-to}(\texttt{a}) &= \{\texttt{X}, \texttt{Y}\} \\ \textit{points-to}(\texttt{b}) &= \{\texttt{X}, \texttt{Y}\} \\ \textit{points-to}(\texttt{c}) &= \{\texttt{X}, \texttt{Y}, \texttt{Z}\} \end{aligned}$$

When analyzing large programs, a key problem is that the number of points-to sets and the size of each set may become very large. Various techniques [Das00, FFSA98, HT01, Lho02, LH03, LPH01, RMR01, SH97, WL02] have been studied for compactly representing the points-to sets and efficiently solving the subset constraints. In this chapter, we review one such technique [BLQ+03] that we have developed, which is to use BDDs to compactly represent the points-to sets and BDD operations to efficiently propagate them along subset constraints.

2.2 Binary Decision Diagrams

A BDD [Bry92] is a representation of a boolean-valued function of n boolean *BDD variables*. Equivalently, it can be thought of as representing a set of binary vectors of length n; the set includes precisely those vectors which the function maps to the value 1.

Physically, a BDD is a trie-like rooted directed acyclic graph (DAG) of nodes. The DAG has two terminal nodes $\boxed{0}$ and $\boxed{1}$ with no successor, and every non-terminal node has two successors called the *0-successor* and the *1-successor*. As in a trie, to determine whether a given binary vector is in the set represented by the BDD, one starts at the root node of the BDD, and follows either the 0- or 1-successor depending on the value of each bit in the vector. If the traversal ends at the $\boxed{1}$ node, the vector is in the set; if the traversal ends at the $\boxed{0}$ node, the vector is not in the set.

To use a concrete example, we will now show how the points-to sets computed for the statements in Figure 2.1 can be encoded in a BDD. We could write the points-to sets as a set of points-to pairs, with each pair indicating that a given pointer may point to a given object, as follows:

$$\{(\texttt{a},\texttt{X}),(\texttt{a},\texttt{Y}),(\texttt{b},\texttt{X}),(\texttt{b},\texttt{Y}),(\texttt{c},\texttt{X}),(\texttt{c},\texttt{Y}),(\texttt{c},\texttt{Z})\}$$

Using 00 to represent `a` and `X`, 01 to represent `b` and `Y`, and 10 to represent `c` and `Z`, we can encode these points-to pairs as the set of binary vectors

$$\{0000, 0001, 0100, 0101, 1000, 1001, 1010\}$$

A BDD representing this set of binary vectors is shown in Figure 2.2. The pointers `a`, `b`, and `c` are encoded in first two bit positions of the BDD, and the objects `X`, `Y`, and `Z` are encoded third and fourth bit positions. We follow the common convention of drawing the 0-successor of each node as a dashed arrow, and the 1-successor as a solid arrow.

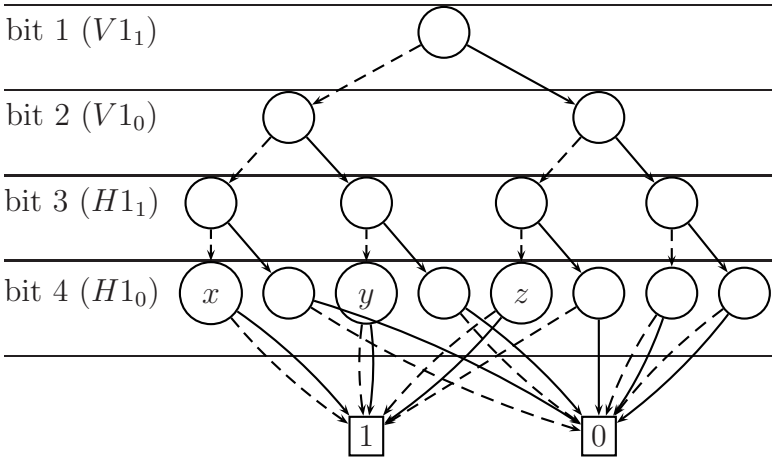


Figure 2.2: Unreduced BDD for points-to example

The nodes marked x, y, and z in Figure 2.2 are at the same bit position and have the same successors, because they all represent the same subset of objects $\{\texttt{X}, \texttt{Y}\}$. Since these nodes are the same, they could be merged into a single node, making the BDD smaller without changing the set that it represents. Furthermore, since their 0- and 1-successor are the same (the $\boxed{1}$ node), the value of the bit that they test does not affect the successor, so the bit does not need to be tested and the nodes could be removed entirely. If we repeatedly reduce the BDD in this way by finding mergeable and unnecessary nodes, we obtain the *reduced* BDD shown in Figure 2.3. The BDD represents the same set as the original unreduced BDD, but it is smaller.

For the purposes of our discussion, we presented an unreduced BDD first, then reduced it. In actual BDD implementations, however, the reduction rules are applied to each node as the BDD is being constructed. Therefore, in a real implementation, every BDD is kept fully reduced at all times.

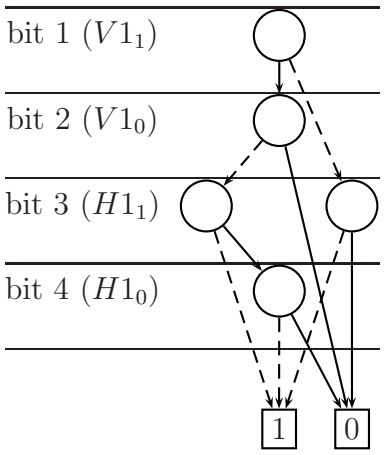


Figure 2.3: Reduced BDD for points-to example

It is convenient to group the bit positions representing a given element under a common name. Throughout this thesis, we will use the term ***physical domain***[1] to refer to a collection of bit positions representing an element such as a pointer or object. For example, the first two bit positions represent a pointer **v**ariable, so we call them the physical domain $V1$. Similarly, we call the third and fourth bit positions $H1$, because they represent an abstract **h**eap location. We use a subscript to denote a specific bit within a physical domain. For example, $V1_0$ denotes the zeroth (least significant) bit in the $V1$ physical domain, which in this case is the second bit in the BDD.

In our discussion so far, we have presented the encoding of points-to sets in a BDD interpreted as a set of binary vectors. For completeness, we now also present the equivalent boolean function. Following our earlier choice of binary encodings of the pointers and abstract objects, the boolean functions representing these elements are shown in the third column of Table 2.1. A points-to pair is represented by the conjunction of the pointer and the abstract object to which it points. For example, `b` pointing to `Z` is represented by the formula $V1_1 = 0 \wedge V1_0 = 1 \wedge H1_1 = 1 \wedge H1_0 = 0$.

[1]In BDD literature, a physical domain is often called just "domain". However, the same word is used in relational database literature with a different meaning (we will define it in Section 3.2.1). To distinguish the two, we use the term "physical domain" for a domain in the BDD sense, and simply "domain" for a domain in the relational database sense.

element	binary encoding	boolean formula
a	00	$V1_1 = 0 \wedge V1_0 = 0$
b	01	$V1_1 = 0 \wedge V1_0 = 1$
c	10	$V1_1 = 1 \wedge V1_0 = 0$
X	00	$H1_1 = 0 \wedge H1_0 = 0$
Y	01	$H1_1 = 0 \wedge H1_0 = 1$
Z	10	$H1_1 = 1 \wedge H1_0 = 0$

Table 2.1: Encodings of elements in terms of physical domains

A set of points-to pairs is represented by the disjunction of their formulas. So, the points-to sets from our running example would be represented by the formula

$$
\begin{aligned}
&\mathit{POINTSTO} \triangleq \\
&\quad (V1_1 = 0 \wedge V1_0 = 0 \wedge H1_1 = 0 \wedge H1_0 = 0) \quad \vee \\
&\quad (V1_1 = 0 \wedge V1_0 = 0 \wedge H1_1 = 0 \wedge H1_0 = 1) \quad \vee \\
&\quad (V1_1 = 0 \wedge V1_0 = 1 \wedge H1_1 = 0 \wedge H1_0 = 0) \quad \vee \\
&\quad (V1_1 = 0 \wedge V1_0 = 1 \wedge H1_1 = 0 \wedge H1_0 = 1) \quad \vee \\
&\quad (V1_1 = 1 \wedge V1_0 = 0 \wedge H1_1 = 0 \wedge H1_0 = 0) \quad \vee \\
&\quad (V1_1 = 1 \wedge V1_0 = 0 \wedge H1_1 = 0 \wedge H1_0 = 1) \quad \vee \\
&\quad (V1_1 = 1 \wedge V1_0 = 0 \wedge H1_1 = 1 \wedge H1_0 = 0)
\end{aligned}
$$

This formula is equivalent to the set of binary vectors given earlier.

In the BDDs that we have seen so far, the bits have always been tested in the same order, $V1_1V1_0H1_1H1_0$. However, any ordering can be used, as long as it is used consistently. For example, if the bits were tested in the order $H1_0V1_0H1_1V1_1$, the BDD for our example set would look like Figure 2.4. Although this BDD represents the same set as the BDD in Figure 2.3, it has 8 nodes rather than 5. When using BDDs, it is important to find an ordering which keeps the BDDs small. Unfortunately, finding the optimal ordering is NP-hard in general [BW96, THY93]. In [BLQ+03], we found an ordering that works well for points-to analysis. The JEDD system, which we

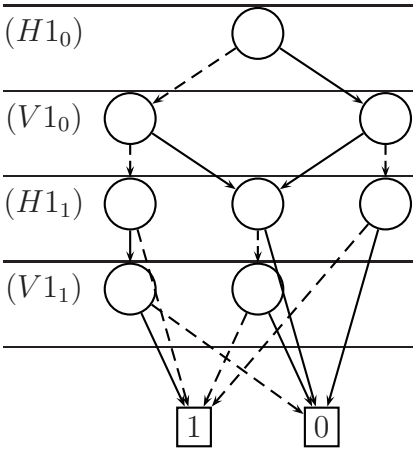


Figure 2.4: BDD for points-to sets using alternative ordering $H1_0V1_0H1_1V1_1$

present in Chapter 3, provides a profiling and visualization tool intended to help find good orderings for specific analyses by identifying the BDDs that affect performance the most, and showing their shape under a given ordering.

The basic set operations (union, intersection, complement, set difference) on the sets represented by BDDs are implemented using a recursive algorithm [Bry92] which traverses the argument BDDs and builds up the resulting BDD. The cost of these operations depends on the number of nodes in the BDDs involved, not the sizes of the sets that they represent. Therefore, large sets represented by small BDDs can be manipulated efficiently.

Like the points-to sets, the subset constraints induced by the pointer assignments in the program can be encoded in a BDD. We reuse the physical domain $V1$ to represent the source of each assignment, and introduce a new two-bit physical domain $V2$ to represent the target of each assignment. Thus, the three assignments from our example, `a=b`, `b=a`, and `c=b`, are encoded by the BDD representing the function

$$
\begin{aligned}
&\mathit{ASSIGN} \triangleq \\
&\quad (V1_1 = 0 \wedge V1_0 = 1 \wedge V2_1 = 0 \wedge V2_0 = 0) \quad \vee \\
&\quad (V1_1 = 0 \wedge V1_0 = 0 \wedge V2_1 = 0 \wedge V2_0 = 1) \quad \vee \\
&\quad (V1_1 = 0 \wedge V1_0 = 1 \wedge V2_1 = 1 \wedge V2_0 = 0)
\end{aligned}
$$

To propagate points-to sets, three additional BDD operations are needed, existential quantification, relational product, and replace.

The ***existential quantification*** operation makes a given BDD f independent of a given bit position b by constructing a function that is true whenever there exists a value of b (either 0 or 1) that makes f true. By applying existential quantification to all the bit positions of a physical domain, we make a BDD independent of the physical domain. For example, if we existentially quantify the *POINTSTO* BDD defined earlier with respect to the $V1$ domain, we obtain the boolean function in which each clause is made independent of $V1$:

$$\begin{aligned}
&\exists_{V1} POINTSTO = \\
&\quad (H1_1 = 0 \wedge H1_0 = 0) \quad \vee \\
&\quad (H1_1 = 0 \wedge H1_0 = 1) \quad \vee \\
&\quad (H1_1 = 0 \wedge H1_0 = 0) \quad \vee \\
&\quad (H1_1 = 0 \wedge H1_0 = 1) \quad \vee \\
&\quad (H1_1 = 0 \wedge H1_0 = 0) \quad \vee \\
&\quad (H1_1 = 0 \wedge H1_0 = 1) \quad \vee \\
&\quad (H1_1 = 1 \wedge H1_0 = 0)
\end{aligned}$$

This formula simplifies to

$$\begin{aligned}
&\exists_{V1} POINTSTO = \\
&\quad (H1_1 = 0 \wedge H1_0 = 0) \quad \vee \\
&\quad (H1_1 = 0 \wedge H1_0 = 1) \quad \vee \\
&\quad (H1_1 = 1 \wedge H1_0 = 0)
\end{aligned}$$

The resulting function represents the set containing every abstract object for which there exists a pointer that points to it (that is, the union of all the points-to sets).

The ***relational product*** operation is equivalent to performing set intersection (boolean conjunction) followed by existential quantification, but is implemented more efficiently than when these operations are performed separately. We illustrate the relational product operation using the points-to set propagation example. Consider the BDD representing the original points-to pairs $\{(\texttt{a}, \texttt{X}), (\texttt{b}, \texttt{Y}), (\texttt{c}, \texttt{Z})\}$ induced by the

three allocation statements in Figure 2.1:

$$
\begin{aligned}
&\mathit{ORIG\text{-}POINTSTO} \triangleq \\
&\quad (V1_1 = 0 \wedge V1_0 = 0 \wedge H1_1 = 0 \wedge H1_0 = 0) \quad \vee \\
&\quad (V1_1 = 0 \wedge V1_0 = 1 \wedge H1_1 = 0 \wedge H1_0 = 1) \quad \vee \\
&\quad (V1_1 = 1 \wedge V1_0 = 0 \wedge H1_1 = 1 \wedge H1_0 = 0)
\end{aligned}
$$

We would like to propagate the points-to pairs across the pointer assignments encoded in the *ASSIGN* BDD shown earlier. Since the $V1$ physical domain is common to both BDDs, a conjunction will find all pairs of clauses from the two formulas which match in the $V1$ physical domain:

$$
\begin{aligned}
&\mathit{ORIG\text{-}POINTSTO} \wedge \mathit{ASSIGN} = \\
&\quad (V1_1 = 0 \wedge V1_0 = 0 \wedge V2_1 = 0 \wedge V2_0 = 1 \wedge H1_1 = 0 \wedge H1_0 = 0) \quad \vee \\
&\quad (V1_1 = 0 \wedge V1_0 = 1 \wedge V2_1 = 0 \wedge V2_0 = 0 \wedge H1_1 = 0 \wedge H1_0 = 1) \quad \vee \\
&\quad (V1_1 = 0 \wedge V1_0 = 1 \wedge V2_1 = 1 \wedge V2_0 = 0 \wedge H1_1 = 0 \wedge H1_0 = 1)
\end{aligned}
$$

After existentially quantifying with respect to $V1$, we obtain

$$
\begin{aligned}
&\mathit{NEW\text{-}POINTSTO} \triangleq \\
&\mathrm{relprod}(\mathit{ORIG\text{-}POINTSTO}, \mathit{ASSIGN}, V1) = \\
&\exists_{V1}(\mathit{ORIG\text{-}POINTSTO} \wedge \mathit{ASSIGN}) = \\
&\quad (V2_1 = 0 \wedge V2_0 = 1 \wedge H1_1 = 0 \wedge H1_0 = 0) \quad \vee \\
&\quad (V2_1 = 0 \wedge V2_0 = 0 \wedge H1_1 = 0 \wedge H1_0 = 1) \quad \vee \\
&\quad (V2_1 = 1 \wedge V2_0 = 0 \wedge H1_1 = 0 \wedge H1_0 = 1)
\end{aligned}
$$

This formula encodes the new points-to pairs $\{(\mathtt{b}, \mathtt{X}), (\mathtt{a}, \mathtt{Y}), (\mathtt{c}, \mathtt{Y})\}$ arising from propagating the original points-to pairs along the pointer assignments. Figure 2.5 shows the effect of the relational product operation on the BDD representation. The *ORIG-POINTSTO* and *ASSIGN* BDDs are shown in parts (a) and (b), respectively, and the result of the relational product is shown in part (c) of the figure.

Next, we would like to find the union of the set of new points-to pairs and the original set. However, the original points-to pairs are encoded using the physical domains $V1$ and $H1$, while the new points-to pairs are encoded using the physical

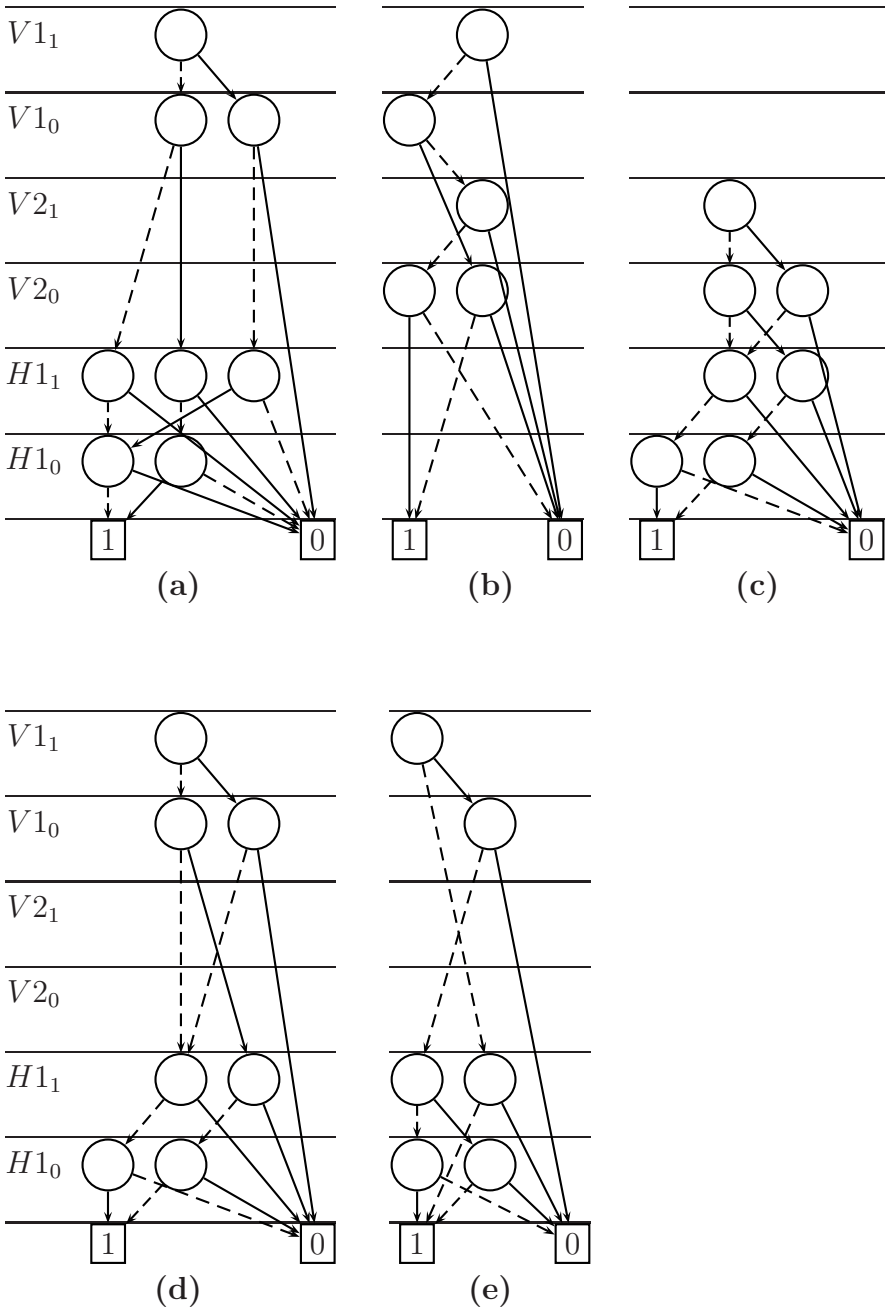


Figure 2.5: BDD representation of (a) *ORIG-POINTSTO* (b) *ASSIGN* (c) *NEW-POINTSTO* (d) *REPLACED-POINTSTO* (e) *PROPAGATED-POINTSTO*

domains $V2$ and $H1$. Before we can find the union, we must use the ***replace*** operation to replace the $V2$ physical domain with $V1$ in the *NEW-POINTSTO* BDD, to make its physical domains match the *ORIG-POINTSTO* BDD :

$$\begin{aligned}
&\mathit{REPLACED\text{-}POINTSTO} \triangleq \\
&\text{replace}(\mathit{NEW\text{-}POINTSTO}, V2, V1) = \\
&\quad (V1_1 = 0 \wedge V1_0 = 1 \wedge H1_1 = 0 \wedge H1_0 = 0) \quad \vee \\
&\quad (V1_1 = 0 \wedge V1_0 = 0 \wedge H1_1 = 0 \wedge H1_0 = 1) \quad \vee \\
&\quad (V1_1 = 1 \wedge V1_0 = 0 \wedge H1_1 = 0 \wedge H1_0 = 1)
\end{aligned}$$

The BDD representation of this function is shown in Figure 2.5(d).

Finally, we can now compute the union

$$\begin{aligned}
&\mathit{PROPAGATED\text{-}POINTSTO} \triangleq \\
&\quad \mathit{ORIG\text{-}POINTSTO} \vee \mathit{REPLACED\text{-}POINTSTO}
\end{aligned}$$

This gives the points-to sets after one step of propagation. The BDD representation is shown in Figure 2.5(e). To obtain the final points-to set BDD that we showed in Figure 2.3, the propagation step must be repeated a second time.

The process of propagating points-to sets using the three operations that we have just described (relational product, replace, and union) is summarized in the BDD code snippet shown in Figure 2.6, which calls into the BuDDy library to implement each operation. Line 4 performs a relational product of the `edgeSet` and `pointsTo` BDDs with respect to the $V1$ physical domain. Line 5 replaces the physical domain $V2$ with $V1$ in the result. Finally, line 6 adds the new points-to pairs into the `pointsTo` BDD. The operations are enclosed in a loop which iterates until a fixed point is reached.

2.3 BDD-based Points-to Analysis

Having illustrated the key BDD operations, we can now present the complete implementation of our original BDD-based points-to analysis [BLQ+03]. The analysis is a subset-based, flow- and context-insensitive, but field-sensitive points-to analysis for Java, based on the analyses that we implemented in the Spark [Lho02, LH03]

```
1  repeat
2    oldPt:[V1xH1] = pointsTo:[V1xH1];
3
4    /* (c) */ newPt1:[V2xH1] = relprod(edgeSet:[V1xV2], pointsTo:[V1xH1], V1);
5    /* (d) */ newPt2:[V1xH1] = replace(newPt1:[V2ToV1], V2ToV1);
6    /* (e) */ pointsTo:[V1xH1] = pointsTo:[V1xH1] ∪ newPt2:[V1xH1];
7
8  until pointsTo:[V1xH1] == oldPt:[V1xH1]
```

Figure 2.6: BDD code for propagating points-to sets along assignment constraints

framework. Like the SPARK analyses, this analysis processes four kinds of constraints, shown in Figure 2.7. The allocation and simple assignment constraints are the same as in Section 2.1. The new field store and load constraints model stores and loads to fields of heap objects.

allocation	$a : l := new\ C$	$o_a \in \textit{points-to}(l)$
simple assignment	$l_2 := l_1$	$l_1 \rightarrow l_2$
field store	$q.f := l$	$l \rightarrow q.f$
field load	$l := p.f$	$p.f \rightarrow l$

Figure 2.7: The four kinds of points-to constraints

In our original implementation, we assume that all the constraints have been generated before the points-to analysis begins. In Chapter 4, we will extend the analysis to handle new constraints generated while the analysis proceeds.

In addition to computing points-to sets for pointer variables, the analysis also computes points-to sets for pointers in fields of heap objects. That is, the points-to fact $o_1 \in \textit{points-to}(o_2.f)$ means that the field f of an object allocated at allocation site o_2 may point to an object allocated at allocation site o_1.

The points-to constraints are solved using the inference rules shown in Figure 2.8. The rules are implemented in BDDs, and are applied iteratively until a fixed point is

reached. The first rule models simple assignments: if l_1 points to o, and is assigned to l_2, then l_2 also points to o. The second rule models field stores: if l points to o_2, and l is stored into $q.f$, then for each o_1 pointed to by q, $o_1.f$ also points to o_2. Similarly, the third rule models field loads: if l is loaded from $p.f$, and p points to o_1, then l points to any o_2 that $o_1.f$ points to.

$$\frac{l_1 \rightarrow l_2 \quad o \in \textit{points-to}(l_1)}{o \in \textit{points-to}(l_2)} \tag{2.1}$$

$$\frac{o_2 \in \textit{points-to}(l) \quad l \rightarrow q.f \quad o_1 \in \textit{points-to}(q)}{o_2 \in \textit{points-to}(o_1.f)} \tag{2.2}$$

$$\frac{p.f \rightarrow l \quad o_1 \in \textit{points-to}(p) \quad o_2 \in \textit{points-to}(o_1.f)}{o_2 \in \textit{points-to}(l)} \tag{2.3}$$

Figure 2.8: Inference rules

For the simple points-to set propagation in Section 2.2, we needed three physical domains, $V1$, $V2$, and $H1$. To represent the constraints and points-to sets of a field-sensitive analysis, two additional physical domains are needed: a second physical domain of objects ($H2$) to represent points-to facts of the form $o_1 \in \textit{points-to}(o_2.f)$, which have two objects, and a physical domain of fields, FD.

We now describe the most important BDDs used in the algorithm, along with the physical domains in which they are encoded.

- $\mathit{pointsTo} \subseteq V1 \times H1$ is the set of points-to pairs for simple variables, of the form $o \in \textit{points-to}(l)$.

- $\mathit{fieldPt} \subseteq (H1 \times FD) \times H2$ is the set of points-to facts for fields of heap objects, of the form $o_1 \in \textit{points-to}(o_2.f)$.

- $\mathit{edgeSet} \subseteq V1 \times V2$ is the set of simple assignment constraints of the form $l_1 \rightarrow l_2$.

- $\mathit{stores} \subseteq V1 \times (V2 \times FD)$ is the set of field store constraints of the form $l_1 \rightarrow l_2.f$.

- $loads \subseteq (V1 \times FD) \times V2$ is the set of field load constraints of the form $l_1.f \rightarrow l_2$.
- $typeFilter \subseteq V1 \times H1$ is a set of constraints specifying which objects each pointer can point to based on its declared type. This is used to restrict the points-to sets for pointers to only contain objects of compatible type.

The full algorithm is given in Figure 2.9. The algorithm consists of an inner loop nested within an outer loop. We have annotated each BDD in the algorithm with the physical domains that it uses. Lines 1.1 to 1.2 implement the first inference rule. In line 1.1, the *edgeSet* and *pointsTo* BDDs are combined. This relational product operation computes the set of facts satisfying the first rule:

$$\{(l_2, o) \mid \exists l_1 : l_1 \rightarrow l_2 \wedge o \in \textit{points-to}(l_1)\}$$

In line 1.2, the set is converted to use physical domains $V1$ and $H1$ rather than $V2$ and $H1$, and in line 1.4, it is added into *pointsTo*. Line 1.3 will be explained later.

Lines 2.1 to 2.3 implement the second rule. Line 2.1 computes the intermediate result of the first two pre-conditions:

$$tmpRel1 = \{(o_2, q.f) \mid \exists l : o_2 \in \textit{points-to}(l) \wedge l \rightarrow q.f\}$$

In line 2.2, $tmpRel1$ is changed to physical domains suitable for the next computation. In line 2.3, the resulting set of facts satisfying all three pre-conditions is computed as

$$\{o_2 \in \textit{points-to}(o_1.f) \mid \exists q : (o_2, q.f) \in tmpRel1 \wedge o_1 \in \textit{points-to}(q)\}$$

In a similar way, lines 3.1 to 3.3 implement the third rule. Again, the first two pre-conditions are first combined to form a temporary BDD (line 3.1), then combined with the results from the second rule (line 3.2). After changing the result to the appropriate physical domains (line 3.3), we obtain new points-to pairs, which are added into the *pointsTo* BDD in line 4.2.

In our earlier work [Lho02, LH03] with the Spark points-to analysis framework, we observed that limiting points-to sets to include only objects of a type compatible with the declared type of the pointer significantly improves both analysis precision

```
1  repeat
2    outerOldPt:[V1×H1] = pointsTo:[V1×H1];
3
4    repeat
5      innerOldPt:[V1×H1] = pointsTo:[V1×H1];
6
7      /* --- rule 1 --- */
8      /* 1.1 */ newPt1:[V2×H1] = relprod(edgeSet:[V1×V2], pointsTo:[V1×H1], V1);
9      /* 1.2 */ newPt2:[V1×H1] = replace(newPt1:[V2ToV1], V2ToV1);
10
11     /* --- apply type filtering and add into pointsTo BDD  --- */
12     /* 1.3 */ newPt3:[V1×H1] = newPt2:[V1×H1] ∩ typeFilter:[V1×H1];
13     /* 1.4 */ pointsTo:[V1×H1] = pointsTo:[V1×H1] ∪ newPt3:[V1×H1];
14   until pointsTo:[V1×H1] == innerOldPt:[V1×H1]
15
16   /* --- rule 2 --- */
17   /* 2.1 */ tmpRel1:[(V2×FD)×H1] = relprod(stores:[V1×(V2×FD)], pointsTo:[V1×H1], V1);
18   /* 2.2 */ tmpRel2:[(V1×FD)×H2] = replace(tmpRel1:[(V2×FD)×H1], V2ToV1 & H1ToH2);
19   /* 2.3 */ fieldPt:[(H1×FD)×H2] = relprod(tmpRel2:[(V1×FD)×H2], pointsTo:[V1×H1], V1);
20
21   /* --- rule 3 --- */
22   /* 3.1 */ tmpRel3:[(H1×FD)×V2] = relprod(loads:[(V1×FD)×V2], pointsTo:[V1×H1], V1);
23   /* 3.2 */ newPt4:[V2×H2] = relprod(tmpRel3:[(H1×FD)×V2], fieldPt:[(H1×FD)×H2], H1xFD);
24   /* 3.3 */ newPt5:[V1×H1] = replace(newPt4:[V2×H2], V2ToV1 & H2ToH1]);
25
26   /* --- apply type filtering and add into pointsTo BDD --- */
27   /* 4.1 */ newPt6:[V1×H1] = newPt5:[V1×H1] ∩ typeFilter:[V1×H1];
28   /* 4.2 */ pointsTo:[V1×H1] = pointsTo:[V1×H1] ∪ newPt6:[V1×H1];
29 until pointsTo:[V1×H1] == outerOldPt:[V1×H1]
```

Figure 2.9: Basic BDD-based points-to analysis algorithm from [BLQ+03]

and efficiency. To implement this type filtering in the BDD algorithm, we use the *typeFilter* BDD, which is precomputed to contain all pairs (p, o) of pointers p and objects o such that the run-time type of o is compatible with the declared type of p. In lines 1.3 and 4.2, the sets of new points-to pairs are intersected with the *typeFilter* set, so that only type-compatible points-to pairs are added to *pointsTo*.

The algorithm in Figure 2.9 is very similar to our actual C++ code implementing the analysis using the BuDDy [LN] BDD library. The main difference is that the actual code lacks the physical domain annotations, although we have documented the physical domains of the most important BDDs in comments.

In order to make the implementation reasonably efficient, we had to tune it in two key ways. First, different bit orderings affected analysis time by multiple orders of magnitude. We observed that the relational product operation in line 1.1 of the algorithm took the vast majority of the computation time. After experimenting with different orderings, we found one which made this key operation fast: first testing the bits of physical domains $V1$ and $V2$ interleaved, then testing the bits of the physical domain $H1$.

Second, we obtained an additional two- to ten-fold speedup by incrementalizing the algorithm. In the algorithm as shown in Figure 2.9, all points-to facts are propagated in every iteration. We transformed the algorithm to avoid propagating points-to facts known to have been propagated in an earlier iteration. We refer the reader to [BLQ+03] for details. The resulting incremental implementation is about twice as long as the basic version in Figure 2.9, and appears in [BLQ+03, Appendix A].

After these two optimizations were applied, the BDD-based implementation was measured to be nearly as fast as the highly-tuned traditional points-to analysis implementation in the Spark [Lho02, LH03] framework, and significantly better in terms of memory requirements. Therefore, we conjectured that BDD-based implementations would make it possible to study analyses that have so far required too much memory to be feasible for large programs, such as context-sensitive analysis.

2.4 Conclusion

In this chapter, we have provided background information about BDDs, and reviewed how they were used to implement a basic subset-based points-to analysis for Java [BLQ+03]. The techniques that we have presented here are sufficient for implementing analyses of similar complexity as the simple points-to analysis. In the next chapter, we will explain some of the difficulties that arise when attempting to implement more complicated BDD-based analyses, and present a system to make it possible to implement them. In Chapter 4, we will complement the points-to analysis with a framework of other related BDD-based interprocedural analyses, and extend it to deal with new constraints introduced while the analysis executes, and to be context-sensitive.

Chapter 3

Extending Java with Relations

In this chapter, we present JEDD, a language that we have developed for expressing program analyses in terms of relations, and a system for implementing the analyses using BDDs. We first provide the motivation for and an overview of our approach in Section 3.1. In Section 3.2, we present relations, and show how they can be represented by BDDs. Then, in Section 3.3, we provide details about our design of the JEDD language. In Section 3.4, we illustrate the overall process of using JEDD to implement a program analysis by walking through a complete JEDD reimplementation of the original BDD-based points-to analysis [BLQ+03] that we showed in Figure 2.9 of Chapter 2. The most significant challenge in generating an efficient BDD implementation of a JEDD program is the assignment of physical domains to relation attributes; we provide our solution to this problem in Section 3.5. In Section 3.6, we discuss the JEDD runtime system, and in Section 3.7, we compare the execution speed of JEDD-generated and hand-coded BDD code, and provide measurements of compile-time speed. We survey work related to JEDD in Section 3.8, and conclude in Section 3.9.

3.1 Jedd Motivation and Overview

The simple points-to analysis we described in Section 2.3 and in more detail in [BLQ+03] was our first experiment in using BDDs to implement program analyses. Encouraged by the performance of this analysis, we decided to express more complicated program analyses for Java. As we began this work, we quickly found that implementing our analyses directly in terms of a BDD library was not a good solution, for several reasons. First, because the interface provided by a BDD library is very low level, understanding and maintaining our code became difficult as it grew larger than our initial points-to analysis. Moreover, programming at such a low level was error prone, and the BDD library did not check for many of the errors; instead, our errors caused the library to either crash, or, worse, to execute successfully but produce incorrect results. The implicit nature of the BDD representation made the errors difficult to track down. Although BDD libraries include garbage collectors for the BDD nodes, they require the programmer to manage the root set explicitly using reference counts, and this burden becomes significant in larger programs. Although BDD libraries make it easy to vary the BDD variable ordering, the physical domain assignment is inherent in the code and difficult to change. Both of these parameters have an enormous effect on the performance of relation-based analyses, so we needed to be able to experiment with both of them. Tuning a BDD-based algorithm requires profiling information about the size and shape of the underlying BDDs at each program step. We had previously developed some *ad hoc* methods for visualizing this information, but a more automated approach was needed.

Our solution to these problems, which we call Jedd, consists of several parts.

1. We have defined an extension to the Java language by adding relations and relational operations, so that we can express program analyses as relations within the Soot framework, which is written in Java.

2. We have developed a translator which automatically translates Jedd code to Java code that implements the high-level relational operations by calling into a low-level BDD library.

3. We have developed a run-time support library for interacting with the BDD back-end, which provides automatic memory management and facilities for debugging and profiling BDD operations.

We now briefly describe the key features and contributions of our approach.

BDDs abstracted as relations: Rather than expose BDDs and their low-level operations directly, our JEDD language makes possible a more abstract representation using relations and relational operations. In developing program analyses using BDDs, we have found this to be an appropriate level of abstraction.

Static and dynamic type checking: When using a BDD library directly, there is very little type information to help the programmer determine whether BDD operations are used in a way that makes sense. In JEDD, each relation has a type specifying its schema, and all operations on relations are checked statically to ensure that the schemas of their operands are compatible. Properties that cannot be checked statically, such as the number of bits required to represent all elements of a domain, are enforced by runtime checks. Together, the static and dynamic checks catch many programmer errors that would otherwise make a complicated BDD-based analysis infeasible to implement correctly.

Code generation strategy: JEDD generates low-level BDD code automatically from program analyses expressed at a high level in terms of relations.

Algorithm for physical domain assignment: When programming directly with BDDs, the programmer must explicitly specify a physical domain for every attribute of every relation in the program. This is a tedious process. Furthermore, a small change in physical domain assignment may require many changes in the program. When specifying a program analysis using the JEDD language, the user need provide only a minimal amount of input about the desired assignment, and the translator automatically generates a reasonable assignment for the whole program. However, the programmer retains complete control over the assignment. In those parts of the program where it is desired, the programmer

can provide a more detailed specification to carefully tune the physical domain assignment for efficiency. The problem of assigning physical domains turns out to be NP-complete. We provide an algorithm to express it as an instance of the SAT problem, and we show that, using modern SAT solvers, the time to find a solution is a negligible part of the compilation process. In cases where no solution exists, we provide detailed information precisely indicating the source of the error to the programmer.

Run-time support for memory management: Unlike low-level BDD libraries, which require the programmer to explicitly manage the root set of live BDDs using reference counts, JEDD reclaims BDD nodes automatically as soon as it is safe to do so using a combination of static analysis and interaction with the Java garbage collector.

BDD profiler: In our work with BDDs, we found that to tune the BDD-based algorithms, we needed profile information about the size and shape of the BDD data structures at each program point. Our JEDD system automatically collects this information, and allows the programmer to browse it in an organized way using a web browser. JEDD reports the time taken and number of BDD nodes involved in each operation, and provides graphical figures showing the size and shape of the BDDs at each program point.

A high-level overview of the complete JEDD system is given in Figure 3.1. JEDD programs are written in the JEDD language, an extension of Java, and are provided as input to the `jeddc` compiler. The `jeddc` compiler is composed of a front-end (parser and semantic analysis) and a back-end (physical domain assignment and code generation). The physical domain assignment module uses an external SAT solver. The output of `jeddc` is in the form of standard Java files which can be incorporated into any Java project. The Java files produced by `jeddc`, along with other ordinary Java source making up a project, are compiled to class files using a standard Java compiler such as `javac`. Unless the code written in JEDD is modified, `jeddc` is not needed when recompiling the Java part of the project. The resulting class files

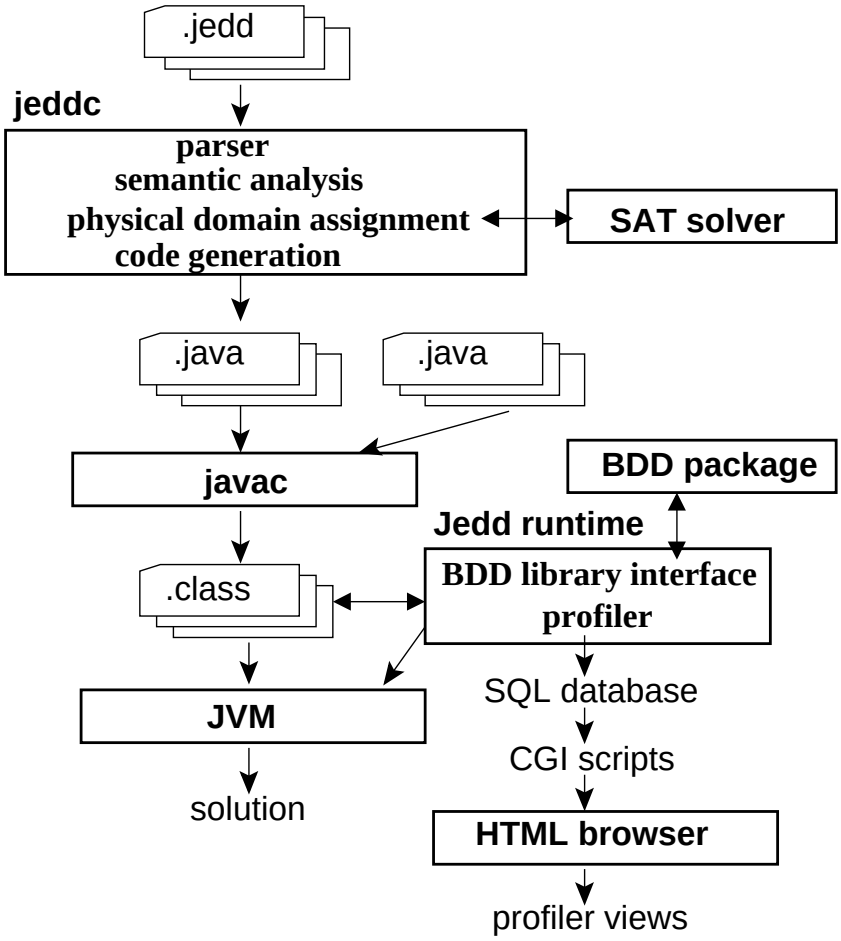


Figure 3.1: Overview of JEDD system

contain calls to the JEDD runtime library, which interfaces using the Java Native Interface (JNI) to a BDD package. A Java Virtual Machine (VM) is used to execute the classes along with the JEDD runtime. The runtime also includes a profiler, which writes profile information into a Structured Query Language (SQL) database. When combined with Common Gateway Interface (CGI) scripts accessing the database, a web browser can be used to navigate profiler views of BDD operations.

3.2 Relations

In JEDD, BDDs are abstracted as relations, and JEDD code is written in terms of relations, rather than directly in terms of BDDs. In this section, we define the terminology we will use to talk about relations, and show how relations are encoded and manipulated in BDDs.

3.2.1 Definitions

We generally follow the accepted terminology used in relational database work (see, for example, [GMUW01, Section 3.1]).

To illustrate the terms on a concrete example, in Figure 3.2, we present two relations representing (a) the initial points-to pairs and (b) the assignment constraints from our points-to analysis example from Chapter 2.

pointer	object
a	X
b	Y
c	Z

(a)

source	dest
b	a
a	b
b	c

(b)

Figure 3.2: Example relations (a) initial points-to pairs (b) assignment constraints

A ***domain***[1] is a set of basic ***elements*** from which we construct relations. In our points-to analysis, we use a domain of pointers, $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, and a domain of abstract objects, $\{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$.

An ***attribute*** is a domain along with an associated name. We use attributes to distinguish different instances of the same domain. For example, in the assignment constraints relation in Figure 3.2(b), source and dest are two attributes with the same domain, pointers.

[1]The term "domain" is used in both the BDD literature and database literature with two different meanings. In this thesis, we say "physical domain" when we mean the BDD sense of the word, and simply "domain" for the database sense.

A ***tuple*** is a collection of elements indexed by attribute. The element corresponding to each attribute is in the domain of that attribute. In Figure 3.2, each row of each relation is a tuple. For example, the first tuple in Figure 3.2(a) contains the element `a` in the pointer attribute and the element `X` in the object attribute.

A ***relation*** is a set of tuples, each with the same attributes. This common set of attributes is the ***schema*** of the relation. The relations in Figure 3.2 have schemas $\{\text{pointer}, \text{object}\}$ and $\{\text{source}, \text{dest}\}$.

3.2.2 Encoding relations in BDDs

To prepare for encoding relations in BDDs, we first assign to each element of every domain a binary vector which is unique within the domain. Within each domain, every binary vector must be of the same length. Continuing with our example from Chapter 2, we may, for instance, assign the binary vector 00 to `a` and `X`, 01 to `b` and `Y`, and 10 to `c` and `Z`.

To represent a relation by a BDD, we first assign a physical domain to each attribute of the relation. Recall that each tuple contains an element for each attribute. To represent an element, we express its binary vector in the physical domain that was assigned to its attribute; we combine the vectors of the elements into a single binary vector for the whole tuple. For example, if we assigned the attributes source and dest of the assignment constraint relation in Figure 3.2(b) to the physical domains $V1$ and $V2$, respectively, the first tuple (`b`, `a`) would be represented by the binary vector 0100, with 01 in the $V1$ physical domain, and 00 in $V2$. A relation is represented by the BDD encoding the set of bit vectors representing its tuples. Therefore, the relations in Figure 3.2 would be encoded by the same BDDs as in Figures 2.5(a) and (b) in Chapter 2, provided that the attributes were assigned to the appropriate physical domains (pointer and object to $V1$ and $H1$, and source and dest to $V1$ and $V2$, respectively).

3.2.3 Manipulating relations in BDDs

Given two relations with the same schema, the operations **union**, **intersection**, **set difference**, and **equality testing** are defined on them as the corresponding operations on the set of their tuples. In BDDs, these relational operations are implemented directly by the corresponding BDD operations. However, each of these operations requires that its operand relations be encoded with the same physical domain assignment. When this is not the case, a replace BDD operation must first be performed to make the physical domain assignments consistent.

The **projection**, **attribute renaming**, and **attribute copying** operations modify the schema of a relation.

The projection operation selects a subset of the attributes from the relation and removes all other attributes. Within each tuple, only the elements associated with the projected attributes are kept; all other elements are removed. Recall that relations are sets of tuples with no duplicates. Since removing an attribute from two tuples that differ only in that attribute makes the tuples equal, a projection may reduce the number of tuples in a relation. Projection is implemented in a BDD by applying the existential quantification operation to each bit position of every physical domain corresponding to an attribute not present in the projection.

Attribute renaming substitutes one attribute for another, without changing the element for the attribute in each tuple. Renaming an attribute of a relation requires no change to the BDD representing it. Only the mapping from attribute to physical domain needs to be updated, with the new attribute replacing the old.

Attribute copying adds a new attribute to a relation, copying the elements of an existing attribute into it. That is, within each tuple, we make a copy of the element for the attribute being copied, and the copy becomes the element for the new attribute. Attribute copying is implemented by first constructing a BDD for the identity relation on the physical domains of the old and new attributes, and intersecting it with the original BDD.

The **join** operation combines the information from two relations into a single relation. Given input relations R, R' and an arbitrary user-specified condition on

tuples, a general join computes the relation consisting of all the tuples of the cross product $R \times R'$ that satisfy the given condition. A common example of such a condition is that the elements of a given list of attributes from R be respectively equal to the elements of a given list of attributes from R'. For example, given the relations shown in Figure 3.2, we may wish to find all tuples in their cross product which match in the pointer and source attributes. A join with this kind of condition is an *equijoin*. In applying BDDs to program analysis, we limit ourselves only to equijoins rather than general joins. Because the elements of the attributes being compared in an equijoin always appear twice in the resulting relation (once coming from an attribute of R and once from the corresponding attribute of R'), we omit the copy coming from R'.

To implement a join in BDDs, we must first carefully set up the physical domain assignment. The attributes being compared must be assigned to the same physical domains in the left and right relations. The remaining attributes must be assigned to physical domains not used by the other relation, so that their elements do not interfere with each other. Assuming we have such a physical domain assignment, the join is computed as the intersection of the BDDs representing the relations.

The **composition** operation is similar to join, but while a join omits one copy of each attribute being compared to an attribute of the other relation, a composition omits both copies. Therefore, a composition is equivalent to a join followed by a projection of the appropriate attributes, and indeed can be implemented this way. We mention it separately for two reasons. First, it tends to be very common in program analyses. Second, it can be implemented by the relational product BDD operation, which is more efficient than an intersection followed by an existential quantification.

3.3 Jedd Language

In this section, we describe the JEDD language for expressing program analyses using relational operations and implementing them using BDDs. To give an idea of what JEDD code looks like, we begin by showing, in Figure 3.3, the JEDD implementation

of the points-to set propagation example from Section 2.2. This JEDD code performs the same propagation as the BDD code that we saw in Figure 2.6.

```
1  <pointer:V1, object:H1> pointsTo;
2  <source, dest:V2> assign;
3
4  <pointer, object> oldPt;
5
6  do {
7      oldPt = pointsTo;
8      <dest, object> tmp = pointsTo{pointer} <> assign{source};
9      pointsTo |= (dest=>pointer) tmp;
10 } while( oldPt != pointsTo );
```

Figure 3.3: JEDD implementation of simple points-to set propagation

Several characteristics of JEDD are apparent from the example. First, JEDD code is written in terms of relations and the relational operations explained in Section 3.2, rather than directly in terms of BDDs and BDD operations. The composition operation is denoted by `<>` (see line 8), union is denoted `|`, and an assignment of the form `pointsTo = pointsTo | ...` can be abbreviated as `pointsTo |= ...` (see line 9). Second, the schema of each relation variable is explicit in its declared type. This makes it possible for the JEDD translator to check that the schemas of the relations involved in each operation are consistent. Third, physical domains can be specified for some attributes; in this case, they are specified for three of the attributes (`pointer` and `object` in line 1 and `dest` in line 2). The JEDD translator automatically finds a reasonable[2] physical domain assignment for those attributes for which physical domains are not explicitly specified. In particular, this includes the various subexpressions within each expression. Each physical domain to be used in the assignment must be mentioned explicitly at least once in the program, but the programmer may choose to make the assignment explicit in additional key relations where desired. A typical

[2]We will give a precise definition of a reasonable physical domain assignment in Section 3.5.3.

system of program analyses, such as the PADDLE system described in Chapter 4, contains on the order of twenty physical domains and thousands of attribute instances,[3] so the requirement to explicitly mention each physical domain at least once is not a significant burden. In comparison, an implementation using a low-level BDD library directly would have to specify a physical domain for every attribute instance.

3.3.1 Grammar

Because JEDD is an extension of Java, we used the Java grammar [GJS96, ch. 19] as a starting point for a JEDD grammar, and removed and added some alternatives and productions. The changes to the grammar are shown in Figure 3.4. Non-terminals from the original Java grammar appear in italics.

First, we added a relation schema as a new kind of type specification. A relation schema consists of a set of attributes, optionally with physical domains to which they are to be assigned. Both attributes and physical domains are specified by class names.

Second, we added the various relational operations. The original Java grammar contains a chain of non-terminals representing different kinds of expressions at successive levels of precedence. For JEDD, we have inserted two kinds of expressions, ⟨RelExprJoin⟩ and ⟨RelExpr⟩, with precedence in between ⟨*UnaryExpressionNotPlusMinus*⟩ and ⟨*PostfixExpression*⟩. The complete chain of non-terminals for expressions is shown in Figure 3.5. A ⟨RelExprJoin⟩ can be a join or composition (denoted with the symbols `><` and `<>`, respectively, suggested by the standard notation ⋈ and ∘), or an expression of higher precedence. Join and composition have equal precedence. A ⟨RelExpr⟩ can be an attribute operation (projection, renaming, or copy), or an expression of higher precedence. The attribute operations are expressed as a list of replacements. Each replacement specifies the original attribute to be affected, followed by the symbol `=>`, followed by zero, one, or two attributes, indicating that the attribute be removed, renamed to a different attribute, or copied

[3] We use the term *attribute instance* to distinguish the instances of the same attribute appearing in different relations. For example, the code in Figure 3.3 contains two instances of the attribute `dest`, in the relations `assign` and `tmp`.

Added alternatives and productions:

⟨*Type*⟩ ::= *(standard Java alternatives)* | ‘<’ ⟨AttributePhys⟩ (‘,’ ⟨AttributePhys⟩)* ‘>’

⟨AttributePhys⟩ ::= ⟨Attribute⟩ | ⟨Attribute⟩ ‘:’ ⟨Attribute⟩

⟨Attribute⟩ ::= ⟨*ClassOrInterfaceType*⟩

⟨*UnaryExpressionNotPlusMinus*⟩ ::= *(standard Java alternatives)* | ⟨RelExprJoin⟩

⟨RelExprJoin⟩ ::= ⟨RelExpr⟩ | ⟨Join⟩

⟨Join⟩ ::= ⟨RelExprJoin⟩ ⟨AttrList⟩ ⟨JoinSym⟩ ⟨RelExpr⟩ ⟨AttrList⟩

⟨AttrList⟩ ::= ‘{’ ⟨Attribute⟩ (‘,’ ⟨Attribute⟩)* ‘}’

⟨JoinSym⟩ ::= ‘>’ ‘<’ | ‘<’ ‘>’

⟨RelExpr⟩ ::= ⟨Replace⟩ | ⟨*PostfixExpression*⟩

⟨Replace⟩ ::= ‘(’ ⟨Replacement⟩ (‘,’ ⟨Replacement⟩)* ‘)’ ⟨RelationExpr⟩

⟨Replacement⟩ ::= ⟨Attribute⟩ ‘=>’ | ⟨Attribute⟩ ‘=>’ ⟨Attribute⟩
| ⟨Attribute⟩ ‘=>’ ⟨Attribute⟩ ⟨Attribute⟩

⟨*Literal*⟩ ::= *(standard Java alternatives)*
| ‘new’ ‘{’ ⟨LiteralPiece⟩ (‘,’ ⟨LiteralPiece⟩)* ‘}’ | ‘0B’ | ‘1B’

⟨LiteralPiece⟩ ::= ⟨*Expression*⟩ ‘=>’ ⟨AttributePhys⟩

Removed alternative:

⟨*UnaryExpressionNotPlusMinus*⟩ ::= ~~⟨*PostfixExpression*⟩~~
| *(other standard Java alternatives)*

Figure 3.4: JEDD grammar productions

to two attributes, respectively. Because the attribute operations change the type of a relation, the replacement list is enclosed in parentheses, like a Java cast.

Third, we added two new kinds of literals. The constant literals `0B` and `1B` represent the empty relation and the full relation (containing all possible tuples of the schema), respectively. Much like Java's `null` constant, they are comparable and assignable to any relation type, and assume the schema imposed by the type to which they are compared or assigned. JEDD also provides an easy way to create new tuples from Java objects. For example, the expression `new {srcPtr=>source, dstPtr=>dest}` creates a relation consisting of one tuple, with the Java objects `srcPtr` and `dstPtr` in the attributes `source` and `dest`, respectively.

Unfortunately, Java's C roots make it difficult to write a clean LALR(1) grammar for it; some of the necessary workarounds are discussed in the introduction to the grammar itself [GJS96, ch. 19]. Keeping the JEDD extension of the grammar LALR(1) proved to be difficult as well. If we applied the changes in Figure 3.4 directly to the original Java grammar, it would no longer be LALR(1). The operands of a join or composition can be primaries, which in Java include class instance creation expressions, which have an optional trailing class body enclosed in curly braces. A LALR(1) parser cannot distinguish this body from the attribute list following the operand in the join or composition. However, the type of a class instance creation is never a relation type, so a class instance creation is never a legal operand to a join or composition, so we can exclude it in this case. Therefore, prior to applying the changes in Figure 3.4, we performed a series of language preserving transformations, removing class instance creation expressions from primaries, and adding them in all places where primaries can occur (except the join production that we added). These modifications are listed in Figure 3.6. The result is a LALR(1) grammar for JEDD which extends Java in a natural way.

$$
\begin{aligned}
\langle \mathit{Expression} \rangle &\Rightarrow \\
\langle \mathit{AssignmentExpression} \rangle &\Rightarrow \\
\langle \mathit{ConditionalExpression} \rangle &\Rightarrow \\
\langle \mathit{ConditionalOrExpression} \rangle &\Rightarrow \\
\langle \mathit{ConditionalAndExpression} \rangle &\Rightarrow \\
\langle \mathit{InclusiveOrExpression} \rangle &\Rightarrow \\
\langle \mathit{ExclusiveOrExpression} \rangle &\Rightarrow \\
\langle \mathit{AndExpression} \rangle &\Rightarrow \\
\langle \mathit{EqualityExpression} \rangle &\Rightarrow \\
\langle \mathit{RelationalExpression} \rangle &\Rightarrow \\
\langle \mathit{ShiftExpression} \rangle &\Rightarrow \\
\langle \mathit{AdditiveExpression} \rangle &\Rightarrow \\
\langle \mathit{MultiplicativeExpression} \rangle &\Rightarrow \\
\langle \mathit{UnaryExpression} \rangle &\Rightarrow \\
\langle \mathit{UnaryExpressionNotPlusMinus} \rangle &\Rightarrow \\
\langle \mathsf{RelExprJoin} \rangle &\Rightarrow \\
\langle \mathsf{RelExpr} \rangle &\Rightarrow \\
\langle \mathit{PostfixExpression} \rangle &\Rightarrow \\
\langle \mathit{Primary} \rangle &\Rightarrow \\
\langle \mathit{PrimaryNoNewArray} \rangle &\Rightarrow \\
\langle \mathit{Literal} \rangle &
\end{aligned}
$$

Figure 3.5: Chain of expression precedences in Java and Jedd

Added alternatives:

⟨*ArrayAccess*⟩ ::= *(standard Java alternatives)*
| ⟨*ClassInstanceCreationExpression*⟩ '`[`' ⟨*Expression*⟩ '`]`'

⟨*ExplicitConstructorInvocation*⟩ ::= *(standard Java alternatives)*
| ⟨*ClassInstanceCreationExpression*⟩ '`.`' '`this`' '`(`' ⟨*ArgumentListOpt*⟩ '`)`' '`;`'
| ⟨*ClassInstanceCreationExpression*⟩ '`.`' '`super`' '`(`' ⟨*ArgumentListOpt*⟩ '`)`' '`;`'

⟨*ClassInstanceCreationExpression*⟩ ::= *(standard Java alternatives)*
| ⟨*ClassInstanceCreationExpression*⟩ '`.`' '`new`' ⟨*SimpleName*⟩ '`(`' ⟨*ArgumentListOpt*⟩ '`)`'
| ⟨*ClassInstanceCreationExpression*⟩ '`.`' '`new`' ⟨*SimpleName*⟩ '`(`' ⟨*ArgumentListOpt*⟩ '`)`' ⟨*ClassBody*⟩

⟨*FieldAccess*⟩ ::= *(standard Java alternatives)*
| ⟨*ClassInstanceCreationExpression*⟩ '`.`' IDENTIFIER

⟨*MethodInvocation*⟩ ::= *(standard Java alternatives)*
| ⟨*ClassInstanceCreationExpression*⟩ '`.`' IDENTIFIER '`(`' ⟨*ArgumentListOpt*⟩ '`)`'

⟨*UnaryExpressionNotPlusMinus*⟩ ::= *(standard Java alternatives)*
| ⟨*ClassInstanceCreationExpression*⟩

Removed alternative:

⟨*PrimaryNoNewArray*⟩ ::= ~~⟨*ClassInstanceCreationExpression*⟩~~
| *(other standard Java alternatives)*

Figure 3.6: Grammar transformations to keep JEDD grammar LALR(1)

3.3.2 Declaring domains, attributes, physical domains, and numberers

All domains, attributes, physical domains, and numberers used in a JEDD program must be declared by the programmer. Each of these entities is declared by writing a class implementing, respectively, the `jedd.Domain`, `jedd.Attribute`, `jedd.PhysicalDomain`, or `jedd.Numberer` interface. The interfaces ensure that the required information about each entity is available at run time. However, for domains, attributes, and physical domains, some of the information is required by the JEDD translator, and must therefore be available at compile time. We have slightly extended the syntax of class declarations to allow the programmer to annotate domains, attributes and physical domains with this compile-time information.[4]

3.3.2.1 Domains

To declare a domain, the programmer must specify the number of BDD bits that will be required to encode each element of the domain, and the mapping between Java objects and binary vectors. An example domain declaration for the domain of pointer variables in our points-to analysis example is shown in Figure 3.7.

```
1 public class VarDomain(2) extends Domain {
2     public Numberer numberer() { return new VarNodeNumberer(); }
3 }
```

Figure 3.7: Example domain declaration

The number of bits (two, in our example) is specified in parentheses immediately after the name of the domain. The JEDD translator ensures that any physical domain in which elements of the domain may be encoded contains at least this many bits.

[4]If JEDD were based on Java 1.5, it would be appropriate to use the standard Java annotation mechanism to specify these annotations. However, JEDD was written before Java 1.5 was defined, so we had to add an annotation syntax of our own. As soon as Polyglot [NCM03] supports Java 1.5-style annotations, we anticipate that it will be a simple task to modify JEDD to use them instead.

The JEDD run-time system ensures that the binary-vector encoding of any element of the domain consists of at most this many bits.

The mapping between Java objects and binary vectors is only needed at run-time. It is specified for the domain by implementing the `numberer()` method to return a numberer object that will convert between Java objects and binary-vector representations. In our example, the method returns a `VarNodeNumberer` object, which we will implement below in Section 3.3.2.4.

3.3.2.2 Attributes

An attribute declaration must specify the domain of the attribute in parentheses after the attribute name. Figure 3.8 shows an example declaration of the `src` attribute from our running example, with the domain `VarDomain`.

```
1 public class src(VarDomain) extends Attribute {}
```

Figure 3.8: Example attribute declaration

3.3.2.3 Physical domains

A declaration of a physical domain does not require any additional information besides its name. An example declaration of the `V1` physical domain from our running example is shown in Figure 3.9.

```
1 public class V1() extends PhysicalDomain {}
```

Figure 3.9: Example physical domain declaration

3.3.2.4 Numberers

The purpose of a numberer is to map Java objects to the binary vectors that encode them in BDDs, and *vice versa*. The `jedd.Numberer` interface requires a numberer to implement two methods:

- `Object get(long)` takes a binary vector stored as a 64-bit integer and returns the corresponding Java object, and
- `long get(Object)` takes a Java object and returns the corresponding binary vector in a 64-bit integer.

The example numberer shown in Figure 3.10 implements the numbering of pointer variables in our running example. The pointer variables `a`, `b`, and `c` are mapped to the binary vectors 00 (0), 01 (1), and 10 (2), respectively.

```
1  public class varnodenumberer implements numberer {
2      public object get(long number) {
3          switch(number) {
4              case 0: return varnode.v("a");
5              case 1: return varnode.v("b");
6              case 2: return varnode.v("c");
7          }
8      }
9      public long get(object o) {
10         if(o.equals(varnode.v("a"))) return 0;
11         if(o.equals(varnode.v("b"))) return 1;
12         if(o.equals(varnode.v("c"))) return 2;
13     }
14 }
```

Figure 3.10: Example numberer

3.3.2.5 Specifying physical domain ordering

The order of the bit positions of physical domains in the BDDs manipulated by JEDD is specified by calling the method `jedd.Jedd.setOrder(jedd.order.Order)`. This method takes as its argument a tree data structure representing the desired ordering. Each subtree of the tree specifies a sequence of the bit positions of the physical domains; the complete tree specifies the complete sequence of all physical domains. Each leaf of the tree is a physical domain, and each internal node is one of the implementors of the `jedd.order.Order` interface, each of which specifies a different way to order the bit positions of its subtrees relative to each other. The following five node implementations are included in JEDD because they were found to be useful in developing the program analyses described in this thesis. JEDD users can implement additional kinds of nodes as needed by implementing the interface, which requires writing a method to generate the desired ordering of bits.

`Seq:` The `Seq` node arranges the bit positions of its subtrees sequentially. All bits of the first subtree are placed first, followed by all bits of the second subtree, then all bits of the third subtree, and so on.

`Interleave:` The `Interleave` node interleaves the bit positions of its subtrees. It first returns the first bit of every subtree, followed by the second bit of every subtree, then the third bit of every subtree, and so on.

`Rev:` The `Rev` node has exactly one child. It returns the bit positions of its subtree in reverse order.

`AsymInterleave:` Like the `Interleave` node, the `AsymInterleave` node interleaves the bit positions of its subtrees. However, rather than taking one bit from each subtree at a time, it can take different numbers of bits from different subtrees. Each subtree is annotated with the number of bits that should be taken from it on each iteration. For example, if an `AsymInterleave` node has two subtrees annotated two and three, it constructs an order consisting of bits one and two of the first subtree, followed by bits one, two, and three of the second subtree,

```
1    Jedd.v().setOrder( new Seq( FD.v(),
2                                new Interleave( V1.v(),
3                                                V2.v()),
4                                H1.v(),
5                                H2.v()));
```

Figure 3.11: Example of setting the bit position ordering

followed by bits three and four of the first subtree, followed by bits four, five, and six of the second subtree, and so on.

`Permute:` Like the `Rev` node, the `Permute` node has exactly one child, but additionally takes an integer argument k. It constructs a permutation of the bit positions of its subtree by taking every kth bit until the end of the bit sequence, then starting again from the first bit that has not yet been taken. For example, if k is three, the resulting sequence consists of bits one, four, seven, ... of the subtree, followed by bits two, five, eight, ..., followed by bits three, six, nine, ...

At run time, JEDD checks that the tree passed to the `setOrder()` method contains exactly one instance of every physical domain declared in the program.

Figure 3.11 shows an example of setting the bit position ordering to the ordering that we found to work well for points-to analysis [BLQ+03]. The bits of the `FD` physical domain are tested first, followed by the bits of the `V1` and `V2` physical domains interleaved, followed by the bits of the `H1` physical domain, and finally the bits of the `H2` physical domain.

3.3.3 Extracting information from relations

An important part of a language extension integrating relations into Java are facilities for extracting information from relations back to Java. JEDD provides two versions of `java.util.Iterator` for iterating over the tuples of a relation. The first works on relations with a single attribute, and in each iteration returns the single object in

each tuple. The example code in Figure 3.12 shows how this iterator is used to print the points-to set for a given pointer variable. The second iterator works on relations of any size, and iterates over the tuples, returning each tuple as an array of objects. An example of how this iterator is used to iterate over the simple assignment relation of subset constraints is shown in Figure 3.13. These iterators are used to implement a `toString()` method on relations, which is very useful for debugging JEDD programs. Without such a method, it would be very difficult to interpret the structure of a BDD to determine the relation it represents.

```
1   /** Prints the targets of the pointer variable represented by vn. */
2   void printPointsToSet(<pointer, object> pointsTo, VarNode vn ) {
3       // extract points-to set for pointer vn
4       <object> pointsToSet = pointsTo{pointer} <> new{vn=>pointer}{pointer};
5
6       // iterate over points-to set
7       Iterator it = pointsToSet.iterator();
8       while( it.hasNext() ) {
9           AllocNode an = (AllocNode) it.next();
10          System.out.println( an.toString() );
11      }
12  }
```

Figure 3.12: Example use of single-attribute iterator

JEDD also provides a `size()` method that returns the number of tuples in a relation. JEDD provides additional statistics about the BDD representations of relations as part of its profiling framework, which is described in Section 3.6.3.

3.3.4 Type checking

Polyglot includes a complete semantic checker for Java. We extended this checker to infer the schemas of relational expressions from their subexpressions, and statically enforce the properties shown in Figure 3.14. The most important properties are that

```
1   /** Prints the pointer assignment edges. */
2   void printEdges(<source, dest> edges) {
3       // Iterate over assignment edges, specifying that
4       // the source attribute to be the zeroth array element, and
5       // the dest attribute to be the first array element.
6       Iterator it = edges.iterator(new Attribute[] {source.v(), dest.v()});
7       while( it.hasNext() ) {
8           Object[] edge = (Object[]) it.next();
9           System.out.println( "Pointer assignment from "+edge[0].toString()+
10              " to "+edge[1].toString() );
11      }
12  }
```

Figure 3.13: Example use of multi-attribute iterator

no relation may have more than one instance of the same attribute, that operands of set and comparison operations have compatible schemas, and that the attributes mentioned in attribute manipulation, join, and composition expressions exist in the corresponding operands.

3.4 Complete Example

We now illustrate all the steps in the development of a Jedd program by walking through a complete reimplementation in Jedd of our original BDD-based points-to analysis solver [BLQ+03]. While Figure 2.9 showed only the core of the BDD-based implementation of the points-to set propagation algorithm, in this section, in Figures 3.15 through 3.19, we present the entire Jedd implementation. The code shown in these figures can be run through the Jedd translator, and the resulting Java bytecode can then be executed by a Java virtual machine. Specifically, the Jedd code shown in Figure 3.16 of this section corresponds to the portion of the BuDDy code that was shown in Figure 2.9.

$$\frac{a_i = a_j \Rightarrow i = j \quad a_i <: \texttt{jedd.Attribute}}{\texttt{new}\ \{o_1\texttt{=>}a_1, \ldots, o_n\texttt{=>}a_n\} : \{a_1, \ldots, a_n\}}[\textsf{Literal}]$$

$$\frac{x : T \quad a \in T \quad a <: \texttt{jedd.Attribute}}{(a\texttt{=>})x : T \setminus \{a\}}[\textsf{Project}]$$

$$\frac{x : T \quad a \in T \quad b \notin T \quad a, b <: \texttt{jedd.Attribute}}{(a\texttt{=>}b)x : (T \setminus \{a\}) \cup \{b\}}[\textsf{Rename}]$$

$$\frac{\begin{array}{c} x : T \quad a \in T \quad b, c \notin T \setminus \{a\} \\ b \neq c \quad a, b, c <: \texttt{jedd.Attribute} \end{array}}{(a\texttt{=>}b\ c)x : (T \setminus \{a\}) \cup \{b, c\}}[\textsf{Copy}]$$

$$\frac{x : T \quad y : T}{x \odot y : T \text{ where } \odot \in \{\texttt{\&}, \texttt{|}, \texttt{-}\}}[\textsf{SetOp}]$$

$$\frac{x : T \quad y : T \vee y \in \{\texttt{0B}, \texttt{1B}\}}{x \odot y : T \text{ where } \odot \in \{\texttt{=}, \texttt{\&=}, \texttt{|=}, \texttt{-=}\}}[\textsf{Assign}]$$

$$\frac{x : T \vee x \in \{\texttt{0B}, \texttt{1B}\} \quad y : T \vee y \in \{\texttt{0B}, \texttt{1B}\}}{x \odot y : \texttt{boolean} \text{ where } \odot \in \{\texttt{==}, \texttt{!=}\}}[\textsf{Compare}]$$

$$\frac{\begin{array}{c} x : T \quad y : U \quad U' = U \setminus \{b_1, \ldots b_n\} \quad T \cap U' = \emptyset \\ \{a_1, \ldots, a_n\} \subseteq T \quad \{b_1, \ldots, b_n\} \subseteq U \\ a_i = a_j \Rightarrow i = j \quad b_i = b_j \Rightarrow i = j \\ a_i, b_i <: \texttt{jedd.Attribute} \end{array}}{x\{a_1, \ldots, a_n\}\texttt{><}y\{b_1, \ldots, b_n\} : T \cup U'}[\textsf{Join}]$$

$$\frac{\begin{array}{c} x : T \quad y : U \quad T' \cap U' = \emptyset \\ T' = (T \setminus \{a_1, \ldots, a_n\}) \quad U' = (U \setminus \{b_1, \ldots b_n\}) \\ \{a_1, \ldots, a_n\} \subseteq T \quad \{b_1, \ldots, b_n\} \subseteq U \\ a_i = a_j \Rightarrow i = j \quad b_i = b_j \Rightarrow i = j \\ a_i, b_i <: \texttt{jedd.Attribute} \end{array}}{x\{a_1, \ldots, a_n\}\texttt{<>}y\{b_1, \ldots, b_n\} : T' \cup U'}[\textsf{Compose}]$$

Figure 3.14: Typing rules

```
1  import jedd.*;
2  import jedd.order.*;
3
4  class Propagator {
5    public static <pointer,object> propagate(
6      <source,dest> assign,
7      <object,pointer> allocs,
8      <source,field,dest> stores,
9      <source,field,dest> loads,
10     <pointer,object> typeFilter
11   ) {
12     <pointer,object> pointsTo = allocs;
13     <pointer,object> outerOldPt;
14     <pointer,object> innerOldPt;
```

Figure 3.15: Complete JEDD code for points-to analysis of [BLQ+03] (part 1 of 5)

We begin the presentation of the solver in Figure 3.15. Lines 1–2 import the packages of the JEDD run-time library. In line 4, we begin a `Propagator` class containing a static method `propagate()`, which will implement the points-to set propagation algorithm. The method takes relations containing the points-to set constraints to be solved as parameters. The return value of the method is a relation that will contain the computed points-to relation. In lines 12–14, we declare three local relation variables for use by the propagation algorithm. The `pointsTo` variable will store the points-to relation computed so far. We initialize it in line 12 with the initial points-to pairs due to allocation statements. The other two variables will be used to save the old points-to relation at the beginning of each loop of the algorithm; at the end of each loop, they will be used to determine whether the points-to relation has changed in the current iteration.

Figure 3.16 shows the JEDD implementation of the core points-to set propagation algorithm. The abstract algorithm is the same as in the BUDDY implementation in Figure 2.9, but now it is expressed in terms of JEDD relational operations, which are

independent of any specific BDD implementation details. Specifically, each relation is defined over a set of abstract attributes, and at this point, we have not specified the physical domain of BDD variables in which each attribute instance will be encoded. In the BuDDy implementation, we had to insert BDD `replace` operations to move attributes to specific physical domains required by each BDD operation. In the Jedd version, we express only the relational operations that we want performed, and leave it to the Jedd translator to allocate the attribute instances to appropriate physical domains, and move them with `replace` operations when necessary.

From the high-level Jedd code shown in Figure 3.16, the Jedd translator will automatically generate low-level BDD code like the code we showed in Figure 2.9.

In Figure 3.17, we show how to define the numberer, domains, and attributes used by the Jedd implementation of the algorithm in Figure 3.16.

The numberer defines a bijective mapping between the Java objects that we want to store in relations and `long` integers. Our original points-to constraint solver [BLQ+03] read points-to constraints from an input file in which each pointer variable, abstract heap object, and field was designated by a unique integer. Therefore, for this particular program, the Java objects to be stored in relations are all of type `java.lang.Long`, and the bijection is easy to define: each `java.lang.Long` object is mapped to the `long` returned by its `longValue()` method. In general, any Java objects could be stored in relations. Indeed, the Paddle framework which we describe in Chapter 4 uses different types of Java objects for each kind of element that it stores in relations. Each numberer must define two `get()` methods, one to convert a `long` integer to the corresponding Java object, and the other to convert a Java object to the corresponding `long` integer. The implementation of these two methods for the points-to set propagation example is shown in lines 56 and 57, respectively.

In lines 62 through 67, we define the three domains used in the algorithm: pointer variables, abstract heap objects, and fields. We specify that up to 10 bits may be required to encode fields, and up to 20 bits may be required to encode objects in the other two domains. In general, the Java objects to be stored in each domain could use a distinct numberer, but in this example, we reuse the same numberer for all four domains.

```
15
16      do {
17        outerOldPt = pointsTo;
18
19        do {
20          innerOldPt = pointsTo;
21
22          /* --- rule 1 --- */
23          <pointer,object> newPt = pointsTo{pointer}
24                  <> (dest=>pointer) assign{source};
25
26          /* --- apply type filtering and add into pointsTo BDD  --- */
27          newPt &= typeFilter;
28          pointsTo |= newPt;
29        } while( pointsTo != innerOldPt );
30
31        /* --- rule 2 --- */
32        <object,pointer,field> objectsBeingStored = (dest=>pointer) stores{source}
33                                                              <> pointsTo{pointer};
34        <base,field,object> fieldPt = objectsBeingStored{pointer}
35                              <> (object=>base) pointsTo{pointer};
36
37        /* --- rule 3 --- */
38        <base,field,dest> loadsFromHeap = loads{source}
39                    <> (object=>base) pointsTo{pointer};
40        <pointer,object> newPt = (dest=>pointer) loadsFromHeap{base,field}
41                                                    <> fieldPt{base,field};
42
43        /* --- apply type filtering and add into pointsTo BDD --- */
44        newPt &= typeFilter;
45        pointsTo |= newPt;
46      } while( pointsTo != outerOldPt );
47      return pointsTo;
48    }
49  }
```

Figure 3.16: Complete JEDD code for points-to analysis of [BLQ+03] (part 2 of 5)

Finally, in lines 71 through 78, we associate the attributes used in the points-to set propagation algorithm with their domains. The attributes `source`, `dest`, and `pointer` are of the domain `Var` of pointer variables, the attributes `object` and `base` are of the domain `Obj` of abstract heap objects, and the attribute `field` is of the domain `Field` of fields.

In Figure 3.18, we begin to connect the high-level relational specification of our algorithm with an actual BDD representation. In lines 82 through 86, we declare the five physical domains of BDD variables in which the relations will be encoded. These are the same five physical domains that we used in the BDD implementation of the algorithm in Figure 2.9. In line 88, we begin the class `Main` which will contain the `main()` method, and also generate the input points-to set constraints to be solved. The relations to store the input points-to set constraints are declared in lines 89 through 93.

At this point, we have decided to specify the physical domains in which some of the attribute instances will be encoded (`V1`, `V2`, `H1`, and `FD` in lines 89 through 91). We expect the JEDD translator to automatically find a reasonable assignment of physical domains to all other attribute instances of all relations in the program, including all the relations in the propagation algorithm in Figure 3.16. However, if we run the JEDD translator on the program in its current form, it will output an error indicating a problem with the `fieldPt` relation declared on line 34, namely that the attributes `base` and `object` must be assigned to distinct physical domains, but only the `H1` physical domain is available for both of them. Recall that every physical domain that JEDD is to use in its assignment must be explicitly specified at least once in the program. So far, we have not yet explicitly specified the `H2` physical domain for any attribute instance, so it cannot be used. Therefore, in light of the error report at line 34, we decide to explicitly assign the `object` attribute of the `fieldPt` relation to the `H2` physical domain by modifying line 34 as shown at the bottom of Figure 3.18. In general, once we have defined a high-level relational implementation of an algorithm, finding a physical domain assignment is an iterative process: we first specify physical domains for a small number of attribute instances, then run the JEDD translator to find relations for which it cannot find a reasonable

```
50
51 // Define numberer
52 ////////////////////////////////////////////////////////////////////
53 class LongNumberer implements Numberer {
54   private static LongNumberer instance = new LongNumberer();
55   public static LongNumberer v() { return instance; }
56   public Object get( long number ) { return new Long(number); }
57   public long get( Object o ) { return ((Long) o).longValue(); }
58 }
59
60 // Define domains
61 ////////////////////////////////////////////////////////////////////
62 class Var(20) extends Domain
63   { public Numberer numberer() { return LongNumberer.v(); } }
64 class Obj(20) extends Domain {
65   { public Numberer numberer() { return LongNumberer.v(); } }
66 class Field(10) extends Domain {
67   { public Numberer numberer() { return LongNumberer.v(); } }
68
69 // Define attributes
70 ////////////////////////////////////////////////////////////////////
71 class source(Var) extends Attribute {}
72 class dest(Var) extends Attribute {}
73 class pointer(Var) extends Attribute {}
74
75 class object(Obj) extends Attribute {}
76 class base(Obj) extends Attribute {}
77
78 class field(Field) extends Attribute {}
79
```

Figure 3.17: Complete JEDD code for points-to analysis of [BLQ+03] (part 3 of 5)

```
80  // Define physical domains
81  ////////////////////////////////////////////////////////////////
82  class V1() extends PhysicalDomain {}
83  class V2() extends PhysicalDomain {}
84  class H1() extends PhysicalDomain {}
85  class H2() extends PhysicalDomain {}
86  class FD() extends PhysicalDomain {}
87
88  public class Main {
89    <source:V1,dest:V2> mAssign;
90    <object:H1,pointer> mAllocs;
91    <source,field:FD,dest:V2> mStores;
92    <source,field,dest> mLoads;
93    <pointer,object> mTypeFilter;
94
95    public static final void main( String args ) {
96      Jedd.v().setBackend("buddy");
97      Jedd.v().setOrder(
98        new Seq(FD.v(), new Interleave(V1.v(), V2.v()), H1.v(), H2.v()) );
99
```

```
34        <base,field,object:H2> fieldPt = objectsBeingStored{pointer}
```

Figure 3.18: Complete JEDD code for points-to analysis of [BLQ+03] (part 4 of 5)

physical domain assignment, and add explicitly specified physical domains for those relations. In the case of our points-to set propagation example, a single iteration of this process is enough: the physical domains that we have specified in lines 89 through 91 and in line 34 are sufficient for JEDD to automatically find a reasonable physical domain assignment for all other relations in the program. In case we are not satisfied with some part of the physical domain assignment (for example, if we find a performance bottleneck using the profiler), we could constrain it further by continuing to specify additional physical domains explicitly, and using JEDD to automatically find a reasonable physical domain assignment for the rest of the program.

The `main()` method begins in line 95. First, in line 96, it initializes JEDD and selects the BUDDY backend. In lines 97 through 98, the `main()` method sets the relative ordering of physical domains. The `FD` physical domain appears first in each BDD, followed by the `V1` and `V2` physical domains interleaved, followed by `H1` and `H2`. This was the physical domain ordering that we found to be most efficient for points-to set propagation [BLQ+03].

Figure 3.19 shows the remainder of the `main()` method and `Main` class. The `main()` method generates a sample set of input points-to constraints (line 101), calls the propagation algorithm to solve them (lines 102 through 103), and prints out the resulting points-to relation (lines 105 through 106). The `initializeConstraints()` method in lines 109 through 130 loads the points-to constraints of one of our test cases into the points-to constraint relations in the `Main` class.

3.5 Assigning Physical Domains to Attributes

As we have seen in the example in the previous section, one important problem when implementing algorithms using BDDs is deciding how to assign each attribute instance to a physical domain of BDD variables. We now show how JEDD automates this task. First, in Section 3.5.1, we present the objectives which motivated the design of the physical domain assignment algorithm. In Section 3.5.2, we formalize these objectives as explicit constraints that a reasonable assignment must satisfy.

```
100     Main m = new Main();
101     m.initializeConstraints();
102     <pointer,object> pointsTo = Propagator.propagate(
103       m.mAssign, m.mAllocs, m.mStores, m.mLoads, m.mTypeFilter);
104
105     System.out.println("Points-to relation:");
106     System.out.println(pointsTo.toString());
107   }
108
109   private void initializeConstraints() {
110     mAllocs |= new{new Long(1)=>object, new Long(1)=>pointer};
111     mAllocs |= new{new Long(2)=>object, new Long(1)=>pointer};
112     mAllocs |= new{new Long(2)=>object, new Long(2)=>pointer};
113     mAllocs |= new{new Long(3)=>object, new Long(2)=>pointer};
114     mAllocs |= new{new Long(3)=>object, new Long(3)=>pointer};
115     mAllocs |= new{new Long(4)=>object, new Long(3)=>pointer};
116     mAllocs |= new{new Long(5)=>object, new Long(5)=>pointer};
117
118     mAssign |= new{new Long(3)=>source, new Long(4)=>dest};
119     mAssign |= new{new Long(7)=>source, new Long(4)=>dest};
120
121     mStores |= new{new Long(4)=>source, new Long(1)=>dest, new Long(1)=>field};
122     mStores |= new{new Long(5)=>source, new Long(2)=>dest, new Long(1)=>field};
123     mStores |= new{new Long(6)=>source, new Long(3)=>dest, new Long(1)=>field};
124
125     mLoads |= new{new Long(1)=>source, new Long(1)=>field, new Long(7)=>dest};
126     mLoads |= new{new Long(2)=>source, new Long(1)=>field, new Long(8)=>dest};
127     mLoads |= new{new Long(3)=>source, new Long(1)=>field, new Long(9)=>dest};
128
129     mTypeFilter = 1B;
130   }
131 }
132
```

Figure 3.19: Complete JEDD code for points-to analysis of [BLQ+03] (part 5 of 5)

Next, in Section 3.5.3, we present our algorithm for solving the constraints. Finally, in Section 3.5.4, we present the error recovery mechanism which provides meaningful error messages to the programmer.

3.5.1 Objectives

Our objectives for the design of the physical domain assignment algorithm fall into three main categories. First, we aimed to minimize the amount of work required of the programmer. Second, we wanted to make it possible to precisely specify different physical domain assignments, and to easily change them, with the overall goal of finding assignments that make the analysis execute efficiently. Third, we wanted an algorithm which could be practically implemented in a usable tool. In the rest of this section, we explain these objectives in more detail.

The first two objectives seem contradictory, since a very flexible system can be obtained by requiring the user to specify every detail, while an automatic system offering no choices requires little from the user. Therefore, one of the challenges was to find a reasonable compromise between these two extremes.

A programmer using a BDD library directly must map each attribute instance to a physical domain by hand, and write the program in terms of the physical domains, rather than attributes. For simple programs of several BDD expressions with two or three attributes, this is acceptable. However, for more complicated programs, assigning a valid physical domain to each attribute of every subexpression is both tedious and error-prone. It is tedious because there are so many attributes to which physical domains must be assigned, and it is error-prone because the many replace operations which move data to the assigned physical domains must be inserted by hand, with no automatic verification of their correctness, either at compile time or run time. This makes it easy to make mistakes, and difficult to find the sources of the errors that do occur. Therefore, we would like JEDD to relieve the user from having to perform the full assignment by automatically generating a reasonable assignment from a minimum amount of user input. To prevent errors, we would like JEDD to automatically insert the correct replace operations to implement the assignment.

Since JEDD is a tool designed mainly for research into implementing program analyses using BDDs, it should make it possible to experiment with different physical domain assignments. It has been widely noted that the ordering of variables in a BDD determines its size, and therefore the speed of operations performed on it. The variable ordering is closely related to the physical domain assignment, since physical domains are groups of BDD variables; the combination of the assignment of attributes to physical domains and the ordering of the variables of those physical domains together determine the relative ordering of the BDD variables implementing the attributes. Therefore, the physical domain assignment chosen has an important effect on the performance of algorithms implemented with BDDs. Unfortunately, with our currently limited knowledge of implementing program analyses using BDDs, we do not know of any easy ways to determine a near-optimal physical domain assignment even by hand, let alone automatically. Some input from the programmer about the desired physical domain assignment is therefore necessary. Indeed, it is desirable to allow the researcher to specify the assignment, to make it possible to experiment with different assignments. These experiments are necessary to improve our knowledge of what makes a good assignment, and will hopefully one day lead to a fully automated physical domain assignment algorithm. However, we must remember to balance flexibility with ease of specification. Ideally, JEDD would allow the program to initially contain a minimum of physical domain information, and would automatically generate a reasonable complete assignment. Later, based on profiling information, the programmer would tune the critical parts of the program and specify the assignment for those parts in more detail.

In order for the physical domain assignment algorithm to be useful, it must be implemented in a practical tool that is usable by programmers. When the programmer-specified part of the physical domain assignment contains errors (i.e., part of the physical domain assignment is inconsistent), the algorithm should be able to indicate the source of the error with meaningful error messages. In the absence of errors, the algorithm should always find a reasonable assignment; it should not be a heuristic that fails for certain difficult inputs, since these difficult problems are likely to also be difficult for the programmer to solve by hand. Since JEDD will be run each time the

program is compiled, and since the point of JEDD is to make it easier to implement non-trivial program analyses using BDDs, the algorithm should be able to process these non-trivial programs in a reasonable amount of time.

JEDD addresses these objectives in the following ways. For each attribute instance, the programmer may optionally specify a physical domain assignment, and JEDD automatically inserts the correct replace operations to implement the assignment. This makes it easy to tweak the assignment without having to rewrite the replace operations. When the programmer specifies physical domains for only a small subset of the attributes, JEDD automatically completes the assignment using the algorithm described in the next section. Should the programmer not be satisfied with specific parts of the automatically generated assignment, physical domains may be specified for these expressions explicitly, and JEDD will find a reasonable assignment for the rest of the program. If the programmer-specified portion of the physical domain assignment contains an inconsistency and an assignment cannot be found, JEDD reports the specific expression and attributes to which physical domains cannot be assigned, as described in section 3.5.4.

3.5.2 Formal physical domain assignment requirements

In order to correctly implement a JEDD program in BDDs, a physical domain assignment must satisfy the following constraints:

1. [conflict] Within every relation, each attribute must be assigned to a *distinct* physical domain.

2. [equality] Each relational operation implemented using BDDs requires certain attributes of its operands to be assigned to the *same* physical domain. In particular,

   - set union, intersection, and difference operations, relation comparison, and assignment of relations all require corresponding attributes of their operands to be assigned to the same physical domains, and

- composition and join require the attributes being compared to be assigned to the same physical domains.[5]

We adopt the term ***valid*** for a physical domain assignment satisfying these constraints. Finding a valid assignment for a JEDD program usually requires the operands of some operations to be wrapped in BDD replace operations in order to move them to physical domains that satisfy the constraints. It is always possible to construct a valid assignment if all operands of all operations are wrapped in replace operations.

Although a valid physical domain assignment is sufficient for a *correct* implementation of a JEDD program, it may not necessarily lead to an *efficient* implementation. In particular, the requirement that an assignment be valid does not limit the number of physical domains used, or the number of expensive BDD replace operations needed to implement it. To obtain reasonably efficient physical domain assignments, we must impose additional constraints.

We define a physical domain assignment to be ***reasonable*** if it is valid, and if every attribute is assigned to its physical domain for a *reason*, rather than arbitrarily. Specifically, the following are the allowed reasons for assigning a physical domain P to an attribute instance A:

1. The physical domain P was explicitly specified for the attribute instance A in the JEDD program.

2. A is involved in an operation requiring it to have the same physical domain as another attribute instance A', and A' has already been assigned the physical domain P. If we were to assign a physical domain other than P to A, a replace operation would have to be introduced before the operation to move A and A' into the same physical domain.

A reasonable physical domain assignment has several desirable properties.

[5]Composition and join also require the attributes *not* being compared to be assigned to physical domains distinct from any used in the other operand. However, this constraint is implied by the conflict constraints on the operands and result of the composition or join, so we need not consider it explicitly.

First, the set of physical domains allowed to be used is limited to those explicitly mentioned somewhere in the program. The physical domain assignment algorithm cannot introduce additional physical domains not mentioned by the programmer. This is important because the programmer must specify a BDD variable ordering of all the physical domains, and therefore must be aware of all the physical domains that are used.

Second, every replace operation implied by the physical domain assignment is necessary in the following sense. Suppose that attribute instances A and A' are assigned distinct physical domains P and P', but are involved in an operation that requires a replace between them. Then there is a *reason* that A and A' were assigned distinct physical domains: specifically, there is a chain C (C') of operations from A (A') to some attribute instance to which the programmer has explicitly assigned the physical domain P (P'). In order for the physical domain assignment to be valid, there *must* be a replace operation somewhere along the combined chain consisting of C, the operation involving A and A', and C'. Although it is not necessary for the replace to be in the specific position that it is, a replace is necessary somewhere along the chain.

Third, requiring a reason to assign a physical domain rather than doing so arbitrarily maintains control over fine-tuning the assignment in the hands of the programmer. Specifically, the programmer can explicitly force a desired attribute instance to a physical domain, and other attribute instances involved in operations with it are likely to be assigned to the same physical domain.

A reasonable physical domain assignment does not necessarily have the minimum possible static number of replaces. However, the static number of replaces is a poor predictor of run-time performance, because different replaces may be executed a very different number of times and have very different costs. Furthermore, for typical JEDD programs, there are often many valid assignments with the minimum static number of replaces but very different performance. If JEDD relied on a global property such as the total number of replaces, it could not allow the programmer local control over specific expensive replaces. The flexibility to tune the *run-time* behaviour of the few expensive replaces is more important to us than the *static* total number of replaces.

3.5.3 Physical domain assignment algorithm

Unfortunately, finding a reasonable physical domain assignment for a JEDD program is not easy.

Proposition 1 *The problem of finding a reasonable physical domain assignment is NP-complete.*

Proof: See Appendix A.

Several heuristics that we implemented to solve this NP-complete problem failed on common example programs. More importantly, an incomplete heuristic (which may fail to find a solution even when one exists) is undesirable for this problem. The case when a heuristic would fail to find a solution is precisely when the programmer very much wants to know whether a solution exists (and is therefore worth searching for by hand) or does not exist (and the code must therefore be modified so that a solution does exist). Therefore, the potentially very high cost of an exhaustive search is justified, and our intuition told us that although the problem in general is NP-complete, typical instances would be relatively easy to solve. However, we realized that implementing a smart exhaustive solver that would handle the easy cases efficiently would be difficult, and we would be duplicating much of the work that has been done on the boolean satisfiability (SAT) problem. We therefore encode the physical domain assignment problem as a SAT problem, and call a SAT solver to solve it for us.

Given a boolean formula over a set of variables, a SAT solver finds a truth assignment to those variables that makes the formula evaluate to true. We therefore encode the physical domain assignment problem into a boolean formula in such a way that we can recover a physical domain assignment from a truth assignment of its variables, and such that the formula evaluates to true exactly when the physical domain assignment satisfies our constraints.

Most SAT solvers require the input boolean formula to be in Conjunctive Normal Form (CNF). A formula in CNF is a conjunction of disjunctions of literals, where each literal is a variable or a negated variable. In the discussion that follows, we present our

formula for the physical domain assignment problem in the form of clauses (conjuncts) of a CNF formula. However, in the interest of clarity, we do not immediately convert each clause into a disjunction of literals. We defer this conversion until Figure 3.21 at the end of this section, in which we show all the clauses fully converted to CNF.

Our initial encoding of the physical domain assignment problem as a SAT formula was presented in [LH04]. This simple encoding worked well for several months of our work with JEDD. However, as we implemented more and more program analyses, the complexity of our code eventually caused the SAT formula to become prohibitively large. The problem was not that the SAT solver could not solve the formula; rather, the formula itself was too large for the JEDD translator to generate it. Therefore, we have devised an improved encoding which guarantees a SAT formula with a number of literals quadratic in the program size in the worst case, and typically linear. We now present the improved encoding.

We represent the constraints in an ***attribute def-use graph***. For each attribute instance of each subexpression in the program, this graph contains two vertices, a def vertex and a use vertex. The def vertex represents the attribute instance in the subexpression itself. Each subexpression can potentially be wrapped in a BDD replace operation, and the use vertex represents the attribute instance after this potential replace. After the algorithm assigns a physical domain to each vertex, it must wrap a replace operation around each subexpression for which the use vertex has been assigned to a different physical domain than the def vertex. The vertices of the graph are connected by three kinds of edges. A conflict edge between two vertices indicates that they must be assigned to *distinct* physical domains. An equality edge between two vertices indicates that they must be assigned to the *same* physical domain. These two kinds of edges are generated to enforce the constraints for the physical domain assignment to be valid, as defined in Section 3.5.2. Finally, an assignment edge between two vertices indicates that they *should* be assigned to the same physical domain. An assignment edge is generated between each def vertex and its corresponding use vertex. As long as both vertices are assigned to the same physical domain, no replace is needed.

The attribute def-use graph for the example JEDD code from Figure 3.3 is shown

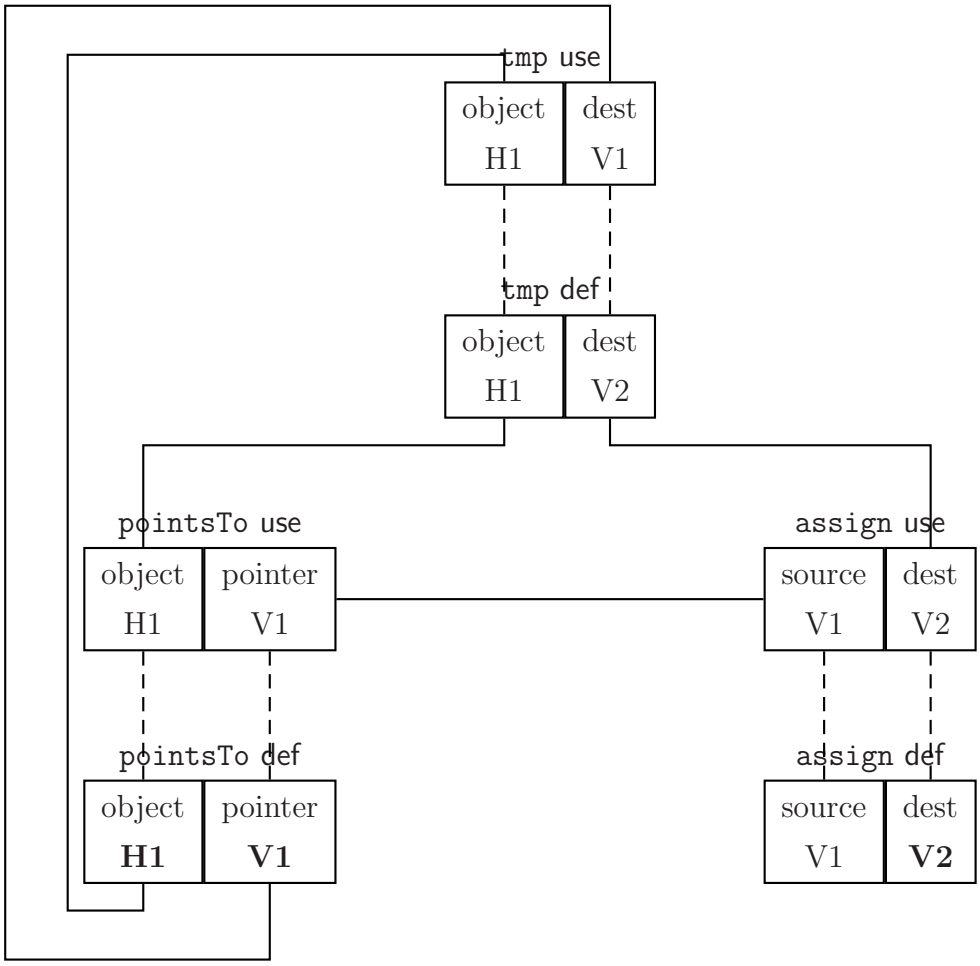


Figure 3.20: Example of physical domain assignment constraints

in Figure 3.20. Equality constraints are shown as solid lines and assignment constraints as dashed lines. Conflict constraints exist between the two attribute instances that compose each definition and each use, but they are not shown in the figure to avoid clutter. The physical domains shown in each vertex form a valid physical domain assignment with no unnecessary replaces. The three physical domains that were specified in the code in Figure 3.3 are indicated in bold. The assignment contains only one assignment edge that will generate a replace, namely the edge between the use and def vertices of the dest attribute in the `tmp` relation. This replace is necessary because it is on the path from the def vertex of the pointer attribute of the `pointsTo` relation and the def vertex of the dest attribute of the `assign` relation, for which the programmer has specified the physical domains V1 and V2, respectively.

To obtain a valid physical domain assignment, we must assign a physical domain to each vertex of the graph in a way that satisfies the constraints imposed by the edges of the graph. Since an equality edge requires its endpoints to be assigned to the same physical domain, every vertex in a component connected by equality edges is assigned the same physical domain.[6] We therefore merge all vertices in each such connected component into a single vertex. In the discussion that follows, we refer only to the simplified graph that results from this merging. For each vertex v in the simplified graph, and for each physical domain p, we define a SAT variable for the pair $(v:p)$. If the satisfying assignment found by the SAT solver sets this variable to true, v is assigned to the physical domain p.

To ensure that any satisfying assignment of the SAT formula corresponds to a *valid* physical domain assignment, the following clauses are needed. In the clauses below, we use V to denote the set of all vertices and P to denote the set of all physical domains.

[6]It is not possible for multiple vertices for which the programmer has specified distinct physical domains to be connected by equality edges, as a consequence of the following two facts. By construction of equality edges, at least one endpoint of every equality edge is a use vertex generated by JEDD, for which the programmer cannot have specified a physical domain. In addition, each such use vertex has at most one outgoing equality edge. Therefore, every path of equality edges starting at a vertex for which a physical domain has been specified has a generated use vertex as its very next vertex, and cannot continue any further from it.

Each vertex is assigned to some physical domain.

$$\bigwedge_{v \in V} \bigvee_{p \in P} (v : p) \tag{3.1}$$

No vertex is assigned to multiple physical domains.

$$\bigwedge_{v \in V} \bigwedge_{p,p' \in P, p \neq p'} \neg \left((v : p) \wedge (v : p') \right) \tag{3.2}$$

Any attribute with an explicitly specified physical domain is assigned that physical domain.

$$\bigwedge_{(v,p) \in SPECIFIED} (v : p) \tag{3.3}$$

For each conflict edge between v and v', the vertices v and v' must not be assigned to the same physical domain.

$$\bigwedge_{(v,v') \in CONFLICT} \bigwedge_{p \in P} \neg \left((v : p) \wedge (v' : p) \right) \tag{3.4}$$

The clauses 3.1 through 3.4 together express the requirement that the physical domain assignment be valid.

Encoding the requirement that the assignment be reasonable is less straightforward, because the definition of reasonable implicitly relies on the order in which attributes are assigned to physical domains, but the SAT solver computes a variable assignment which *simultaneously* satisfies all clauses of the formula. A vertex A can be assigned to the physical domain P if it is connected by an assignment edge to A', and A' has *previously* been assigned to P. Without the ordering requirement, it would be permitted to assign an arbitrary domain P' to both A and A', since each of them is connected to the other, and the other is also assigned P'. We must therefore consider the order when encoding the problem as a SAT formula.

We encode the reasonableness requirement in several steps, which we detail in the following paragraphs. First, we define a relation $\prec$, such that $a \prec b$ if and only if the reason for assigning b its physical domain was that a was assigned the same physical domain before it, and an assignment edge exists between a and b. We give

the SAT solver constraints that force it to compute such a relation $\prec$. We also create constraints which ensure that there exists a total order $\leq$ in which the vertices may have been assigned physical domains which is consistent with the physical domain assignment and the computed $\prec$ relation. More precisely, the SAT solver outputs enough information to prove the existence of a total order $\leq$ for which $a \prec b \Rightarrow a < b$. Since we are interested in the physical domain assignment itself, rather than the order in which the vertices were assigned physical domains, the SAT solver need not compute $\leq$ itself (which would require a larger SAT formula), but only provide enough information to prove its existence.

For each assignment edge (v, v'), we define a pair of SAT variables $(v \prec v')$ and $(v' \prec v)$. If $(v \prec v')$ is true in the satisfying assignment, it indicates that $v \prec v'$. We use the following clause to ensure that $(v \prec v')$ and $(v' \prec v)$ cannot both be true simultaneously:

$$\bigwedge_{(v,v') \in ASSIGNMENT} \neg \left((v \prec v') \wedge (v' \prec v) \right) \tag{3.5}$$

Since $v \prec v'$ indicates that v' was assigned a physical domain because v had already been assigned the same physical domain, we ensure that v' is assigned the same physical domain as v:

$$\bigwedge_{(v,v') \in ASSIGNMENT} \bigwedge_{p \in P} (v \prec v') \Rightarrow ((v : p) \Rightarrow (v' : p)) \tag{3.6}$$

If the programmer did not specify a physical domain for v', there must be some v such that $v \prec v'$:

$$\bigwedge_{v' \in V \mid \nexists p : (v',p) \in SPECIFIED} \bigvee_{v \in V \mid (v,v') \in ASSIGNMENT} (v \prec v') \tag{3.7}$$

To prove the existence of a total order in which the vertices may have been assigned physical domains, we make use of the following proposition. For now, we will make use of only the equivalence of statements 2 and 3 of the proposition.

Proposition 2 *Let G be an attribute def-use graph, and let $\prec$ be an antisymmetric binary relation on its vertices such that $a \prec b$ implies that a and b are connected by an* assignment *edge in G. Then the following four statements are all equivalent:*

1. *$\prec$ is a well-founded relation.*

2. *There exists a total order $\leq$ such that $a \prec b \Rightarrow a < b$. (This is the order in which physical domains could be assigned the vertices.)*

3. *There exists a total antisymmetric relation $\sqsubseteq$ such that $a \prec b \Rightarrow a \sqsubset b$ and there is no triple of distinct vertices a, b, c such that $a \prec b \sqsubset c \sqsubset a$.*

4. *On the vertices of every biconnected component $C = (V_C, E_C)$ of the graph formed by* assignment *edges, there exists a total antisymmetric relation $\sqsubseteq_C$ such that $\forall a, b \in V_C . a \prec b \Rightarrow a \sqsubset_C b$ and there is no triple of distinct vertices a, b, c such that $a \prec b \sqsubset_C c \sqsubset_C a$.*

Proof: See Appendix A.

To prove the existence of the total order $\leq$ (statement 2 of the proposition), the SAT solver need only produce the total relation $\sqsubseteq$ (proving statement 3), which can be specified with a much smaller SAT formula.

For every unordered pair $\{v, v'\}$ of distinct vertices, we arbitrarily choose one of the vertices (say v), and define a single SAT variable $(v \sqsubset v')$ indicating that $v \sqsubset v'$ if the variable is true, and $v' \sqsubset v$ if it is false. For convenience, we permit ourselves to write $(v' \sqsubset v)$ to mean $\neg(v \sqsubset v')$, but note that $(v \sqsubset v')$ and $(v' \sqsubset v)$ both refer to the same physical SAT variable, possibly negated. This definition ensures that the $\sqsubseteq$ relation found by the SAT solver is total and antisymmetric.

Next, we encode the requirement that $a \prec b \Rightarrow a \sqsubset b$:

$$\bigwedge_{(a,b) \in ASSIGNMENT} (a \prec b) \Rightarrow (a \sqsubset b) \tag{3.8}$$

Finally, we encode the requirement that there be no triple of distinct vertices a, b, c such that $a \prec b \sqsubset c \sqsubset a$:

$$\bigwedge_{(a,b) \in ASSIGNMENT} \bigwedge_{c \in V \setminus \{a,b\}} \neg \left((a \prec b) \wedge (b \sqsubset c) \wedge (c \sqsubset a) \right) \tag{3.9}$$

This clause completes the SAT formula. Figure 3.21 shows all the clauses of the formula converted to CNF.

$$\bigwedge_{v \in V} \bigvee_{p \in P} (v : p) \tag{3.1}$$

$$\bigwedge_{v \in V} \bigwedge_{p,p' \in P, p \neq p'} \neg (v : p) \vee \neg (v : p') \tag{3.2}$$

$$\bigwedge_{(v,p) \in SPECIFIED} (v : p) \tag{3.3}$$

$$\bigwedge_{(v,v') \in CONFLICT} \bigwedge_{p \in P} \neg (v : p) \vee \neg (v' : p) \tag{3.4}$$

$$\bigwedge_{(v,v') \in ASSIGNMENT} \neg (v \prec v') \vee \neg (v' \prec v) \tag{3.5}$$

$$\bigwedge_{(v,v') \in ASSIGNMENT} \bigwedge_{p \in P} \neg (v \prec v') \vee \neg (v : p) \vee (v' : p) \tag{3.6}$$

$$\bigwedge_{v' \in V \mid \nexists p : (v',p) \in SPECIFIED} \bigvee_{v \in V \mid (v,v') \in ASSIGNMENT} (v \prec v') \tag{3.7}$$

$$\bigwedge_{(a,b) \in ASSIGNMENT} \neg (a \prec b) \vee (a \sqsubset b) \tag{3.8}$$

$$\bigwedge_{(a,b) \in ASSIGNMENT} \bigwedge_{c \in V \setminus \{a,b\}} \neg (a \prec b) \vee \neg (b \sqsubset c) \vee \neg (c \sqsubset a) \tag{3.9}$$

Figure 3.21: Complete formula for physical domain assignment problem in CNF

3.5.3.1 Additional optimizations

The asymptotically largest number of literals in the SAT formula comes from this last clause, which introduces $3m(n-2)$ literals, where m is the number of assignment edges and n is the number of vertices. In typical attribute def-use graphs, m is approximately equal to n. In the program analyses for which JEDD is intended, n and m can be up to 1000, leading to 3 million literals in the SAT formula. Based on our experience with the zChaff SAT solver, it is capable of working with a formula of this size. However, we can make the formula significantly smaller still by making use of the fourth statement of Proposition 2.

The properties required of the relations $\sqsubseteq_C$ in statement 4 of the proposition are similar to those required of $\sqsubseteq$ in statement 3, but $\sqsubseteq_C$ is only defined on the much smaller biconnected components of the graph, rather than on the whole graph. Therefore, if we change the SAT formula generated by JEDD to construct $\sqsubseteq_C$ rather than $\sqsubseteq$, the size of a SAT formula can be made proportional not to the square of the size of the entire graph, but to the sum of squares of the sizes of the biconnected components. In our experience, most biconnected components are no larger than ten edges, with the largest being on the order of 100 edges. The SAT formula is therefore significantly smaller.

To find the biconnected components of the graph, JEDD uses the well-known algorithm [Tar72] based on depth-first search (DFS). To construct $\sqsubseteq_C$ rather than $\sqsubseteq$, only clauses 3.8 and 3.9 of the SAT formula need to be modified, and the necessary modification is quite simple. Only the vertices over which the clauses range are modified; the bodies of the clauses are not changed. The pairs (a, b) in both clauses, which range over all assignment edges, are changed to range over only those assignment edges whose endpoints are in the same biconnected component. The vertex c in clause 3.9, which ranges over all vertices in the graph except a and b, is changed to range over all vertices in the same biconnected component as a and b excluding a and b themselves.

JEDD performs one additional optimization to make the SAT formula smaller. Several of the clauses (3.1, 3.2, 3.4, and 3.6) quantify over all physical domains defined

for the JEDD program. However, because a reason is required to assign a physical domain p to a vertex v, v can never be assigned p unless v is connected by some path of assignment edges to some vertex v' to which p has been assigned explicitly in the JEDD program. JEDD partitions the graph of assignment edges into connected components using a DFS, and for each connected component, collects the set of all physical domains explicitly assigned to a vertex in the component. The SAT variable $(v{:}p)$ cannot be true if p is not in this set for the connected component containing v. Therefore, $(v{:}p)$ is removed from all clauses (disjunctions), since it cannot make them true. In addition, all clauses (disjunctions) containing $\neg(v{:}p)$ are necessarily true, so they are removed from the overall conjunction. The connected components are also used in error reporting, which we discuss next in Section 3.5.4.

3.5.4 Error reporting

One challenge with using a black box such as a SAT solver in a compiler is reporting errors to the user. When the SAT solver determines that no reasonable physical domain assignment exists, it reports that the boolean formula is unsatisfiable. While this fact is useful for the programmer to know, it is not very helpful in pinpointing the cause of the error.

To improve the error reporting, we took advantage of a new feature recently implemented in the zChaff SAT solver, unsatisfiable core extraction [ZM03]. When the SAT solver determines that the boolean formula is unsatisfiable, it also outputs a small subset of the clauses whose conjunction is still unsatisfiable.

There are two potential reasons why no reasonable physical domain assignment may exist. First, there may be a vertex v not connected by any path to any other vertex for which a physical domain has been specified. In this case, the list of explicitly assigned physical domains for the connected component containing v is empty, and JEDD detects this when constructing the SAT formula. Second, it may not be possible to assign physical domains to the vertices in a way that respects all the conflict constraints. In this case, the SAT formula is unsatisfiable. The following proposition suggests a way to report the source of the problem to the programmer.

Proposition 3 *When the SAT formula produced for the physical domain assignment problem is unsatisfiable, every unsatisfiable core contains at least one clause of type 3.4 (conflict clause).*

Proof: See Appendix A.

Therefore, the small unsatisfiable core returned by the SAT solver must contain at least one clause of type 3.4. From this clause, JEDD extracts the attribute instances to which physical domains could not be assigned, and the physical domain(s) that were considered for assignment. This information is reported to the programmer along with the position of the expression in the source file. An easy way for the programmer to fix the problem is to introduce a new physical domain, and explicitly assign it to one of the attributes of the unsatisfiable conflict constraint.

To illustrate the error reporting with a typical error, consider what would happen if the attribute `dest` of the relation `assign` in line 2 of the code in Figure 3.3 were not explicitly assigned a physical domain. As a result, there would be no reasonable physical domain assignment for the program, since there would be only two physical domains, `H1` and `V1`, but the composition in line 8 requires three. JEDD would output the following error message:

```
1 Prop.jedd:8: Conflict between attributes dest and source of replaced version
2     of
3             <dest, object> tmp = pointsTo{pointer} <> assign{source};
4                                                      ^----^
5 over physical domain V1
```

The error message indicates the location of the error, the expression in question (`assign`), the attributes to which a physical domain could not be assigned (`dest` and `source`), and the single physical domain which is available for the two attributes (`V1`). To fix this error, the programmer would specify that one of the attributes should be assigned to a new physical domain. For example, in the original code in Figure 3.3, `dest` was explicitly assigned to the physical domain `V2`.

3.6 Jedd Runtime

3.6.1 Backends

One of the benefits of expressing BDD algorithms in Jedd is that these algorithms can be executed, without modification, using various BDD libraries as backends. This allows us to compare the performance of different backends on the same problem. Jedd can currently use the BuDDy [LN], CUDD [Som], SableJBDD [Qia], or JavaBDD [Whab] libraries as backends. Because BuDDy and CUDD are written in C, they are called from Jedd using the JNI.

3.6.2 Memory management issues

BDD libraries use reference counts of external references to identify unused BDD nodes to be reclaimed. A disadvantage of this approach is that a programmer using the library is required to explicitly increment and decrement the reference count whenever BDDs are assigned or a reference to a BDD goes out of scope. In C++, it is possible to use overloaded assignment operators and destructors to relieve the programmer of much of this burden. The lack of operator overloading makes this impossible in Java. If Jedd were a library rather than a language extension, the programmer would have to explicitly manipulate reference counts. Memory management is yet another tedious and error-prone aspect of working with BDDs. Since Jedd is an extension to the language, we can design it to update reference counts automatically, without any help from the programmer.

For performance reasons, it is particularly important that the reference count be decremented as soon as possible after a reference becomes unreachable, because it may be the root of a BDD consisting of many other nodes. When dead nodes are not freed in a garbage collection, fewer nodes remain for future computation, so garbage collection is required more frequently. In addition, BDD libraries use a cache to speed up the basic operations on nodes. Large numbers of unfreed obsolete nodes may pollute this cache. In general, we cannot rely solely on the Java garbage collector to determine when relations are unreachable, particularly short-lived temporary

relations. This is because unlike allocations of Java objects, an allocation of a BDD node will not trigger a Java garbage collection when no more memory is available. It is possible to allocate many large temporary BDDs in several iterations of a loop and have the BDD library run out of memory without a Java garbage collection ever being triggered.

A BDD can become unreachable in one of four ways. First, a subexpression of an expression becomes unreachable when the overall expression is evaluated. Second, the BDD may be stored in a local variable or field, and be overwritten by another BDD. Third, the BDD may be stored in a local variable which goes out of scope. Fourth, the BDD may be stored in a field, and the object containing the field may become unreachable. For temporary values, the first two cases are the most common and therefore the most important.

To handle the first case, we implement the convention that each BDD operation decrements the reference count of its arguments and increments the reference count of its result before returning it. Therefore, the reference count of a subexpression is decremented as soon as it is used in the overall expression. This convention is partly imposed by the requirement of the BDD libraries that any BDDs passed to library functions have non-zero reference counts.

For a clean implementation of the remaining cases, we create a relation container object for each local variable and field of relation type. In the generated Java code, the corresponding variable or field points to its relation container throughout its entire lifetime; this is enforced by making the generated variable or field final. The BDD itself is stored as a private field of the relation container, and can be updated only through an assignment method which also updates the reference counts. This ensures that when a BDD is overwritten by another, the reference count of the overwritten BDD is immediately decremented.

To handle the third and fourth cases, the finalizer of every relation container (which is called when the relation container is garbage collected) decrements the reference count of the BDD stored in it. In the case of a local variable going out of scope, the finalizer of the relation container ensures that the reference count will eventually be decremented, but this may be a significant amount of time after the

variable goes out of scope. To improve on this, we perform a static liveness analysis on all relation variables, and at each point where a variable may be live and is known to become dead, we decrement the reference count of any BDD it may contain and remove the BDD from the container. In the face of exceptional interprocedural control flow, this is not always possible. We assume such control flow to be unusual, and fall back on the finalizer to decrement the reference count in such cases.

In the case of an object containing a BDD becoming unreachable, the relation container is normally garbage collected in the same garbage collection as the object containing it.[7] The finalizer decrements the reference count in the same garbage collection.

To summarize, JEDD manages BDD reference counts automatically without any help from the programmer. In all four cases, it frees BDDs as soon as it becomes safe to do so, so its performance should be no worse than that of a hand-coded reference counting solution.

3.6.3 Profiler

A common problem when tuning any algorithm using BDDs is choosing an efficient *variable ordering*, the relative order of the individual bits of the physical domains. In complicated programs with many relations and attributes, a related problem is tuning the physical domain assignment, and the replace operations which it implies. Specifically, we are interested in removing those replace operations which are particularly expensive by modifying the physical domain assignment to make them unnecessary. For these tuning tasks, we need some insight into the runtime behaviour of our program. In particular, we want to know which operations are expensive in terms of time and BDD size (and therefore space), in order to either remove them, or make them cheaper by modifying the variable ordering. For tuning the variable ordering, knowing the shape of the BDDs involved in the operation is also useful, as we will see

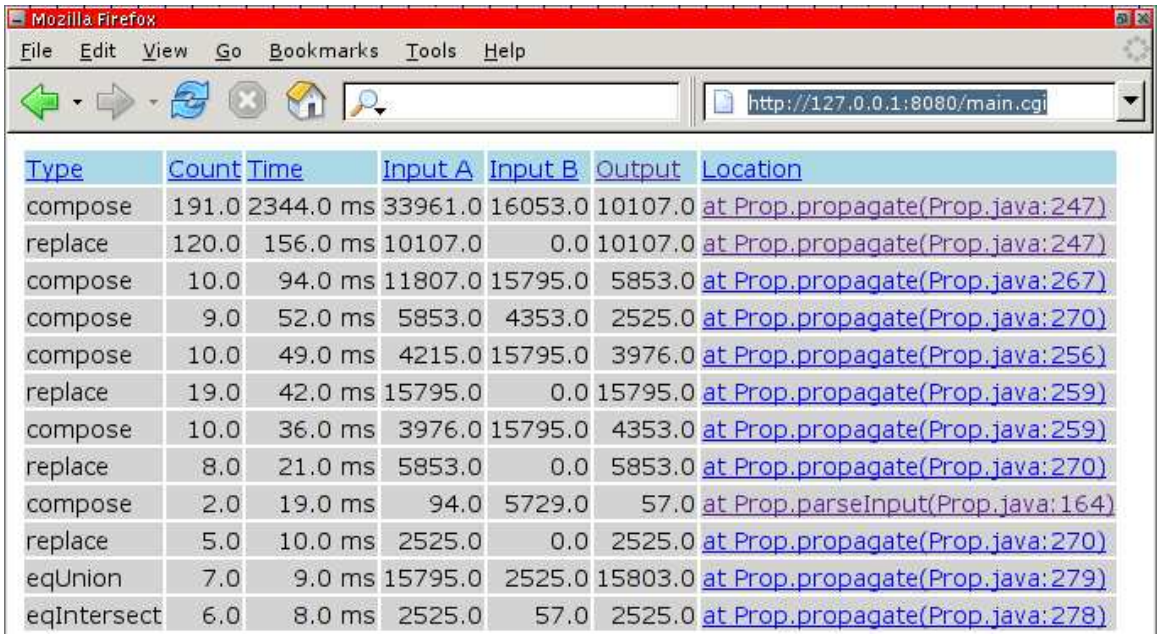
[7] Here, we assume that the garbage collector collects all unreachable objects in each collection. However, even when this is not true in general, such as in a generational collector, it is very likely that the object containing the field and the relation container will be reclaimed in the same collection, since they are allocated close together: the latter is allocated in the constructor of the former.

with several examples at the end of this section. The shape of a BDD is the number of nodes at each level (testing each variable) of the BDD.

In the code generated by JEDD, relational operations are implemented as calls into the JEDD runtime library. The runtime library optionally makes calls to a profiler which records, for each operation, the time taken and the number of nodes and shape of the operand and result BDDs. This information is written out as a SQL file to be loaded into a database, which provides a flexible data store on which arbitrary queries can be performed to present the data to the user. JEDD also includes CGI scripts to provide access to the profiling data through a web browser. We use SQLite [Hip] for the database and `thttpd` [Pos] for the web server because of their ease of installation, but in principle, any SQL database and any web server would work. The overall profile view shows, for each relational operation in the program, the number of times it was executed, the total time taken, and the maximum size of the BDDs involved (see Figure 3.22). Clicking on an operation brings up a detailed view with a line of information for each time the operation was executed. Clicking on a specific execution of the operation generates a graphical representation of the shape of the BDDs involved in the operation. Figure 3.23 shows an example of this graphical representation for a typical replace operation. In this case, the relation consists of two attributes, the first mapped to the physical domain V1 ranging from levels 20 to 39 of the BDD, and the second being moved from the physical domain H2 at levels 80 to 99 of the BDD to a different physical domain H1 at levels 60 to 79.

Once an unacceptably large BDD has been identified, its shape often provides insight into why it is so large, and how the program can be changed to make it smaller. In Figures 3.24 to 3.27, we present some typical BDD shapes that may be observed when tuning a JEDD program, and explain what they suggest about the physical domain assignment and bit ordering. The shape graphs in these figures are of BDDs synthesized to highlight patterns that were observed during tuning of the PADDLE framework described in Chapter 4.

When a relation has a large number of attributes, often only some of the attributes are responsible for making the BDD large. The physical domains to which these important attributes are assigned affect the BDD size the most. For example, in the



Type	Count	Time	Input A	Input B	Output	Location
compose	191.0	2344.0 ms	33961.0	16053.0	10107.0	at Prop.propagate(Prop.java:247)
replace	120.0	156.0 ms	10107.0	0.0	10107.0	at Prop.propagate(Prop.java:247)
compose	10.0	94.0 ms	11807.0	15795.0	5853.0	at Prop.propagate(Prop.java:267)
compose	9.0	52.0 ms	5853.0	4353.0	2525.0	at Prop.propagate(Prop.java:270)
compose	10.0	49.0 ms	4215.0	15795.0	3976.0	at Prop.propagate(Prop.java:256)
replace	19.0	42.0 ms	15795.0	0.0	15795.0	at Prop.propagate(Prop.java:259)
compose	10.0	36.0 ms	3976.0	15795.0	4353.0	at Prop.propagate(Prop.java:259)
replace	8.0	21.0 ms	5853.0	0.0	5853.0	at Prop.propagate(Prop.java:270)
compose	2.0	19.0 ms	94.0	5729.0	57.0	at Prop.parseInput(Prop.java:164)
replace	5.0	10.0 ms	2525.0	0.0	2525.0	at Prop.propagate(Prop.java:270)
eqUnion	7.0	9.0 ms	15795.0	2525.0	15803.0	at Prop.propagate(Prop.java:279)
eqIntersect	6.0	8.0 ms	2525.0	57.0	2525.0	at Prop.propagate(Prop.java:278)

Figure 3.22: Overall profile view

BDD in Figure 3.24, the vast majority of BDD nodes test physical domains PD3, PD4, and PD5, and very few nodes test physical domains PD1 and PD2. Therefore, changing the relative ordering of the bit positions in PD3, PD4, and PD5 will have a much stronger effect on the BDD size than changing the relative ordering of the bit positions in PD1 and PD2.

In some BDD shape graphs, the number of nodes testing each bit position of a physical domain remains constant or nearly constant, as for the PD2 domain in Figure 3.25(a). This suggests that testing a bit of the physical domain provides little information about whether a given binary vector is in the set represented by the BDD. In the BDD of Figure 3.25(a), information about bits in both PD1 and PD3 is required to decide whether a binary vector is in the set. Therefore, to test a given binary vector, the information about PD1 must be carried through PD2 to PD3, leading to a large number of nodes in PD2. If the bit ordering is changed so that PD2 no longer separates PD1 and PD3, the BDD becomes much smaller, as shown in Figure 3.25(b).

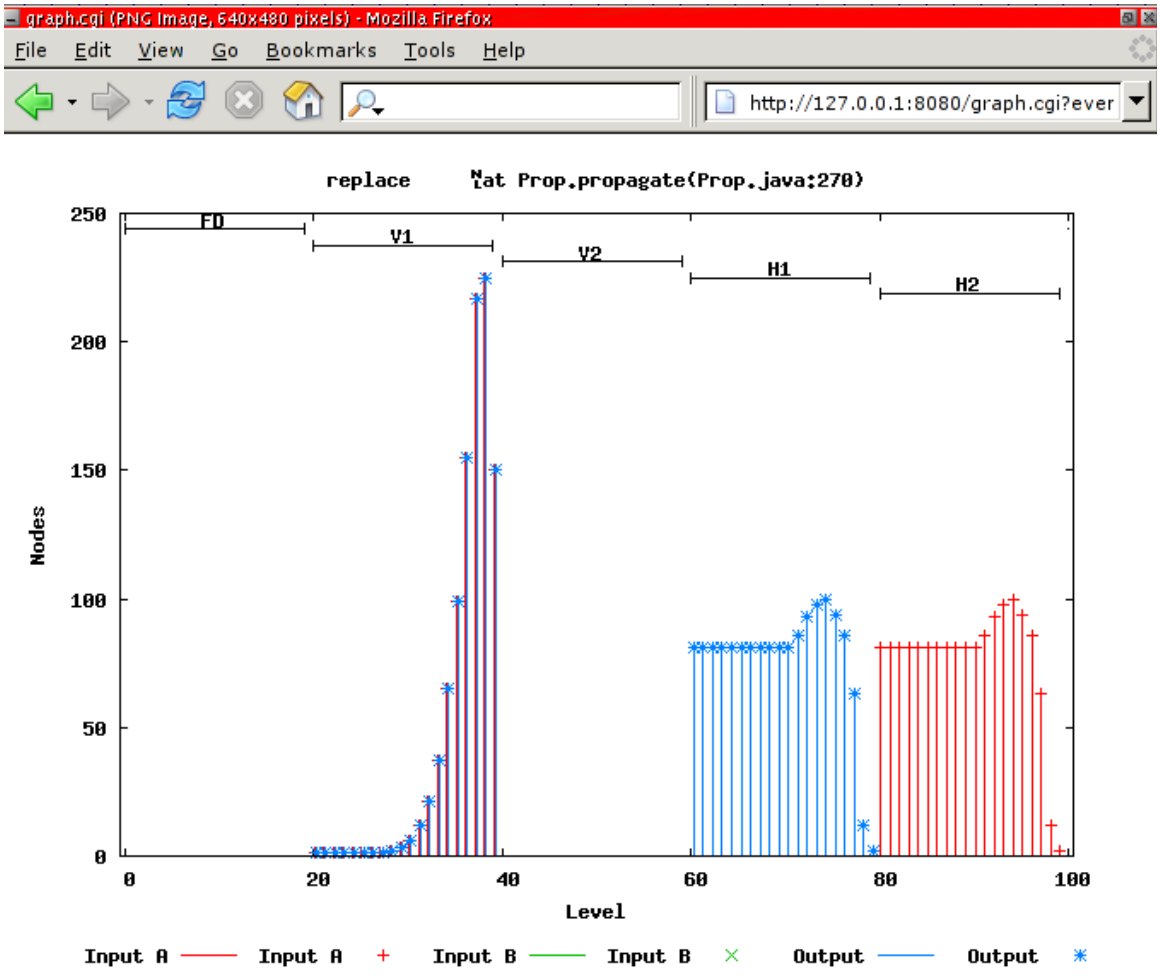
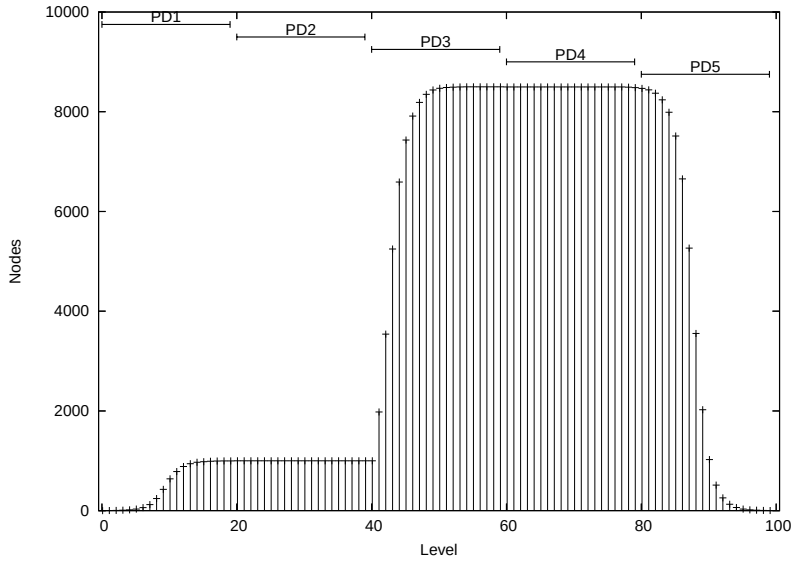
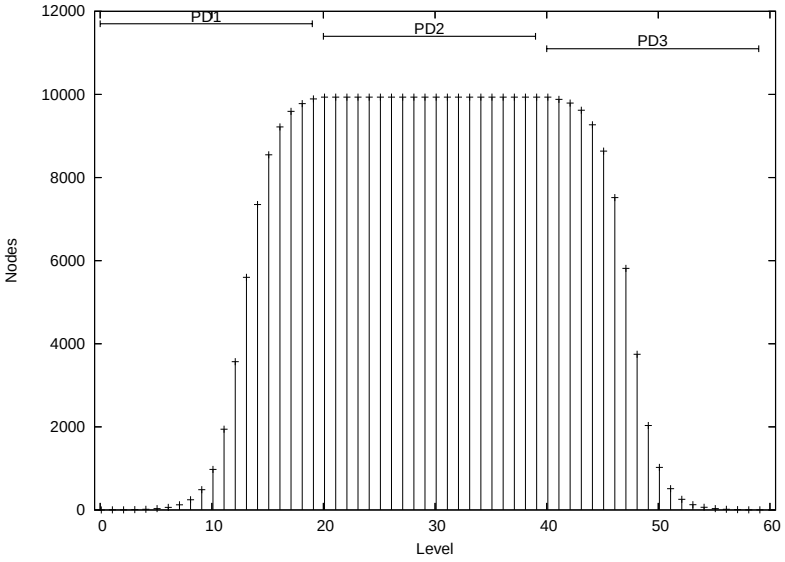


Figure 3.23: Graphical representation of BDD in replace operation



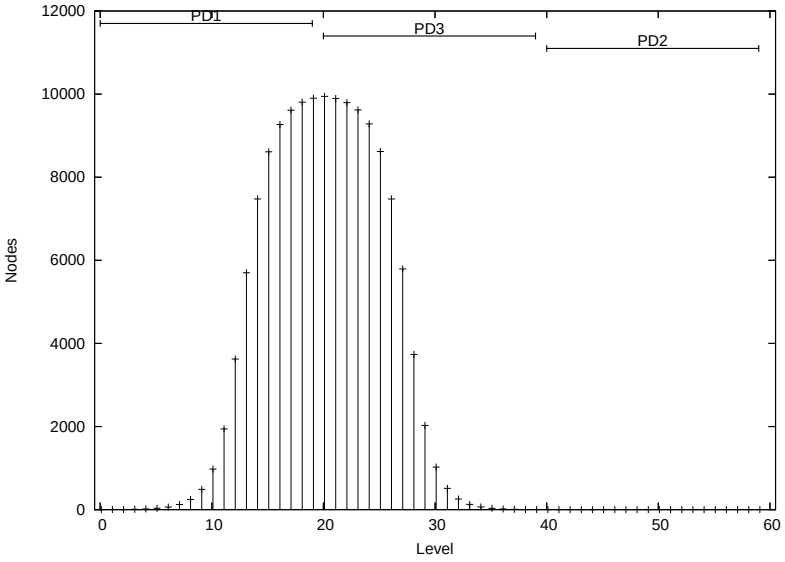
Total nodes: 411791

Figure 3.24: Example shape graph in which most of the nodes test physical domains PD3, PD4, and PD5



Total nodes: 344846

(a)



Total nodes: 145929

(b)

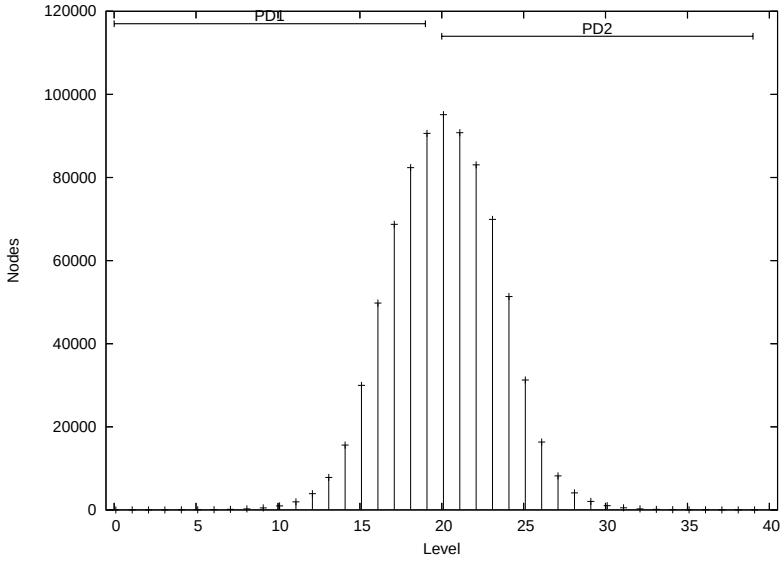
Figure 3.25: Example shape graph in which the number of nodes testing each bit of PD2 is high and constant

Some BDDs exhibit a sharp spike near the boundary of two physical domains, as in Figure 3.26(a). After the bits of PD1 have been tested, many BDD nodes are required to remember which of the many distinct binary sub-vectors has been observed in PD1. As soon as a few bits of PD2 have been tested, however, the number of distinct states that must be remembered quickly goes down. This suggests that if some bits of PD2 were tested earlier, the BDD may not grow as wide. The example relation represented by the BDD of Figure 3.26(a) can be represented by the much smaller BDD in Figure 3.26(b) if the bits of PD2 are interleaved with the bits of PD1, rather than being tested after all the bits of PD1.

However, when certain attributes of a relation are not closely correlated, interleaving their physical domains is a mistake. A symptom of this problem is a sharp spike in the shape graph within an area of interleaved physical domains, as shown in Figure 3.27(a). Each BDD node in the spike carries information about some of the bits of PD1 as well as about some of the bits of PD2. Instead, if we first test all the bits of one physical domain and then the other, as in Figure 3.27(b), the BDD is much smaller. Note that in this case, the BDD in Figure 3.27(b) is smaller by so much that we have magnified the scale of the y axis by 100 to make its shape visible.

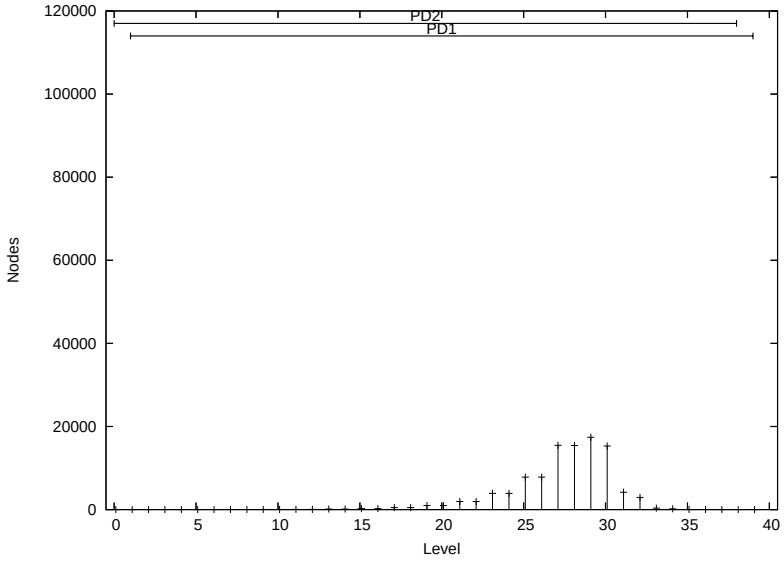
3.7 Jedd Performance

We have implemented in Jedd several test examples, our BDD points-to analysis algorithm from [BLQ+03], and the Paddle framework of interrelated whole-program analyses that we describe in Chapter 4. Without Jedd, the latter would not have been feasible, since it would require us to assign physical domains by hand to the attributes of thousands of relation instances, with no automated way to verify that we had not made mistakes. We first wrote the analyses without specifying any physical domains at all, and when it came time to compile, we assigned only enough attributes to physical domains to allow the physical domain assignment algorithm to find a reasonable assignment for the rest. In this process, Jedd's error reporting pointed us directly to the expressions that needed to have physical domains assigned by hand.



Total nodes: 806506

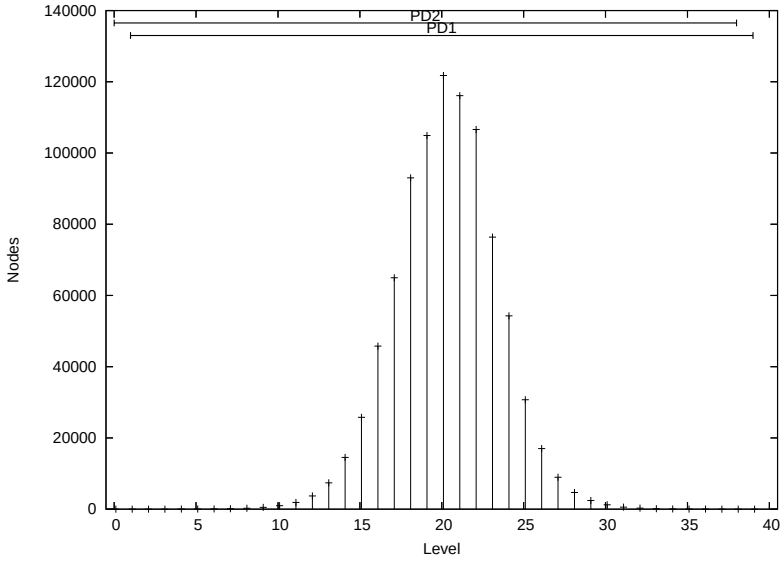
(a)



Total nodes: 102503

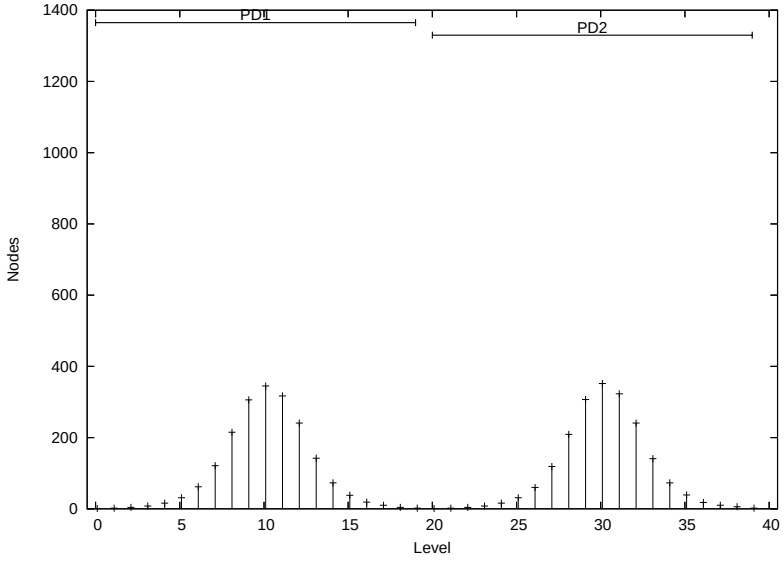
(b)

Figure 3.26: Example shape graph with a spike at the boundary of PD1 and PD2



Total nodes: 896905

(a)



Total nodes: 3918

(b)

Figure 3.27: Example shape graph in which unrelated physical domains are interleaved

To measure the runtime overhead of JEDD compared to using a BDD library directly from C++ code, we timed the C++ and JEDD versions of our analysis from [BLQ+03] on five benchmarks. Both versions used the BUDDY BDD library as the backend. The timings are shown in Table 3.1. The overhead varied from 0.5% to 4%, which we attribute to having to have the Java virtual machine in memory, and to the internal Java threads that run even when not executing Java code.

Benchmark	Std. lib. version	C++	JEDD
javac	1.1.8	3.4 s	3.5 s
compress	1.3.1	21.7 s	22.4 s
javac	1.3.1	25.3 s	26.3 s
sablecc	1.3.1	25.4 s	26.1 s
jedit	1.3.1	41.1 s	41.3 s

Table 3.1: Running time comparison of hand-coded C++ [BLQ+03] and JEDD points-to analysis

To evaluate the compile-time performance of the physical domain assignment algorithm, we used JEDD to compile each revision in the source repository of the PADDLE framework. We will describe the PADDLE framework in detail in Chapter 4. Here, we use PADDLE only as a benchmark to evaluate the compile-time performance of JEDD. The PADDLE framework was developed over a period of two years, with new features and analyses being added to it throughout this time. Figure 3.28 shows the growth in the size of the SAT formula (measured as the total number of literals) compared to the growth of the PADDLE framework (measured as the total number of attribute instances in the code). The size of the SAT formula increases predictably and linearly with the size of the code.

Figure 3.29 shows the time taken by the zChaff [MMZ+01] SAT solver to solve the SAT formula derived from each revision of the PADDLE framework, again compared to the number of attribute instances in the code. These times were measured on a machine with a 1833 MHz AMD Athlon CPU and 512 MB of RAM. Although the SAT solver is very fast when compiling revisions of PADDLE with up to 10000 attribute

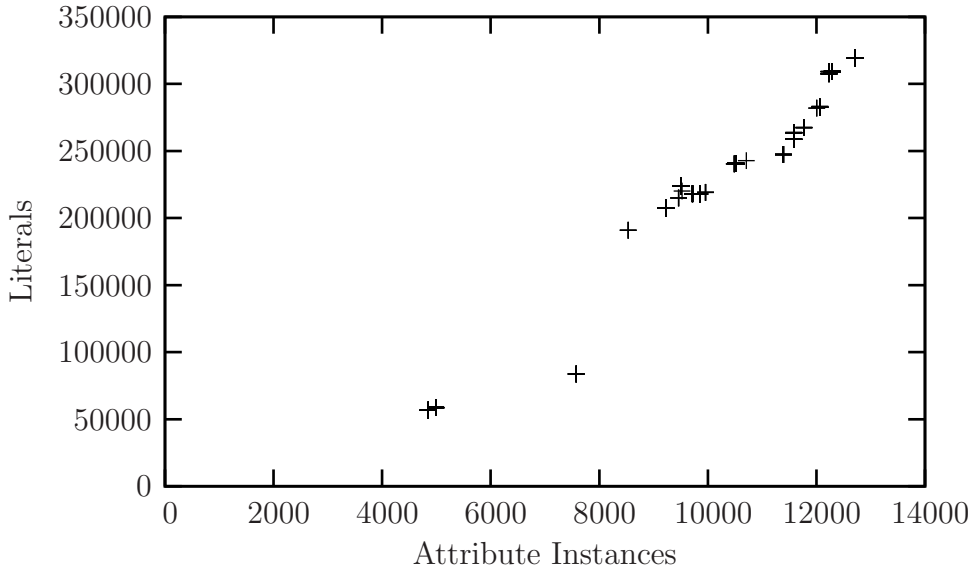


Figure 3.28: Size of SAT formula

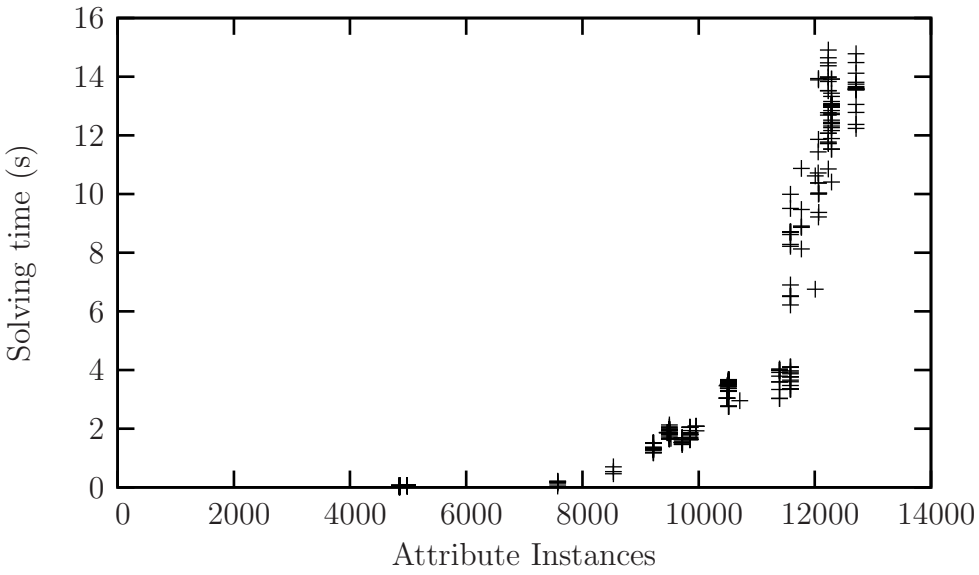


Figure 3.29: SAT solving time

instances, it starts to take significantly more time when PADDLE grows beyond this size. Because the physical domain assignment is an NP-complete problem, this growth is to be expected.

In order for JEDD to be practical for programs much larger than the current PADDLE framework, it is likely that further improvements in SAT solving will be needed. However, the current version of the PADDLE framework includes all of the analyses that we had planned to implement using BDDs, including client analyses for both Java and AspectJ. The 15 seconds required to find a physical domain assignment for this large collection of analyses is only a small part of the overall 5 minute compilation time of the SOOT framework in which PADDLE has been implemented.

Therefore, we conclude that JEDD makes it practical to develop analysis frameworks as complicated as PADDLE.

3.8 Related Work

We have organized related work into three categories. We first sample the abundance of work on languages for expressing relational computation. In Section 3.8.2, we present various tools that have been written to interface with BDDs at a low level. Finally, some work has been done on abstracting BDDs as relations, and we compare this work with JEDD in Section 3.8.3.

3.8.1 Languages with relations

The relational data model based on relational algebra was proposed by Codd [Cod70], and has since been used for many applications, particularly as the basis of relational databases. SQL has become a standard way of expressing relational operations in database systems, and snippets of SQL code are often embedded in programs written in other languages. Prolog [CM87] and its derivatives are based on querying and updating a database of *facts*, which are analogous to relational tuples. Relations as first-class objects have appeared in many general-purpose languages ever since the days of SETL [SDDS86], which included binary relations as one of its basic

data types. Support for n-ary relations is often present in languages for writing "glue" code between database systems and client interfaces, such as the `<bigwig>` project [BMS02]. The increasing popularity of Extensible Markup Language (XML) is fuelling work on adapting languages for manipulating XML fragments, which often resemble tuples, but are generally less homogeneous. Two recent examples of this work are the JWIG project [CMS03], which integrates the `<bigwig>` programming model into Java, and an extension to C# for expressing both relational and XML data [MS03].

JEDD is similar to these languages in that it adds relations as a data type to Java. In contrast to these languages whose primary goal is to provide access to relations, the primary focus of JEDD is to enable program analysis developers to exploit the compact data representation provided by BDDs, using relations as an abstraction to make programming with BDDs manageable.

3.8.2 Interfacing with BDDs

JEDD is built on top of the BuDDy [LN] and CUDD [Som] BDD libraries, which provide a low-level interface to a BDD implementation. These libraries implement the basic operations on BDDs, with few higher-level abstractions. The *finite domain blocks* of BuDDy are one small exception; they provide a convenient way to group together BDD variables, much like the physical domains in JEDD.

Several small interactive languages have been developed for experimenting with BDDs directly. One example is BEM-II [MS97], designed for manipulating Arithmetic BDDs and solving 0-1 integer programming problems. Another is IBEN [Beh], which provides a command-line user interface to directly call the BuDDy library functions, as well as BDD visualization facilities.

The JNI allows Java code to use BDD libraries written in C through specially written wrappers. We have found it very convenient to use the wrapper generator Swig [Bea96] to automatically generate these wrappers for us. However, others have chosen to write such wrappers by hand, resulting in JBDD [Vah], a Java interface to both BuDDy and CUDD, later extended and renamed JavaBDD [Whab]. Unlike

JEDD, these JNI wrappers provide no abstraction over the underlying BDD libraries. They simply allow the library functions to be called from Java.

3.8.3 Relations with BDD back-ends

Although relations have been included in many languages, and several BDD implementations and interfaces exist, the use of BDDs as back-ends for implementing relations has been comparatively rare.

The RELVIEW system is an interactive manipulator of binary relations with a graphical user interface for visualizing them. It supports multiple back-ends, and one of the newer back-ends stores relations in BDDs [BLM02]. The fundamental difference between RELVIEW and JEDD is that RELVIEW is designed around binary relations, while much of the complexity of JEDD stems from the need to represent n-ary relations. As pointed out by Fahmy, Holt, and Cordy [FHC01], binary relations are insufficient for expressing certain problems in representing and querying graphs. Even in the case of program analysis problems which can be represented by binary relations, such a representation may be more cumbersome than with n-ary relations.

GBDD [Nil] is a C++ library providing an abstraction of BDDs based on relations of integers. Although it has partial support for n-ary relations, some operations (such as composition) require binary relations. Compared to JEDD, GBDD lacks static type checking (the type of a relation is not known until run-time), the concept of abstract attributes to be assigned to physical domains, automatic memory management, and a profiler.

The language most closely related to JEDD is CrocoPat [BNL03], a tool for querying relations representing software architecture extracted from source code. Like JEDD, CrocoPat is based on n-ary relations. CrocoPat uses a declarative, Prolog-like syntax in which attributes are identified implicitly by their position, rather than explicitly by name, as in JEDD. CrocoPat also differs from JEDD in that it is primarily a query language rather than an extension of a general-purpose language. The issue of assigning attributes to physical domains is not discussed in the CrocoPat paper.

The bddbddb tool [Whaa, WL04] approaches the same problem as JEDD– the need for a high-level notation for BDD-based program analyses – in a different way. The bddbddb language is based on Datalog [Ull88, Ull89]. A bddbddb program is a set of potentially recursive subset constraints on relations. For example, the constraint `C(x,y) :- A(x,z), B(z,y)` states that the relation `C` is a superset of the composition `A` ∘ `B`. Given such a system of constraints, bddbddb generates BDD code to find their least fixed point. The key difference between bddbddb and JEDD is that a JEDD program expresses relational operations, while a bddbddb program expresses subset constraints. If a problem has already been expressed as a system of subset constraints, it is easy to encode it in bddbddb. However, encoding all the details of a complicated program analysis problem (such as the interrelated analyses presented in Chapter 4) *purely* in terms of subset constraints may be difficult or impossible. Therefore, the requirement for a system of subset constraints is a key limitation of bddbddb. In contrast, JEDD programs can express arbitrary algorithms composed of relational operations and Java code, seamlessly integrated. The current version of bddbddb requires the programmer to assign attributes to physical domains by hand; the JEDD physical domain assignment algorithm described in Section 3.5 could be adapted to bddbddb to greatly reduce this burden.

3.9 Conclusion

In this chapter, we have presented JEDD, a language, compiler, and run-time system for expressing program analyses at a high level in terms of relations, and implementing them efficiently using BDDs. JEDD makes it feasible to implement complicated BDD-based analyses by providing static type checking and an algorithm for assigning attributes of relations to physical domains of BDDs. The JEDD runtime automatically manages the memory storing BDD nodes, and includes a profiler for tuning the BDD representation of relations. In the following chapters, we discuss the program analyses that JEDD has made it possible for us to develop.

Chapter 4
Applying BDDs to Interprocedural Program Analysis

In this chapter, we describe PADDLE, a framework of context-sensitive interprocedural program analyses for Java implemented in JEDD. The design of PADDLE was influenced by our earlier SPARK framework [Lho02, LH03], which was context-insensitive and did not make use of BDDs, and by our initial BDD-based points-to analysis [BLQ+03]. This initial work showed that BDDs can effectively represent the large sets that are needed to perform subset-based points-to analyses, and suggested that BDDs may make context-sensitive analyses feasible for programs of significant size. The key improvement of PADDLE over our earlier work is its support for variations of context sensitivity, including call site context sensitivity [SP81, Shi88] and object sensitivity [MRR02, MRR05]. In Chapter 5, we will use PADDLE to perform a study of the effect of context sensitivity variations on analysis precision.

This chapter is structured as follows. We begin in Section 4.1 by positioning PADDLE in the context of related work on interprocedural program analysis of object-oriented languages, particularly context-sensitive and BDD-based analysis. Then, in Section 4.2, we outline the key contributions of the PADDLE framework. In Section 4.3, we present the most significant part of PADDLE, the points-to analysis and call graph construction. We first give a high-level overview of its overall structure, then discuss some of its key components in more detail. The points-to information

and call graph are used by several client analyses, which we describe in Section 4.4. We conclude in Section 4.5.

4.1 Background and Related Work

4.1.1 Points-to analysis and call graph construction

Program analyses for languages with pointers to memory must take into account the effects of operations performed through pointers. Some estimate of the possible targets of pointers is therefore necessary. The purpose of a points-to analysis [EGH94] is to approximate, for each pointer in the program, the set of locations to which it could point at run time. Points-to analysis has been the subject of a large body of existing work, which has been surveyed by Hind [Hin01]. To classify the many variations of points-to analysis that have been studied, Ryder [Ryd03] proposes a set of ***dimensions*** of analysis variations which determine the relative precision and cost of analyses. We now position PADDLE within the body of work on points-to analysis by specifying where it fits with respect to each of these dimensions.

Flow sensitivity: A flow-sensitive analysis considers the order in which statements may be executed, and computes possibly different analysis information for each point in the program. In contrast, a flow-insensitive analysis computes a single analysis result valid for the entire program. PADDLE uses a hybrid approach [HH98] of first converting the program into an intermediate representation in which the control flow dependencies are captured in data dependencies, then performing a flow-insensitive analysis. Specifically, PADDLE can use either the Jimple or Shimple intermediate representations, in which variables are split along DU-UD webs [Muc97, Section 16.3.3] or converted to static single assignment (SSA) form [AWZ88], respectively. Thanks to these representations, PADDLE achieves the same precision [HH98] as an analysis which treats local variables in a flow-sensitive way (such as [VR01, WR99, WL02]) with the simplicity of a flow-insensitive analysis. However, some flow-sensitive analyses

(*e.g.* [EGH94]) additionally maintain must-points-to information, which can be used to further improve precision: when a pointer p is known to point to a unique memory location that holds a pointer q, the points-to set of q can be destructively updated at an indirect write through p.

Context sensitivity: A context-insensitive analysis produces a single analysis result for each procedure in the program. However, a given procedure may have different behaviours each time it is invoked. Therefore, a context-sensitive analysis, which produces possibly multiple analysis results for each procedure depending on how it is invoked, is potentially more precise. PADDLE supports several variations of context sensitivity. We defer a detailed discussion to Section 4.1.2.

Call graph construction:[1] In object-oriented languages with virtual dispatch, the method to be invoked at a virtual call site depends on the run-time type of the receiver object pointed-to by the call site. A points-to analysis is therefore required in order to construct a call graph; however, most points-to analyses in turn require a call graph, so a cyclic dependency exists. A simple way to break the cycle is to first use trivial points-to information to build an imprecise call graph (for example, using Class Hierarchy Analysis [DGC95]), then use the call graph to perform the points-to analysis. A preferred [Ryd03, GC01], more precise alternative is to perform call graph construction on-the-fly as the points-to analysis proceeds and new points-to pairs are discovered. PADDLE is the first BDD-based analysis which implements call graph construction in BDDs, and can therefore use the preferable on-the-fly call graph construction.

Some analyses [RMR01, WL04] construct a call graph only *partly* on-the-fly in that they require an initial call graph to determine which methods are reachable, but construct a second, more precise call graph as the points-to analysis proceeds. These partly on-the-fly analyses generate intraprocedural points-to constraints at the very beginning to model all assignments within methods

[1] In [Ryd03], this dimension is called "Program representation (calling structure)".

reachable in the initial call graph, but they generate interprocedural points-to constraints as they add call edges to the more precise call graph that they built. Therefore, the precision of partly on-the-fly analyses is in between that of ahead-of-time and fully on-the-fly call graph analyses; they model intraprocedural pointer flow like the ahead-of-time analyses, and interprocedural pointer flow like the fully on-the-fly analyses.

In Paddle, the call graph is constructed fully on-the-fly in the default setting, but Paddle can also use a call graph constructed ahead-of-time for comparison purposes.

Object representation: A points-to analysis manipulates a static abstraction of each object that may be pointed to by a pointer at run time. Two commonly-used abstractions are the run-time type of the object (*e.g.* [BS96, SHR+00, DMM96]), and the allocation site at which the object was allocated (*e.g.* [RMR01, LH03, WL02]). Paddle supports both of these abstractions (allocation site being the default setting), and provides flexibility for defining others. Furthermore, while many context-sensitive analyses use context to refine only the pointer representation, Paddle can additionally use context to refine the object representation, a technique sometimes called heap specialization [BCCH97, NKH04].

Reference (pointer) representation: A pointer abstraction represents each pointer that may occur at run time with some static abstract pointer; a points-to analysis computes a points-to set for each such abstract pointer. A common pointer abstraction is to use an abstract pointer for each variable of pointer type appearing in the program. However, some less precise abstractions have been studied, such as Rapid Type Analysis (RTA) [BS96], which uses a single abstract pointer to represent all pointers in the program. Several variations in between these two choices were studied by Tip and Palsberg [TP00]. Paddle directly supports both RTA and using an abstract pointer for each Jimple or Shimple variable (which is slightly more precise than one for each pointer variable, since

a variable in the original program may be split into multiple Jimple or Shimple variables). The design of PADDLE is flexible in terms of pointer abstraction, so other variations (such as those studied in [TP00]) could be implemented.

Field sensitivity: Certain fields of objects in the heap are also pointers, and field sensitivity defines how they are abstracted. In a field-sensitive analysis, each run-time field f of run-time object o is abstracted as the pair $\langle A(o), f \rangle$, where $A(o)$ is the object abstraction of o. Either of the two components may be ignored in the abstraction, resulting in either a field-based analysis (in which f alone is used as the abstraction) or a field-insensitive analysis (in which f is ignored and only $A(o)$ is used). Field-insensitive analysis is used for analyzing languages such as C whose type-unsafe pointer operations make it difficult to determine the field being accessed. In the context of Java, our earlier work [Lho02, LH03] showed that field-sensitive analysis is more precise but more costly than field-based analysis. PADDLE implements both field-sensitive and field-based analysis.

Directionality: An assignment of the value of a pointer p to another pointer q constrains the points-to set of p to be a subset of the points-to set of q, since q may point to any object to which p was pointing. A subset-based analysis [And94] solves only these necessary constraints. One can sacrifice precision to reduce analysis cost with an equality-based analysis [Ste96], in which the necessary subset constraints are strengthened to be bidirectional (equality constraints). As a consequence, any two points-to sets in the solution are either equal or disjoint, and a fast union-find [Tar75] algorithm can be used to compute equivalence classes of points-to sets. On Java programs, subset-based analysis has been observed [LPH01] to be significantly more precise than equality-based analysis, and efficient implementation techniques [HT01, LH03, Lho02, PKH04] have made subset-based analyses sufficiently fast for most applications. PADDLE implements subset-based analysis. The lower precision of equality-based analysis could be simulated in PADDLE by making the subset constraints bidirectional.

Type filtering:[2] When analyzing languages which enforce declared types of pointers, such as Java, a points-to analysis can filter the elements of points-to sets to exclude those incompatible with the declared type of the pointer. Our earlier work showed that type filtering both improves precision and reduces cost in both traditional [Lho02, LH03] and BDD-based [BLQ+03] points-to analyses. PADDLE performs type filtering by default, but provides an option to disable it so that its effect on precision and analysis cost can be measured.

4.1.2 Context sensitivity

Interprocedural program analyses model the effects of not only individual methods, but also of the interactions between methods. A ***context-insensitive*** analysis computes, for each method, a single analysis result that holds for all executions of the method. Because different invocations of a method may have different behaviours, it may be more precise to perform a ***context-sensitive*** analysis, which can produce different analysis results for different invocations. In general, a ***context*** is some static abstraction of a set of run-time invocations of a method. A context-sensitive analysis produces an analysis result for each pair of method and context. Different levels of context sensitivity can be achieved by choosing different abstraction functions to abstract run-time invocations as static contexts. Two common choices of context are the call site from which the method is called, and a static abstraction of the parameters passed to the method.

In general, traditional implementations of context-sensitive analyses have been too costly to scale to programs as large as recent versions of the Java standard libraries. BDD-based analyses make it feasible to study the effects of context sensitivity on these realistic programs.

Sharir and Pnueli [SP81] defined two approaches to performing context-sensitive program analysis, the ***functional*** approach and the ***call-strings*** approach. The approaches vary in two ways: in the algorithm used to compute the analysis, and in

[2]Type filtering was not included as a dimension in [Ryd03], but it has been shown [Lho02, LH03] to significantly affect analysis precision and cost.

the context abstraction that was chosen for use with each approach. In the functional approach, the effect of each method is first captured in a summary function which maps each context to the effect of the method on the analysis facts in that context. The summary function is then evaluated for each context in which the method is invoked. In the call-strings approach, the facts processed by the analysis are tagged with context, and the analysis propagates the tagged facts along the flow graph of the program. Sharir and Pnueli present their two approaches using two specific context abstractions: method arguments for the functional approach and strings of call sites for the call string approach. Analyses using call site strings as context are often called k-CFA analyses (where k is an integer limit on the length of each context string), a term coined by Shivers [Shi88]. We explain call site string context-sensitive points-to analysis in detail below in Section 4.1.2.1.

The PADDLE BDD-based encoding of context-sensitive program analyses shares some characteristics of both approaches. Like in the functional approach, PADDLE first captures the data flow graph of each method, independent of context, in a BDD analogous to the summary function. Next, the BDD is joined with the set of all contexts in which the method may be executed to form a new BDD representing the (context-sensitive) subset constraints. Finally, the subset constraints are solved to compute a points-to set for each pair of pointer and context, as in the call-strings approach. The PADDLE implementation is parameterized to allow any context abstraction to be used, including both method arguments and call sites.

In the specific area of points-to analysis, researchers have experimented with several different context abstractions. The initial points-to work by Emami, Ghiya, and Hendren [EGH94] used a string of call sites as context. They did not limit the length of each call string, but truncated the string at the first repetition of a call site in the case of recursion. Their analysis was flow sensitive, and computed an intraprocedural fixed point within each procedure; in the case of recursion, this fixed-point computation was performed over each cluster of mutually recursive procedures rather than a single procedure at a time. Another context abstraction particularly popular in alias analyses for C has been the set of alias relationships at the call

site of the procedure [WL95, LRZ93]. More recently, Milanova, Rountev, and Ryder [MRR02, Mil03, MRR05] argued that for analyzing object-oriented languages such as Java, a representation of the receiver object of each method call would be a more appropriate context abstraction. We explain object-sensitive points-to analysis in detail below in Section 4.1.2.2. Like object sensitivity, the Cartesian Product Algorithm [Age95, WS01] uses abstract objects as the context abstraction, but includes all method parameters as context, rather than only the receiver parameter.

4.1.2.1 Call site context-sensitive analyses

The example code shown in Figure 4.1 illustrates why context-insensitive points-to analysis can be imprecise. In the example, the `id()` method simply returns its argument. The method `f()` creates two objects and assigns them to `a` and `b`. It then assigns the object in `a` to `c` and the object in `b` to `d` indirectly through the `id()` method. A precise analysis would determine that `c` may point to the object allocated in line 5 but not to the object allocated in line 6, and *vice versa* for `d`. However, a context-insensitive analysis cannot determine this because it models the parameter and return value of the `id()` method using a single points-to set, which is shared by both invocations of the method. This points-to set contains both the objects allocated at lines 5 and 6, and it is assigned to both `c` and `d`, so the analysis conservatively computes that each of `c` and `d` may point to either of these objects.

A context-sensitive analysis overcomes the problem by modelling each method separately for each abstract context in which it is called. The call site from which the method is called is a popular choice of context abstraction. When analyzing the example in Figure 4.1, a call site context-sensitive analysis would analyze the `id()` method twice as if it were two separate methods, one called from line 7 and the other from line 8. In the first context, the parameter and return value of `id()` would point only to the object allocated in line 5, and in the second context, they would point only to the object allocated in line 6. Therefore, the analysis would be able to determine that `c` points only to the object allocated in line 5 and `d` points only to the object allocated in line 6.

```
1  Object id(Object o) {
2      return o;
3  }
4  void f() {
5      Object a = new Object();
6      Object b = new Object();
7      Object c = id(a);
8      Object d = id(b);
9  }
```

Figure 4.1: Imprecision of context-insensitive analysis

```
1  Object id(Object o) {
2      return id2(o);
3  }
4  Object id2(Object o) {
5      return o;
6  }
7  void f() {
8      Object a = new Object();
9      Object b = new Object();
10     Object c = id(a);
11     Object d = id(b);
12 }
```

Figure 4.2: Imprecision of 1-call-site context-sensitive analysis

The example in Figure 4.2 shows that sometimes, using a single call site as context is insufficient. This example adds an extra layer of indirection in the `id()` method. Instead of returning its argument directly, it returns it indirectly through the `id2()` method. A call site context-sensitive analysis will analyze the `id()` method twice, once for each of the call sites from which it is called. However, the `id2()` method is only called from a single call site (line 2), so it will be analyzed only once, and both objects will again be mixed together in the points-to set of its argument.

A solution to this imprecision is to use strings of multiple call sites as the context abstraction, rather than just a single call site. When analyzing `id2()`, we can include in its context not only the site that it was called from, but also the site that its caller, in turn, was called from. In general, the strings of call sites can be of any length. In our example, the `id2()` method would be analyzed twice, in the two contexts:

1. (`id()` called from line 10, `id2()` called from line 2) and

2. (`id()` called from line 11, `id2()` called from line 2).

In each context, only one of the objects would appear in the points-to set of the parameter and return value of `id2()`, and the analysis could again determine that `c` points only to the object allocated in line 8, and `d` points only to the object allocated in line 9.

So far, we have been specializing pointers and their points-to sets for different contexts. The example in Figure 4.3 illustrates why we may also want to specialize abstract heap objects. The code creates two objects, and assigns one to `a` and the other to `b`. The object creation has been encapsulated in the `alloc()` method. Therefore, in an analysis that models objects simply by their allocation site, both objects are represented by the same abstract object, namely the allocation site in line 2. Therefore, the analysis cannot determine that `a` and `b` point to distinct objects.

To eliminate this imprecision, objects may be modelled not only by their allocation site, but by a combination of the allocation site and the calling context in which the method containing it is called [BCCH97, NKH04]. Thus, in the example in Figure 4.3, the object assigned to `a` would be modelled by the allocation site in line 2 in the

```
1  Object alloc() {
2      return new Object();
3  }
4  void f() {
5      Object a = alloc();
6      Object b = alloc();
7  }
```

Figure 4.3: Imprecision of context-insensitive modelling of abstract heap objects

context of the call site in line 5, while the object assigned to `b` would be modelled by the same allocation site in the context of the call site in line 6. Thus, the two objects would be distinguished by the analysis.

4.1.2.2 Object-sensitive analyses

Milanova, Rountev, and Ryder [MRR02, MRR05] noted that when analyzing an object-oriented language such as Java, an abstraction of the receiver object of a method call may be a better choice of context abstraction than the call site. Specifically, they suggested using the allocation site of the receiver object as the context abstraction. They proposed a collection of such object-sensitive analyses parameterized according to which pointers and abstract heap objects are to be modelled context-sensitively, and how long a context string of receiver objects may be used for each of them.

We will illustrate object-sensitive analysis using the example shown in Figure 4.4. The code contains a `Container` class, which can store some `Item` in its field `item`. A setter method is provided to store an item into the field. The `go()` method creates two containers and two items, and stores the first item in the first container and the second item in the second container. A context-insensitive analysis would analyze the `setItem()` method only once, so its parameter `i` would be deemed to possibly point to both the `Items`. As a result, the points-to sets of the field `item` in both `Container` objects would contain both `Item` objects.

```
1  class Container {
2      private Item item;
3      public void setItem( Item i ) {
4          this.item = i;
5      }
6  }
7
8  void go() {
9      Container c1 = new Container();
10     Item i1 = new Item();
11     c1.setItem(i1);
12
13     Container c2 = new Container();
14     Item i2 = new Item();
15     c2.setItem(i2);
16 }
```

Figure 4.4: Example code illustrating 1-object-sensitive analysis

In a 1-object-sensitive analysis, each method would be analyzed in the context of the allocation site of the receiver object on which it was called. In particular, the `setItem()` method would first be analyzed in the context of the `Container` object allocated in line 9. In that context, the parameter of `setItem()` would be the `Item` allocated in line 10. Therefore, only this `Item` would be added to the points-to set of the field `item` of `Container` objects allocated in line 9. The `setItem()` method would then be analyzed a second time in the context of the `Container` object allocated in line 13. In this case, the parameter to `setItem()` would be the `Item` allocated in line 14, so only this `Item` would be added to the points-to set of the field `item` of the `Container` object allocated in line 13. Thus, the analysis would be able to show that `c1.item` and `c2.item` point to distinct `Item`s.

Now consider the slightly modified version of the code that appears in Figure 4.5. The difference compared to the previous example is that the assignment to the field `item` has now been delegated to an `ItemSettingVisitor` implementing the visitor design pattern [GHJV95]. The `go()` method now applies the visitor to each container, and the `visitContainer()` method in the visitor stores the `Item` in the container. A 1-object-sensitive analysis would not be able to distinguish the two `Item`s stored in the `item` field of the two `Container`s, because both are written into the field inside the `visitContainer()` method called on the same receiver object, namely the `visitor`.

In a 2-object-sensitive analysis, each method would be analyzed in the context of strings of up to two receiver object allocation sites. Specifically, the `apply()` method in the `Container` class would be analyzed twice, once for each of the `Container` allocation sites. Then, because the `visitContainer()` method is called from `apply()`, it would also be analyzed twice, in the contexts of the following two receiver object strings:

1. (`Container` allocated in line 21, `Visitor` allocated in line 11) and
2. (`Container` allocated in line 25, `Visitor` allocated in line 11).

In each of these contexts, only one of the `Item` objects would be passed through the `arg` parameter of the `apply()` and `visitContainer()` methods. Therefore, the analysis would distinguish the two `Item` objects stored in the two `Container` objects.

```
1  interface Visitor {
2      public void visitContainer( Container c, Object arg );
3  }
4
5  class ItemSettingVisitor implements Visitor {
6      public void visitContainer( Container c, Object arg ) {
7          c.item = (Item) arg;
8      }
9  }
10
11 static Visitor visitor = new ItemSettingVisitor();
12
13 class Container {
14     Item item;
15     public void apply( Visitor v, Object arg ) {
16         v.visitContainer( v, arg );
17     }
18 }
19
20 void go() {
21     Container c1 = new Container();
22     Item i1 = new Item();
23     c1.apply(visitor, i1);
24
25     Container c2 = new Container();
26     Item i2 = new Item();
27     c2.apply(visitor, i2);
28 }
```

Figure 4.5: Example code illustrating k-object-sensitive analysis

The example in Figure 4.6 is another small variation of Figure 4.4. In this case, the `item` field of the `Container` object has been replaced with an array. A similar pattern is commonly used in the collections classes in the Java standard library. In a points-to analysis that models heap objects by their allocation site only, all instances of the `Item[]` array would be modelled as a single object, because they are all allocated at the same allocation site, in line 4. Therefore, all `Item` objects stored in any `Container` would be added to the single points-to set representing the contents of the `Item[]` array, and the analysis would not be able to distinguish `Item` objects added to different `Container`s.

```
1  class Container {
2      private Item[] item;
3      public Container() {
4          item = new Item[1];
5      }
6      public void setItem( Item i ) {
7          this.item[0] = i;
8      }
9  }
10
11 void go() {
12     Container c1 = new Container();
13     Item i1 = new Item();
14     c1.setItem(i1);
15
16     Container c2 = new Container();
17     Item i2 = new Item();
18     c2.setItem(i2);
19 }
```

Figure 4.6: Example code illustrating object-sensitive heap abstraction

To eliminate this imprecision, an analysis must distinguish the instances of the `Item[]` array allocated for different instances of `Container`. This can be done by

modelling the `Item[]` array not only by its allocation site, but by its allocation site annotated with the allocation site of the receiver object of the method in which the array is allocated. In the example, the `Container` constructor is called on two receiver objects, namely the `Container` object allocated in line 12 and the `Container` object allocated in line 16. Therefore, each of the `Item[]` arrays allocated in the constructor of one of these objects can be abstractly represented by its allocation site annotated with the allocation site of the receiver of the constructor, namely the `Container` object to which the array corresponds. This abstract representation of heap objects distinguishes the two `Item[]` arrays created for the two different `Container` objects, and therefore makes it possible for the analysis to distinguish `Item`s stored in the two different `Container`s.

4.1.2.3 Zhu/Calman/Whaley/Lam algorithm

Zhu and Calman [ZC04] and Whaley and Lam [WL04] have developed an algorithm for efficiently representing k-CFA call graphs in BDDs, where k is the depth of the longest possible non-recursive call chain in the program. The algorithm takes a complete context-insensitive call graph constructed ahead-of-time as input, and transforms it into a k-CFA context-sensitive one. The process consists of the following steps, which we illustrate in Figure 4.7.

1. An arbitrary context-insensitive call graph such as the one shown in Figure 4.7(a) is made into a DAG by merging every strongly-connected component into a single node. The DAG resulting from merging the strongly-connected component consisting of nodes D and E is shown in Figure 4.7(b).

2. Every node in the DAG with multiple incoming edges is cloned once for every incoming edge. This is performed recursively until every node has at most one incoming edge (*i.e.* the result is a tree). The tree resulting from our example is shown in Figure 4.7(c). Since the tree contains a cloned node for every path through the DAG, it may be very large. However, the Zhu/Calman/Whaley/Lam algorithm constructs a compact BDD representation of the tree. The key to constructing this representation quickly is a special

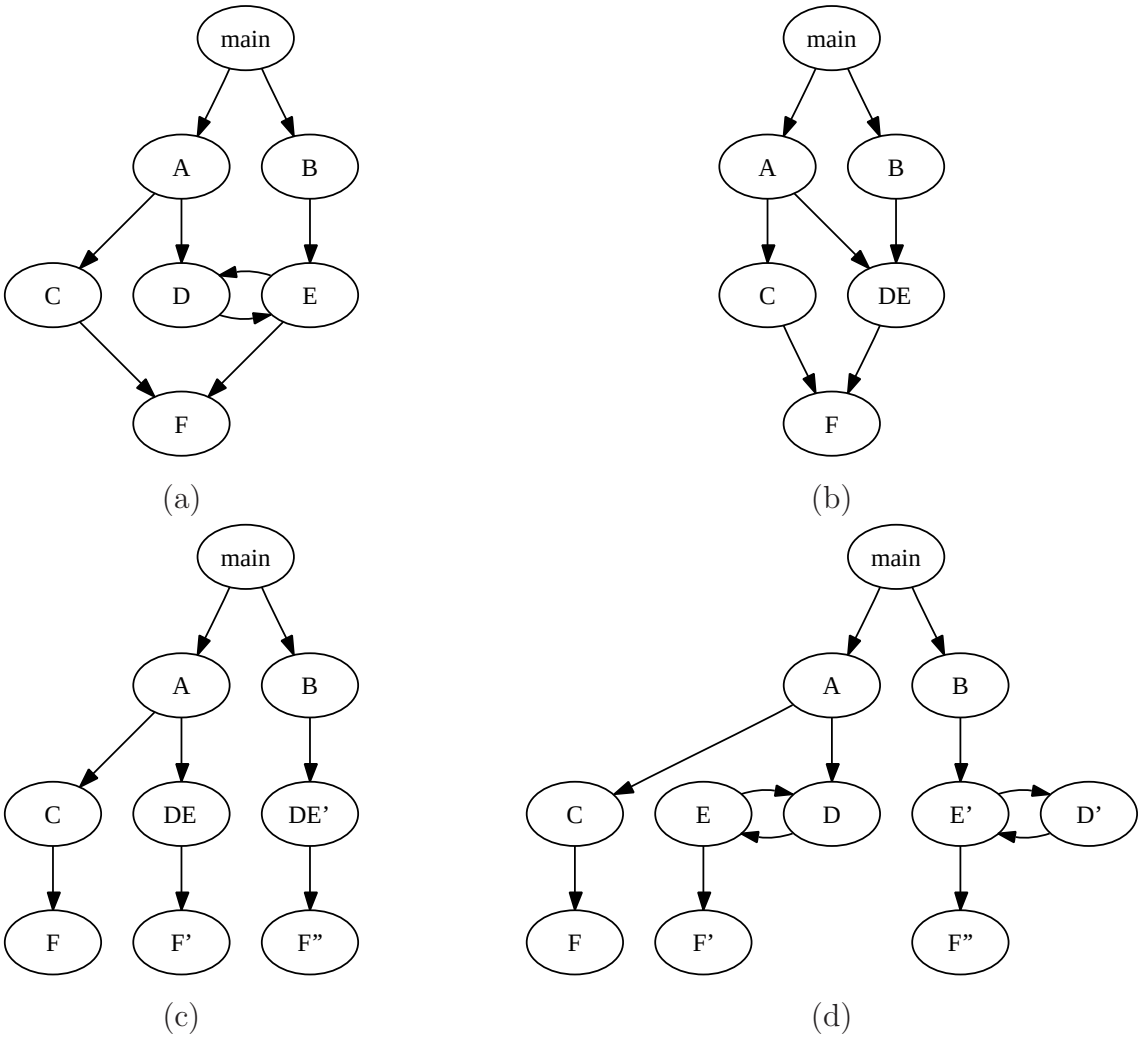


Figure 4.7: Steps of Zhu/Calman/Whaley/Lam algorithm applied to example graph

BDD operation based on a binary adder circuit, which is described in detail by Zhu and Calman [ZC04, Section 4.2].

3. The strongly-connected components are un-merged into their original methods. The context-insensitive call edges that were originally within each strongly-connected component in the context-insensitive graph are reintroduced into each clone of the component. Every call edge that led into or out of a method of a strongly-connected component in the original call graph now just leads into or out of a clone of the strongly-connected component as a whole. When the strongly-connected component is un-merged, these cloned edges are made to lead into or out of the clone of the specific method that they led into or out of in the original call graph. The resulting call graph for our example is shown in Figure 4.7(d). Although this final step was not mentioned explicitly by Whaley and Lam [WL04], it is a crucial part of the algorithm.

Once a context-sensitive call graph such as the one in Figure 4.7(d) has been constructed, it can be used to perform a points-to analysis. Whaley and Lam [WL04] used the context-sensitive call graph to generate subset constraints for a field-sensitive subset-based analysis. Their points-to analysis modelled pointer variables context-sensitively using the k-CFA context strings from the call graph, and heap objects context-insensitively using only their allocation site. For comparison with the other variations of context sensitivity, we have implemented the Zhu/Calman/Whaley/Lam algorithm within PADDLE.

4.1.3 BDD-based program analyses

Several researchers have recently used BDDs to implement program analyses, including both points-to analyses similar to our work, as well as very different kinds of analyses.

4.1.3.1 Points-to and call graph analyses

Concurrently with our initial BDD-based points-to analysis for Java [BLQ+02, BLQ+03], Zhu [Zhu02] devised a similar BDD-based points-to analysis for hardware synthesis programs written in C. Zhu and Calman [ZC04] and Whaley and Lam [WL04] designed an algorithm for computing k-CFA call graphs from context-insensitive call graphs using BDDs. We described this algorithm in detail above in Section 4.1.2.3. Both groups performed points-to analysis on the resulting context-sensitive call graph: Zhu and Calman applied Zhu's [Zhu02] points-to analysis for C, while Whaley and Lam applied our [BLQ+02, BLQ+03] points-to analysis for Java.

4.1.3.2 Other program analyses

In a very different use of BDDs, Ball and Rajamani [BR01] lifted a flow-sensitive finite-set dataflow analysis to keep track of *a set* of dataflow sets for each program point, in order to track correlations between elements of dataflow sets, achieving a path-sensitive analysis. They used BDDs to compactly represent the large sets of sets.

Sagiv, Reps, and Wilhelm [SRW02] have constructed a framework based on three-valued logic for expressing program analyses, particularly heap shape analyses. Although very expressive, this framework has memory requirements that are often prohibitive when analyzing non-trivial programs. Manevich *et al.* [MRF+02, Man03] compared the original representation of these data structures in their Three-Valued Logic Analysis (TVLA) framework with two new representations, one using BDDs, and one using a novel BDD-like data structure that they developed for representing maps [Man03, Section 3.2.3]. The memory requirements of both new representations were found to be about an order of magnitude lower than the original representation. Analysis times were found to be about the same with all three representations.

Sittampalam, de Moor, and Larsen [SdML04] formulated program analyses using conditions on control flow paths. These conditions contain free metavariables corresponding to program elements (such as variables and constants). To perform an

analysis, these metavariables were instantiated with specific elements from the particular program being analyzed. BDDs were used to efficiently represent and search the large space of possible instantiations.

4.2 Key Contributions of the Paddle Framework

Having placed Paddle in the context of existing work, we now outline the key contributions of the Paddle framework.

On-the-fly call graph construction: Object-oriented languages such as Java support virtual method dispatch, which means that the method invoked from a call site depends on the run-time type of the receiver. The run-time type, and therefore the call target, can be approximated precisely using a points-to analysis. However, performing an interprocedural points-to analysis requires a call graph of call site targets, so there is a circular dependence between call graph construction and points-to analysis. Existing work on BDD-based Java points-to analysis [BLQ+03, WL04] resolves this cyclic dependence by first constructing a call graph based on conservative, imprecise assumptions about receiver types, using the call graph to generate points-to constraints and encode them in BDDs, and finally performing the points-to analysis by solving the constraints. It is generally accepted [Ryd03, GC01] that this approach is significantly less precise than the alternative approach of iterating both the call graph construction and the points-to set propagation together until an overall fixed point is reached. In Paddle, we have implemented the latter, more precise approach. We have also implemented the less precise ahead-of-time call graph construction for comparison.

BDD-based prerequisite and client analyses: In previous work on BDD-based points-to analysis, only the points-to set propagation was performed using BDDs. However, points-to analysis relies on other prerequisite information about the program being analyzed, which was previously computed using traditional analyses. In Paddle, we show how these prerequisite analyses can

be implemented in BDDs as well. In particular, in PADDLE, we use a BDD representation to compute subtype relationships, resolve virtual method calls, keep track of call edges between methods, and determine which methods are reachable in the call graph. In addition, we have implemented BDD-based client analyses that make use of the points-to and call graph information once it has been computed. We present the client analyses for Java in Section 4.4 of this chapter, and the client analyses for AspectJ in Chapter 6.

Reducing the cost of encoding prerequisite analysis results in BDDs: The process of converting traditional representations of large relations into a BDD representation is very costly in terms of execution time. When large relations such as the call graph and subtype relationships are constructed using traditional analyses and later converted to BDDs, as is done in existing BDD-based points-to analyses, the conversion often takes more time than the subsequent points-to analysis itself. Encoding this required information in BDDs is therefore an important barrier to the overall efficiency of the analysis. However, as described above, PADDLE computes these large prerequisite relations in BDDs, rather than using traditional analyses. Therefore, only the small, initial relations needed by these analyses need to be converted from traditional representations to BDDs, greatly reducing the conversion cost.

Parameterized context sensitivity: While existing work [ZC04, WL04] has shown that context sensitivity is feasible in BDD-based analyses, little is known about how different variations of context sensitivity affect the precision of analysis results on benchmarks of significant size. BDDs make context sensitivity feasible, but is it worthwhile? PADDLE makes it possible to experiment with different variations of context sensitivity, including object sensitivity [MRR02, MRR05], a form of context sensitivity which promises to be particularly effective for object-oriented languages such as Java. We have used PADDLE to perform an in-depth study of the effects of context sensitivity variations on analysis precision; we discuss the study and its results in Chapter 5.

Modular design: The PADDLE framework is designed as a collection of simple components connected by worklists. This modular design makes it easy to modify the system to implement new analyses by adding or replacing some of the components. Each component is implemented in both a BDD-based version and a traditional version. Normally, all components are instantiated in the same version to avoid the cost of repeatedly converting between BDD-based and traditional representations. For debugging purposes, a mixture of BDD-based and traditional components can be instantiated to help locate the cause of any discrepancies between the two versions of the analysis.

4.3 Points-to Analysis and Call Graph Construction

In this section, we present the core part of PADDLE, the points-to analysis and call graph construction. We first give a high-level overview of its structure in Section 4.3.1. In Sections 4.3.2 to 4.3.5, we provide more detail about its key parts. Finally, we discuss using an existing call graph instead of constructing one on-the-fly in Section 4.3.6.

4.3.1 High-level structure

A very high level view of the analyses and their dependencies is shown in Figure 4.8. Call graph construction determines which methods of the program are reachable during execution, and the possible targets of each call site. Points-to constraints are generated to model the effects of each reachable method and the flow of parameters and return values along each call edge. Points-to sets are propagated along the constraints. The computed points-to sets of call site receivers are used to resolve virtual calls, generating additional call edges and reachable methods.

At a finer level of detail, each of the boxes of Figure 4.8 is implemented in PADDLE as a collection of components, each performing some basic analysis, connected by worklists expressing the dependencies between the analyses. We will present the full list of components later in Figure 4.9 and Sections 4.3.2 through 4.3.5. Each component defines an `update()` method which processes the new analysis facts appearing

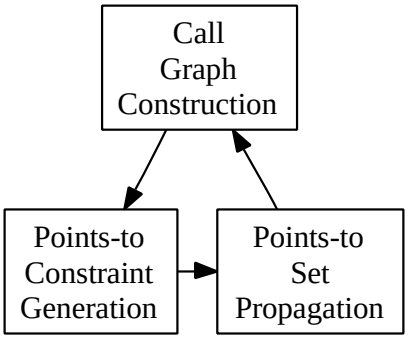


Figure 4.8: Very high level overview of call graph and points-to analyses

on its input worklists, uses them to compute new analysis facts, and adds the new facts to its output worklists for other components to use.

A separate scheduler maintains a global worklist of components which need to be updated, and calls their `update()` methods in turn, until an overall global fixed point is reached. A component is added to the global worklist whenever an analysis fact is added to one of its input worklists.

A given worklist may have multiple components adding facts into it. For example, the fact that a given pointer points to a given object may be generated by the component that processes simple pointer assignments, or by the component that processes loads from fields of heap objects. A worklist may also be the input to multiple components. Every analysis fact added to the worklist is seen by all components that read from it, as if each of these components had their own worklist, and each analysis fact were added to all of them. This is needed because some facts must be processed by several components. For example, a new call edge added to the call graph must be processed both by the component which creates points-to constraints modelling the flow of the parameters and return value of the call, and the component which keeps track of which methods are reachable in the call graph.

Each component and the worklists are implemented in two versions, a traditional version and a BDD-based version. Since BDD operations process an entire relation at a time, a BDD component generally processes the whole batch of new analysis facts appearing on its input worklist in one step, producing a batch of new analysis facts

to be added to its output worklist. A traditional component processes one analysis fact at a time. In normal operation, components and worklists are instantiated either all in their BDD-based version, or all in their traditional version. However, the two versions share the same interfaces, so it is possible to mix traditional and BDD-based versions. Therefore, if a user of PADDLE prototypes a new component in only one of the versions, it can interoperate with both versions of the other components. Mixing traditional and BDD-based versions of components is also useful for tracking down any discrepancies in the outputs of the two versions.

The BDD-based version of a worklist is implemented as a JEDD relation, with each tuple representing an analysis fact. Components add relations of new facts to it using the union operation, and a component that reads the whole worklist resets the relation to the empty relation. When multiple components are reading a worklist, a separate relation is maintained for each reader.

The traditional version of a worklist is implemented as a chunked array. A pointer is maintained to the first free element of the array,[3] where new analysis facts are added. Each component reading from the worklist maintains a pointer to the next element to be read. Thus, all the readers can share the same chunked array. When all readers have read the elements of a chunk, there are no longer any references to it, and the chunk is automatically reclaimed by the garbage collector.

Figure 4.9 shows the specific components and connecting worklists which make up the call graph and points-to analyses of PADDLE. Each component is shown as an oval, and each worklist as a rectangle. The names of components and worklists correspond to the names in the PADDLE code. Each worklist stores analysis facts encoded as tuples of a single type. In the figure, the type of the tuples stored in each worklist is given by the sequence of letters under the worklist name, with each letter representing a given type. For example, the worklist named receivers contains tuples consisting of a local variable (L), method (M), statement (S), method signature (I), and kind (K).

[3]Since Java does not allow pointers to individual array elements, the pointer is implemented as a reference to the chunk, along with an integer index into the chunk.

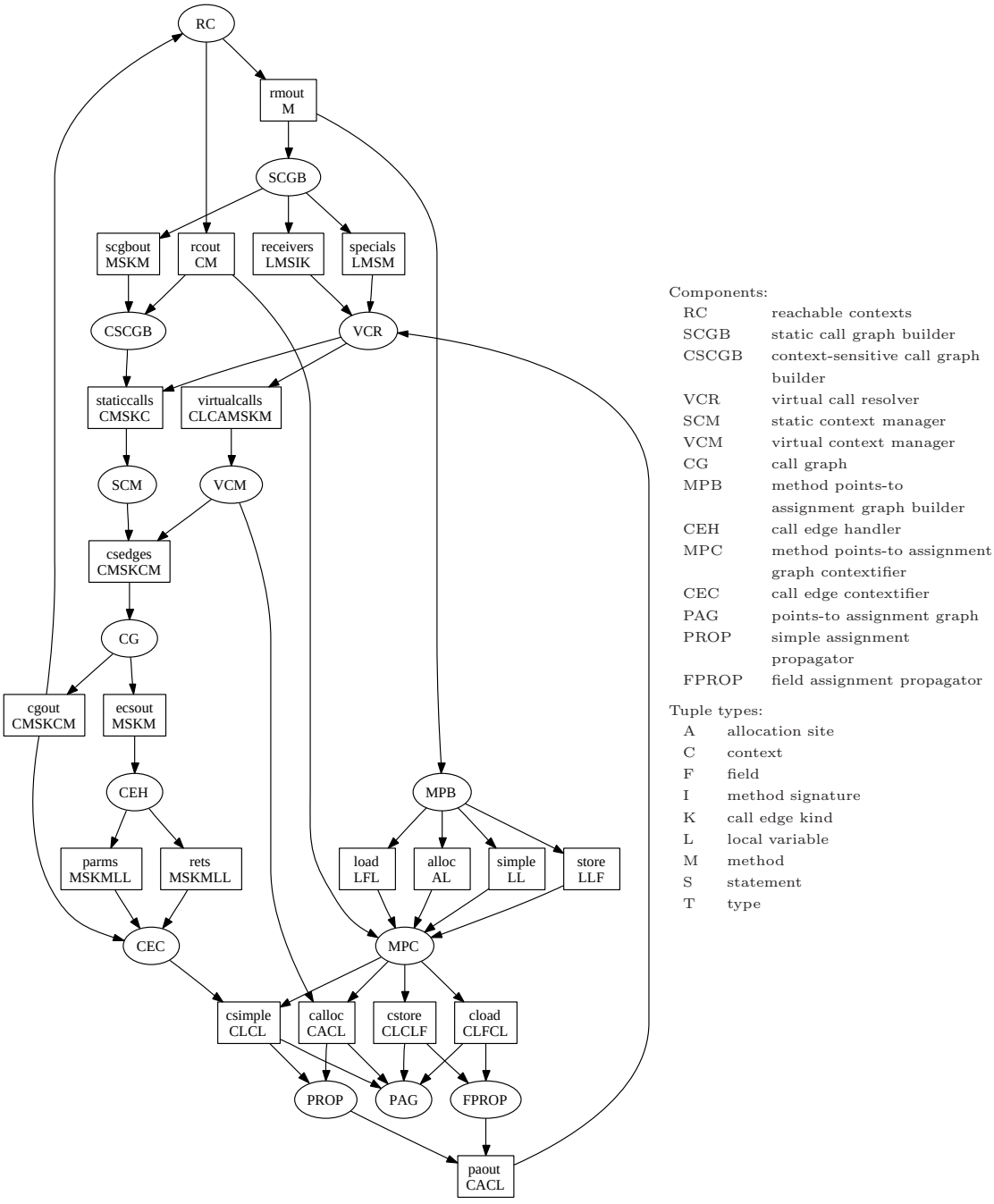


Figure 4.9: Components of call graph and points-to analyses in the default on-the-fly call graph configuration of Paddle

Call graph construction, which is discussed in detail in Section 4.3.2, is performed by the reachable contexts (**RC**), static call graph builder (**SCGB**), context-sensitive call graph builder (**CSCGB**), virtual call resolver (**VCR**), static context manager (**SCM**), virtual context manager (**VCM**), and call graph (**CG**) components. Points-to constraints, discussed in detail in Section 4.3.3, are generated by the method points-to assignment graph builder (**MPB**), call edge handler (**CEH**), method points-to assignment graph contextifier (**MPC**) and call edge contextifier (**CEC**) components. Propagation of points-to sets, discussed in Section 4.3.4, is performed by the points-to assignment graph (**PAG**), simple assignment propagator (**PROP**) and field assignment propagator (**FPROP**) components.

4.3.2 Call graph construction

A key problem in context-sensitive call graph construction is that the number of contexts can grow intractably large. BDDs help by representing the sets of contexts implicitly. In designing a BDD-based context-sensitive analysis, we must be careful to keep the sets of contexts implicit in the BDDs, and not explicitly enumerate them during the analysis. Although the PADDLE framework constructs a context-sensitive call graph, some operations in the overall analysis require call graph information to be extracted from the BDDs and made explicit. To do this efficiently, PADDLE constructs a context-insensitive view of the call graph within BDDs, makes this context-insensitive view explicit, uses it to perform the operation and encode its result in BDDs, then specializes the BDD-encoded results of the operation for the relevant contexts. We will see examples of this technique later in this section and in Section 4.3.4.

We start our discussion of call graph construction with the reachable contexts (**RC**) component. This component keeps track of the set of methods that have so far been found to be reachable through the call graph, and the contexts in which they are reachable. It produces output into two worklists. Each pair of method and context in which it is reachable is added to the **rcout** worklist. Each unique method found to be reachable, regardless of context, is added to the **rmout** worklist, which is a context-insensitive view of the **rcout** worklist for those operations that require

it. At the beginning of the analysis, the reachable contexts (**RC**) component is initialized with the entry points of the program in the null context. The component then processes newly discovered call edges from the **cgout** worklist to find newly reachable method-context pairs, and adds them to its output worklists.

The static call graph builder (**SCGB**) is one of the components that require a context-insensitive view of the call graph. The component processes each reachable method from the **rmout** worklist, extracts information about call sites from its Jimple representation, and encodes it into tuples added to its output worklists. This must be done without context to avoid re-processing the Jimple code of the method multiple times for the many contexts in which it may be called. Later, in the context-sensitive call graph builder (**CSCGB**) and virtual call resolver (**VCR**) components, the call site information generated by the static call graph builder (**SCGB**) component will be specialized for the contexts in which the method is called.

At some call sites (see below), the target of the call is independent of the context of the call; these call sites are resolved immediately, and the resulting call edge is inserted into the **scgbout** worklist. Call sites whose target depends on information about some object (such as the receiver of the call) may have different targets in different contexts, since the relevant object may be different in different contexts. Therefore, these call sites are not resolved immediately, but information about them is inserted into the **receivers** and **specials** worklists, to be processed later when points-to information is available.

Call sites whose target is independent of context include:

- staticinvoke instructions,
- implicit calls to static initializers from instructions that may trigger static initialization,
- implicit calls to any `finalize()` methods of every object allocated, and
- at a call to `Class.newInstance()`, implicit calls to constructors of any classes that the user has specified as potentially loaded by reflection.

Call sites whose target depends on the actual type of the receiver, which are added to the **receivers** worklist, include:

- virtualinvoke and interfaceinvoke instructions,
- implicit calls to `Thread.run()` at call sites of `Thead.start()`,
- implicit calls to the `run()` method of `PrivilegedAction` and `PrivilegedExceptionAction` at call sites of `AccessController.doPrivileged()`, and
- implicit calls to the static initializer of classes loaded reflectively using `Class.forName()` (the specific class can be determined if the argument to `Class.forName()` is known to point to a constant string).

The specialinvoke instruction is treated differently than other method calls. Although the target of a specialinvoke is independent of the run-time type of the receiver, the call only succeeds if the receiver is non-null. Therefore, if the points-to set of the receiver of a specialinvoke is empty, the specialinvoke can never call its target, so analysis precision can be improved by excluding the call from the call graph. Therefore, the static call graph builder (**SCGB**) immediately resolves the targets of special calls, but the call edges are not added to the **scgbout** worklist. Instead, they are added along with their receiver to the separate **specials** worklist, so that they can later be added to the call graph only when the points-to set of the receiver, in the relevant context, is determined to be non-empty.

The context-sensitive call graph builder (**CSCGB**) composes the context-independent call edges generated by the static call graph builder in the **scgbout** worklist with the reachable method-context pairs in the **rcout** worklist. Each context-independent call edge from a given method is thereby implicitly cloned once for each context in which the method is reachable. The resulting context-sensitive edges are added to the **staticcalls** worklist.

The virtual call resolver (**VCR**) combines points-to information (computed using points-to set propagation, which we will discuss in Section 4.3.4) with the information about call sites in the **receivers** and **specials** worklists to determine the targets of

these calls. Depending on the kind of call site, the points-to information determines the target of the call in one of three ways. First, for virtual calls, the target is determined based on the run-time type of the receiver object according to the procedure specified in the Java Virtual Machine Specification [LY99, Chapter 6]. Second, for special calls, the target has already been determined in the static call graph builder, and is available in the **specials** worklist. As soon as the points-to set of the receiver is non-empty, the call edge is copied to the **virtualcalls** worklist. Third, for implicit invocations of static initializers caused by calls to `Class.forName()`, the call target depends on the parameter passed to `Class.forName()`. If the points-to set of the parameter contains only string constants, each constant specifies the name of the class whose static initializer is called. If the points-to set contains an object that is not known to be a string constant, it could be an arbitrary string, so the analysis generates call edges to the static initializers of all classes that the user has specified as possibly loaded by reflection (using the Soot command-line switch `-dynamic-class`).

Because the virtual call resolver must implement the complicated resolution procedures of the Java Virtual Machine Specification and handle all the special kinds of call edges, it is one of the most complicated components of Paddle. Therefore, we will return to it in more detail in Section 4.3.5, and explain how it is implemented in terms of Jedd BDD operations.

The call edges generated by the context-sensitive call graph builder (**CSCGB**) and the virtual call resolver (**VCR**) are stored in the **staticcalls** worklist if their target method is static, or the **virtualcalls** worklist if their target method is an instance method. At this point, each call edge has a context associated with its source call site, but not with its target method. The static context manager (**SCM**) and virtual context manager (**VCM**) determine the context to be associated with the target of the call, which depends on the context abstraction being used. Paddle contains the following implementations of the context managers, each implementing a different context abstraction. We described the different context abstractions in detail in Section 4.1.2.

1. The context-insensitive context managers assign each call target the null context.

2. The call-site (1-CFA) context managers select the source statement (call site) of the call as the context for the target method.

3. The object-sensitive virtual context manager selects the abstract object representing the receiver of the call as the context for the target method. For static methods, there is no receiver object, so the object-sensitive static context manager just copies the context of the source method as the context for the target method.

4. The call-site-string (k-CFA) context managers copy the context of the source method, append the source statement of the call, then truncate the resulting call string to at most k entries.

5. The object-string (k-object-sensitive) virtual context manager copies the context of the source method, appends the abstract object representing the receiver of the call, then truncates the resulting call string to at most k entries. The object-string static context manager just clones the source context as the context for the target method.

6. The unique-object-string (unique-k-object-sensitive) virtual context manager copies the context of the source method and appends the abstract object representing the receiver of the call, but only if the context string does not already contain it. Thus, within every object string, each abstract object is unique. The object-string static context manager just clones the source context as the context for the target method. When analyzing calls on the `this` pointer, which are very common in Java programs, unique-object-string context sensitivity is more precise than object-string context sensitivity. The target of the call on the `this` pointer has the same receiver as the caller, so the same abstraction of the receiver object is added twice to the context string. Since the length of the context string is limited, this redundant abstract receiver object causes some

other, potentially useful, abstract receiver to be pushed out from the context string. In a unique-object-string context-sensitive analysis, however, each abstract receiver is added to the string at most once, so other abstract receivers are not needlessly pushed out.

The resulting complete, context-sensitive call edges are inserted into the **csedges** worklist.

The call graph (**CG**) component stores all the context-sensitive call edges and indexes them to support queries for all edges originating from a given method or statement, or all edges whose target is a given method. Each context-sensitive call edge is inserted into the **cgout** worklist if the same edge has not been inserted into it before. Therefore, the **cgout** worklist contains all call edges in the context-sensitive call graph, with each call edge appearing exactly once. This worklist is used as input to the reachable contexts (**RC**) component to find method-context pairs that become reachable through the call graph. The call graph (**CG**) also maintains a context-insensitive view of the call graph in the **ecsout** worklist. For each context-sensitive call edge processed, the context is removed, and the resulting context-insensitive edge is added to the **ecsout** worklist if the same edge has not been added into it before. This context-insensitive view of the call graph will be needed to generate points-to constraints to model pointer flow through method parameters and return values, as discussed in the next section.

4.3.3 Points-to constraint generation

Points-to constraints are generated to model the flow of objects along assignments between pointers in the program. Each of these assignments is either intraprocedural in that it is implied by the execution of some method (for example, by an explicit assignment statement in the method), or interprocedural due to parameter and return value passing at a method call. The constraints modelling the former are generated by the method points-to assignment graph builder (**MPB**), and the constraints modelling the latter are generated by the call edge handler (**CEH**).

In both cases, each statement of the program being analyzed and each call edge must be processed individually to generate the specific constraint that it induces. It is important that this processing only be done once for each method or call edge, rather than once for each context in which the method is reachable or in which the call may occur. The number of contexts may be very large, and it would be prohibitively costly to reexamine the code for each context. Instead, PADDLE first generates context-insensitive versions of the points-to constraints for each method and for each context-insensitive call edge, encodes them in BDDs, than specializes the constraints for the relevant contexts by implicitly making a copy of them for each context in which the method is reachable or in which the call edge occurs. The context specialization is done as a BDD operation for all the relevant contexts at once; it does not have to be done one context at a time. Intraprocedural points-to constraints are specialized by the method points-to assignment graph contextifier (**MPC**). Interprocedural points-to constraints are specialized by the call edge contextifier (**CEC**).

When specializing points-to constraints for different contexts, it is important to distinguish local (stack-allocated) variables, whose lifetime is a single method call, so they are necessarily distinct variables in distinct calling contexts, from global variables (such as static fields), whose values persist between method calls and therefore between different contexts. It would be unsound to model global variables as separate variables in different contexts, because values written to them in one context persist and may be read out of them in any other context. Milanova, Rountev, and Ryder [MRR02, MRR05] suggest that the set of variables modelled context-sensitively may be varied to achieve different tradeoffs between analysis efficiency and precision. Indeed, each local variable may be modelled context-sensitively or context-insensitively according to the wishes of the analysis designer without sacrificing soundness, but global variables must be modelled context-insensitively. In PADDLE, the following are treated context-sensitively:

1. each local variable,

2. for each method, the parameters and return value, and, in the case of an instance method, the implicit `this` parameter, and

3. each temporary variable generated by PADDLE to hold the result of each cast expression and the intermediate arrays created by each `multianewarray` bytecode instruction.

The following are treated context-insensitively:

1. static fields and, in a field-based analysis, instance fields,
2. the global variable representing all exceptions potentially thrown,
3. the finalizer queue to which the garbage collector adds finalizers to be executed, and
4. the temporary variables generated by PADDLE to hold
   (a) each string constant,
   (b) the default class loader, main thread group, and main thread instantiated by the VM,
   (c) the string created by the VM containing the name of the main class,
   (d) the array of command-line arguments created by the VM and the string arguments that it contains,
   (e) the abstract object representing all objects potentially instantiated using reflection.

Like pointer variables, some abstract heap objects can also be distinguished by the context in which they are allocated, if it is known. In PADDLE, abstract heap objects representing explicit allocation sites in the program are modelled context-sensitively. The following abstract heap objects are treated as global and always modelled context-insensitively:

1. the abstract object representing each string constant,
2. each abstract object representing the run-time type of the object rather than its allocation site.

3. the default class loader, main thread group, and main thread instantiated by the VM,

4. the string created by the VM containing the name of the main class,

5. the array of command-line arguments created by the VM and the string arguments that it contains,

6. the abstract object representing all objects potentially instantiated using reflection.

The method points-to assignment graph builder (**MPB**) processes each method in the **rmout** worklist of unique reachable methods, and inserts the corresponding context-insensitive points-to set constraints into the **simple**, **alloc**, **store**, and **load** worklists. The **simple** worklist holds simple subset constraints between pairs of pointer variables. The **alloc** worklist holds allocation site constraints of the form $o \in pt(v)$, where o is an abstract object and v is a pointer variable. The **store** and **load** worklists hold field-sensitive field store and load constraints of the form $v \subseteq b.f$ and $b.f \subseteq v$, respectively, where v and b are pointer variables, f is a field, and $b.f$ is a field dereference expression. The method points-to assignment graph contextifier (**MPC**) takes these worklists as input, along with the **rcout** worklist of all method-context pairs in which each method is reachable. Each points-to constraint involving a local pointer variable or abstract object is implicitly copied for each context in which the method is reachable. Global pointer variables and global abstract heap objects are always assigned the single global null context. The resulting context-sensitive points-to constraints are added to the context-sensitive constraint worklists, **csimple**, **calloc**, **cstore**, and **cload**.

The call edge handler (**CEH**) reads the **ecsout** worklist of context-insensitive call edges. For each call edge, it generates points-to assignment constraints modelling the flow of method parameters into the called method, and of the return value out of the called method, and inserts them into the **parms** and **rets** worklists, respectively. Each tuple representing a constraint also contains the context-insensitive call edge that induced the constraint. The call edge contextifier (**CEC**) matches each of these

context-insensitive call edges against all the context-sensitive call edges in the **cgout** worklist to find all the contexts in which the call occurs. The points-to constraints are then specialized for these contexts (by adding context to all local pointer variables), and inserted into the **csimple** worklist.

Separating the constraints modelling method parameters and method return values into the two worklists **parms** and **rets** is necessary because they must be specialized differently. In a constraint representing a method parameter, the source of the pointer flow is the argument being passed in from the caller, and is assigned the context of the caller, while the destination of the pointer flow is the parameter in the callee, and is assigned the context of the callee. In a constraint representing a return value, it is the opposite. The source of the pointer flow is the return value in the callee, so it is assigned the context of the callee. The destination of the pointer flow is the variable in the caller to which the return value is stored, so it is assigned the context of the caller.

The receiver of a virtual method call, passed to the implicit `this` variable of the called method, may be modelled in one of two ways. First, it may be modelled like any other method parameter, by a subset constraint indicating flow from the receiver at the call site to the `this` pointer of the callee. A second, more precise alternative is to limit the objects that flow from the receiver at the call site to the `this` pointer of the callee to only those whose run-time type causes that specific callee to be resolved. This second alternative has been observed to be more precise [Buc04], but it is more complicated, because it cannot be modelled with a simple subset constraint. In addition, the second alternative is applicable only when the call graph is generated on-the-fly as the points-to analysis proceeds, rather than ahead-of-time, because it depends on each call edge being annotated with the specific abstract heap objects (from the points-to analysis) that caused the call target to be resolved. In Paddle, we have implemented both alternatives of modelling the receiver because we wish to experiment with both on-the-fly and ahead-of-time call graph construction. To implement the first alternative, the call edge handler (**CEH**) has a setting which causes it to generate points-to constraints for the receiver in the same way as for the explicit method parameters. The second alternative requires information about the

specific object that caused each context-sensitive call edge to be added. In PADDLE, this information is available only in the **virtualcalls** worklist when it is processed by the virtual context manager (**VCM**), so we have added code to generate the relevant context-sensitive points-to constraints in the virtual context manager. The constraints are of the form $o \in \textit{points-to}(p)$, much like the constraints generated at object allocation sites, so the virtual context manager adds them to the **calloc** worklist. The objects passed from the receiver to the `this` pointer may be different in different calling contexts, so these constraints cannot be generated once in a context-insensitive way, and specialized later. The additional complexity of the more precise alternative is compounded by context sensitivity, and the precision improvement that it brings comes at a significant cost in the complexity and reduced modularity of the analysis.

4.3.4 Points-to set propagation

The context-sensitive points-to constraints generated by the method points-to assignment graph contextifier (**MPC**) and the call edge contextifier (**CEC**) are read into the points-to assignment graph (**PAG**), which indexes them in order to answer queries such as finding all the assignment edges originating at a given variable. The points-to set propagator can obtain the points-to constraints from two sources. First, it can issue queries to the points-to assignment graph, which yield information about all the constraints that have been generated so far that involve a given variable. Second, the propagator reads constraints from the context-sensitive constraint worklists, **csimple**, **calloc**, **cload**, and **cstore**. By reading all constraints from these worklists whenever it is executed, the propagator obtains a list of all new constraints that were generated since the last time that the propagator ran.

Points-to sets are propagated using two components, the simple assignment propagator (**PROP**), which processes simple subset constraints from the **csimple** worklist and allocation constraints from the **calloc** worklist, and the field assignment propagator (**FPROP**), which processes field store and load constraints from the **cstore**

and **cload** worklists. For each propagation algorithm, the simple assignment propagator (**PROP**) and field assignment propagator (**FPROP**) components are designed to work together, so we implement both in a single class. The `update()` method is called to update the simple assignment propagator (**PROP**) component, and a separate `fieldUpdate()` method is called to update the field assignment propagator (**FPROP**) component. Each of these methods returns a boolean value indicating whether it produced any new output (points-to pairs). A true return value indicates to the scheduler that other components that depend on points-to information should also be updated.

The PADDLE framework contains three traditional and two BDD-based implementations of points-to set propagation algorithms. The traditional algorithms are based on the iterative, incremental worklist, and incremental alias edge propagation algorithms that we described in detail in [Lho02]. The BDD-based algorithms are derived from the basic and incremental BDD algorithms that we developed and described in detail in [BLQ+02, BLQ+03]. Compared to these simpler algorithms, the algorithms implemented in PADDLE have two significant extensions. First, they have been extended to handle new points-to constraints being introduced as a result of constructing the call graph on-the fly. Second, they have been extended to be context-sensitive. In this chapter, we limit our detailed discussion to the two BDD based points-to propagation algorithms.

In practice, the incremental BDD-based propagation algorithm is the most efficient when PADDLE is using BDD-based versions of its other components, and the worklist propagation algorithm is the most efficient when PADDLE is using traditional versions of its other components.

4.3.4.1 Basic propagation algorithm

The PADDLE implementation of the basic propagation algorithm is presented in Figures 4.10 and 4.11. The algorithm maintains two fields, `pt` storing the points-to relation for simple variables, and `fieldPt` storing the points-to relation for fields of

```
1  <varc,var,objc,obj> pt = 0B;
2  <basec,base,fld,objc,obj> fieldPt = 0B;
3
4  boolean update() {
5      <varc,var,objc,obj> oldPt = pt;
6      pt |= calloc.get();
7      pt |= propSimple(pt, pag.allSimple().get());
8      return pt != oldPt;
9  }
10 <varc,var,objc,obj> propSimple(
11     <varc,var,objc,obj> pt,
12     <srcc,src,dstc,dst> simple) {
13
14     <varc,var,objc,obj> ret = 0B;
15     while(true) {
16         pt = (dstc=>varc, dst=>var)
17                                     simple {srcc,src}
18                                      <> pt {varc,var};
19         pt -= ret;
20         if(pt == 0B) break;
21         ret |= pt;
22     }
23     return ret;
24 }
```

Figure 4.10: JEDD code for basic propagation algorithm for simple assignments

```
1  boolean fieldUpdate() {
2      <varc,var,objc,obj> oldPt = pt;
3      fieldPt |= propStore(pt, pag.allStore().get(), pt);
4      pt |= propLoad(fieldPt, pag.allLoad().get(), pt);
5      return pt != oldPt;
6  }
7  <basec,base,fld,objc,obj> propStore(
8      <varc,var,objc,obj> pt,
9      <srcc,src,fld,dstc,dst> store,
10     <varc,var,objc,obj> storePt) {
11
12     <objc,obj,varc,var,fld> objectsBeingStored =
13         (dstc=>varc, dst=>var) store {srcc,src}
14                                <> pt {varc,var};
15     return               objectsBeingStored {varc,var}
16         <> (objc=>basec, obj=>base) storePt {varc,var};
17 }
18 <varc,var,objc,obj> propLoad(
19     <basec,base,fld,objc,obj> fpt,
20     <srcc,src,fld,dstc,dst> load,
21     <varc,var,objc,obj> loadPt) {
22
23     <basec,base,fld,dstc,dst> loadsFromHeap;
24     loadsFromHeap =                    load{srcc,src}
25         <> (objc=>basec, obj=>base) loadPt{varc,var};
26     return (dstc=>varc, dst=>var) loadsFromHeap {basec,base,fld}
27                                          <> fpt {basec,base,fld};
28 }
```

Figure 4.11: JEDD code for basic propagation algorithm for field loads and stores

heap objects. The `update` method first reads any new allocation edges from the **calloc** worklist. Since an allocation edge is a constraint of the form $o \in \textit{points-to}(p)$, the set of allocation edge tuples is added directly to the points-to set relation. The second step in the basic propagator is to propagate points-to sets along all simple assignment edges. Each simple assignment edge is a constraint of the form $\textit{points-to}(p) \subseteq \textit{points-to}(q)$, so the points-to set of p must be added into the points-to set of q. The `propSimple` method takes two relations as parameters, a points-to relation, and a relation of simple assignment edges, and propagates the points-to relation along the assignment edges iteratively until a fixed-point is reached. The propagation is implemented by the composition operation in line 18, much like in the simple example we showed in Figure 3.3 of Chapter 3. An important difference in the PADDLE version of the implementation is that each variable and object now has an associated context, which is stored in a separate attribute of each relation. For example, the assignment edges relation `simple` now has the additional attributes `srcc` and `dstc` to store the context for the source and destination pointer variable, respectively. In the basic propagation algorithm, on each update, the `propSimple` method is called with the complete points-to set and the complete set of assignment edges from the points-to assignment graph. In the incremental propagation algorithm, we will improve on this by propagating only the new part of the points-to relation.

The `fieldUpdate` method first propagates points-to sets along store edges (using the `propStore` method) to the field points-to relation, then propagates the field points-to relation along load edges (using the `propLoad` method) back to the points-to sets for simple variables. The field points-to relation represents facts of the form $o \in \textit{points-to}(b.f)$, indicating that field f of the object b may point to the object o.

The `propStore` method takes three relation parameters. The `pt` relation contains the points-to pairs to be propagated along stores. The `store` relation contains the store edges, with each edge representing a store instruction of the form $v_d.f := v_s$. Finally, the `storePt` relation is the points-to relation used to determine the potential objects that the target of the store (v_d) may point to. The stores are processed in two steps. First, in line 14, the sources of the store edges are looked up in the points-to relation, yielding a relation of tuples of the form (o, v_d, f) indicating that the object

o is being stored into $v_d.f$. In the second step, in line 16, v_d is looked up in `storePt` to determine which objects are being stored into.

The `propLoad` method also takes three relation parameters. The `fpt` relation contains the points-to sets of fields of heap objects. The `load` relation contains the set of load edges of the form $v_d := v_s.f$. The `loadPt` relation is the points-to relation used to determine the objects that the source of the store (v_s) may point to. Like stores, loads are processed in two steps. First, in line 25, the load sources are looked up in `loadPt`, giving a relation of tuples of the form (b, f, v), indicating that the points-to set of $b.f$ is being propagated into the points-to set of v. Second, in line 27, the points-to set of $b.f$ is looked up and added into the points-to set of v in the `fpt` relation.

Like the propagation along simple assignment edges, the heap field propagation code has an extra attribute for each variable and object to store its context.

4.3.4.2 Incremental propagation algorithm

The PADDLE implementation of the incremental propagation algorithm is presented in Figures 4.12 and 4.13. The incremental algorithm reuses the propagation methods `propSimple`, `propStore` and `propLoad` of the basic propagation algorithm, but they are called to propagate only the new points-to tuples, instead of propagating all tuples in every iteration.

In addition to the `pt` and `fieldPt` fields of the basic propagation algorithm, the incremental propagation algorithm maintains a third field `ptFromLoad`, which acts as a worklist for the `fieldUpdate` method to communicate new points-to pairs resulting from field loads to the `update` method.

The `update` method processes new information from three sources. First, for each newly introduced allocation edge, a tuple is stored into the `ptFromAlloc` relation. Second, all newly introduced simple assignment edges must be processed. Specifically, the `update` method propagates the *full* points-to relation along only the *newly introduced* simple assignment edges. The points-to pairs resulting from this propagation are stored in the `ptFromSimple1` relation. Third, the `update` method must

also process any new points-to pairs that were generated in the `fieldUpdate` method due to field loads since the last execution of `update`. These points-to pairs are retrieved from the `ptFromLoad` field, and the field is then cleared. Finally, the three kinds of *new* points-to pairs must be propagated along *all* existing simple assignment edges. The resulting points-to pairs are saved in the `ptFromSimple2` relation. Finally, all the new points-to relations except `ptFromLoad` (that is, the `ptFromAlloc`, `ptFromSimple1`, and `ptFromSimple2` relations) are added to the `pt` output worklist. The `ptFromLoad` relation does not need to be added, because it is already added to `pt` in the `fieldUpdate` method.

The `fieldUpdate` method in the incremental propagation algorithm performs the same operations as in the basic propagation algorithm. The only difference is that in addition to adding new points-to pairs from field loads to the `pt` relation, it also adds them to the `ptFromLoad` relation for use by the `update` method.

4.3.5 Virtual call resolution

The virtual call resolver is one of the most complicated components of Paddle, because it must implement the complicated resolution procedures defined by the Java Virtual Machine Specification, as well as handle the special kinds of implicit call edges. To our knowledge, this is the first time that virtual method resolution has been implemented in BDDs. In this section, we present a simplified version of the virtual call resolver to demonstrate how it is implemented using Jedd and BDDs. The simplified virtual call resolver presented in this section resolves only explicit virtual calls due to virtualinvoke and interfaceinvoke instructions. The complete virtual call resolver[4] actually implemented in Paddle resolves not only these calls, but also all the special kinds of calls that were described in Section 4.3.2.

For a given method signature and actual receiver type, the virtual call resolver determines which method will actually be invoked. This is done by searching the class hierarchy from the receiver type upwards for a class implementing a method with the

[4]The virtual call resolver is implemented in the class `BDDVirtualCalls`.

```
1  <varc,var,objc,obj> pt = 0B;
2  <basec,base,fld,objc,obj> fieldPt = 0B;
3  <varc,var,objc,obj> ptFromLoad = 0B;
4
5  boolean update() {
6
7      <varc,var,objc,obj> oldPt = pt;
8
9      <varc,var,objc,obj> ptFromAlloc = calloc.get();
10     <varc,var,objc,obj>
11         ptFromSimple1 = propSimple(pt, csimple.get());
12     <varc,var,objc,obj>
13         ptFromAllocAndSimple1 = ptFromAlloc|ptFromSimple1;
14     <varc,var,objc,obj>
15         ptFromSimple2 = propSimple(ptFromAllocAndSimple1|ptFromLoad,
16             pag.allSimple().get());
17     ptFromLoad = 0B;
18     pt |= ptFromAllocAndSimple1|ptFromSimple2;
19     return pt != oldPt;
20 }
```

Figure 4.12: JEDD code for incremental propagation algorithm for simple assignments

```
1  final boolean fieldUpdate() {
2      <varc,var,objc,obj> oldPt = pt;
3
4      fieldPt = propStore( pt, pag.allStore().get(), pt );
5      <varc,var,objc,obj> ptFromThisLoad |=
6          propLoad( fieldPt, pag.allLoad().get(), pt );
7      pt |= ptFromThisLoad;
8      ptFromLoad |= ptFromThisLoad;
9
10     return pt != oldPt;
11 }
```

Figure 4.13: JEDD code for incremental propagation algorithm for field loads and stores

given signature. The PADDLE virtual call resolver does this for an entire relation of receiver types and method signatures at once.

The JEDD code for the algorithm is shown in Figure 4.14. We will walk through the code, explaining how it would resolve the example virtual calls shown in the relations in Figure 4.15. The algorithm starts with the relation `receiverTypes`, with each tuple specifying a receiver type and a method signature. An example of such a relation is shown in Figure 4.15(a), specifying the receiver type B at two call sites with signatures foo() and bar(). Before starting to walk up the hierarchy starting from the receiver type, the algorithm first saves a copy of the original receiver type in each tuple using the attribute copying operation in line 6. In the resulting `toResolve` relation, each tuple contains the method signature and two copies of the receiver type (see Figure 4.15(b)). As the algorithm searches for the target method, one copy (in tgttype) of the receiver type will be moved up the class hierarchy, while the other copy (in rectype) will be kept unchanged to keep track of the original receiver type.

The next step is to determine whether the class of the receiver type implements a method with the signature. This is done by joining the `toResolve` relation with the `implementsMethod` relation shown in Figure 4.15(c), which keeps track of the

```
1  <rectype, signature, tgttype, method> answer = 0B;
2
3  public void resolve( <rectype, signature> receiverTypes,
4                       <subtype, supertype> extend ) {
5
6      <rectype, signature, tgttype> toResolve =
7          (rectype=>rectype, rectype=>tgttype) receiverTypes;
8
9      do {
10         <rectype, signature, tgttype, method> resolved =
11              toResolve     {tgttype, signature} ><
12              declaresMethod{type,    signature};
13          answer |= resolved;
14          toResolve -= (method=>) resolved;
15          toResolve = (supertype=>tgttype)
16             (toResolve{tgttype} <> extend{subtype});
17     } while( toResolve != 0B );
18 }
```

Figure 4.14: JEDD code for virtual call resolution

(a)

type	signature
B	foo()
B	bar()

(b)

rectype	signature	tgttype
B	foo()	B
B	bar()	B

(c)

type	signature	method
A	foo()	A.foo()
B	bar()	B.bar()

(d)

rectype	signature	tgttype	method
B	bar()	B	B.bar()

(e)

rectype	signature	tgttype
B	foo()	B

(f)

subtype	supertype
B	A

(g)

rectype	signature	supertype
B	foo()	A

(h)

rectype	signature	tgttype	method
B	foo()	A	A.foo()

Figure 4.15: Example of resolving virtual method calls
(a) `receiverTypes` (b) `toResolve` in line 6 (c) `implementsMethod` (d) `resolved` in first iteration (e) `toResolve` in line 15 (f) `extend` (g) result of composition in line 15 (h) `resolved` in second iteration

methods implemented by each class and their signatures. This join, which appears on line 11, matches the current class (tgttype attribute of `toResolve`) with the class implementing the method (type attribute of `implementsMethod`), and the method signature (signature attribute of `toResolve`) with the method signature of the implemented method (signature attribute of `implementsMethod`). For each class and method signature being resolved, if the class implements a method with the matching signature, then the resulting relation `resolved` contains a tuple with the method signature, two copies of the receiver type, and the target method. In our example, the only match is type B and signature bar(), resulting in the `resolved` relation in Figure 4.15(d). In general, these are the method calls that we have just resolved by finding a method with the desired signature, so in line 13, we add them to our answer.

The next step is to remove the resolved call sites from the set of sites left to resolve. The `resolved` relation has the method attribute which `toResolve` lacks, so it is removed using projection in line 14 before the resolved call sites are subtracted. After doing this to our example, we obtain the `toResolve` relation in Figure 4.15(e).

The final step is to move up the class hierarchy by replacing each class in the tgttype attribute with its immediate superclass. This is done with a composition (in line 15) of the `toResolve` relation with the `extend` relation passed in from the class hierarchy, which encodes the immediate superclass (extends) relationship. In our example, as Figure 4.15(f) shows, B is a subclass of A. The tgttype attribute is matched with the subtype attribute in the `extend` relation, and a composition rather than a join is used because the attributes being compared (the subtype) are not needed; from the `extend` relation, only the supertype attribute is needed. The resulting relation has replaced each object in the tgttype attribute of `toResolve` with its immediate superclass, as shown in Figure 4.15(g). Before it can be assigned to `toResolve`, the supertype attribute must be renamed to tgttype to match the schema of `toResolve`. Finally, if the set of call sites to be resolved is not yet empty, the algorithm starts another iteration of the loop to resolve them. Figure 4.15(h) shows the call resolved in the second iteration. Together, the relations in Figures 4.15(d) and (h) show the final result: the targets of calling foo() and bar() with a receiver of type B are A.foo() and B.bar().

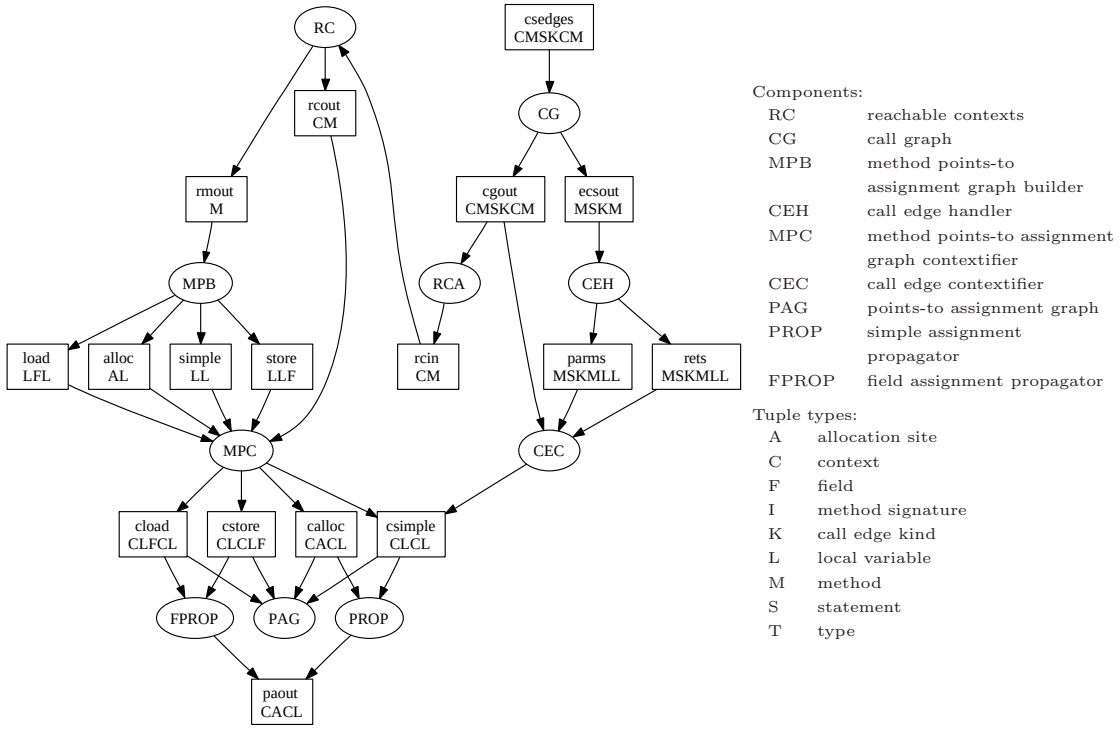


Figure 4.16: Components of call graph and points-to analyses in the ahead-of-time call graph configuration

4.3.6 Reusing an existing call graph

The default configuration of Paddle as shown in Figure 4.9 builds a call graph on-the-fly as the points-to analysis proceeds. Paddle can also be configured to use an existing call graph to compute only points-to information. This makes it possible to compare the results of Paddle against the results of other context-sensitive analysis techniques which inherently require the call graph to be built ahead-of-time in a separate step, such as the technique of Zhu and Calman [ZC04] and Whaley and Lam [WL04].

The ahead-of-time call graph configuration of Paddle is shown in Figure 4.16. It is similar to the on-the-fly call graph configuration in Figure 4.9, but lacks the **SCGB**, **CSCGB**, **VCR**, **SCM** and **VCM** components and associated worklists. The edges of the ahead-of-time call graph must be inserted into the **csedges** worklist

before PADDLE begins processing. One additional difference is that in the ahead-of-time call graph configuration, PADDLE cannot implement the precise propagation of method call receiver objects to `this` pointers that was described at the end of Section 4.3.3, because it depends on building the call graph on-the-fly. Instead, the call edge handler treats `this` pointers like any other parameter, and generates simple assignment constraints from each receiver to the `this` pointer of each method that may be invoked on it.

The initial call graph can be constructed by running PADDLE in the on-the-fly call graph configuration. Thus, an ahead-of-time call graph analysis involves two separate instances of PADDLE, the first in the on-the-fly call graph configuration, and the second in the ahead-of-time call graph configuration. After the first instance finishes, the resulting points-to sets are discarded, and the resulting call graph is used as input to the second instance. The results (points-to sets and call graph) of the second instance are deemed the results of the overall analysis.

In between the two instances of PADDLE, the initial call graph may be made context-sensitive using the algorithm proposed by Zhu and Calman [ZC04] and Whaley and Lam [WL04] that we described in Section 4.1.3. This setup implements the Zhu/Calman/Whaley/Lam analysis within the PADDLE framework, so its results can be readily compared with the default configuration of PADDLE. We explained the Zhu/Calman/Whaley/Lam algorithm in detail in Section 4.1.2.3.

Implementors of the Zhu/Calman/Whaley/Lam analysis may be interested in constructing the initial call graph using Class Hierarchy Analysis [DGC95] to avoid having to perform the points-to analysis twice (once to construct the initial call graph, and a second time for the final context-sensitive analysis). To measure the precision of this approach, the instance of PADDLE building the initial call graph can be configured to simulate Class Hierarchy Analysis by assuming that every pointer can point to every object.

Figure 4.17 summarizes the possible configurations of PADDLE. As shown on the left, the default configuration is the on-the-fly call graph version of PADDLE detailed in Figure 4.9. The dashed box on the right contains the ahead-of-time call graph variations. First, the initial context-insensitive call graph may be constructed either

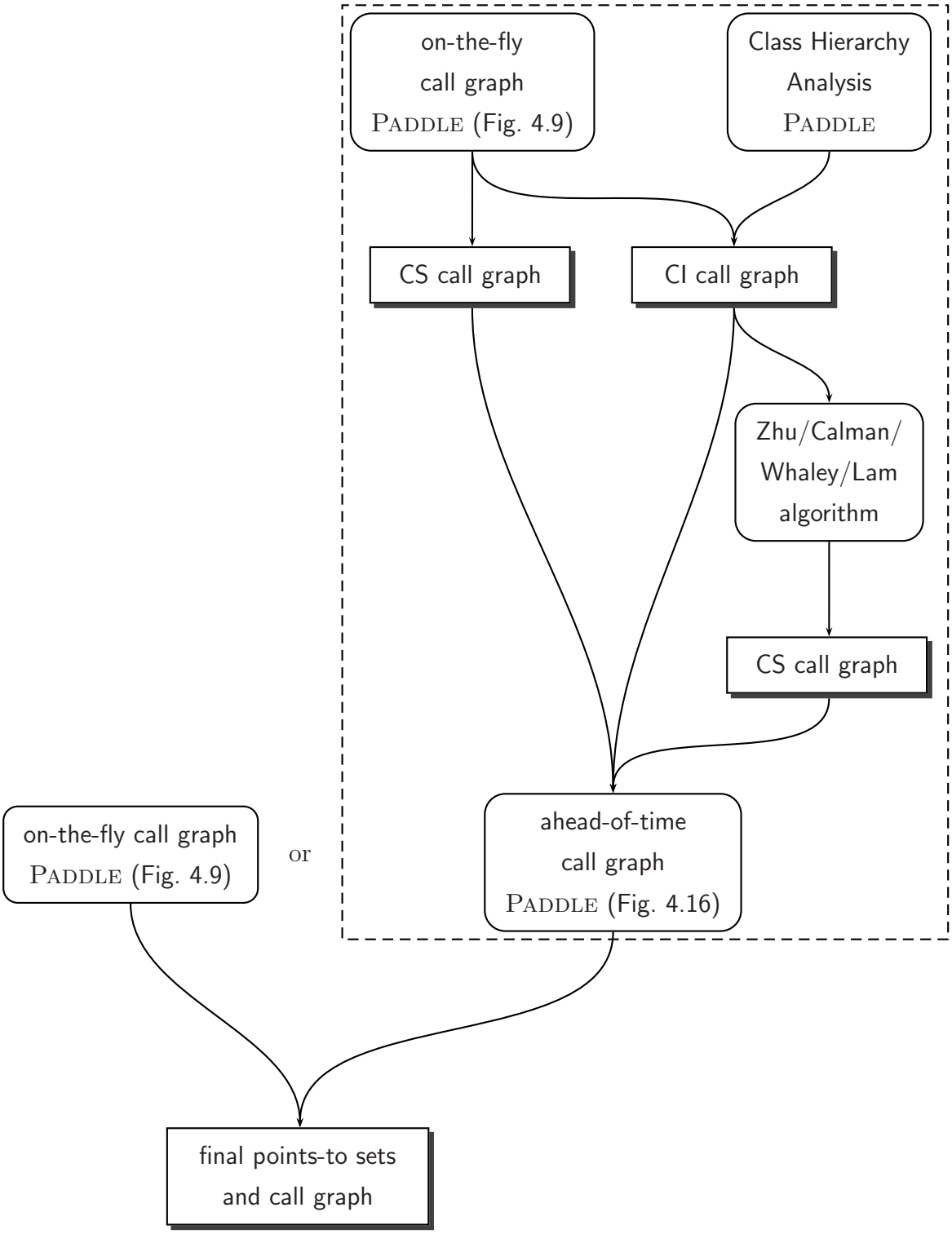


Figure 4.17: Summary of PADDLE configurations

using an on-the-fly call graph version of PADDLE, or using a version of PADDLE simulating Class Hierarchy Analysis. The resulting initial call graph can either be used as is, or it can be made context-sensitive using the Zhu/Calman/Whaley/Lam algorithm. Finally, the call graph is used by the ahead-of-time call graph variation of PADDLE detailed in Figure 4.16 to compute points-to sets.

4.4 Client Analyses

In this section, we describe client analyses which use the points-to sets and call graph computed by PADDLE to generate additional analysis information useful for program optimization and for program understanding. In Chapter 5, we will explore the effects of differences in the precision of the points-to sets and call graph on the precision of these client analyses. All of the client analyses have been implemented within PADDLE in terms of BDDs, using the JEDD language.

The call graph and points-to sets computed by PADDLE are context-sensitive. Where applicable, the client analyses are also performed context-sensitively. However, all context information is removed from the final results of the client analyses, because practical applications require the properties determined by the client analyses to hold in all contexts. In addition, we wish to compare the precision of the client analyses when using points-to sets and call graphs computed with different variations of context sensitivity, so the context information must be removed for the client analysis results to be comparable.

4.4.1 Monomorphic call sites

In object-oriented languages such as Java, the target of a method invocation depends on the run-time type of the receiver object on which the method is invoked. Determining and invoking the correct method can be a major source of run-time overhead. Moreover, the uncertainty about which method will be invoked hinders interprocedural optimizations such as method inlining. In typical programs, most invocation sites actually only invoke a single target method during execution. Various techniques

have therefore been proposed to determine the targets of these monomorphic call sites (*e.g.* [SHR+00, IKY+00, TLSS99, GDDC97]).

The call graph generated by PADDLE can be used to statically determine the targets of monomorphic call sites. Since a call site must call the same target method in every context to be considered monomorphic, the monomorphic call site analysis considers the context-insensitive call graph edges from the **ecsout** worklist. The analysis iterates through all virtual and interface edges in the call graph. The first time a call edge originates at a given call site, the call site is marked as having one target method. When another call edge originates at a call site that has already been marked, the call site is marked as polymorphic. All call sites that are not found to be polymorphic are considered monomorphic.

4.4.2 Cast safety analysis

In Java, a cast expression `(Type) o` checks that the run-time type of the object pointed to by `o` is a subtype of `Type`. If it is, the cast expression evaluates to the object `o`, but has compile-time type `Type`; otherwise, evaluating the cast raises a `ClassCastException`. An analysis which statically proves that `o` is always a subtype of `Type` is useful both for optimizing away the run-time type check, and for informing the programmer whether the cast may fail at run time.

The points-to sets computed by PADDLE can be used to conservatively estimate the set of casts that must always succeed at run time. The points-to set for each pointer represents all possible targets of the pointer, and each target has a fixed run-time type. If the run-time types of all the objects in the points-to set of `o` are subtypes of `Type`, then the cast `(Type) o` cannot fail at run time.

To perform points-to analysis in PADDLE, we consider the points-to set computed for each pointer that is the argument of a cast. If the points-to set contains an abstract object whose type is not a subtype of the declared type of the pointer, the cast is marked as potentially failing; otherwise, the cast cannot fail.

4.4.3 Side-effect analysis

A side-effect analysis computes, for each instruction in the program, an abstraction of the set of memory locations that may be read and written during the execution of the instruction. Specifically, for Java programs, a side-effect analysis determines which static fields and which instance fields of which abstract objects may be read and written by each instruction.

In general, a side-effect analysis requires both points-to sets and a call graph. For an instruction reading or writing a static field, the field can be determined directly from the instruction. An instruction reading or writing an instance field expression of the form $v.f$ reads or writes the field f of every abstract object $o \in \textit{points-to}(v)$, so the points-to set of v is needed to compute the side-effects. The side-effect of a method invocation instruction is the union of the side-effects of all the instructions of all the methods possibly invoked from the instruction, including any methods invoked transitively from those methods. Therefore, to compute the side-effects of method invocation instructions, a call graph is required.

The side-effect analysis implemented in Paddle is the same as the one we implemented in Spark and described in detail in [Lho02, LLH05], except that it is written in Jedd, and the side-effect sets are represented using Jedd relations. Since the side-effect sets are very large and many of them are similar or equal, manipulating them in BDDs reduces the cost of the analysis. The analysis first computes an intraprocedural side-effect set for each instruction, which includes only the effects of the instruction itself, and does not include any side-effects due to methods that may be called from the instruction. The Paddle points-to sets are used to determine the side-effects of reads and writes of instance field expressions. Next, for each method, the union of the side-effects of all the instructions in the method is computed as the overall side-effect for the method. Finally, the transitive closure of the call graph is computed, and the side-effects of all methods transitively callable from each method invocation instruction are added to the side-effect of the instruction.

4.4.4 Escape analysis

As described by Rountev, Milanova, and Ryder [RMR01, Sections 3.3 and 3.4], points-to sets can be used to prove that certain objects do not escape the method in which they are created (*i.e.* no references to them exist when the method returns), and that certain objects do not escape the thread in which they are created (*i.e.* they cannot be accessed during the execution of any other thread). Specifically, objects which are not reachable through the points-to graph from any static field or any field of any class implementing `java.lang.Runnable` cannot escape their creating thread and are said to be *thread-local*. A thread-local object which is additionally unreachable from the parameters and return value of the method in which it is allocated cannot escape the method and is said to be *method-local*.

The results of escape analysis are useful for optimization [ACSE99, Bla99, BH99, CGS+99, GS00, Ruf00, WR99]. In particular, method-local objects can be allocated more efficiently on the stack rather than the heap, and reclaimed immediately when the method returns, rather than later by the garbage collector. The synchronization operations required by the Java Virtual Machine Specification [LY99] can be optimized away for objects known to be thread-local. In addition, programmers may find information about which objects are method-local and thread-local useful for understanding their programs.

In Paddle, escape analysis is implemented according to the specification in [RMR01]. The set of thread-escaping objects is first initialized to the points-to sets of static fields and fields of classes implementing `java.lang.Runnable`. Its closure under the field points-to relation is then iteratively computed. All objects not found to be thread-escaping are identified as thread-local. Next, the set of method-escaping objects is initialized as the set of all thread-escaping objects. All objects in the points-to sets of method parameters and return values are added as method-escaping. Finally, the set of method-escaping objects is closed under the field points-to relation. The result is the complete set of method-escaping objects as defined by Rountev, Milanova, and Ryder [RMR01].

4.5 Conclusions

In this chapter, we have presented the PADDLE BDD-based interprocedural analysis framework. The core part of PADDLE computes points-to sets and constructs a call graph. Because the analyses are implemented in terms of BDDs, which represent contexts implicitly, PADDLE makes it feasible to perform context-sensitive analyses on large Java programs. In PADDLE, the call graph is constructed precisely, on-the-fly as the points-to analysis proceeds. PADDLE supports different variations of context sensitivity, including strings of call sites and strings of abstract receiver objects. We have implemented four client analyses that make use of the points-to sets and call graph computed by PADDLE.

In the next chapter, we will use PADDLE to perform an empirical study of the effect of variations of context sensitivity on the precision of points-to analysis, call graph construction, and the client analyses we described in Section 4.4. In Chapter 6, we will apply PADDLE to an analysis for optimizing the *cflow* construct in AspectJ programs.

Chapter 5
Empirical Study of Context Sensitivity

In this chapter, we report on an in-depth empirical study of several variations of context sensitivity, including object sensitivity [MRR02, MRR05], call site strings as the context abstraction [SP81, Shi88], and the contexts generated by the Zhu/Calman/Whaley/Lam algorithm [ZC04, WL04]. Our goal in this study is to evaluate the effect of these variations of context sensitivity on analysis precision, in order to guide future research. Specifically, we would like to determine which analyses are useful (in the sense that they improve precision) so that we can focus our future attention on practical implementation of only the useful analyses. Practical implementation of a useful context-sensitive analysis is our long-term goal, but not a direct goal of the present study.

Nevertheless, in order to be able to perform our study, our implementations of the analyses must be scalable enough to be able to analyze the significant benchmarks on which we will evaluate them. Indeed, the lack of scalable implementations of these analyses is what has prevented researchers from performing this study in the past. It is the use of BDDs and the Paddle framework that finally makes this study possible. Moreover, some of the characteristics of the analysis results that we are interested in would be very costly to measure on an explicit representation. We have found ways to perform these measurements directly on the BDD representation of the analysis results.

In our study, we compare the relative precision of analyses both quantitatively, by computing summary statistics about the analysis results, and qualitatively, by examining specific code patterns for which a given analysis variation produces better results than other variations. Context-sensitive analyses have been associated with very large numbers of contexts. We want to also determine how many contexts each variation of context sensitivity actually generates, how the number of contexts relates to the precision of the analysis results, and how feasible it is likely to be to implement practical context-sensitive analyses that scale to large benchmarks.

This chapter is organized as follows. In Section 5.1, we list the benchmarks that we used in our study. In Section 5.2, we specify the variations of context sensitivity that we have studied. We have already explained the variations in detail in Section 4.1.2 of Chapter 4. We discuss the number of contexts and its implications on precision and scalability in Section 5.3. In Section 5.4, we examine the effects of context sensitivity on the precision of the call graph. We evaluate opportunities for static resolution of virtual calls in Section 5.5. In Section 5.6, we measure the effect of context sensitivity on cast safety analysis. We surveyed related work on context-sensitive analysis in general in Section 4.1 of Chapter 4; in addition, we compare our empirical study of to other experimental evaluations of context sensitivity in Section 5.7 of this chapter. Finally, we draw conclusions from our experimental results in Section 5.8.

5.1 Benchmarks

We evaluated the different variations of context sensitivity on programs from the JOlden [CM01, CM] benchmark suite, the SpecJVM 98 benchmark suite [Sta], the DaCapo benchmark suite, version beta050224 [DaC], and the Ashes benchmark suite [VR], and on the Polyglot extensible Java front-end [NCM03]. Most of these benchmarks have been used in earlier evaluations of interprocedural analyses for Java. A list of the benchmarks appears in Table 5.1. For each benchmark, the middle section

Benchmark	Total number of classes	Total number of methods	Executed methods benchmark	Executed methods +library
bh	9	86	54	459
bisort	2	14	12	414
em3d	5	31	18	425
health	8	38	26	435
mst	6	32	31	434
perimeter	10	56	42	443
power	6	51	29	427
treeadd	2	10	5	407
tsp	2	12	12	404
voronoi	6	84	44	450
compress	41	476	56	463
db	32	440	51	483
jack	86	812	291	739
javac	209	2499	778	1283
jess	180	1482	395	846
mpegaudio	88	872	222	637
mtrt	55	574	182	616
raytrace	54	570	180	611
soot-c	731	3962	1055	1549
sablecc-j	342	2309	1034	1856
polyglot	502	5785	2037	3093
antlr	203	3154	1099	1783
bloat	434	6125	138	1010
chart	1077	14966	854	2790
jython	270	4915	1004	1858
pmd	1546	14086	1817	2581
ps	202	1147	285	945

Table 5.1: Benchmarks

of the table shows the total number of classes and methods comprising the benchmark. These numbers exclude the Java standard library[1] (which is required to run the benchmark), but include all other libraries that must accompany the benchmark for it to run successfully. The right-most section of the table shows the number of distinct methods that are actually executed in a run of the benchmark, both excluding and including methods of the Java standard library, in the columns labelled "benchmark" and "+library", respectively. The run-time call graphs were collected using the *J tool [Duf04, DDHV03]. About 400 methods of the standard library are executed even for the smallest benchmarks for purposes such as class loading; some of the larger benchmarks make heavier use of the standard library.

The first ten benchmarks (bh through voronoi) are the JOlden suite [CM01, CM]. The suite originated as a collection of pointer-intensive C programs, which were later translated to Java. As can be seen from Table 5.1, each of these benchmarks is fairly small.

The next eight benchmarks (compress through raytrace) are the SpecJVM 98 suite [Sta]. The purpose, origins and sizes of these benchmarks vary. Compress is an implementation of LZW compression [Wel84] ported to Java from C. Db is a program that performs searches and updates on a memory-resident address database. Jack is a parser generator that generates Java code from a description of a grammar. Javac is the Java source to bytecode compiler from the Java Development Kit version 1.0.2. Jess is an expert shell system. Mpegaudio is a decompressor for MPEG Layer-3 sound files. Raytrace and mtrt are two versions of a raytracer; raytrace uses a single thread, while mtrt is multi-threaded.

The next three benchmarks, soot-c, sablecc-j, and polyglot are examples of large applications that make significant use of the object-oriented features of Java. Soot-c and sablecc-j are from the Ashes suite, and polyglot is version 1.0.0 of Polyglot [NCM03] applied to its own source code. Soot-c is an early version of the Soot [VRGH+00] Java bytecode analysis and optimization framework. Sablecc-j is the SableCC [GMN+] parser generator. Given a grammar, SableCC generates not just a parser, but also a

[1] All of the measurements in this chapter were done with version 1.3.1_01 of the Sun Java standard class library.

collection of classes for representing and traversing parse trees. The SableCC grammar parser (for grammar input files) is itself generated by SableCC. Polyglot is an extensible Java front-end that performs all required type checking on Java source, and pretty-prints the final abstract syntax tree. It is intended for the development of extensions to the Java language, and achieves its extensibility through heavy use of object-oriented design patterns.

The final six benchmarks (antlr through ps) are from version beta050224 of the DaCapo suite [DaC], a collection of programs intended to make significant use of the memory management system at run time. From a more static point of view, the benchmarks are examples of large applications that use the object-oriented features of Java. Antlr generates lexers and parsers from a grammar. Bloat is a Java bytecode analysis and optimization system. Chart is a program that plots charts using the JFreeChart [Gil] library. Jython is a compiler from a variant of Python to Java bytecode. Pmd is an extensible code style checker for Java. Ps is a postscript interpreter.

5.2 Context Abstractions

Before we list the specific variations of context sensitivity that we evaluated in our study, we invite the reader to read Section 4.1.2 of Chapter 4, in which we explained the different approaches to context sensitivity in detail with examples.

Context-insensitive analysis variations: In our earlier work [Lho02, LH03] on SPARK, a predecessor of the PADDLE framework, we empirically evaluated context-insensitive analyses to find good tradeoffs between analysis precision and efficiency. Based on this earlier work, we have selected two context-insensitive analyses to serve as a baseline for our measurements of the effects of context sensitivity.

The first configuration was identified as very fast and also quite precise. In the SPARK work, it was denoted ot-aot-fs, indicating on-the-fly enforcement of

declared types, ahead-of-time call graph construction, and field-sensitive modelling of fields. We include it as an example of a practical context-insensitive configuration. In this configuration, three separate steps are performed. First, a call graph is constructed using Class Hierarchy Analysis [DGC95]. Second, subset constraints are generated between pointer variables to model flow of pointers between them. For each method reachable in the call graph, a constraint is generated for every pointer assignment appearing in the method. For each call edge in the call graph, constraints are generated to model pointer flow through method parameters and the return value. Third, a points-to set is computed for every pointer by propagating sets of allocation sites (the abstraction of heap objects) along the subset constraints. Whenever a pointer p may point to an object allocated at allocation site a at run-time, the points-to set of p contains a. Fields of objects are modelled field-sensitively. That is, the analysis maintains a separate points-to set *points-to*$(a.f)$ for every allocation site a and every field f to represent pointers stored in the field f of any object allocated at allocation site a. Declared types of pointers are enforced. That is, an allocation site a allocating an object of run-time type t is not propagated into the points-to set of p unless the declared type of p is a supertype of t. Throughout this chapter, we denote this first context-insensitive analysis **AOT**. In this configuration, client analyses use the call graph computed in the first step and the points-to sets computed in the third step of the analysis as described above.

The second context-insensitive configuration is similar to but even more precise than ot-otf-fs, the most precise configuration that we studied in the work on Spark. We include this configuration as the most precise context-insensitive configuration, to serve as a baseline for comparing the precision of context-sensitive configurations. Like in the AOT configuration, heap objects are modelled by their allocation site, fields are modelled field-sensitively, and declared types are enforced. Instead of using an initial call graph, however, the analysis constructs a call graph on-the-fly as the points-to set propagation proceeds. The three steps — call graph construction, subset constraint generation, and

points-to set propagation — are cyclically dependent. Subset constraints are generated only for the methods reachable through the partial call graph generated so far, and only for call edges already present in the call graph. Points-to sets are then propagated along subset constraints that have been generated so far. Each virtual call in the reachable methods is resolved using the types of the objects in the points-to set of the receiver. New call edges are added to the call graph, which causes new methods to become reachable. The whole process is repeated until an overall fixed point is reached. Client analyses use the resulting call graph and points-to sets. Throughout this chapter, we refer to this configuration as **OTF**.

There is a subtle detail that makes the **OTF** analysis more precise than some other analyses that have been called "on-the-fly" in earlier work, including our own work on SPARK [Lho02, LH03], Rountev, Milanova and Ryder's work [RMR01], and Whaley and Lam's BDD-based analysis [WL04]. These analyses are only *partly* on-the-fly, in the following sense. In the **OTF** analysis, subset constraint generation depends on the call graph in two distinct ways. First, the set of methods reachable in the call graph is required to generate subset constraints for pointer assignments within those methods. Second, the set of call edges in the call graph is required to generate subset constraints for parameters and return values of those calls. In the partly on-the-fly analyses, however, the first kind of subset constraints are generated at the very beginning for all methods, and only the second kind of subset constraints are actually generated on-the-fly as call edges are added to the call graph. Therefore, the points-to sets of the partly on-the-fly analyses reflect the effects of methods that can never execute because they are not reachable in the call graph. The OTF analysis, however, is more precise because it models the effects of only those methods reachable through the call graph.

All of the context-sensitive analyses described below, except the ZCWL analysis, construct the call graph completely on-the-fly like the OTF analysis.

Call site string context-sensitive variations: In Section 4.1.2.1 of Chapter 4, we described how to use call sites as the context abstraction, and provided motivating examples for using strings of multiple call sites, and for modelling abstract heap objects context-sensitively, in addition to pointer variables.

In our present study of context sensitivity, we include three variations of call site string context-sensitive analysis. All three are similar to the more precise context-insensitive variation OTF in that the call graph is constructed on-the-fly as the points-to analysis proceeds, fields are modelled field-sensitively, and declared types are enforced. In the first two variations, which we denote **1 call site** and **2 call site** throughout this chapter, context strings are limited to a length of one and two, respectively, and only pointers are modelled with context, while heap objects are modelled only by their allocation site, without context. We have included these two variations to measure how much lengthening the context strings improves precision, and to determine how lengthening strings of call sites compares with lengthening strings of receiver object allocation sites. In the third variation, which we denote **1H call site** throughout this chapter, context strings are limited to a single call site, and both pointers and abstract heap objects are modelled with context. We have included this variation to measure the effect of modelling abstract heap objects with context on analysis precision, and to compare the effectiveness of call sites and abstract receivers as the context abstraction for abstract heap objects.

Object-sensitive analysis variations: In Section 4.1.2.2 of Chapter 4, we explained the use of allocation sites of method call receiver objects as the context abstraction. We also provided motivating examples for using strings of multiple receiver object allocation sites, and for modelling abstract heap objects context-sensitively, in addition to pointer variables. In our empirical study, we evaluate the effects of these variations.

Specifically, we include five variations of object-sensitive analysis in our study. All of them are similar to the more precise context-insensitive variation OTF in that the call graph is constructed on-the-fly as the points-to analysis proceeds,

fields are modelled field-sensitively, and declared types are enforced. In the first three variations, which we denote throughout this chapter as **1-object-sensitive**, **2-object-sensitive**, and **3-object-sensitive**, all pointer variables are modelled with context strings of up to one, two, and three abstract receiver objects, respectively. Heap objects are modelled only by their allocation site, without context. We use these three variations to evaluate how the length of the context string affects precision. The fourth variation, which we denote **1H-object-sensitive**, is like the 1-object-sensitive variation, but we additionally model heap objects context-sensitively using the allocation site annotated with one abstract receiver object. We include this variation to evaluate the effect of modelling abstract objects with context on analysis precision. The fifth variation, which we denote **2U-object-sensitive** is included to compare unique object sensitivity to normal object sensitivity. It is just like the 2-object-sensitive variation, except we do not add receiver allocation sites to a context string if they are already present in the string. By not adding duplicate abstract receivers, we prevent other, potentially useful, abstract receivers from being forced out of the context string of limited length.

Zhu/Calman/Whaley/Lam algorithm: In Section 4.1.2.3 of Chapter 4, we described the Zhu/Calman/Whaley/Lam algorithm [ZC04, WL04] in detail. Recall that the algorithm requires an initial context-insensitive call graph to be constructed before it can be applied. In contrast, in all of the variations that we have defined so far except the AOT context-insensitive variation, the call graph has been built on-the-fly as the points-to analysis proceeds. Thus, in the dimension of call graph construction, the Zhu/Calman/Whaley/Lam algorithm is most like the AOT context-insensitive variation.

A key parameter is the precision of the initial call graph, which depends on how it is constructed. An obvious choice would be to construct the initial call graph using Class Hierarchy Analysis [DGC95], because it does not require points-to analysis. Recall that when using the Zhu/Calman/Whaley/Lam algorithm, the

initial context-insensitive call graph is first made context-sensitive, and points-to analysis is performed afterwards using the resulting context-sensitive call graph. Therefore, if we also used points-to analysis for the initial call graph construction, we would be performing points-to analysis twice. However, when we applied the Zhu/Calman/Whaley/Lam algorithm to a call graph constructed using CHA, it failed to complete in the available memory[2] on the larger benchmarks, despite extensive tuning of the BDD variable ordering. Therefore, like Whaley and Lam [WL04], we instead evaluated the algorithm using the much more precise call graph constructed by the OTF context-insensitive variation described above. That is, we first performed points-to analysis together with on-the-fly call graph construction to get the same call graph and points-to sets as in the OTF variation. We then discarded the points-to sets, and used the call graph as input to the Zhu/Calman/Whaley/Lam algorithm to construct a context-sensitive call graph. Finally, we performed points-to analysis a second time using the resulting context-sensitive call graph. Like in the other variations, the points-to analysis was field-sensitive and enforced declared types. Pointer variables were modelled with context, but abstract heap locations were modelled context-insensitively, like in the work of both Zhu and Calman [ZC04] and Whaley and Lam [WL04]. Throughout the rest of this chapter, we refer to this analysis variation as **ZCWL**.

5.3 Number of Contexts

Context-sensitive analysis is often considered intractable mainly because, if contexts are propagated from every call site to every called method, the number of resulting context strings grows exponentially in the length of the call chains. The purpose of this section is to shed some light on two issues. First, of the large numbers of contexts, how many are actually useful in improving analysis results? Second, why

[2]All of the results presented in this chapter were obtained with PADDLE using the BUDDY [LN] backend. BUDDY was allowed to allocate a maximum of 41 million BDD nodes (820 million bytes).

can BDDs represent such seemingly large numbers of contexts, and how much hope is there that they can be represented with more traditional techniques?

In the following three subsections, we perform three measurements of the numbers of contexts. First, we measure the total number of abstract contexts that arise with each context abstraction. Second, we define a notion of contexts that are equivalent in the sense that it is not useful to distinguish them, and measure the number of equivalence classes of contexts for each context abstraction. Finally, we measure the number of distinct points-to sets generated with each context abstraction.

5.3.1 Total number of contexts

We begin by comparing the numbers of abstract contexts that arise when a context-sensitive analysis is performed with the different context abstractions. More precisely, we measure the number of contexts that appear in the context-sensitive points-to relation. For the purpose of this measurement, we consider the method to which a context string applies as part of the context, and count the contexts rather than just the context strings. For example, if call sites are being used as the context abstraction, and a given virtual call site has two potential target methods, each of these methods invoked with the call site as the context string is considered a separate context.

Measuring the number of contexts in the context-sensitive points-to relation is straightforward when the relation is encoded in a BDD. First, we join the points-to relation with a relation that specifies for each pointer variable the method containing it. Next, we perform a projection keeping only the context and the method, to obtain a BDD representing the set of all contexts with their final target methods. Finally, the size of the set is found by calling the `size()` method (provided by JEDD) on the relation.

The measurements of the total numbers of contexts are shown in Table 5.2. Each column lists the number of contexts produced by one of the variations of context-sensitive analyses described in Section 5.2. Please refer to that section for an explanation of the analyses denoted by the column headings. The columns labelled

"context-insensitive" show the absolute number of contexts (which is also the number of methods, since in a context-insensitive analysis, every method has exactly one context). All the other columns, rather than showing the absolute number of contexts, which would be very large, instead show the number of contexts as a multiple of the "context-insensitive OTF" column (*i.e.* they show the average number of contexts per method). For example, for the bh benchmark, the total number of 1-object-sensitive contexts is $2583 \times 13.5 = 3.48 \times 10^4$. The empty spots in the table (and in other tables throughout this chapter) indicate configurations in which the analysis did not complete in the available memory, despite being implemented using BDDs.

The generally large numbers of abstract contexts explain why an analysis that represents each context explicitly cannot scale to the programs that we analyze here. While a 1-call-site-sensitive points-to analysis requires 6 to 9 times more data to be stored and processed than a context-insensitive analysis, the ratio grows to 1500 times for a 3-object-sensitive analysis.

When context strings are limited to a length of 1, the 1-object-sensitive analysis produces about twice as many contexts as the 1-call-site-sensitive analysis. However, as the context strings grow longer, the number of contexts in the object-sensitive analyses grows more slowly than in the call site string analyses. This is because it is common in Java programs to invoke a method on the `this` pointer; in this common case, the receiver object of the called method is the same as at the call site, so in many context strings, the same abstract receiver objects are repeated. Notice that in the unique-object-sensitive analysis, in which repeated receiver objects are filtered out, the number of contexts grows much more quickly (compare the 1-object-sensitive column first to the 2-object-sensitive column, then to the 2U-object-sensitive column).

The Zhu/Calman/Whaley/Lam algorithm effectively performs a k-CFA analysis in which the value of k is the maximum call depth in the original call graph after strongly connected components have been merged. This maximum call depth is shown in parentheses in the ZCWL column of Table 5.2. Because k changes from one benchmark to another, the total number of contexts is much more variable than in the other variations of context sensitivity. On the javac, soot-c, sablecc-j, chart, and pmd benchmarks, the algorithm failed to complete in the available memory.

	context-insens.		object-sensitive					call site				
Benchmark	AOT	OTF	1	2	3	1H	2U	1	2	1H	ZCWL	(k)
bh	3160	2583	13.5	111	1491	13.2	1225	6.2	233	6.2	2601	(17)
bisort	3115	2541	13.6	112	1516	13.4	1245	6.2	237	6.2	2042	(14)
em3d	3147	2552	13.6	112	1509	13.3	1239	6.2	236	6.2	1824	(14)
health	3131	2556	13.6	112	1507	13.3	1237	6.2	235	6.2	1914	(14)
mst	3132	2558	13.6	112	1525	13.3	1243	6.2	235	6.2	1665	(14)
perimeter	3143	2568	13.5	111	1500	13.2	1231	6.2	234	6.2	1594	(14)
power	3132	2558	13.6	112	1507	13.3	1236	6.2	235	6.2	2623	(18)
treeadd	3108	2534	13.7	113	1521	13.4	1248	6.2	237	6.2	1565	(14)
tsp	3140	2545	13.6	112	1513	13.3	1243	6.2	236	6.2	2585	(15)
voronoi	3162	2589	13.5	111	1490	13.2	1222	6.2	232	6.2	2464	(15)
compress	10989	2596	13.7	113	1517	13.4	1231	6.5	237	6.5	2.9×10^4	(21)
db	10993	2613	13.7	115	1555	13.4	1239	6.5	236	6.5	7.9×10^4	(22)
jack	11245	2869	13.8	156	1872	13.2	1149	6.8	220	6.8	2.7×10^7	(45)
javac	12120	3780	15.8	297	13289	15.6	1489	8.4	244	8.4		(41)
jess	11620	3216	19.0	305	5394	18.6	1415	6.7	207	6.7	6.1×10^6	(24)
mpegaudio	11198	2793	13.0	107	1419	12.7	1148	6.3	221	6.3	4.4×10^5	(31)
mtrt	11115	2738	13.3	108	1447	13.1	1170	6.6	226	6.6	1.2×10^5	(26)
raytrace	11115	2738	13.3	108	1447	13.1	1170	6.6	226	6.6	1.2×10^5	(26)
soot-c	5502	4837	11.1	168	4010	10.9	847	8.2	198	8.2		(39)
sablecc-j	12816	5608	10.8	116	1792	10.5	660	5.5	126	5.5		(55)
polyglot	6204	5616	11.7	149	2011	11.2	797	7.1	144	7.1	10130	(22)
antlr	4493	3897	15.0	309	8110	14.7	1715	9.6	191	9.6	4.8×10^9	(39)
bloat	6496	5237	14.3	291		14.0		8.9	159	8.9	3.0×10^8	(26)
chart	14804	7069	22.3	500		21.9	1413	7.0	335			(69)
jython	5274	4401	18.8	384		18.3	2264	6.7	162	6.7	2.1×10^{15}	(72)
pmd	8497	7219	13.4	283	5607	12.9	1196	6.6	239	6.6		(55)
ps	11744	3874	13.3	271	24967	13.1	1543	9.0	224	9.0	2.0×10^8	(29)

Table 5.2: Total number of abstract contexts

5.3.2 Equivalent contexts

Next, we consider that many of the large numbers of abstract contexts are equivalent in the sense that the points-to relations computed in many of the abstract contexts are the same. More precisely, we define two method-context pairs, (m_1, c_1) and (m_2, c_2) to be ***equivalent*** if $m_1 = m_2$, and for every local pointer variable p in the method, the points-to set of p is the same in both contexts c_1 and c_2.

We illustrate context equivalence with a concrete example. Consider a method M with four pointer variables a, b, c, and d, which is called in four abstract contexts P, Q, R, and S. Suppose that the points-to sets for these methods in these contexts are found to be as shown in Figure 5.1. In this example, the contexts R and S are equivalent for the method M, because the points-to sets of each of the four pointers a, b, c, and d are the same in the two contexts. Therefore, there are three equivalence classes of contexts, namely $\{(M, P)\}$, $\{(M, Q)\}$, and $\{(M, R), (M, S)\}$.

When two contexts are equivalent, there is no point in distinguishing them, because the resulting points-to relation is independent of the context. In this sense, the number of equivalence classes of method-context pairs reflects how worthwhile context sensitivity is in improving the precision of points-to sets.

The number of equivalence classes of contexts can be measured directly on the BDD representing the context-sensitive points-to relation by performing the following two steps. First, the variable ordering of the BDD must be arranged with the context attribute at the beginning of the BDD, followed by the pointer variable and abstract heap object attributes. We do this in JEDD by copying the points-to relation into a relation for which we have explicitly specified the physical domain assignment. Second, we count the number of BDD nodes strictly below the context attribute (*i.e.* those that test a bit of the pointer variable or abstract heap object attributes, as well as the two terminal nodes) that have one or more incoming edges from a BDD node testing a bit of the context attribute. This count is exactly the number of equivalence classes of contexts. To see why this is the case, note that each unique context-insensitive points-to relation is represented by a unique BDD node below the context attribute, and an incoming edge from a BDD node testing the context

$$
\begin{array}{ll}
\text{In context } P, & \textit{points-to}(a) = \{X, Y\} \\
& \textit{points-to}(b) = \{X, Y\} \\
& \textit{points-to}(c) = \{X, Y\} \\
& \textit{points-to}(d) = \{X, Y\} \\
\\
\text{In context } Q, & \textit{points-to}(a) = \{X, Y\} \\
& \textit{points-to}(b) = \{X, Y\} \\
& \textit{points-to}(c) = \{X, Y, Z\} \\
& \textit{points-to}(d) = \{\} \\
\\
\text{In context } R, & \textit{points-to}(a) = \{X, Y, Z\} \\
& \textit{points-to}(b) = \{\} \\
& \textit{points-to}(c) = \{X, Y, Z\} \\
& \textit{points-to}(d) = \{\} \\
\\
\text{In context } S, & \textit{points-to}(a) = \{X, Y, Z\} \\
& \textit{points-to}(b) = \{\} \\
& \textit{points-to}(c) = \{X, Y, Z\} \\
& \textit{points-to}(d) = \{\}
\end{array}
$$

Figure 5.1: Example context-sensitive points-to relation

attribute indicates that it is the points-to relation for one or more contexts.

To illustrate, let us return to our concrete example. The BDD representing the context-sensitive points-to relation from Figure 5.1 is shown in Figure 5.2. We have used the following assignment of bit strings to contexts, pointer variables, and abstract heap objects:

P 00	a 00	X 00
Q 01	b 01	Y 01
R 10	c 10	Z 10
S 11	d 11	

The top two levels of the BDD (nodes a and b) test the two bits of the context, the next two levels (nodes c and d) test the two bits of the pointer variable, and the final two levels (nodes e, f, and g) test the two bits of the abstract heap object. Recall that there are three equivalence classes, namely $\{(M, P)\}$, $\{(M, Q)\}$, and $\{(M, R), (M, S)\}$. As expected, in the BDD, there are also three nodes that satisfy the criterion, namely nodes c, d, and e. Each of these nodes tests a bit strictly below the context bits, and each has an incoming edge from a node testing a context bit. When the context is P (00), a traversal of the BDD goes through node e, which represents the context-insensitive points-to relation for the context P, namely every pointer having the points-to set $\{X, Y\}$. When the context is Q (01), a traversal of the BDD goes through node c, which represents the context-insensitive points-to relation for that context. Finally, when the context is either R or S, a traversal of the BDD goes through node d, which represents the context-insensitive points-to relation that is common to those two contexts and makes them equivalent, namely the points-to set $\{X, Y, Z\}$ for pointers a and c, and the empty points-to set for pointers b and d.

The measurements of the number of equivalence classes of contexts are shown in Table 5.3. Again, the "context-insensitive" columns show the actual number of equivalence classes of contexts, while the other columns give a multiple of the "context-insensitive OTF" number (*i.e.* the average number of equivalence classes per method).

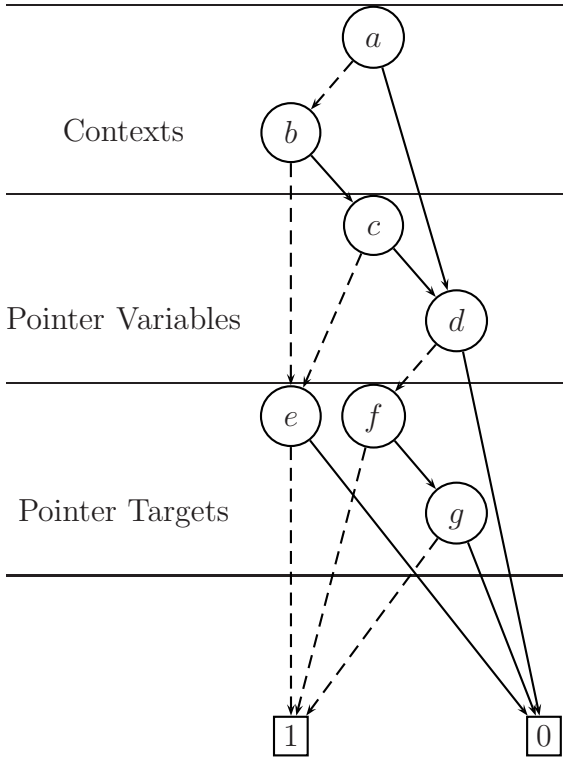


Figure 5.2: BDD for relation from Figure 5.1

	context-insens.		object-sensitive					call site			
Benchmark	AOT	OTF	1	2	3	1H	2U	1	2	1H	ZCWL
bh	3161	2584	8.3	9.3	10.3	11.9	11.2	2.4	3.9	4.7	3.3
bisort	3116	2542	8.4	9.4	10.4	12.0	11.3	2.4	3.9	4.7	3.3
em3d	3148	2553	8.3	9.4	10.4	12.0	11.3	2.4	3.9	4.7	3.3
health	3132	2557	8.3	9.4	10.4	12.0	11.3	2.4	3.9	4.7	3.3
mst	3133	2559	8.3	9.4	10.4	12.0	11.3	2.4	3.9	4.7	3.3
perimeter	3144	2569	8.3	9.4	10.3	11.9	11.2	2.4	3.9	4.7	3.3
power	3133	2559	8.3	9.4	10.3	12.0	11.2	2.4	3.9	4.7	3.3
treeadd	3109	2535	8.4	9.5	10.4	12.1	11.3	2.4	3.9	4.7	3.3
tsp	3141	2546	8.3	9.4	10.4	12.0	11.3	2.4	3.9	4.7	3.3
voronoi	3163	2590	8.3	9.4	10.3	11.9	11.2	2.4	3.9	4.7	3.3
compress	10990	2597	8.4	9.9	11.3	12.1	12.2	2.4	3.9	4.9	3.3
db	10994	2614	8.5	9.9	11.4	12.1	12.3	2.4	3.9	5.0	3.3
jack	11246	2870	8.6	10.2	11.6	11.9	12.3	2.4	3.9	5.0	3.4
javac	12121	3781	10.4	17.7	33.8	14.3	41.3	2.7	5.3	5.4	
jess	11621	3217	8.9	10.6	12.0	13.9	12.9	2.6	4.2	5.0	3.9
mpegaudio	11199	2794	8.1	9.4	10.8	11.5	11.7	2.4	3.8	4.8	3.3
mtrt	11116	2739	8.3	9.7	11.1	11.8	12.0	2.5	4.0	4.9	3.4
raytrace	11116	2739	8.3	9.7	11.1	11.8	12.0	2.5	4.0	4.9	3.4
soot-c	5503	4838	7.1	13.7	18.4	9.8	15.6	2.6	4.2	4.8	
sablecc-j	12817	5609	6.9	8.4	9.6	9.5	10.0	2.3	3.6	3.9	
polyglot	6205	5617	7.9	9.4	10.8	10.2	12.4	2.4	3.7	4.7	3.3
antlr	4494	3898	9.4	12.1	13.8	13.2	14.7	2.5	4.1	5.2	4.3
bloat	6497	5238	10.2	44.6		12.9		2.8	4.9	5.2	6.7
chart	14805	7070	10.0	17.4		18.2	21.4	2.7	4.8		
jython	5275	4402	9.9	55.9		15.6	60.0	2.5	4.3	4.6	4.0
pmd	8498	7220	7.6	14.6	17.0	11.0	19.2	2.4	4.2	4.2	
ps	11745	3875	8.7	9.9	11.0	12.0	16.0	2.6	4.0	5.2	4.4

Table 5.3: Number of equivalence classes of abstract contexts

The relatively small size of these numbers compared to the total numbers of contexts in Table 5.2 explains why a BDD can effectively represent the analysis information, since it automatically merges the representation of equal points-to relations, so each distinct relation is only represented once. If we had some idea before designing an analysis which abstract contexts are likely to be equivalent, we could define a new context abstraction in which these equivalent contexts are merged. That is, each equivalence class of old abstract contexts would be represented by a single new abstract context. If we had such a new context abstraction, the context-sensitive analysis could be implemented without the need for BDDs.

It is interesting to note that in the 1-, 2-, and 1H-object-sensitive analysis, the number of equivalence classes of contexts is generally about 3 times as high as in the corresponding 1-, 2-, and 1H-call site string analysis. This indicates that receiver objects better partition the space of concrete calling contexts that give rise to distinct points-to relations. That is, if at run time, the run-time points-to relation is different in two concrete calls to a method, it is more likely that the two calls will correspond to distinct abstract contexts if receiver objects rather than call sites are used as the context abstraction. This observation leads us to hypothesize that object-sensitive analysis should be more precise than the call site string analysis; we will see more direct measurements of precision in the upcoming sections.

In both object-sensitive and call site string analyses, making the context string longer increases the number of equivalence classes of contexts by only a small amount, while it increases the absolute number of contexts much more significantly. Therefore, increasing the length of the context string is unlikely to result in a large improvement in precision, but will significantly increase analysis cost.

In the two analysis variations in which abstract heap objects are modelled with context (1H-object-sensitive and 1H-call-site), the number of equivalence classes of contexts per method is slightly higher than in the analyses that model abstract heap objects context-insensitively. This is because the abstract heap objects in the points-to sets are annotated with abstract contexts, so it is more likely for two points-to relations to be distinct.

The 2-unique-object-sensitive analysis results in more equivalence classes of contexts than even the 3-object-sensitive analysis. This suggests that retaining a second distinct receiver object in the context string is more useful than retaining the receiver objects of even two additional calls for the purpose of distinguishing (in the abstract) contexts that have different run-time points-to sets.

It was initially rather surprising that in the analysis using the Zhu/Calman/Whaley/Lam algorithm, the number of equivalence classes of abstract contexts is so small, often even smaller than in the 2-call-site-sensitive analysis. The algorithm effectively performs a k-CFA analysis, where k is the maximum call depth in the original call graph; k is likely to be much higher than 2. The number of equivalence classes of contexts when using the Zhu/Calman/Whaley/Lam algorithm is small because the algorithm merges strongly connected components in the call graph, and models all call edges in each such component in a context-insensitive way. In contrast, the 2-call-site-sensitive analysis models all call edges context-sensitively, including those in strongly connected components. Indeed, a very large number of methods are part of some strongly connected component. The initial call graph for each of our benchmarks contains a large strongly-connected component of 1386 to 2926 methods, representing 36% to 53% of all methods in the call graph. In particular, this strongly-connected component always includes many methods for which context-sensitive analysis would be particularly useful, such as the methods of the `String` class and the standard collections classes. These methods are used extensively within the Java standard library, and contain many calls to each other. We examined this large strongly-connected component and found many distinct cycles; there was no single method that, if removed, would break the component. In summary, the reason for the surprisingly small number of equivalence classes of abstract contexts when using the Zhu/Calman/Whaley/Lam algorithm is that it models a large portion of the call graph context-insensitively.

5.3.3 Distinct points-to sets

Finally, we measure the number of distinct points-to sets that appear in the points-to analysis result. This number is an indication of how difficult it would be to efficiently represent the context-sensitive points-to sets in a non-BDD-based analysis implementation, assuming there was already a way to represent the contexts themselves. An increase in the number of distinct points-to sets also suggests an increase in precision, but the connection is very indirect [Hin01, Section 3.2]. We therefore present the number of distinct points-to sets primarily as a measure of analysis cost, and provide more direct measurements of the precision of clients of the analysis later in this chapter. In traditional, context-insensitive, subset-based points-to analyses, the representation of the points-to sets often makes up most of the memory requirements of the analysis. If the traditional analysis stores points-to sets using shared bit-vectors as suggested by Heintze [Hei99], each distinct points-to set need only be stored once. Therefore, the number of distinct points-to sets approximates the space requirements of such a traditional representation.

The number of distinct points-to sets can be measured on the points-to relation BDD using a technique similar to the one we used to measure the number of equivalent contexts. The BDD must first be arranged so that the attribute of abstract heap objects is assigned to the very bottom physical domain of the BDD. In particular, the physical domain assignment that we used to measure the number of equivalent contexts satisfies this requirement. The number of distinct points-to sets is then equal to the number of BDD nodes that test a bit of the abstract heap object attribute or are terminal nodes, and that have one or more incoming edges from a BDD node testing a bit strictly above the abstract heap object attribute. Each such node represents a unique points-to set, and for each points-to set in the points-to relation, there is such a node in the BDD.

To illustrate this, we return to our example points-to relation from Figure 5.1 and the BDD representing it from Figure 5.2. The points-to relation contains three distinct points-to sets, namely $\{\}$, $\{X, Y\}$, and $\{X, Y, Z\}$. As expected, the BDD contains three nodes satisfying the requirement, namely e, f, and the zero terminal

node. Note that node g does *not* satisfy the requirement because it does not have any incoming edges from a node strictly above the abstract heap objects attribute. The node e represents the points-to set $\{X, Y\}$. The node f represents the points-to set $\{X, Y, Z\}$. The zero terminal node represents the empty points-to set.

The measurements of the number of distinct points-to sets arising with each context abstraction are shown in Table 5.4. In this table, all numbers are the absolute count of distinct points-to sets, not multiples of the "context-insensitive" column.

The numbers of distinct points-to sets are fairly constant in most of the analysis variations, including object-sensitive analyses, call site string analyses, and the analysis using the Zhu/Calman/Whaley/Lam algorithm. Therefore, in a traditional points-to analysis implemented using shared bit-vectors, representing the individual points-to sets should not be a source of major difficulty even in a context-sensitive analysis. Future research in traditional implementations of context-sensitive analyses should therefore be directed at the problem of efficiently representing the contexts, rather than representing the points-to sets.

However, when abstract heap objects are modelled context-sensitively, the elements of each points-to set are pairs of abstract object and context, rather than simply abstract objects, and the number of distinct points-to sets increases about 11-fold. In addition, it is likely that the points-to sets themselves are significantly larger. Therefore, in order to implement such an analysis without using BDDs, it would be worthwhile to look for an efficient way to represent points-to sets of abstract objects with context.

5.4 Call Graph

We now turn our attention to the effect of context sensitivity on call graph construction. For the purposes of comparison, we have constructed context-sensitive call graphs, removed their contexts, and measured differences in their context-insensitive projections. We adopted this methodology because context-sensitive call graphs using different context abstractions are not directly comparable. Each node in the graph

	context-insens.		object-sensitive					call site			
Benchmark	AOT	OTF	1	2	3	1H	2U	1	2	1H	ZCWL
bh	3466	3192	3143	3230	3251	34239	4356	3229	3125	36784	3115
bisort	3410	3136	3101	3187	3206	34047	4309	3185	3084	36671	3071
em3d	3446	3152	3111	3200	3220	34080	4323	3199	3096	36708	3087
health	3434	3158	3120	3205	3224	34110	4327	3206	3103	36710	3089
mst	3419	3145	3112	3201	3221	34094	4329	3193	3091	36673	3080
perimeter	3422	3148	3109	3197	3217	34040	4326	3196	3094	36807	3083
power	3436	3162	3118	3206	3226	34072	4329	3206	3101	36699	3094
treeadd	3414	3140	3101	3189	3209	34017	4313	3188	3085	36673	3075
tsp	3434	3140	3100	3188	3208	34019	4311	3188	3085	36678	3073
voronoi	3440	3166	3124	3212	3232	34251	4341	3212	3108	36774	3112
compress	10574	3178	3150	3240	3261	34355	4371	3227	3125	38242	3139
db	10590	3197	3170	3261	3283	34637	4401	3239	3133	38375	3173
jack	10847	3441	3411	3507	3527	37432	4704	3497	3377	40955	3541
javac	11789	4346	4367	4579	4712	55196	16397	4424	4303	54866	
jess	11340	3834	4433	4498	4514	51452	6537	4589	4426	42614	4644
mpegaudio	11627	4228	4179	4272	4293	36563	5402	4264	4157	67565	4175
mtrt	10739	3349	3287	3377	3396	35154	4531	3387	3263	38758	3282
raytrace	10739	3349	3287	3377	3396	35154	4531	3387	3263	38758	3282
soot-c	4937	4683	4565	4670	4657	45974	7400	4722	4550	52937	
sablecc-j	12308	5753	5777	5895	5907	52993	7893	5875	5694	59748	
polyglot	5774	5591	5556	5829	5925	50587	8120	5682	5516	59837	5575
antlr	4792	4520	5259	5388	5448	54942	7388	4624	4535	54176	4901
bloat	5853	5337	5480	5815		55309		5452	5342	49230	6658
chart	16136	9608	9914	10168		233723	16610	9755	9520		
jython	5128	4669	5111	5720		74297	21546	4968	4857	46280	8587
pmd	7980	7368	7679	7832	7930	94403	16706	7671	7502	103990	
ps	11796	4610	4504	4639	4672	47244	22670	4656	4521	58513	4802

Table 5.4: Total number of distinct points-to sets in points-to analysis results

represents a pair of method and abstract context, but the set of possible abstract contexts is different in each context variation. In the context-insensitive projection, each node is simply a method, so the projections are directly comparable. Projecting away context discards some information from the call graph, but only the information which is not directly comparable between different context abstractions. In particular, the context-insensitive projection preserves the set of methods reachable from the program entry points, as well as the set of possible targets of each call site in the program; it is these sets that we measure. The set of reachable methods is particularly important because any conservative interprocedural analysis must analyze all of these methods, so a small set of reachable methods reduces the cost of other interprocedural analyses.

In our study of call graph construction, it does not make sense to include the Zhu/Calman/Whaley/Lam algorithm because the context-sensitive call graph it produces is only as precise as the original context-insensitive call graph that it is given as input. That is, the context-insensitive projection of the context-sensitive call graph produced by the Zhu/Calman/Whaley/Lam algorithm is identical to the context-insensitive OTF call graph that we used as input.

5.4.1 Reachable methods

Table 5.5 shows the number of methods reachable from the program entry points when constructing the call graph using different variations of context sensitivity, excluding methods from the standard Java library. In Table 5.5 and all subsequent tables in this chapter, the most precise entry for each benchmark has been highlighted in bold. In the case of a tie, the most precise entry that is least expensive to compute has been highlighted.

For most of the benchmarks, the call graph generated with OTF context-insensitive points-to analysis is much more precise (smaller) than one generated with Class Hierarchy Analysis (in the AOT column). Any further improvements due to context sensitivity are relatively small. The JOlden benchmarks are so simple that

	context-insens.		object-sensitive					call site			actually
Benchmark	AOT	OTF	1	2	3	1H	2U	1	2	1H	executed
bh	60	**57**	57	57	57	57	57	57	57	57	54
bisort	**14**	14	14	14	14	14	14	14	14	14	12
em3d	**21**	21	21	21	21	21	21	21	21	21	18
health	28	**27**	27	27	27	27	27	27	27	27	26
mst	**32**	32	32	32	32	32	32	32	32	32	31
perimeter	44	**43**	43	43	43	43	43	43	43	43	42
power	**31**	31	31	31	31	31	31	31	31	31	29
treeadd	**6**	6	6	6	6	6	6	6	6	6	5
tsp	**15**	15	15	15	15	15	15	15	15	15	12
voronoi	61	**60**	60	60	60	60	60	60	60	60	44
compress	90	**59**	59	59	59	59	59	59	59	59	56
db	95	65	**64**	64	64	64	64	65	64	65	51
jack	348	317	**313**	313	313	313	313	316	313	316	291
javac	1185	1154	**1147**	1147	1147	1147	1147	1147	1147	1147	778
jess	683	630	629	629	629	**623**	629	629	629	629	395
mpegaudio	306	255	**251**	251	251	251	251	251	251	251	222
mtrt	217	189	**186**	186	186	186	186	187	187	187	182
raytrace	217	189	**186**	186	186	186	186	187	187	187	180
soot-c	2395	2273	**2264**	2264	2264	2264	2264	2266	2264	2266	1055
sablecc-j	1904	1744	1744	1744	1744	**1731**	1744	1744	1744	1744	1034
polyglot	2540	2421	2419	2419	2419	**2416**	2419	2419	2419	2419	2037
antlr	1374	**1323**	1323	1323	1323	1323	1323	1323	1323	1323	1099
bloat	2879	2464	**2451**	2451		2451		2451	2451	2451	138
chart	3227	2081	2080	2080		**2031**	2080	2080	2080		854
jython	2007	1695	1693	1693		**1683**	1694	1694	1693	1694	1004
pmd	4997	4528	4521	4521	4521	**4509**	4521	4521	4521	4521	1817
ps	840	835	835	835	835	**834**	835	835	835	835	285

Table 5.5: Number of reachable benchmark (non-library) methods in call graph

a context-insensitive OTF analysis already generates a call graph that is almost perfectly precise. These call graphs are not much larger than the number of methods actually executed during a run of the benchmark, shown in the right-most column.[3]

For most of the more significant benchmarks, call graph construction benefits slightly from 1-object sensitivity. The largest difference is 13 methods, in the bloat benchmark. All of these methods are visit methods in an implementation of the visitor design pattern, in the class `AscendVisitor`. This class traverses a parse tree from a starting node upwards toward the root of the tree, visiting each node along the way. Some kinds of nodes have no descendants that are ever the starting node of a traversal, so the visit methods of these nodes can never be called. However, in order to prove this, an analysis must analyze the visitor dispatch method context-sensitively in order to keep track of the kind of node from which it was called. Therefore, a context-insensitive analysis fails to show that these visit methods are unreachable.

In jess, sablecc-j, polyglot, chart, jython, pmd, and ps, modelling abstract heap objects object-sensitively further improves the precision of the call graph. In the sablecc-j benchmark, an additional 13 methods are proved unreachable. The benchmark includes its own implementation of maps similar to those in the Java standard library. The maps are instantiated in a number of places, and different kinds of objects are placed in the different maps. Methods such as `toString()` and `equals()` are called on some of the maps but not others. As a result, `toString()` and `equals()` are called on some of the objects placed in the maps, but not on others. However, the objects stored in every map are placed in map entry objects, which are allocated at a single point in the map code. When abstract heap objects are modelled without context, all map entries are modelled by a single abstract object, and the contents of all maps are conflated. When abstract heap objects are modelled with context, the map entries are treated as separate objects depending on which map they were created for. Note

[3]The perfectly precise call graph would contain the union of all methods and call edges executed when the program is run on all inputs. The static call graphs overestimate the perfect call graph, while the dynamic call graphs underestimate it (because they are observed while running the program on only one input). For example, although we do not know the perfect call graph for the bh benchmark, we know that it must contain between 54 and 57 non-library methods. Therefore, we know that the OTF call graph, with 57 non-library methods, is not much bigger than the perfect call graph.

that successfully distinguishing the map entries requires receiver objects to be used as context, rather than call site strings. The code that allocates a new entry is in a method that is always called from the same call site, in another method of the map class. In general, although modelling abstract heap objects with context improved the call graph for some benchmarks in an object-sensitive analysis, it never made any difference in analyses using call site strings as the context abstraction (*i.e.* the 1-call-site and 1H-call-site columns are the same).

Overall, object-sensitive analysis results in slightly smaller call graphs than call site string analysis. The 1-object-sensitive call graph is never larger than the 1-call-site-sensitive call graph, and it is smaller on db, jack, mtrt, raytrace, soot-c, and jython. On the db, jack, and jython benchmarks, the call-site-sensitive call graph can be made as small as the 1-object-sensitive call graph, but it requires 2-call-site rather than 1-call-site analysis.

The cost of client interprocedural analyses depends on the number of methods in the whole call graph, not just the subset excluding the Java standard library. The number of methods in the whole call graph is shown in Table 5.6. All variations of using points-to analysis to construct the call graph result in a much smaller call graph than when using CHA, and therefore are likely to speed up client interprocedural analyses. However, compared to a context-insensitive points-to analysis, the various context-sensitive analyses have little effect on the overall size of the call graph.

Notice that even the most precise context-sensitive analyses produce a call graph much bigger than the set of methods actually executed, shown in the rightmost column of the table. This difference is due not to remaining imprecision in the static call graph construction, but to limited coverage by the benchmarks of rarely-used features of the standard Java library. For example, one cause of a large number of methods in the static call graphs is Java's Jar File signing feature. The Jar Files containing classes to be executed may be cryptographically signed. If they are, the Java VM automatically loads and runs a large amount of cryptography code to verify the signatures. Since it is possible for the the cryptography code to run, it must be included in any conservative call graph. However, none of the runs of any of our benchmarks actually run the cryptography code, because their Jar Files are not signed.

	context-insens.		object-sensitive					call site			actually
Benchmark	AOT	OTF	1	2	3	1H	2U	1	2	1H	executed
bh	3425	2694	2643	2643	2643	**2608**	2645	2647	2643	2647	459
bisort	3377	2649	2598	2598	2598	**2563**	2600	2602	2598	2602	414
em3d	3407	2657	2606	2606	2606	**2571**	2608	2610	2606	2610	425
health	3391	2662	2611	2611	2611	**2576**	2613	2615	2611	2615	435
mst	3393	2665	2614	2614	2614	**2579**	2616	2618	2614	2618	434
perimeter	3405	2676	2625	2625	2625	**2590**	2627	2629	2625	2629	443
power	3394	2666	2615	2615	2615	**2580**	2617	2619	2615	2619	427
treeadd	3367	2639	2588	2588	2588	**2553**	2590	2592	2588	2592	407
tsp	3402	2652	2601	2601	2601	**2566**	2603	2605	2601	2605	404
voronoi	3427	2701	2649	2649	2649	**2614**	2651	2653	2649	2653	450
compress	11492	2706	2655	2655	2655	**2620**	2657	2659	2655	2659	463
db	11499	2725	2671	2671	2671	**2636**	2673	2678	2671	2678	483
jack	11750	2980	2943	2943	2943	**2885**	2945	2962	2943	2962	739
javac	12627	3900	3832	3832	3832	**3797**	3834	3841	3834	3841	1283
jess	12130	3338	3285	3285	3285	**3244**	3287	3292	3286	3292	846
mpegaudio	11706	2911	2850	2850	2850	**2815**	2852	2853	2850	2853	637
mtrt	11621	2858	2801	2801	2801	**2766**	2803	2809	2802	2809	616
raytrace	11621	2858	2801	2801	2801	**2766**	2803	2809	2802	2809	611
soot-c	5789	4964	4922	4922	4922	**4873**	4924	4938	4922	4938	1549
sablecc-j	13306	5776	5686	5686	5686	**5622**	5688	5713	5695	5713	1856
polyglot	6441	5737	5679	5679	5679	**5636**	5681	5703	5680	5703	3093
antlr	4775	4006	3950	3950	3950	**3915**	3952	3958	3950	3958	1783
bloat	6824	5375	5325	5325		**5271**		5341	5326	5341	1010
chart	15488	7302	7228	7228		**7137**	7238	7252	7235		2790
jython	5587	4542	4503	4503		**4442**	4509	4525	4503	4525	1858
pmd	8905	7371	7328	7328	7328	**7285**	7330	7344	7330	7344	2581
ps	12240	4017	3944	3944	3944	**3904**	3946	3987	3945	3987	945

Table 5.6: Total number of reachable methods in call graph

5.4.2 Call edges

Table 5.7 shows the size of the call graph in terms of call edges rather than reachable methods. Only call edges originating from a benchmark (non-library) method are counted.

In general, context sensitivity makes little difference to the size of the call graph when measured this way, with one major exception. In the sablecc-j benchmark, the number of call edges is 17925 in a context-insensitive analysis, but only 5175 in a 1-object-sensitive analysis. This could make a significant difference to the cost of a client analysis whose complexity depends on the number of edges in the call graph. The large difference is caused by the following pattern of code. The sablecc-j benchmark contains code to represent a parse tree, with many different kinds of nodes. Each kind of node implements a method called `removeChild()`. The code contains a large number of calls of the form `this.getParent().removeChild(this)`. In a context-insensitive analysis, `getParent()` is found to possibly return any of hundreds of possible kinds of nodes. Therefore, each of these many calls to `removeChild(this)` results in hundreds of call graph edges. However, in a context-sensitive analysis, `getParent()` is analyzed in the context of the `this` pointer. For each kind of node, there is a relatively small number of kinds of nodes that can be its parent. Therefore, in a given context, `getParent()` is found to return only a small number of kinds of parent node, and therefore each call site of `removeChild()` adds only a small number of edges to the call graph.

5.5 Virtual Call Resolution

Table 5.8 shows the number of virtual call sites for which the call graph contains more than one potential target method. Call sites with at most one potential target method can be converted to cheaper static instead of virtual calls, and they can be inlined, possibly enabling many other optimizations. Therefore, an analysis that proves that call sites are not polymorphic can be used to significantly improve run-time performance.

	context-insens.		object-sensitive					call site			actually
Benchmark	AOT	OTF	1	2	3	1H	2U	1	2	1H	executed
bh	354	**187**	187	187	187	187	187	187	187	187	129
bisort	**57**	57	57	57	57	57	57	57	57	57	30
em3d	162	**78**	78	78	78	78	78	78	78	78	43
health	234	**90**	90	90	90	90	90	90	90	90	70
mst	177	**82**	82	82	82	82	82	82	82	82	50
perimeter	113	**105**	105	105	105	105	105	105	105	105	65
power	**108**	108	108	108	108	108	108	108	108	108	54
treeadd	**32**	32	32	32	32	32	32	32	32	32	19
tsp	**60**	60	60	60	60	60	60	60	60	60	36
voronoi	206	**201**	201	201	201	201	201	201	201	201	92
compress	456	**270**	270	270	270	270	270	270	270	270	118
db	940	434	**427**	427	427	427	427	434	427	434	184
jack	1936	1283	1251	1251	1251	**1250**	1251	1276	1251	1276	833
javac	13146	10360	**10296**	10296	10296	10296	10318	10318	10301	10318	2928
jess	4700	3626	3618	3618	3618	**3571**	3618	3618	3618	3618	919
mpegaudio	1182	858	**812**	812	812	812	812	812	812	812	400
mtrt	925	761	**739**	739	739	739	739	746	746	746	484
raytrace	925	761	**739**	739	739	739	739	746	746	746	478
soot-c	20079	14611	14112	14112	14112	**13868**	14160	14185	14112	14185	2860
sablecc-j	24283	17925	5175	5140	5140	**5072**	5140	5182	5140	5182	2326
polyglot	19898	11768	11564	11564	11564	**11374**	11537	11566	11566	11566	5440
antlr	10769	**9553**	9553	9553	9553	9553	9553	9553	9553	9553	4196
bloat	36863	18586	18143	18143		**17722**		18166	18143	18166	477
chart	24978	9526	9443	9443		**9178**	9443	9443	9443		2166
jython	13679	9382	9367	9367		**9307**	9374	9367	9365	9367	2898
pmd	29401	18785	18582	18582	18580	**18263**	18587	18601	18599	18601	3879
ps	13610	11338	11292	11292	11292	**10451**	11298	11298	11292	11298	705

Table 5.7: Total number of call edges in call graph originating from a benchmark (non-library) method

	context-insens.		object-sensitive					call site		
Benchmark	AOT	OTF	1	2	3	1H	2U	1	2	1H
bh	17	**7**	7	7	7	7	7	7	7	7
bisort	**0**	0	0	0	0	0	0	0	0	0
em3d	20	**0**	0	0	0	0	0	0	0	0
health	10	**0**	0	0	0	0	0	0	0	0
mst	1	**0**	0	0	0	0	0	0	0	0
perimeter	**16**	16	16	16	16	16	16	16	16	16
power	**0**	0	0	0	0	0	0	0	0	0
treeadd	**0**	0	0	0	0	0	0	0	0	0
tsp	**0**	0	0	0	0	0	0	0	0	0
voronoi	2	**0**	0	0	0	0	0	0	0	0
compress	16	**3**	3	3	3	3	3	3	3	3
db	36	5	**4**	4	4	4	4	5	4	5
jack	474	25	23	23	23	**22**	23	24	23	24
javac	908	737	**720**	720	720	720	720	720	720	720
jess	121	**45**	45	45	45	45	45	45	45	45
mpegaudio	40	27	**24**	24	24	24	24	24	24	24
mtrt	20	9	**7**	7	7	7	7	8	8	8
raytrace	20	9	**7**	7	7	7	7	8	8	8
soot-c	1748	983	**913**	913	913	913	913	938	913	938
sablecc-j	722	450	325	325	325	**301**	325	380	325	380
polyglot	1332	744	592	592	592	**585**	592	592	592	592
antlr	1086	**843**	843	843	843	843	843	843	843	843
bloat	2503	1079	962	962		**961**		962	962	962
chart	2782	254	235	235		**214**	235	235	235	
jython	646	347	347	347		**346**	347	347	347	347
pmd	2868	1224	1193	1193	1193	**1163**	1193	1205	1205	1205
ps	321	304	303	303	303	**300**	303	303	303	303

Table 5.8: Total number of potentially polymorphic call sites in benchmark (non-library) code

For all but two of the JOlden benchmarks, even a call graph based on context-insensitive points-to analysis is sufficient to devirtualize all the calls. Note, however, that Class Hierarchy Analysis is insufficient to devirtualize many of these call sites.

In the benchmarks written in an object-oriented style, notably javac, soot-c, sablecc-j, polyglot, bloat, and pmd, a further significant number of call sites can be devirtualized using object-sensitive analysis (compared to context-insensitive analysis). In some cases, call site string analysis gives the same improvement, but never any more, and in the case of soot-c and sablecc-j, the improvement from 1-object-sensitive analysis is much greater than from 1-call-site string analysis.

In sablecc-j, there are three key sets of call sites that can be devirtualized using context-sensitive analysis. Any context-sensitive analysis is sufficient to devirtualize the first set of call sites. Devirtualization of the second set of call sites requires an object-sensitive analysis; an analysis using call sites as the context abstraction cannot prove them to be monomorphic. Devirtualization of the third set of call sites not only requires an object-sensitive analysis, but it also requires that abstract heap objects be modelled with context.

The first set of call sites contains the calls to the `removeChild()` method mentioned in Section 5.4.2. Object sensitivity reduces the number of potential target methods at each of these call sites. At many of them, it reduces the number down to one, so the calls can be devirtualized. The same improvement can be obtained with call site string context sensitivity.

The second set of call sites are calls to methods of iterators over lists. The sablecc-j benchmark contains several implementations of lists similar to those in the standard Java library. A call to `iterator()` on any of these lists invokes `iterator()` on the `AbstractList` superclass, which in turn invokes the `listIterator()` method specific to each list. The actual kind of iterator that is returned depends on which `listIterator()` was invoked, which in turn depends on the receiver object of the call to `iterator()`; it is independent of the call site of `listIterator()`, which is always the same site in `iterator()`. Therefore, calls to `hasNext()` and `next()` on the returned iterator can be devirtualized only with an object-sensitive analysis.

The third set of call sites are calls to methods such as `toString()` and `equals()`

on objects stored in maps. As we explained in Section 5.4.1, object-sensitive modelling of abstract heap objects is required distinguish the internal map entry objects in each separate use of the map implementation. The map entry objects must be distinguished in order to distinguish the objects that are stored in the maps. Therefore, devirtualization of these calls to methods of objects stored in maps requires an object-sensitive analysis that models abstract heap objects with context.

5.6 Cast Safety

We evaluated the precision of the cast safety analysis implemented in PADDLE and described in Section 4.4.2 on the different variations of context sensitivity. Table 5.9 shows the number of casts in each benchmark that cannot be statically proven safe by the cast safety analysis.

Context sensitivity improves precision of the cast safety analysis in the bh, jack, javac, mpegaudio, mtrt, raytrace, soot-c, sablecc-j, polyglot, antlr, bloat, chart, jython, pmd, and ps benchmarks. Object sensitive cast safety analysis is never less precise and often significantly more precise than the call site string context sensitive variations. The improvements due to context sensitivity are most significant in the polyglot and javac benchmarks. In voronoi, db, jack, javac, jess, soot-c, sablecc-j, polyglot, antlr, bloat, chart, jython, pmd, and ps, modelling abstract heap objects with receiver object context further improves precision of cast safety analysis.

The polyglot benchmark contains a hierarchy of classes representing different kinds of nodes in an abstract syntax tree. At the root of this hierarchy is the `Node_c` class. This class implements a method called `copy()` which, like the `clone()` method of `java.lang.Object`, returns a copy of the node on which it is called. In fact, the `copy()` method first uses `clone()` to create the copy of the node, and then performs some additional processing on it. The static return type of the `copy()` method is `java.lang.Object`, but at most sites calling it, the returned value is immediately cast to the static type of the node on which it is called. In the PADDLE framework, the `clone()` method is modelled as returning its receiver; that is, the original object and

	context-insens.		object-sensitive					call site			
Benchmark	AOT	OTF	1	2	3	1H	2U	1	2	1H	ZCWL
bh	14	9	**8**	8	8	8	8	8	8	8	8
bisort	**4**	4	4	4	4	4	4	4	4	4	4
em3d	13	**3**	3	3	3	3	3	3	3	3	3
health	19	**14**	14	14	14	14	14	14	14	14	14
mst	**3**	3	3	3	3	3	3	3	3	3	3
perimeter	**3**	3	3	3	3	3	3	3	3	3	3
power	**5**	5	5	5	5	5	5	5	5	5	5
treeadd	**3**	3	3	3	3	3	3	3	3	3	3
tsp	**3**	3	3	3	3	3	3	3	3	3	3
voronoi	7	7	7	7	7	**6**	7	7	7	7	7
compress	25	**18**	18	18	18	18	18	18	18	18	18
db	32	27	27	27	27	**21**	27	27	27	27	27
jack	151	146	145	145	145	**104**	145	146	145	146	146
javac	412	405	370	370	370	**363**	389	391	370	391	
jess	137	130	130	130	130	**86**	130	130	130	130	130
mpegaudio	56	42	**38**	38	38	38	38	40	40	40	42
mtrt	36	31	**27**	27	27	27	27	27	27	27	29
raytrace	36	31	**27**	27	27	27	27	27	27	27	29
soot-c	972	955	932	932	932	**878**	933	932	932	932	
sablecc-j	385	375	369	369	369	**331**	369	370	370	370	
polyglot	3583	3539	3307	3306	3306	**1017**	3314	3526	3443	3526	3318
antlr	296	295	275	275	275	**237**	275	276	275	276	276
bloat	1355	1241	1207	1207		**1160**		1233	1207	1233	1234
chart	1979	1097	1086	1085		**934**	1086	1070	1070		
jython	599	501	499	499		**471**	499	499	499	499	499
pmd	1477	1427	1376	1375	1375	**1300**	1394	1393	1391	1393	
ps	692	641	612	612	612	**421**	631	612	612	612	612

Table 5.9: Number of casts potentially failing at run time

the cloned version are represented in PADDLE by the same abstract object. Therefore, given a program that calls `clone()` directly, the cast safety analysis in PADDLE correctly determines that the run-time type of the clone is the same as that of the original. However, in polyglot, the call to `clone()` is wrapped inside `copy()`, and the casts appear at sites calling `copy()`. When `copy()` is analyzed in a context-insensitive way, it is deemed to possibly return any of the objects on which it is called throughout the program, so the casts cannot be proven to succeed. In an object-sensitive analysis, however, `copy()` is analyzed separately in the context of each receiver object on which it is called, and in each such context, it returns only an object of the same type as that receiver object. Therefore, the cast safety analysis proves statically that the casts of the return value of `copy()` cannot fail.

The number of potentially failing casts in the polyglot benchmark decreases significantly between the 1-object-sensitive and 1H-object-sensitive columns of Table 5.9, from 3307 to 1017. The vast majority of these casts are in the parser generated by JavaCUP [Ana]. Specifically, the parser uses a `java.util.Stack` as the LR parse stack. Each object popped from the stack is cast to a `Symbol`. The generated polyglot parser contains about 2000 of these casts. The `java.util.Stack` class extends `java.util.Vector`, which uses an internal `elementData` array to store the objects that have been pushed onto the stack. In order to prove the safety of the casts, the analysis must distinguish the array implementing the parse stack from the arrays of other uses of `java.util.Vector` in the program. Since the array is allocated in one place, inside the `java.util.Vector` class, the different array instances can only be distinguished if abstract heap objects are modelled with context. Therefore, modelling abstract heap objects with object sensitivity is necessary to statically prove that these 2000 casts cannot fail.

5.7 Related Work

We refer the reader to Section 4.1 of Chapter 4, in which we presented related work on context-sensitive points-to analysis and call graph construction. In this section,

we specifically compare our empirical study of the effects of context sensitivity to related experiments performed by others.

The work most closely related to our empirical evaluation of context-sensitive interprocedural analyses for Java is the pioneering work on object-sensitive analysis by Milanova, Rountev, and Ryder [MRR05, MRR02]. They implemented a limited form of object sensitivity within their points-to analysis framework based on annotated constraints [RMR01] and built on top of the BANE toolkit [AFFS98]. In particular, they selected a subset of pointer variables (method parameters, the `this` pointer, and the method return value) which they modelled context-sensitively using the receiver object as the context abstraction. All other pointer variables and all abstract heap objects were modelled in a context-insensitive way. The precision of the analysis was evaluated on benchmarks using version 1.1.8 of the Java standard library, and compared to a context-insensitive and to a call site context-sensitive analysis, using call graph construction, virtual call resolution, and mod-ref analysis as client analyses. Our BDD-based implementation of object-sensitive analysis has made it feasible to evaluate it on benchmarks using the much larger version 1.3.1_01 of the Java standard library. Thanks to the better scalability of the BDD-based implementation, we have performed a much broader empirical exploration of the design space of object-sensitive analyses. In particular, we have modelled all pointer variables context-sensitively, rather than only a subset, we have used receiver object strings of length up to three, rather than only one, and we have modelled abstract heap objects context-sensitively.

Whaley and Lam [WL04] suggest several client analyses of the Zhu/Calman/Whaley/Lam algorithm, but state that "[a]n in-depth analysis of the accuracy of the analyses ... is beyond the scope of this paper." They do, however, provide some preliminary data about thread escape analysis and "type refinement analysis", an analysis for finding variables whose declared type could be made more specific. In this chapter, we have compared the precision of the Zhu/Calman/Whaley/Lam algorithm against object-sensitive and call site string context-sensitive analyses using several client analyses, namely call graph construction, virtual call resolution, and cast safety analysis.

5.8 Conclusions

We have performed an in-depth empirical study of the effects of variations of context sensitivity on the precision of call graph construction, points-to analysis, and related client analyses. In particular, we studied five variations of object-sensitive analysis, three variations of context-sensitive analysis using call sites as the context abstraction, and the Zhu/Calman/Whaley/Lam algorithm. We evaluated the effects of these variations of context sensitivity on the number of contexts generated, the number of distinct points-to sets constructed, and on the precision of call graph construction, virtual call resolution, and cast safety analysis. We performed our experiments on a collection of 27 Java benchmarks.

Overall, we found that context sensitivity improved call graph precision by a small amount, improved the precision of virtual call resolution by a more significant amount, and enabled a major precision improvement in cast safety analysis.

Object-sensitive analysis was clearly better than the other variations of context sensitivity that we studied, both in terms of analysis precision and scalability. Client analyses based on object-sensitive analyses were never less precise than those based on call site string context-sensitive analyses or on the Zhu/Calman/Whaley/Lam algorithm, and in many cases, they were significantly more precise. As we increased the length of context strings, the number of abstract contexts produced with object-sensitive analysis grew much more slowly than with the other variations of context sensitivity, so object-sensitive analysis scaled better. However, the number of equivalence classes of contexts was greater with object sensitivity than with the other variations, which indicates that object sensitivity better distinguishes contexts that give rise to differences in points-to sets.

Of the object-sensitive variations, extending the length of context strings caused very few additional improvements in analysis precision compared to 1-object-sensitive analysis. However, modelling abstract heap objects with context did improve precision significantly in many cases. Therefore, we conclude that 1-object-sensitive and 1H-object-sensitive analyses provide the best tradeoffs between precision and analysis

efficiency. Our measurements of the numbers of abstract contexts and distinct points-to sets suggest that it should be feasible to implement an efficient non-BDD-based 1-object-sensitive analysis using current implementation techniques such as shared bit vectors. Efficiently implementing a 1H-object-sensitive analysis without BDDs will require new improvements in the data structures and algorithms used to implement points-to analyses, and we expect that our results will motivate and help guide this future research.

Although the Zhu/Calman/Whaley/Lam algorithm constructs call site strings of arbitrary length, we observed that client analyses based on it were never more precise than those based on object-sensitive analysis. In many cases, analyses based on the Zhu/Calman/Whaley/Lam algorithm were even less precise than those based on 1-call-site-sensitive analysis. We found that the key cause of the disappointing results of this algorithm was its context-insensitive treatment of calls within strongly connected components of the initial call graph — a large proportion of call edges were indeed within such strongly connected components.

Chapter 6
Analyses for AspectJ

In this chapter, we use Jedd and Paddle to implement a novel analysis and optimization for the aspect-oriented programming language AspectJ. In many cases, our analysis completely eliminates the run-time overhead of the *cflow* construct, which has been measured [DGH+04] to be very significant in programs that use *cflow*. Our analysis is implemented in the `abc` [abc, ACH+05a] compiler, which is based on Soot [VRGH+00]. The analysis is written in Jedd, and makes use of a call graph constructed using Paddle.

We first provide some background about aspect-oriented programming, AspectJ, and the `abc` compiler in Section 6.1. Next, in Section 6.2, we present the *cflow* analysis and optimization, and provide experimental results in Section 6.3. We present related work in Section 6.4. Finally, we conclude and suggest an area of future work in Section 6.5.

6.1 Background

6.1.1 AspectJ background

The purpose of aspect-oriented programming [KLM+97] is to improve modularity by separating the implementation of cross-cutting concerns from other parts of the program. For example, in a non-aspect-oriented program, logging code is typically

spread out in each method whose actions are to be logged. In an aspect-oriented program, however, the logging code could be consolidated into a separate logging aspect, which would contain a declarative specification of the places in which it should apply.

An ***aspect*** is a unit of code, much like a class, intended to encapsulate a concern. The key components of an aspect are ***pointcuts*** coupled with ***advice***. Each pointcut is a predicate on ***joinpoints***, which are certain intervals in the dynamic execution trace of the program. Advice is the code to be executed before, after, or instead of each joinpoint matching the pointcut expression. A joinpoint ***shadow*** is the static projection of a pointcut. That is, a shadow consists of a consecutive region of one or more instructions in the code, and each joinpoint is the dynamic interval spanning the time in which these instructions are executed. Note that if the shadow contains an instruction that invokes a method, the execution of the invoked method is included in the joinpoint, although the instructions of the called method are not generally part of the shadow.

AspectJ is a popular aspect-oriented extension of the Java programming language. Two compilers for AspectJ currently exist: the `ajc` [ajc] compiler formerly developed at Xerox PARC and currently managed under the Eclipse project, and `abc` [abc, ACH+05a], a joint project of the Sable Research Group at McGill University and the Programming Tools Group at the University of Oxford. The analyses and optimizations discussed in this chapter have been implemented in the `abc` compiler.

In AspectJ, each pointcut is specified by an expression consisting of ***pointcut designators*** combined using boolean operators. Each pointcut designator expresses a desired property of the joinpoints to be matched. The AspectJ language defines about 17 pointcut designators[1] for expressing both static and dynamic properties of joinpoints. For example, the ***call*** pointcut designator specifies a pattern of method signatures, and matches all instructions invoking methods whose signature matches the pattern. A pointcut designator p is ***static*** if there is a set of shadows such that a joinpoint matches p if and only if its shadow is in the set. A ***dynamic*** pointcut

[1]The AspectJ language is still in development, so the number of pointcut designators changes from version to version.

designator may or may not match joinpoints at a given shadow depending on run-time conditions. To implement static pointcut designators, an AspectJ compiler computes, at compile-time, the set of shadows that it matches. Implementing dynamic pointcut designators requires the compiler to insert a ***dynamic residue*** into the generated code to test whether the run-time conditions required for the pointcut designator to match have been satisfied. The overall motivation of our work is to eliminate these dynamic residues where possible to reduce the overhead of aspect-oriented programming. We perform analyses to find instances of pointcut designators which are dynamic in general, but static in the specific instance.

In this chapter, we focus particularly on the ***cflow*** pointcut designator, along with the related designator ***cflowbelow***. Each of these pointcut designators takes a pointcut p as an argument, and matches all joinpoints contained within a joinpoint matching p. Recall that a joinpoint is an interval in the execution trace of the program; the pointcut *cflow*(p) matches the joinpoint j if the interval of j is included in the interval of some joinpoint j' matched by the pointcut p. The difference between *cflowbelow* and *cflow* is that *cflowbelow* requires the interval containment to be strict, whereas *cflow*(p) also matches every joinpoint matched by p.

We will use the code in Figure 6.1 as a running example to illustrate the *cflow* pointcut designator and our *cflow* analysis. Since pointcuts are predicates on join points, which are intervals in the dynamic execution trace, we show the dynamic execution trace for the example code in Figure 6.2, on the right side. The trace records all events during the execution of the code. To avoid cluttering the trace, in Figure 6.2, we have shown only two kinds of events: the beginning and end of each method call. To the left of the trace, we have delimited eight join points (intervals in the trace), each corresponding to a method execution.

We will now present three sample pointcuts, and explain which join points in the example trace they match. The pointcut `call(C.c())` matches all calls to the method `C.c()`. Therefore, in the example trace, it matches the join points numbered 4 and 6. The pointcut `cflow(call(C.c()))` matches all join points *inclusively* nested within a join point matching `call(C.c())`. These are the join points numbered 4 and 5, because they are nested within join point 4, and the join points 6 and 7, because they

```
1  class C {
2      void main() {
3          a(); // update shadow 1
4          c(); // update shadow 2
5          d(); // update shadow 3
6          z(); // query shadow 1
7      }
8      void a() {
9          b();
10     }
11     void b() {
12         c(); // update shadow 4
13     }
14     void c() {
15         z(); // query shadow 2
16     }
17     void d() {
18         z(); // query shadow 3
19     }
20     void z() {
21     }
22 }
```

Figure 6.1: Base code for AspectJ *cflow* example

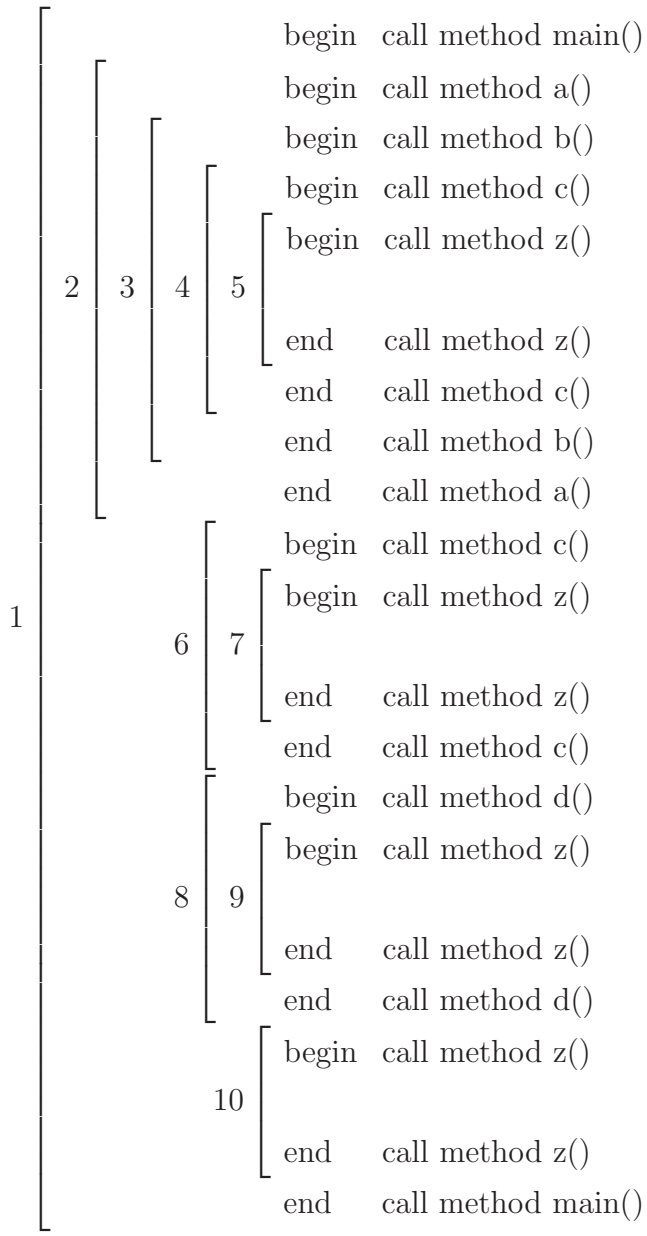


Figure 6.2: Dynamic trace of method call join points

are nested within join point 6. The pointcut `cflowbelow(call(C.c()))` matches all join points *strictly* nested within a join point matching `call(C.c())`. These are the join point numbered 5, because it is strictly nested within join point 4, and the join point numbered 7, because it is strictly nested within join point 6.

In the `ajc` and `abc` compilers, *cflow*(p) is implemented by inserting two kinds of code sequences into the generated code. Before and after each shadow that may match p at runtime, code is inserted to record that a joinpoint matching p has been entered and exited. If the pointcut p is dynamic, this code is made conditional on the dynamic residue of p. We call this shadow an ***update shadow***. At each shadow at which *cflow*(p) is to be tested, code is inserted to test whether the execution is currently inside a joinpoint matching p. We call a shadow at which *cflow*(p) is tested a ***query shadow***.

We illustrate this using the slightly more complicated pointcut

```
1    call(void C.z())
2 && cflow(call(void C.a()) || call(void C.c()) || call(void C.d()))
```

which will be our running example for the remainder of this chapter. The pointcut matches every call to the method `C.z()` which occurs nested within a call to one of the methods `C.a()`, `C.c()`, or `C.d()`. We leave it as an exercise to the reader to confirm that in the trace in Figure 6.2, the pointcut matches join points 5, 7, and 9 In the code in Figure 6.1, the static shadows which may match the argument of the *cflow* at run-time are the call sites of methods `a()`, `c()`, and `d()` at lines 3, 4, 5, and 12. We have therefore marked them in the code with comments as update shadows 1, 2, 3, and 4, respectively. Because the `&&` operator in pointcut expressions is short-circuiting, the *cflow* is tested at all shadows that may match the left operand of the `&&` operator, namely `call(void C.z())`. Therefore, the query shadows for the *cflow* are the call sites of `z()` in lines 6, 15, and 18 of the example code.

In general, joinpoints matching the argument p of *cflow*(p) may nest recursively, so the update shadows must maintain a nesting count. In addition, AspectJ allows each pointcut to bind values from the joinpoint it matches, and these values may be used inside the advice. If the pointcut p binds values, the generated code must

maintain a stack of the bound values of all nested joinpoints matching p. In early implementations of `ajc`, all *cflow* designators were implemented with a stack of bound values. In the common case of a pointcut binding no values, `ajc` created an empty array at each update shadow and pushed it onto the stack. In `abc`, a much faster counter is used when p does not bind any values. The `ajc` compiler has also adopted this optimization as of version 1.2.1. Nevertheless, in programs that use *cflow*, the overhead of updating and checking the counter or stack can be significant. The goal of the optimizations presented in this chapter is to eliminate this overhead.

6.1.2 `abc` background

Development of the `abc` compiler was motivated by the need for a flexible workbench for experimenting with new language features to be added to AspectJ, and with aspect-oriented analyses and optimizations. The implementation of `abc` takes advantage of two existing compiler toolkits. Like PADDLE, `abc` is a built on top of the Soot [Soo, VRGH+00] Java analysis and optimization framework. Soot itself uses the Polyglot [NCM03] extensible Java frontend to perform semantic checks on Java source code, then converts it to its Jimple intermediate representation. The flexible design of Soot and Polyglot made it possible to develop the `abc` compiler for AspectJ as a modular extension of what is usually a compiler for Java. Moreover, because we built `abc` on top of Soot, we can take advantage of the analyses and optimizations already implemented, including, in particular, the PADDLE framework which was presented in Chapter 4.

The high-level structure of `abc` is shown in Figure 6.3. AspectJ source code is parsed and analyzed by the Polyglot-based frontend. The frontend performs the semantic checks for Java included with Polyglot, as well as additional AspectJ-specific checks that were added as part of `abc`. The final pass in the frontend separates the AspectJ abstract syntax tree (AST) into a pure Java AST, and an aspect information data structure containing all the AspectJ-specific information present in the original code. The Java AST is passed to Soot to be converted to Jimple using Soot's standard JavaToJimple module. The matcher finds all the shadows in the Jimple code at which

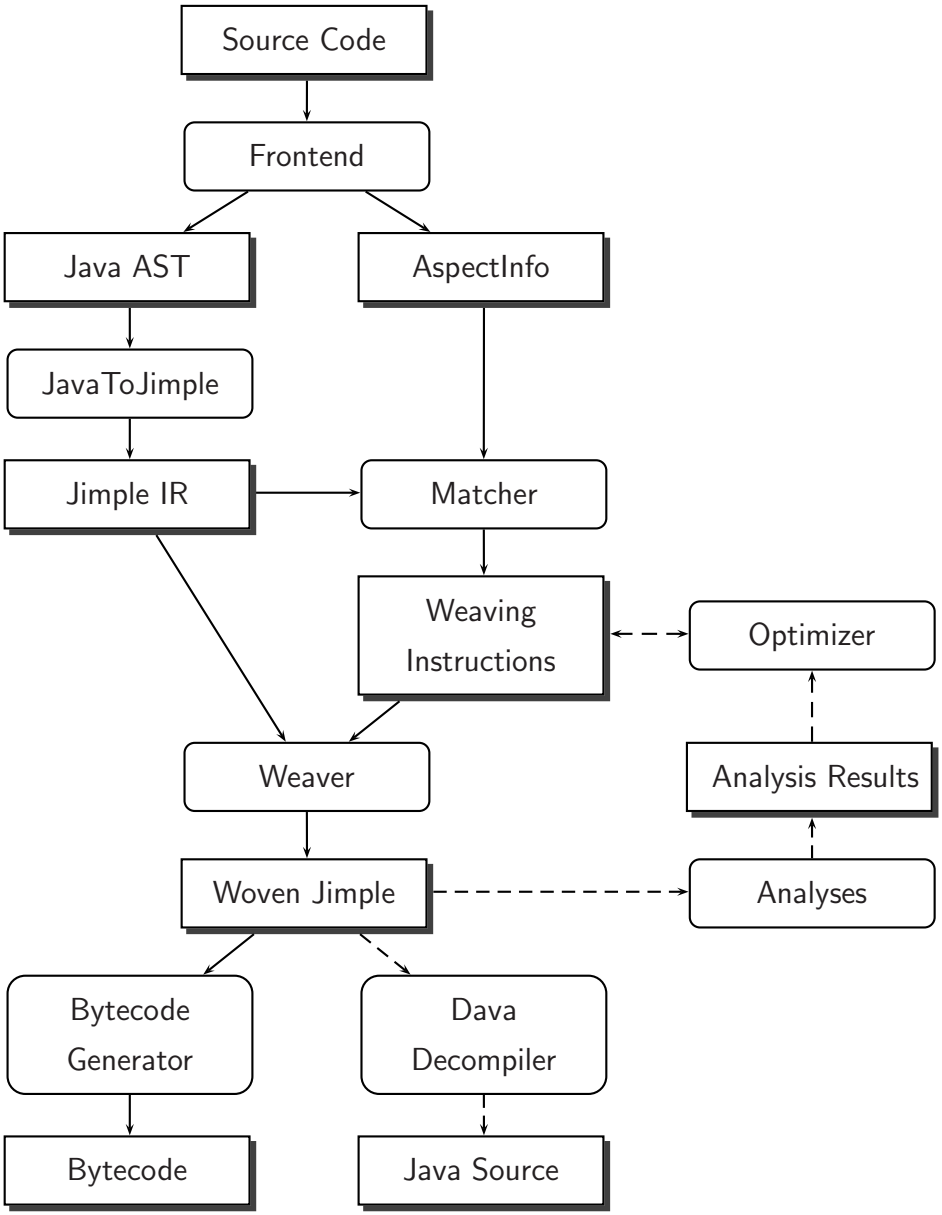


Figure 6.3: High-level structure of the abc AspectJ compiler

each pointcut may match, and produces weaving instructions prescribing where the code for each dynamic residue and advice should be woven. The weaver interprets the weaving instructions and generates the Jimple code to implement the aspects. Finally, Soot converts the Jimple into Java bytecode (or, optionally, decompiles it to Java source code using the Dava [MH02] decompiler).

In designing `abc` for analyzing and optimizing AspectJ code, we wanted to leverage the many analyses existing for Java code, without having to rewrite all of them to be specific to AspectJ. Therefore, `abc` includes a hook to perform analyses on the Jimple code produced immediately after weaving, optimize the naive weaving instructions originally produced by the matcher, and then repeat the weaving process on the original code using the optimized weaving instructions. This is important because the weaving process may change properties on which the optimizations depend. For example, the *cflow* analysis which we present in this chapter requires a call graph which must reflect calls in the woven code, so the call graph must be constructed after weaving. Because the code being analyzed is standard Jimple with no AspectJ-specific constructs, it is possible to apply standard analyses already in Soot and Paddle. Of course, we also implement analyses and optimizations specific to AspectJ, but these are greatly simplified by being able to use the results of Java analyses.

6.2 Cflow Analysis

6.2.1 Desired optimization

The customary implementation of a *cflow* pointcut expression *cflow*(p) incurs overhead at two kinds of shadows. First, at each shadow matching p, a *cflow* stack is pushed and popped to indicate when we are in the dynamic scope of the *cflow*. We denote these shadows with the term ***update shadow***. Second, at each shadow where the *cflow* pointcut could possibly match, we insert a dynamic residue to test whether the *cflow* stack is non-empty. We denote these shadows with the term ***query shadow***.

We wish to perform two kinds of optimization. First, if we can determine *cflow* stack emptiness at a query shadow statically, we can remove the dynamic residue at the query shadow, and possibly other code that becomes unreachable. In our running example, whose code was shown in Figure 6.1, query shadow 1 on line 6 can never execute within the *cflow* of a call to method `a()`, `c()`, or `d()`, so we can statically determine that the *cflow* stack will be empty, and remove the dynamic check at that line. On the other hand, query shadow 3 on line 18 is in the *cflow* of method `d()` every time it executes, so the *cflow* stack is never empty, and we can remove the dynamic check. Second, if we can prove that a *cflow* stack update operation will not be observed by a stack query within the dynamic scope of a given update shadow, we can remove the stack update operations at the update shadow. In our running example, after we have removed the dynamic check at query shadow 3 in line 18, there are no remaining dynamic checks during any execution of method `d()`, so the stack update operations at update shadow 3 in line 5, a call site of `d()`, can be removed.

6.2.2 Analysis prerequisites

Because the *cflow* analysis estimates the calling contexts in which *cflow* shadows execute, a call graph is a key prerequisite. To construct the call graph, we use Paddle in its default configuration, and obtain the context-insensitive call graph from the call graph (**CG**) component. Semantically, *cflow* queries are to be evaluated at run-time on the woven code, so the call graph is built for the woven Jimple code after the initial weaving. In AspectJ, a method m is considered be within the *cflow* of another method m' whenever m executes during the execution of m', regardless of whether m is invoked by an explicit invoke instruction, or implicitly by the VM for one of the reasons listed in Section 4.3.2. Therefore, all the edges in the call graph are relevant to the *cflow* analysis, including the implicit kinds of edges. There is one exception: when a method starts a new thread, the new thread is not considered to be in the *cflow* of the method that started it. Therefore, the *cflow* analysis checks the kind of each call edge, and ignores those edges marked as representing implicit calls to `Thread.run()` from `Thread.start()`.

The `abc` compiler must communicate to the *cflow* analysis the locations of the query and update shadows. For each query and update shadow that it weaves, the weaver records the Jimple instructions that were woven for it. This mapping of shadows to Jimple instructions is passed to the *cflow* analysis to indicate the locations of the shadows in the Jimple code.

6.2.3 Desired analysis results

For each update shadow sh in the program, we define two sets of instructions to be computed, $mayCflow(sh)$ and $mustCflow(sh)$. The set $mayCflow(sh)$ contains every instruction i in the program such that when i is executed, we may be in the dynamic scope of sh. That is, i may execute after the push operation of sh has been performed, but before the corresponding pop operation has been performed. The set $mustCflow(sh)$ contains every instruction i in the program such that whenever i is executed, we must be in the dynamic scope of sh.

Whenever a query shadow is not in $mayCflow(sh)$, we replace the dynamic test with a constant false pointcut designator.[2] Any query shadow in $mustCflow(sh)$ is replaced with a constant true pointcut designator.

In addition, we calculate a subset *necessaryShadows* of update shadows whose effect may be observed at a query shadow. Each update shadow $sh \in necessaryShadows$ satisfies two properties. First, some query shadow qsh that has not been resolved statically may occur in the dynamic scope of sh (*i.e.* $qsh \in mayCflow(sh)$). Second, sh may occur outside the dynamic scope of all update shadows for the same *cflow* stack (*i.e.* $\nexists sh'.sh \in mustCflow(sh')$). This second condition enables us to mark as unnecessary those update shadows at which the stack is always already non-empty. In our running example, update shadow 4 in line 12 is in the *cflow* of a call to method `a()` every time it executes, so the stack is never empty, and the stack update operation at update shadow 4 can be removed.

[2]The *cflow* designator may be part of a more complicated pointcut expression. Constant folding of pointcut expressions is done in a separate phase prior to weaving.

The optimizations become more complicated when the *cflow* binds arguments because, in this case, each query shadow not only tests whether the stack is non-empty, but also observes the entry at the top of the stack. We can still resolve statically those query shadows not in $mayCflow(sh)$, since we know that the stack would always be empty when they are executed. However, at the query shadows where we know the stack is non-empty, we must keep the dynamic residues which read the entry from the stack. In addition, we can no longer remove update shadows just because they are in the *mustCflow* of some other update shadow which will make the stack non-empty, because we also need the correct entry to be pushed onto the stack in addition to the stack being non-empty.

Defining sets of program statements known to execute possibly or definitely within the *cflow* is a natural way of specifying the analysis. However, these sets can be quite large, and it may be prohibitively costly to express them explicitly in an implementation of the analysis. Devising a more compact representation could be difficult. We implement our analysis in the JEDD language and store the sets of program statements in BDDs, which automatically share the representation of common subsets of statements. BDDs provide us with a compact representation of sets of statements without any added complexity in the analysis itself.

6.2.4 Computing analysis results

The exact extent of a *cflow* shadow depends on subtle details of advice precedence and the distinction between *cflow* and *cflowbelow*, and the weaver must respect these details when weaving in the *cflow* stack update operations. Because we perform the analysis on the woven code, we need not consider these details; we simply consider each *cflow* shadow to start immediately after the point where the weaver wove the *cflow* push instruction, and end immediately before the corresponding *cflow* pop instruction. We need to unambiguously classify every instruction in the method as being either within or outside the *cflow* shadow. This requires that there be no jumps into or out of the shadow, which would bypass the push or pop instruction.

Due to details of the weaving process, this requirement is always satisfied, except in the case when the argument p of the *cflow* expression *cflow*(p) is not entirely static, and requires a dynamic residue. In this case, the weaver generates the dynamic test at the update shadow. If the pointcut p does not match, we do not enter the dynamic scope of the *cflow*, so a conditional jump skips the stack update operations. Therefore, when p is not entirely static, the instructions between the push and pop may execute within or outside the dynamic scope of the *cflow*. Since no instruction can be guaranteed to execute only in the dynamic scope of the *cflow*, *mustCflow*(sh) is the empty set in this case.

The JEDD code to compute *mayCflow*(sh) for an update shadow sh is shown in Figure 6.4. The `mayCflow` set is initialized with the set of statements intraprocedurally within the shadow in line 2. Line 6 queries the call graph for the target methods of all call statements in the `mayCflow` set. Line 7 adds all statements in those methods to the `mayCflow` set. This process is repeated until a fixed point is reached.

```
1  <stmt> mayCflow(Shadow sh) {
2      <stmt> mayCflow = stmtsWithin(sh);
3      do {
4          old = mayCflow;
5
6          <method> targets = mayCflow{stmt} <> callTargets{stmt};
7          mayCflow |= targets{method} <> stmtsIn{method};
8
9      } while( mayCflow != old );
10     return mayCflow;
11 }
```

Figure 6.4: JEDD code to compute *mayCflow* for one update shadow

A *cflow* pointcut designator may have many update shadows, and the code in Figure 6.4 has to be executed separately for each one. We can improve on this by computing the *mayCflow* sets for all the update shadows at once, as shown in Figure 6.5. The key modification is that a shadow attribute has been added to the

`mayCflow` and `targets` relations, so that instead of storing a single *mayCflow* set, they instead store a relation of all the *mayCflow* sets, indexed by shadow. Specifically, the `mayCflow` relation contains all pairs (sh, st) such that $st \in mayCflow(sh)$. Computing the *mayCflow* sets all at the same time allows the BDDs to take advantage of any similarities in the sets. This modification is an example of a general approach often applicable when expressing computation relationally; rather than writing algorithms that manipulate a single fact at a time, we write them to manipulate relations of many facts in a single operation.

```
1  <shadow, stmt> mayCflow() {
2      <shadow, stmt> mayCflow = 0B;
3      for( Shadow sh : shadows ) {
4          mayCflow |= new {sh=>shadow}{} >< stmtsWithin(sh){};
5      }
6      do {
7          old = mayCflow;
8
9          <shadow, method> targets = mayCflow{stmt} <> callTargets{stmt};
10         mayCflow |= targets{method} <> stmtsIn{method};
11
12     } while( mayCflow != old );
13     return mayCflow;
14 }
```

Figure 6.5: JEDD code to compute *mayCflow* for all update shadows at once

Like the *mayCflow* sets, *mustCflow* is also computed for all the update shadows at once. However, in this case, the analysis only needs to know that there is some update shadow in whose *cflow* the query must be, but it does not need to know which update shadow it is. That is, the analysis only needs to compute the set $\{st \mid \exists sh : st \in mustCflow(sh)\}$. Since the *mustCflow* algorithm does not need to keep track of the update shadows, its relations do not need a shadow attribute.

We show the JEDD code for computing *mustCflow* in Figure 6.6. If the *cflow* argument has a dynamic residue, the *mustCflow* set is empty (line 2), as discussed earlier. Otherwise, the code first initializes `shadowStmts` with all the statements intraprocedurally within some update shadow (lines 4 to 6), which definitely are in the *mustCflow*. The `mustCflow` set starts with all the statements in the program (line 7); the loop in lines 10 to 19 will eventually remove all statements that can be reached without passing through a statement in `shadowStmts`. The `badMethods` set stores the methods found not to be in the *mustCflow*. In line 8, it is initialized to the entry points of the program. The `badStmts` set stores the statements found not to be in the *mustCflow*. These are the statements in the `badMethods` (line 13), but not any of the statements in `shadowStmts` (line 14), since those *are* in the *cflow*. The `badStmts` are removed from `mustcflow` (line 16), and any methods called from them become `badMethods` (line 17). The process repeats until a fixed point is reached.

The JEDD code to compute *necessaryShadows* is shown in Figure 6.7. It begins with the set of all query statements (line 2). On line 3, it removes those known statically to be false (those which are not in the *mayCflow* of any update shadow). On line 4, it also removes those known statically to be true (those in the *mustCflow* of some update shadow), unless the *cflow* binds arguments. This leaves the query statements that will be tested dynamically. The necessary shadows are those update shadows in whose *mayCflow* any dynamic query statement appears (line 5). Unless the *cflow* binds arguments, we can also remove those update shadows which are already in the *mustCflow* of another update shadow (line 6).

6.3 Experimental Results

We evaluated the *cflow* optimizations on benchmarks from a wide range of sources. The benchmarks are listed in Table 6.1. The figure benchmark is a demo from the AspectJ programming guide [Asp]. The quicksort benchmark is the example from [Sd03] with modifications suggested by Gregor Kiczales. The sablecc benchmark is a compiler written using the SableCC [GMN+] compiler generator, with an aspect applied

```
1  <stmt> mustCflow() {
2      if(dynamicArgument) return 0B;
3      <stmt> shadowStmts = 0B;
4      for( Shadow sh : shadows ) {
5          shadowStmts |= stmtsWithin(sh);
6      }
7      <stmt> mustCflow = 1B;
8      <method> badMethods = entryPoints;
9      <stmt> old;
10     do {
11         old = mustCflow;
12
13         <stmt> badStmts = badMethods{method} <> stmtsIn{method};
14         badStmts -= shadowStmts;
15
16         mustCflow -= badStmts;
17         badMethods = badStmts{stmt} <> callTargets{stmt};
18
19     } while( old != mustCflow );
20     return mustCflow;
21 }
```

Figure 6.6: JEDD code to compute *mustCflow*

```
1  <shadow> necessaryShadows() {
2      <stmt> queryStmts = allQueryStmts;
3      queryStmts &= (shadow=>) mayCflow();
4      if(!bindsArgs) queryStmts -= mustCflow();
5      <shadow> necessaryShadows = mayCflow{stmt} <> queryStmts{stmt};
6      if(!bindsArgs) necessaryShadows -=
7          mustCflow{stmt} <> shadowOfStmt{stmt};
8  }
```

Figure 6.7: JEDD code to compute *necessaryShadows*

to count memory allocations in each of its phases. The ants benchmark is a simulator of an ant colony designed completely in an aspect-oriented style. The benchmarks LoD-sim and LoD-weka consist of the Law of Demeter [LLW03] style-checking aspect applied to two base programs, Certrevsim, a discrete event simulator for certificate revocation simulation [Årn], and Weka, part of the Weka machine learning library [WF00]. Cona [SL04] is a framework for specifying and checking pre- and post-conditions using aspects. It was applied to the stack example from the Cona paper, and to Certrevsim.

Benchmark	Source Lines of Code
figure	94
quicksort	72
sablecc	31233
ants	939
LoD-sim	1586
LoD-weka	3912
Cona-stack	291
Cona-sim	1942

Table 6.1: Benchmarks

Static results of our interprocedural *cflow* analysis are shown in Table 6.2. The "query shadows" column shows, for each *cflow* pointcut designator, the total number of query shadows and, of those, how many the analysis determined to be unreachable, how many are determined to never or always match, and how many cannot be determined statically and therefore still require a dynamic test. The "update shadows" column shows the total number of update shadows and the number that the analysis determines to be necessary, and must remain as dynamic updates even after the analysis.

With the exception of one *cflow* pointcut designator in sablecc, the analysis was able to statically determine the outcome of all *cflow* queries, and therefore entirely

Benchmark	Query shadows					Update shadows	
	Total	Unreach.	Never	Always	Dynamic	Total	Dynamic
figure	6	0	2	4	0	6	0
quicksort	6	0	2	4	0	3	0
sablecc	985	388	299	298	0	698	0
	985	388	332	260	5	1	1
ants	84	0	84	0	0	1	0
LoD-sim	1313	798	515	0	0	41	0
LoD-weka	7031	3501	3530	0	0	41	0
Cona-stack	16	0	14	2	0	27	0
Cona-sim	2	0	2	0	0	2	0
	3	3	0	0	0	18	0
	4	3	1	0	0	31	0
	0	0	0	0	0	2	0
	7	5	2	0	0	20	0
	0	0	0	0	0	6	0
	4	0	4	0	0	5	0
	0	0	0	0	0	3	0

Table 6.2: Static interprocedural optimization counts

remove the dynamic updates and queries of the *cflow* stacks or counters. The imprecision in the sablecc case is due to query shadows in a static initializer. Since any instruction that uses a class could potentially cause the class to be initialized, the static initializer could be called from many different places in the program.

Even though the *cflow* pointcut in ants binds an argument, it can be eliminated because it is never queried. This is because the pointcut is being used as an assertion to find an error condition. By determining that the *cflow* never matches, we have statically verified the assertion.

We were pleasantly surprised that the interprocedural analysis was so effective in resolving *cflow* statically. To confirm that these analysis results are indeed correct, we ran all the benchmarks with a special dynamic residue woven in to check that the static analysis results always agreed with the run-time behaviour.

We compiled and timed the benchmarks on a machine with two AMD Athlon MP 2000+ CPUs running at 1667 MHz, with 2 GB RAM, running Linux version 2.4.20 and the Sun Java HotSpot Client Virtual Machine version 1.4.2-b28. The benchmark running times are presented in Table 6.3. The three middle columns show the running times of the benchmarks when compiled with the latest version of the `ajc` compiler, the `abc` compiler in its default configuration (without interprocedural *cflow* optimizations), and the `abc` compiler with the `-O3` flag, which enables the interprocedural *cflow* optimizations. The default configuration of `abc` includes all of the intraprocedural techniques to reduce the cost of *cflow* described in [ACH+05b]; notice that it already significantly outperforms `ajc` on most of the benchmarks. The right-most column shows the additional speedup provided by the interprocedural *cflow* optimizations compared to the default configuration of `abc`.

The largest speedups are in the figure benchmark, which makes very significant use of *cflow*, and in the ants benchmark, in which the *cflow* binds an argument, and must therefore be tracked with a stack instead of a counter. The speedups on the other benchmarks are also significant, except for quicksort, in which the overhead of *cflow* is small, and Cona-sim, which becomes slightly slower when the *cflow* operations are removed from it, presumably due to chaotic interactions with the virtual machine and hardware.

Benchmark	ajc 1.2.1 default	abc 1.0.2 default	abc 1.0.2 -O3	Speedup
figure	167.7	20.3	1.96	936.%
quicksort	28.9	27.4	27.3	0.366%
sablecc	24.2	22.5	20.4	10.3%
ants	32.9	17.9	13.1	36.6%
LoD-sim	35.3	26.2	23.7	10.5%
LoD-weka	113.5	75.2	66.3	13.4%
Cona-stack	56.0	27.4	23.1	18.6%
Cona-sim	69.0	72.0	73.6	-2.17%

Table 6.3: Benchmark running times (seconds)

6.4 Related Work

The work described in this chapter was motivated by an empirical study measuring the run-time overheads of aspect-oriented features [DGH+04]. The study showed that this overhead can be very significant, particularly in programs containing *cflow* pointcuts and around advice. We therefore focused on reducing the overhead of these two constructs [ACH+05b] by devising intraprocedural techniques that reduce the overhead of *cflow*, the interprocedural analyses presented in this chapter that eliminate *cflow* overhead entirely, and efficient implementation techniques for around advice.

The *cflow* analyses presented in this chapter were inspired by earlier work by Sereni and de Moor [Sd03] on a simple procedural language. For each shadow testing a *cflow*-like pointcut designator, they computed a regular language over-approximating the set of all call stack configurations possible at the shadow. The pointcut designator was tested against these call stacks; if it matched all of them or none of them, it could be resolved statically.

Costanza [Cos03] noted the resemblance between *cflow* pointcuts and dynamically scoped functions in languages such as Lisp. The state captured at a *cflow* update

shadow is made available at queries within its dynamic scope. In the same way, state stored in variables of dynamically scoped functions is available throughout the dynamic scope of the function. Neubauer and Sperber [NS01] presented a dynamic scope analysis intended for automatic program translation from Emacs Lisp to languages without dynamically scoped variables.

6.5 Conclusions

We have presented an interprocedural analysis and optimization for *cflow* pointcuts in AspectJ. In cases where the analysis resolves all *cflow* queries statically, it removes all overhead of dynamic *cflow* tests. In our experiments, the analysis did indeed resolve all queries statically in all but one benchmark. In six of our eight benchmarks, the *cflow* overhead that was removed accounted for over 10% of execution time, even after all intraprocedural techniques for reducing the cost of the *cflow* tests from [ACH+05b] had been applied.

Our implementation of the *cflow* analysis relies on both JEDD and PADDLE. The JEDD implementation of the *cflow* analysis is concise, and takes advantage of BDDs to efficiently represent the large sets of statements that may or must execute within a given *cflow* pointcut. The PADDLE framework is used to construct the call graph needed by the *cflow* analysis.

The remarkable static precision of the *cflow* analysis suggests an area of future work in extending AspectJ to allow "dynamic" pointcuts such as *cflow* to be used in contexts where currently only "static" pointcuts are allowed, such as in the **declare error** construct. Currently, this construct allows programmers to specify very simple program-specific properties to be verified at compile time. Allowing additional pointcut designators would make it possible to specify more interesting properties. A sufficiently precise analysis could check the properties at compile time. Should the analysis fail to verify a property statically, it could produce a warning and weave in an assertion to check the property at run time.

Chapter 7
Conclusions and Future Work

Compilers and software engineering tools require increasingly precise and efficient interprocedural program analyses. A key problem in implementing these analyses is representing the collections of large sets that these analyses manipulate. We have shown that BDDs are a general-purpose data structure for compactly storing and efficiently manipulating these sets. In particular, we have shown that a BDD-based implementation makes it possible for context-sensitive analyses to scale to large Java programs. In addition, the use of BDDs frees analysis designers from having to design special-purpose data structures customized for each program analysis, and therefore makes it easier to develop and experiment with new analyses.

7.1 The Jedd Language and Compiler

We have presented Jedd, a programming language and compiler that makes it feasible to implement complicated, interrelated program analyses using BDDs. In the Jedd language, program analyses are expressed at a high level in terms of relations, and the Jedd compiler translates the relational operations into low-level BDD operations. In the process, Jedd performs static and dynamic type checking to catch the inconsistent uses of relations that would make it infeasible to write the analyses in terms of BDDs directly. We have designed and implemented within Jedd an algorithm for finding a reasonable assignment of relation attributes to BDD physical domains. The

programmer may specify part of the physical domain assignment by hand to tune performance-critical computations, and use the algorithm to automatically complete the assignment for the rest of the program. JEDD also provides support for tuning the BDD representation, including a profiler that graphically displays the shapes of the BDDs that are constructed during execution. We have identified several patterns of the BDD shapes associated with common inefficiencies, and suggested techniques for tuning the physical domain assignment and BDD variable ordering when these patterns are observed. Finally, we have shown that JEDD introduces almost no overhead into the performance of program analyses (compared to program analyses implemented directly in terms of BDD operations), and that the JEDD translator scales to programs as large as our PADDLE interprocedural analysis framework.

In summary, we have shown that BDDs are a flexible way to prototype and experiment with novel interprocedural program analyses.

7.2 The Paddle Interprocedural Analysis Framework

Using JEDD, we have implemented PADDLE, a framework of interrelated interprocedural program analyses. PADDLE consists of a BDD-based implementation of points-to analysis with on-the-fly call graph construction, and related prerequisite and client analyses. PADDLE supports several variations of context sensitivity, including using strings of call sites and strings of receiver objects as the context abstraction. Because PADDLE represents context information using BDDs, these context-sensitive analyses scale to much larger Java programs than earlier implementations.

The two key analyses in the PADDLE framework, points-to analysis and call graph construction, are prerequisites for many interprocedural program analyses for Java required by optimizing compilers and software engineering tools. For example, optimizing compilers can make use of interprocedural analysis information to reduce the overhead of virtual calls and remove redundant heap accesses. Software engineering

tools such as bug detectors, program verifiers, and race detectors make use of call graphs and points-to information.

7.3 Empirical Evaluation of Context Sensitivity

We have used Paddle to perform an empirical evaluation of the effect of variations of context-sensitive analyses on the precision of call graph construction, points-to analysis, and related client analyses. Thanks to our use of BDDs to implement the analyses, we were able to include in our study variations of context sensitivity that could not be included in earlier studies because their non-BDD implementations were not sufficiently scalable. We showed that object sensitivity [MRR02, MRR05] in particular does significantly improve precision of interprocedural program analyses. Among the variations of object sensitivity, extending the length of context strings beyond one receiver object does not further improve precision of client analyses. However, modelling abstract heap objects with context does improve precision compared to an analysis that models only pointer variables with context, although it also increases the cost of the analysis. Therefore, we conclude that 1-object-sensitive analyses, with or without context-sensitive modelling of abstract heap objects, are the best tradeoffs between analysis precision and cost.

7.4 Analysis of the cflow Construct

We have designed and implemented in the Jedd language a static analysis of the *cflow* construct in the aspect-oriented language AspectJ. The analysis builds on top of the call graph constructed by Paddle. By implementing the analysis at a high level in Jedd and deferring low-level concerns about efficient data representation to BDDs, we were able to easily experiment with variations of the analysis during its development. The final implementation of the analysis very closely resembles its specification.

The analysis itself was very successful. In seven of our eight benchmarks, the analysis resolved all *cflow* queries statically, removing all overhead due to the *cflow* construct. In six of the eight benchmarks, this overhead due to *cflow* represented over 10% of execution time.

The high precision of the analysis results suggests that it could be used not only for reducing the overhead of *cflow*, but for statically verifying program specifications expressed using the *cflow* construct.

7.5 Future Work

The JEDD system that we have developed and presented in Chapter 3 provides an ideal platform for prototyping program analyses. In the future, we will continue to use it to experiment with new static analyses, particularly for emerging languages such as AspectJ.

The PADDLE framework that we have developed is a key foundation for applications of precise interprocedural analyses for Java. We plan to continue to use it to develop new compiler optimizations, particularly for new language features being proposed. For example, we recently defined tracematches [AAC+05] as a mechanism for expressing sequences of events in program traces, and for triggering actions when these sequences are observed. In order to be efficient enough to be practical, tracematches are likely to require sophisticated interprocedural analyses and optimizations. Furthermore, PADDLE will serve as the foundation of software engineering tools, including visualizers and verifiers. A simple example of such a tool would be to integrate the cast safety analysis described in Section 4.4.2 into an Integrated Development Environment, where it would warn programmers about potentially failing casts during program development.

In our study of the effects of context sensitivity on analysis precision, we have identified object sensitivity as a technique deserving further research. In particular, we have shown that object-sensitive analyses are more precise than other variations, and that it is likely that efficient implementations of object-sensitive analyses can be

found. Our search for these efficient implementations will be guided by the observations from our study.

The very high precision of our analysis of the *cflow* construct in AspectJ suggests a new area of application of aspect-oriented techniques: static verification of program properties expressed using aspects. In aspect-oriented languages, aspects are a natural way for programmers to express assertions about the intended behaviour of their programs. In current aspect-oriented systems, these assertions can be checked at run time. By developing precise analyses of aspect-oriented features, we will make it possible to check these assertions statically.

Appendix A
Proofs

Proposition 1 *The problem of finding a reasonable physical domain assignment is NP-complete.*

Proof: We prove that the problem is NP-hard by constructing a polynomial reduction of the NP-complete graph vertex k-colouring problem to it. For a given graph G, we construct a JEDD program for which a reasonable physical domain assignment exists if and only if G has a k-colouring.

Let $G = (V, E)$ be a graph for which a k-colouring is to be found. Construct a JEDD program from it as follows:

1. Declare attributes `a`, `b`, and `c`.

2. Declare $k + 1$ physical domains $d_0 \ldots d_k$.

3. For each vertex $v_i \in V$, declare a JEDD relation variable x_i with schema `<a,b>`, and no physical domains specified.

4. For each j with $1 \leq j \leq k$ and for each $v_i \in V$, add an assignment of a relation literal:
   x_i `= new {o1=>a:`d_j`, o2=>b:`d_0`}`.

5. For each edge $(v_i, v_j) \in E$, add a statement computing
   x_i`{b} <> ((a=>c)` x_j`){b}`.

Now, if G has a k-colouring, the following physical domain assignment is reasonable. For each v_i coloured $c(v_i)$, assign the physical domain $d_{c(v_i)}$ to the following attribute instances:

1. attribute `a` of x_i,
2. attribute `a` of the literal with `a` explicitly specified to be assigned to d_{c_i},
3. attribute `a` of the result of each composition having x_i as its left argument, and
4. attribute `c` of the result of the composition having `((a=>c)` x_j`)` as its right argument.

Conversely, suppose a valid physical domain assignment with no unnecessary replaces has been found for the JEDD program. Attribute `a` of each x_i must be assigned to some physical domain, and there is a reason to assign it to any of the physical domains $d_1, \ldots, d_k$ because of the k literals. Attributes `a` and `c` of the result of each composition can only be assigned with reason to the same physical domain as the corresponding operand. Each composition with arguments x_i and x_j therefore forces attribute `a` of x_i and x_j to be assigned to distinct physical domains. Now, whenever attribute `a` of x_i is assigned to d_j, we colour v_i with the colour j. For each edge (v_i, v_j) in G, the corresponding composition ensures that v_i and v_j are coloured with different colours, so we have obtained a k-colouring of G.

Therefore, G is k-colourable if and only if a reasonable physical domain assignment exists for the constructed JEDD program, so the physical domain assignment problem is NP-hard.

Given a physical domain assignment, it can be checked in polynomial time that it is reasonable. Therefore, finding a reasonable physical domain assignment is in NP. Since it is also NP-hard, it is NP-complete. □

Proposition 2 *Let G be an attribute def-use graph, and let $\prec$ be an antisymmetric binary relation on its vertices such that $a \prec b$ implies that a and b are connected by an* assignment *edge in G. Then the following four statements are all equivalent:*

1. *$\prec$ is a well-founded relation.*

2. *There exists a total order $\leq$ such that $a \prec b \Rightarrow a < b$. (This is the order in which physical domains could be assigned the vertices.)*

3. *There exists a total antisymmetric relation $\sqsubseteq$ such that $a \prec b \Rightarrow a \sqsubset b$ and there is no triple of distinct vertices a, b, c such that $a \prec b \sqsubset c \sqsubset a$.*

4. *On the vertices of every biconnected component $C = (V_C, E_C)$ of the graph formed by* assignment *edges, there exists a total antisymmetric relation $\sqsubseteq_C$ such that $\forall a, b \in V_C . a \prec b \Rightarrow a \sqsubset_C b$ and there is no triple of distinct vertices a, b, c such that $a \prec b \sqsubset_C c \sqsubset_C a$.*

Proof:

1 $\Rightarrow$ 2:

Suppose $\prec$ is a well-founded relation, so that every non-empty subset of vertices contains a minimal element. Define recursively the sequence of sets $S_1, S_2, \ldots, S_n = \emptyset$ by $S_1 = V$, the set of all vertices, and for each $i \geq 1$, $S_{i+1} = S_i \setminus \{a_i\}$, where a_i is a minimal element of S_i. Then let $\leq$ be the total order defined by $i \leq j \Leftrightarrow a_i \leq a_j$.

If $a_i \prec a_j$, then since a_j is a minimal element of S_j, a_i cannot be in S_j. But since $S_j = V \setminus \{a_k \mid k < j\}$, it must be the case that $a_i \in \{a_k \mid k < j\}$. Therefore, $i < j$, so $a_i < a_j$. So, for all a, b, $a \prec b \Rightarrow a < b$, satisfying statement 2.

2 $\Rightarrow$ 3:

The total order $\leq$ satisfies the properties required of the relation $\sqsubseteq$ by statement 3. It is a total antisymmetric relation, and $a \prec b \Rightarrow a < b$. Given any triple of distinct vertices a, b, c, it is not the case that $a \leq b \leq c \leq a$, since transitivity and antisymmetry would imply $a = b = c$. Since $a \prec b \Rightarrow a \leq b$ and $a < b \Rightarrow a \leq b$, it is also not the case that $a \prec b < c < a$.

3 $\Rightarrow$ 4:

Let $\sqsubseteq$ be a relation satisfying statement 3. Given a biconnected component $C = (V_C, E_C)$ of the graph formed by assignment edges, let $\sqsubseteq_C$ be the relation $\{(a, b) \mid a, b \in V_C \wedge a \sqsubseteq b\}$. Then $\sqsubseteq_C$ satisfies the required conditions of statement 4.

4 $\Rightarrow$ 1:

We prove this by contradiction. Suppose that statement 4 holds, yet $\prec$ is not well-founded, so there exists an infinite descending chain $\ldots \prec a_2 \prec a_1 \prec a_0$. Since the number of attribute instances is finite, some a_i must be repeated; that is, there exist indices i and j, $j < i$, with $a_j = a_i$.

Let $n = j - i$ and define $b_k = a_{i+k}$ for all k, giving the cycle $b_0 = b_n \prec b_{n-1} \prec \ldots \prec b_0$. Without loss of generality, we can choose the smallest cycle, so that all the b_i's from b_1 to b_n are distinct. Since every pair $x \prec y$ implies an assignment edge between x and y, the b_i's form a cycle in the graph of assignment edges, so they are all in the same biconnected component. Because $\prec$ is antisymmetric, the length of the cycle, n, is at least 3.

Let $\sqsubseteq_C$ be the relation that satisfies statement 4 on the biconnected component containing the b_i's. We show by induction on i that $b_i \sqsubset_C b_0$ for all $1 \leq i \leq n-1$. When $i = 1$, $b_i \sqsubset_C b_0$ follows from $b_1 \prec b_0$. Suppose that for some k with $2 \leq k < n-2$, $b_k \sqsubset_C b_0$. It cannot be the case that $b_0 \sqsubset_C b_{k+1}$, since $\sqsubseteq_C$ satisfies statement 4, which states that there is no triple b_{k+1}, b_k, b_0 for which $b_{k+1} \prec b_k \sqsubset_C b_0 \sqsubset_C b_{k+1}$. Since $\sqsubseteq_C$ is total, it must be the case that that $b_{k+1} \sqsubset_C b_0$. Therefore, $b_k \sqsubset_C b_0$ implies $b_{k+1} \sqsubset_C b_0$. By induction, $b_i \sqsubset_C b_0$ for all $1 \leq i \leq n-1$.

Now we have $b_{n-1} \sqsubset_C b_0 = b_n$, but $b_n \prec b_{n-1}$ implies $b_n \sqsubset_C b_{n-1}$. This contradicts antisymmetry of $\sqsubset_C$. $\square$

Proposition 3 *When the SAT formula produced for the physical domain assignment problem is unsatisfiable, every unsatisfiable core contains at least one clause of type 3.4 (conflict clause).*

Proof: The key idea of the proof is that if clauses of type 3.4 are removed, the SAT formula ignores the requirement that conflict edges be respected. We will show that it is always possible to find a reasonable assignment if all conflict edges are removed, and such a physical domain assignment therefore corresponds to a satisfying assignment of the remaining clauses.

We will first construct the total order $\leq$ in which physical domains are assigned to vertices. We represent the order by numbering the vertices with consecutive natural numbers. To each vertex v, we assign a natural number $o(v)$ and define $o(v) \leq$

$o(v') \Leftrightarrow v \leq v'$. We begin by assigning the lowest natural numbers arbitrarily to the vertices with explicitly specified physical domains. Define the sequence of sets $A_i = \{v \mid o(v) \leq i\}$, so A_i is the set of the first i vertices to be assigned a physical domain. Let k be the number of vertices with explicitly specified physical domains, so A_k is the set of these vertices. Recall that there exists a path of assignment edges from every vertex to a vertex in A_k. Therefore, for all $i > k$, there must be an assignment edge from some vertex v in $V \setminus A_i$ to some vertex v' in A_i (as long as $V \setminus A_i$ is not empty). Otherwise, there would be no path from any vertex in $V \setminus A_i$ to any vertex in A_i, but there has to be a path from every vertex in $V \setminus A_i$ to A_k, and $A_k \subseteq A_i$. Having defined the sets A_i, we can inductively number all the vertices in the following way: for each $i > k$, find a vertex $v \in V \setminus A_i$ such that an assignment edge connects v to some vertex in A_i, and define $o(v) = i + 1$.

Having thus defined the order in which physical domains are to be assigned to vertices, we can construct a reasonable assignment following the order. We first assign physical domains to the k vertices for which they have been explicitly specified. Then each vertex v with $o(v) > k$ has at least one neighbouring vertex $v' \in A_{o(v)-1}$ connected by an assignment edge. Since $v' \in A_{o(v)-1}$, $o(v') \leq o(v) - 1 < o(v)$. Therefore, if we assign physical domains in order, v' will be assigned a physical domain before v, so there is a reason to assign v the same physical domain as v'.

It can be checked that a physical domain assignment constructed in this way satisfies clauses 3.1, 3.2, and 3.3. Define $v' \prec v$ if and only if $o(v') < o(v)$, and v is assigned the same physical domain as v'. By the definition of $\prec$, SAT clauses 3.5, 3.6, and 3.7 are satisfied. Since $\prec$ satisfies statement 1 of Proposition 2, statement 4 also holds. By statement 4, there exists a relation $\sqsubseteq_C$ such that $\prec$ and $\sqsubseteq_C$ satisfy SAT clauses 3.8 and 3.9.

We have constructed an assignment satisfying all clauses except those of type 3.4. Therefore, every unsatisfiable subset of clauses must contain a clause of type 3.4. $\square$

Appendix B
Jedd Usage Notes

This appendix provides additional practical information for programmers intending to write JEDD code. Programmers should first read Chapter 3 to learn about the JEDD system in general. The appendix contains implementation-specific details about using the JEDD translator and runtime system.

B.1 Example

The JEDD distribution includes a directory called `examples` containing sample JEDD code. The directory `examples/pointsto` contains a complete JEDD implementation of the BDD-based points-to analysis from [BLQ+03].

B.2 Jedd Source Files

Source files to be processed by JEDD must have one of the extensions `.jedd` or `.java`. It is customary to use the extension `.jedd` for files containing JEDD-specific constructs, and `.java` for files containing plain Java.

JEDD files should import the package `jedd.*` from the JEDD runtime library. This package contains interface classes with methods that can be called by JEDD programs. In particular, the `jedd.Jedd` class is a singleton containing methods affecting the

behaviour of JEDD in general, and `jedd.Relation` is an interface listing the methods that can be called on any JEDD relation type. JEDD files should not import the package `jedd.internal.*`.

B.3 Selecting a Backend

JEDD currently supports four different BDD libraries as backends: BuDDy, CUDD, SableJBDD, and JavaBDD. BuDDy is the backend which has the most complete support in JEDD, which is the most tested, and which tends to perform best. BuDDy and CUDD are C libraries, so they require that their shared library (`.so` or `.dll`) files be available on the `LD_LIBRARY_PATH`. Before the program instantiates any relations, it must must select one of the backends by calling `jedd.Jedd.v().setBackend()`. The argument to this method should be one of `"buddy"`, `"cudd"`, `"sablejbdd"` or `"javabdd"`.

B.4 Compiling Jedd Code

The JEDD compiler is invoked with the command `java jedd.Main`. It uses the same command-line format as Polyglot, with two additional switches for specifying the path to a SAT solver (`-s`) and a SAT core extractor (`-sc`). The simplest way to compile a project is to list all the `.jedd` files on the command line. This will compile them to `.java` files, and run `javac` on them to compile them to classfiles. The `-c` switch disables the `javac` pass. If your project consists of both `.jedd` and `.java` files, you can put them all on the command line, but be warned that Polyglot will overwrite your `.java` files unless you specify an alternate output directory with the `-d` switch.

The points-to analysis example provided with JEDD includes a simple Ant build file which can be modified for use in other projects.

B.5 Using the Profiler

To use the profiler, it must be enabled before the computation to be profiled begins by calling `jedd.Jedd.v().enableProfiling()` with a `java.io.PrintStream` to which the profile will be written. At the end of the computation, the profile file must be closed by calling `jedd.Jedd.v().outputProfile()`. See the file `examples/pointsto/src/Prop.jedd` for an example use of the profiler.

Viewing the profile data requires a SQL database and a web server supporting the CGI. The CGI scripts (found in the `profile_view` directory in the JEDD distribution) are specific to SQLite, but should work with any web server. They expect the profiling data in a database called `profile.db`, in the same directory as the scripts. This file can be generated by piping the SQL file generated by the JEDD runtime to SQLite with the command:

```
cat profile.sql | sqlite profile.db
```

The web server can be started with the command:

```
/usr/sbin/thttpd -d /directory/with/cgi/scripts -p 8080 -c '*.cgi'
```

This starts the web server on port 8080. To view the profiling data, point your web browser to:

`http://127.0.0.1:8080/main.cgi`.

Appendix C
Paddle User's Guide

This appendix describes how to invoke the Paddle framework and how to retrieve the analysis results that it generates in a client analysis. Before reading this appendix and using Paddle, users are encouraged to read Chapter 4 which explains the features and design of Paddle in detail.

C.1 Invoking Paddle

Paddle is implemented as a Soot whole-program phase, and is invoked from the Soot command line. The Paddle phase is called `cg.paddle`, and appears within Soot's call graph construction pack, `cg`. In order to run Paddle or any other interprocedural analysis, Soot must be told to run in whole-program mode using the `-w` command line switch. The Paddle phase can then be enabled using the phase switch `-p cg.paddle on`.

The following example command line invokes Paddle with its default settings on the Java program whose main class is `Main`:

```
java soot.Main -w -p cg.paddle on Main
```

In the rest of this section, we describe the command line options that control features of the Paddle framework. A quick summary of all the options can be obtained from Soot using the command:

```
java soot.Main -phase-help cg.paddle
```

Like all Soot phase options, Paddle options are given on the Soot command line following the phase option switch `-p cg.paddle`. The option name and its corresponding value are separated with a colon. For example, the `verbose` option is enabled by adding `-p cg.paddle verbose:true` to the command line.

C.1.1 General options

`verbose` (default value: `false`)

The `verbose` option causes Paddle to print detailed information about its execution.

C.1.2 Analysis implementation options

`bdd` (default value: `false`)

Paddle contains both BDD-based and traditional implementations of each of its components. The `bdd` option controls which of these implementations will be instantiated. Setting the option to `true` instantiates the BDD-based implementation of each component; setting the option to `false` instantiates the traditional implementation.

`propagator` (default value: `auto`)

The `propagator` option controls which points-to set propagation algorithm will be instantiated.

Allowed values:

`auto`	By default, the propagation algorithm is selected based on the value of the `bdd` option. When BDD-based components are being used, the incremental BDD-based propagation algorithm that was described in Section 4.3.4 is used. When traditional components are being used, the worklist propagation algorithm is used.

`bdd`	The `bdd` setting causes PADDLE to use the basic BDD-based propagation algorithm that was described in Section 4.3.4.
`incbdd`	The `incbdd` setting causes PADDLE to use the incremental BDD-based propagation algorithm that was described in Section 4.3.4.
`iter`	The `iter` setting causes PADDLE to use the naive iterative propagation algorithm based on the iterative algorithm in SPARK [Lho02].
`worklist`	The `worklist` setting causes PADDLE to use the fast worklist propagation algorithm based on the worklist algorithm in SPARK [Lho02].
`alias`	The `alias` setting causes PADDLE to use the alias-edge propagation algorithm base on the alias-edge alias-edge in SPARK [Lho02].

`conf` (default value: `ofcg`)

The `conf` option determines how the components of PADDLE should be connected together, in order to either construct the call graph on-the-fly as the points-to analysis proceeds, use an existing call graph, or use the algorithm of Zhu, Calman, Whaley and Lam [ZC04, WL04] to construct a context-sensitive call graph from an existing context-insensitive one. These configurations were discussed in detail in Section 4.3.6 and summarized in Figure 4.17.

Allowed values:

`ofcg`	The default `ofcg` setting causes PADDLE to build the call graph on-the-fly as the points-to analysis proceeds.

`cha-aot` The `cha-aot` setting causes PADDLE to

1. first build a call graph using Class Hierarchy Analysis [DGC95],
2. then perform a points-to analysis using this call graph constructed ahead-of-time.

`ofcg-aot` The `ofcg-aot` setting causes PADDLE to

1. first build a call graph on-the-fly as a points-to analysis proceeds, as with the `ofcg` option,
2. then discard the computed points-to sets,
3. and finally perform a second points-to analysis using the call graph constructed ahead-of-time.

`cha-context-aot` The `cha-context-aot` setting causes PADDLE to

1. first build a call graph using Class Hierarchy Analysis [DGC95],
2. then make the call graph context-sensitive using the algorithm of Zhu, Calman, Whaley, and Lam [ZC04, WL04],
3. and finally perform a points-to analysis using the context-sensitive call graph constructed ahead-of-time.

`ofcg-context-aot` The `ofcg-context-aot` setting causes PADDLE to

1. first build a call graph on-the-fly as a points-to analysis proceeds, as with the `ofcg` option,
2. then discard the computed points-to sets,
3. then make the call graph context-sensitive using the algorithm of Zhu, Calman, Whaley, and Lam [ZC04, WL04],
4. and finally perform a points-to analysis using the context-sensitive call graph constructed ahead-of-time.

C.1.3 Paddle context sensitivity options

The following options control which variation of context sensitivity PADDLE uses in its analyses.

`context` (default value: `insens`)

The `context` option controls which kind of context abstraction PADDLE will use. The supported context abstractions were described in detail in Section 4.3.2.

Allowed values:

`insens` The `insens` setting causes PADDLE to perform context-insensitive analyses.

`1cfa` The `1cfa` setting causes PADDLE to perform 1-CFA [Shi88] context-sensitive analyses.

`kcfa` The `kcfa` setting causes PADDLE to perform k-CFA [Shi88] context-sensitive analyses, for some fixed value of k. See the `k` option below to set the value of k.

`objsens`	The `objsens` setting causes PADDLE to perform 1-object-sensitive [MRR02] analyses.
`kobjsens`	The `kobjsens` setting causes PADDLE to perform k-object-sensitive [MRR02] analyses, for some fixed value of k. See the `k` option below to set the value of k.
`uniqkobjsens`	The `uniqkobjsens` setting causes PADDLE to perform unique-k-object-sensitive analyses, for some fixed value of k. See the `k` option below to set the value of k.

`k` (default value: `2`)

The `k` option controls the maximum length of a call string or receiver object string used as the context abstraction when the `context` option is set to `kcfa`, `kobjsens`, or `uniqkobjsens`.

`context-heap` (default value: `false`)

The `context-heap` option causes PADDLE to model abstract heap locations in a context-sensitive way. When the `context-heap` option is `false`, only pointer variables are modelled context-sensitively.

C.1.4 BDD backend options

The following options control the BDD backend used by PADDLE.

`backend` (default value: `buddy`)

The `backend` option selects which BDD library will be used to implement BDDs.

Allowed values:

`buddy`	The `buddy` setting causes PADDLE to use the BUDDY [LN] BDD library as the backend.
`cudd`	The `cudd` setting causes PADDLE to use the CUDD [Som] BDD library as the backend.

`sable`	The `sable` setting causes PADDLE to use the Sable-JBDD [Qia] BDD library as the backend.
`javabdd`	The `javabdd` setting causes PADDLE to use the JavaBDD [Whab] BDD library as the backend.

`profile` (default value: `false`)

The `profile` option turns on the JEDD profiler to profile all PADDLE BDD operations. The profiler output is compressed and written to a file named `profile.sql.gz` in the current working directory.

C.1.5 Miscellaneous analysis precision options

`this-edges` (default value: `false`)

When PADDLE is building a call graph on-the-fly, it models the flow of objects from the receiver of a method call to the `this` pointer of the called method precisely, by propagating only those abstract objects whose type would cause that particular method to be invoked. The `this-edges` option causes PADDLE to instead model this flow using the simpler technique of adding a subset constraint between the receiver and the `this` pointer. The effects of this option were discussed in detail in Section 4.3.3.

`field-based` (default value: `false`)

The `field-based` option causes PADDLE to perform a field-based rather than field-sensitive points-to analysis. In a field-based analysis, each field of a class is modelled as a single pointer, corresponding to all instances of the field in all objects of the class. A field-sensitive analysis uses points-to information to distinguish provably distinct objects, and models their fields separately. A field-sensitive analysis is more precise but generally more expensive to perform than a field-based analysis.

`types-for-sites` (default value: `false`)

The `types-for-sites` option causes PADDLE to abstractly model each object using the run-time type of the object, rather than its allocation site. Using allocation sites is more precise but generally more costly than using run-time types to abstractly model objects.

C.2 Analysis Results

After PADDLE constructs a call graph and performs points-to analysis, it stores these analysis results into the singleton class `soot.jimple.paddle.Results`, so they can be retrieved by client analyses.

The analysis results are returned in the form of ***readers***, a generalization of iterators that can be used by both traditional and BDD-based client analyses. Each reader represents a relation of analysis results. The `get()` method of the reader returns a BDD representation of the relation for use by BDD-based client analyses. The `iterator()` method of the reader returns an iterator over the tuples of the relation for use by traditional client analyses.

This `soot.jimple.paddle.Results` class contains the following three methods:

`public AbsCallGraph callGraph()`

The `AbsCallGraph` object represents the call edges in the call graph. Its `csEdges()` method returns a reader of the set of all context-sensitive call edges. The `edgesOutOf(Context, SootMethod)` method returns a reader of only the context-sensitive call edges originating from the specified method in the specified context.

The `AbsCallGraph` object can also provide a context-insensitive projection of the call edges by removing the context. The `ciEdges()` method returns a reader of the context-insensitive projection of the set of all call edges, and the `edgesOutOf(SootMethod)` method returns a reader of the context-insensitive projection of only the call edges originating from the specified method.

`public AbsReachableMethods reachableMethods()`

The `AbsReachableMethods` object represents the set of methods reachable

through the call graph, and the contexts in which they are reachable. The `contextMethods()` method returns a reader of all the method and context pairs in which each method is reachable. The `methods()` method returns a reader of the set of all methods reachable through the call graph in any context.

`public AbsP2Sets p2sets()`

The `AbsP2Sets` object represents the points-to sets computed by PADDLE. Its `getReader()` method returns a reader of the context-sensitive points-to relation of all points-to pairs. It also provides a `fieldPt()` method, which returns a reader of the context-sensitive field points-to relation describing the points-to sets of fields of heap objects.

Bibliography

[AAC+05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 345–364, New York, NY, USA, 2005.

[abc] abc: The AspectBench Compiler for AspectJ. `http://aspectbench.org/`.

[ACH+05a] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98. 2005.

[ACH+05b] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–128, New York, NY, USA, 2005.

[ACSE99] Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, and Susan J. Eggers. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22-24, 1999, Proceedings*, volume 1694 of *Lecture Notes in Computer Science*, pages 19–38. 1999.

[AFFS98] A. Aiken, M. Faehndrich, J. S. Foster, and Z. Su. A Toolkit for Constructing Type- and Constraint-Based Program Analyses. In *Types in Compilation: Second International Workshop, TIC'98*, volume 1473 of *Lecture Notes in Computer Science*, pages 78–96, 1998.

[Age95] Ole Agesen. The Cartesian Product Algorithm. In *ECOOP '95, Object-Oriented Programming: 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 2–51, 1995.

[ajc] ajc: The Eclipse AspectJ Compiler. `http://www.eclipse.org/aspectj/`.

[Ana] C. Scott Ananian. JavaCUP. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[Årn] André Årnes. PKI Certificate Revocation. `http://www.pvv.ntnu.no/~andrearn/certrev/`.

[Asp] AspectJ Team. The AspectJ Programming Guide. `http://eclipse.org/aspectj`.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–11. 1988.

[BCCH97] M. Burke, P. Carini, J. Choi, and M. Hind. Interprocedural Pointer Alias Analysis. Technical Report RC 21055, IBM T. J. Watson Research Center, December 1997.

[Bea96] D. M. Beazley. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pages 129–139, July 1996.

[Beh] Gerd Behrmann. The Interactive BDD Environment. `http://iben.sourceforge.net/`.

[BH99] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 35–46. 1999.

[Bla99] Bruno Blanchet. Escape analysis for object-oriented languages: application to Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 20–34. 1999.

[BLM02] R. Berghammer, B. Leoniuk, and U. Milanese. Implementation of relational algebra using binary decision diagrams. In *6th International Conference RelMiCS 2001*, volume 2561 of *LNCS*, pages 241–257, December 2002.

[BLQ+02] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. Technical Report 2002-10, McGill University, Sable Research Group, 2002. `http://www.sable.mcgill.ca/publications/techreports`.

[BLQ+03] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 103–114. 2003.

[BMS02] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The `<bigwig>` project. *ACM Transactions on Internet Technology (TOIT)*, 2(2):79–114, 2002.

[BNL03] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Simple and Efficient Relational Querying of Software Structures. In Arie van Deursen, Eleni Stroulia, and Margaret-Anne D. Storey, editors, *10th Working Conference on Reverse Engineering (WCRE 2003), 13-16 November 2003, Victoria, Canada*, pages 216–227. 2003.

[BR01] Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 97–103. 2001.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341. 1996.

[Buc04] Thorsten Buckley. KABA als Fallstudie für das Soot-Framework. Master's thesis, Universität Passau, November 2004.

[BW96] Beate Bollig and Ingo Wegener. Improving the Variable Ordering of OBDDs Is NP-Complete. *IEEE Trans. Comput.*, 45(9):993–1002, 1996.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19. 1999.

[CM] Brendon Cahoon and Kathryn S. McKinley. JOlden Benchmarks. `ftp://ftp.cs.umass.edu/pub/osl/benchmarks/jolden.tar.gz`.

[CM87] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag New York, Inc., 1987.

[CM01] Brendon Cahoon and Kathryn S. McKinley. Data Flow Analysis for Software Prefetching Linked Data Structures in Java. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, Washington, DC, USA, 2001.

[CMS03] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, November 2003.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cos03] Pascal Costanza. Dynamically scoped functions as the essence of AOP. *SIGPLAN Not.*, 38(8):29–36, 2003.

[DaC] DaCapo Project. The DaCapo Benchmark Suite. `http://www-ali.cs.umass.edu/DaCapo/gcbm.html`.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 35–46. 2000.

[DDHV03] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 149–168. 2003.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP '95, Object-Oriented Programming: 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, 1995.

[DGH+04] Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 150–169. 2004.

[DMM96] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–305. 1996.

[Duf04] Bruno Dufour. Objective Quantification of Program Behaviour using Dynamic Metrics. Master's thesis, McGill University, June 2004.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256. 1994.

[FFSA98] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 85–96. 1998.

[FHC01] Hoda Fahmy, Richard C. Holt, and James R. Cordy. Wins and Losses of Algebraic Transformations of Software Architectures. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*, pages 51–62. 2001.

[GC01] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, November 2001.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 108–124. 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[Gil] David Gilbert. JFreeChart. `http://www.jfree.org/jfreechart/index.php`.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[GMN+] Etienne M. Gagnon, Ben Menking, Mariusz Nowostawski, Komivi Agbakpem, and Kis Gergely. SableCC parser generator. `http://sablecc.org/`.

[GMUW01] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[GS00] David Gay and Bjarne Steensgaard. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In *International Conference on Compiler Construction (CC'2000)*, volume 1781 of *Lecture Notes in Computer Science*. 2000.

[Hei99] Nevin Heintze. Analysis of large code bases: the compile-link-analyze model, 1999. `http://cm.bell-labs.com/cm/cs/who/nch/cla.ps`.

[HH98] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 97–105. 1998.

[Hin01] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61. 2001.

[Hip] D. Richard Hipp. SQLite: An Embeddable Database Engine. `http://www.sqlite.org/`.

[HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 254–263. 2001.

[IKY+00] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a Java Just-In-Time compiler. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 294–310. 2000.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP'97 — Object-Oriented Programming: 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, 1997.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java Points-to Analysis Using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003.

[LH04] Ondřej Lhoták and Laurie Hendren. Jedd: a BDD-based relational extension of Java. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 158–169. 2004.

[Lho02] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[LLH05] Anatole Le, Ondřej Lhoták, and Laurie Hendren. Using inter-procedural side-effect information in JIT optimizations. In R. Bodik, editor, *Compiler Construction, 14th International Conference*, volume 3443 of *LNCS*, pages 287–304, Edinburgh, April 2005.

[LLW03] Karl Lieberherr, David H. Lorenz, and Pengcheng Wu. A case for statically executable advice: checking the law of demeter with AspectJ. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 40–49. 2003.

[LN] Jørn Lind-Nielsen. BuDDy, A Binary Decision Diagram Package. `http://www.itu.dk/research/buddy/`.

[LPH01] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for Java. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79. 2001.

[LRZ93] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 56–67. 1993.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[Man03] Roman Manevich. Data Structures and Algorithms for Efficient Shape Analysis. Master's thesis, Tel-Aviv University, School of Computer Science, Tel-Aviv, Israel, January 2003.

[MH02] Jerome Miecznikowski and Laurie Hendren. Decompiling Java Bytecode: Problems, Traps and Pitfalls. In *Compiler Construction: 11th International Conference, CC 2002*, volume 2304 of *Lecture Notes in Computer Science*, pages 111–127, 2002.

[Mil03] Ana Milanova. *Precise and Practical Flow Analysis of Object-Oriented Software*. PhD thesis, Rutgers University, August 2003.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Conference on Design Automation*, pages 530–535. 2001.

[MRF+02] R. Manevich, G. Ramalingam, J. Field, D. Goyal, and M. Sagiv. Compactly Representing First-Order Structures for Static Analysis. In *Static Analysis: 9th International Symposium, SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 196–212, 2002.

[MRR02] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–11. 2002.

[MRR05] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[MS97] S. Minato and F. Somenzi. Arithmetic Boolean Expression Manipulator Using BDDs. *Formal Methods in System Design*, 10(2/3):221–242, 1997.

[MS03] Erik Meijer and Wolfram Schulte. Unifying Tables, Objects, and Documents. In *Workshop on Declarative Programming in the Context of Object-Oriented Languages*, pages 145–166, August 2003.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[NCM03] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *12th International Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 138–152, 2003.

[Nil] Marcus Nilsson. GBDD – A package for representing relations with BDDs. `http://user.it.uu.se/~marcusn/projects/rmc/docs/gbdd/index.html`.

[NKH04] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of the ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 43–48. 2004.

[NS01] Matthias Neubauer and Michael Sperber. Down with Emacs Lisp: dynamic scope analysis. In *ICFP '01: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 38–49, New York, NY, USA, 2001.

[PKH04] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for C. In *Proceedings of the ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 37–42. 2004.

[Pos] Jef Poskanzer. thttpd: tiny/turbo/throttling HTTP server. `http://www.acme.com/software/thttpd/`.

[Qia] Feng Qian. SableJBDD, a Java Binary Decision Diagram Package. `http://www.sable.mcgill.ca/~fqian/SableJBDD/`.

[RMR01] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of the OOPSLA '01 Conference on Object-Oriented Programming Systems Languages and Applications*, pages 43–55. 2001.

[Ruf00] Erik Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 208–218. 2000.

[Ryd03] Barbara G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 126–137. 2003.

[Sd03] Damien Sereni and Oege de Moor. Static analysis of aspects. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 30–39. 2003.

[SDDS86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets - an Introduction to Setl*. Springer, New York, 1986.

[SdML04] Ganesh Sittampalam, Oege de Moor, and Ken Friis Larsen. Incremental execution of transformation specifications. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 26–38. 2004.

[SH97] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14. 1997.

[Shi88] O. Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 164–174. 1988.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 264–280. 2000.

[SL04] Therapon Skotiniotis and David H. Lorenz. Cona: aspects for contracts and contracts for aspects. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 196–197. 2004.

[Som] Fabio Somenzi. CUDD: CU Decision Diagram Package. `http://vlsi.colorado.edu/~fabio/CUDD/`.

[Soo] Soot: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S Muchnick and Neil D Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.

[Sta] Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. `http://www.spec.org/osg/jvm98/`.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41. 1996.

[Tar72] R. E. Tarjan. Depth First Search and Linear Graph Algorithms. *Journal of Computing*, 1(2):146–160, 1972.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[THY93] Seiichiro Tani, Kiyoharu Hamaguchi, and Shuzo Yajima. The Complexity of the Optimal Variable Ordering Problems of Shared Binary Decision Diagrams. In *ISAAC '93: Proceedings of the 4th International Symposium on Algorithms and Computation*, pages 389–398, London, UK, 1993.

[TLSS99] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–305. 1999.

[TP00] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293. 2000.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press, 1989.

[Vah] Arash Vahidi. Arash's Java interface to BDDs. `http://www.chl.chalmers.se/~vahidi/bdd/bdd.html`.

[VR] Raja Vallée-Rai. Ashes Suite Collection. `http://www.sable.mcgill.ca/ashes/`.

[VR01] Frédéric Vivien and Martin Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 35–46. 2001.

[VRGH+00] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34, 2000.

[Wel84] Terry A. Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, 17(6):8–19, 1984.

[WF00] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java implementations*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2000.

[Whaa] John Whaley. bddbddb. `http://bddbddb.sourceforge.net`.

[Whab] John Whaley. JavaBDD. `http://javabdd.sourceforge.net`.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 1–12. 1995.

[WL02] John Whaley and Monica S. Lam. An Efficient Inclusion-Based Points-To Analysis for Strictly-Typed Languages. In *Static Analysis: 9th International Symposium, SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 180–195, 2002.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of*

the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, pages 131–144. 2004.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206. 1999.

[WS01] Tiejun Wang and Scott F. Smith. Precise Constraint-Based Type Inference for Java. In *ECOOP 2001 — Object-Oriented Programming: 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 99–117, 2001.

[ZC04] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 145–157. 2004.

[Zhu02] Jianwen Zhu. Symbolic pointer analysis. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 150–157. 2002.

[ZM03] L. Zhang and S. Malik. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In *Proceedings of Design, Automation and Test in Europe (DATE2003)*, pages 880–885, 2003.