SPARK: A FLEXIBLE POINTS-TO ANALYSIS FRAMEWORK FOR JAVA

by
Ondřej Lhoták

School of Computer Science
McGill University, Montreal

December 2002

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE

Copyright © 2003 by Ondřej Lhoták

Abstract

Many compiler analyses and optimizations require precise information about the behaviour of pointers in order to be effective. Points-to analysis is a technique for computing this information that has been studied extensively over the last decade. Most of this research has focused on points-to analyses for C. The behaviour of points-to analysis on higher-level languages such as Java appears very different than on C. Moreover, most proposed points-to analysis techniques were evaluated in disparate analysis systems and benchmarks, making it difficult to compare their effectiveness.

To address these issues, this thesis introduces SPARK, a flexible framework for experimenting with points-to analyses for Java. SPARK is intended to be a universal framework within which different points-to analyses can be easily implemented and compared in a common context. Currently, SPARK supports equality- and subset-based analyses, variations in field sensitivity, respect for declared types, variations in call graph construction, off-line simplification, and several points-to set propagation algorithms.

A substantial study of factors affecting precision and efficiency of points-to analyses has been performed as a demonstration of SPARK in action. The results show that SPARK is not only flexible and modular, but also very efficient compared to other points-to analysis implementations.

Two client analyses that use the points-to information are described, call graph construction and side-effect analysis. The side-effect information can be encoded in Java class file attributes, so that it can later be used for optimization by other compilers and virtual machines.

SPARK has been demonstrated to be a flexible and efficient framework for Java points-to analysis. Several experiments that could be performed with it are suggested.

Résumé

Afin d'être efficaces, beaucoup d'analyses et optimisations de compilateur exigent des informations précises sur le comportement des pointeurs. L'analyse dite *points-to* (pointe sur) est une technique visant à calculer cette information qui a été étudiée intensivement au cours de la dernière décennie. La majeure partie de cette recherche s'est concentrée sur les analyses pour C. Le comportement de l'analyse *points-to* appliquée à des langages de plus haut niveau tels que Java semble très différent de celui observé pour C. D'ailleurs, la plupart des techniques d'analyse *points-to* qui ont été proposées ont été évaluées dans des systèmes d'analyse divers et sur les différents programmes d'évaluation, ce qui rend difficile la comparaison de leur efficacité.

Pour répondre à ces problèmes, cette thèse présente SPARK, un cadre d'application flexible pour expérimenter avec des analyses *points-to* pour Java. SPARK est destiné à être un cadre universel dans lequel peuvent être facilement implantées de différentes analyses *points-to*, afin de pouvoir être comparées dans un contexte commun. Actuellement, SPARK supporte des analyses basées sur les contraintes d'égalité ainsi que de sous-ensemble, des variations en le traitement des champs, en le respect pour les types déclarés, et en la méthode de construction du graphe des appels, un algorithme de simplification des contraintes, et plusieurs algorithmes de propagation des ensembles *points-to*.

Une étude importante sur les facteurs influant la précision et l'efficacité des analyses *points-to* a été effectuée comme démonstration de l'utilisation de SPARK. Les résultats démontrent que SPARK est non seulement flexible et modulaire, mais également très efficace comparé à d'autres réalisations d'analyse *points-to*.

Deux analyses clientes qui profitent de l'information *points-to* sont décrites, la

construction du graphe d'appel et l'analyse d'effets secondaires. L'information sur les effets secondaires peut être codé en des attributs dans les fichiers de code objet Java, pour qu'elle puisse être employée à des fins d'optimisation par d'autres compilateurs et machines virtuelles.

Il a été démontré que SPARK est un cadre flexible et efficace pour l'analyse *points-to* de Java. Plusieurs expériences qui pourraient être effèctuées avec SPARK sont suggérées.

Acknowledgments

I am very grateful to my advisor, Laurie Hendren, for leading the Sable research group so well. Her suggestions for my work were plentiful, and always resulted in a significant improvement. Her encouragement and enthusiasm kept me going.

The Sable group has been a pleasant environment in which to work, thanks to all its members. In particular, I want to thank the pointer group of Feng Qian, John Jorgensen, Felix Kwok, Marc Berndl, and Navindra Umanee for the many discussions, Sable group alumni Rhodes Brown, Patrick Lam, Etienne Gagnon, Jerome Miecznikowski, and Derek Rayside for all the bits of advice, and Bruno Dufour for always being eager to help with whatever needs to be done.

I learned a lot during my M.Sc. work, in my courses as well as in my research. Thanks to Karel Driesen, Doina Precup, Laurie Hendren, Xiao-Wen Chang, and Prakash Panangaden for teaching them.

This work was supported financially by the taxpayers of Canada through NSERC, and by a Richard H. Tomlinson fellowship.

I cannot forget my time in Waterloo, where I gained the background to start this work. I am thankful to everyone at UW, in the CEMC, and at Watcom.

My family continues to be supportive, even though they are far away. I am especially grateful to my wife Jennifer for coming with me to Montreal, and for her constant friendship, patience, and love.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Motivation

Accurate information about the behaviour of pointers is a prerequisite for many analyses and optimizations of programs written in languages with pointers. The exact runtime values of each pointer in a program are, in general, uncomputable [Lan92]. Various approximation algorithms have therefore been the subject of active research for over a decade. Unfortunately, these variations were implemented within different compiler frameworks, making them difficult to compare. Moreover, pointer analysis researchers have not yet agreed on an objective metric of the precision of a pointer analysis. Although much work has been done, the problem of efficiently and accurately predicting the behaviour of pointers is far from solved.

In recent years, Java, and other similar languages with dynamic dispatch and strong typing, have been growing in popularity. These language features make the development of software easier and less error-prone, but have significant costs in performance and compiler complexity. Pointer analyses must be adapted to deal with new features not present in simpler languages like C. On the other hand, the type-safety properties of these languages should be exploited to improve efficiency and accuracy of the analysis.

This thesis aims to address these problems by introducing SPARK, a flexible framework for points-to analysis of Java programs, and by reporting on an extensive study of Java points-to analysis variations that was performed using SPARK.

All features of Java are considered by SPARK, making it an ideal framework for experimenting with different representations of these features in pointer analyses. SPARK is designed to be modular, in that different implementations of its various components can be interchanged. This allows experimentation with specific implementation details of pointer analysis algorithms, an area which has been largely neglected in recent pointer analysis research. By setting various parameters within SPARK, and possibly by implementing additional SPARK modules, researchers can easily instantiate efficient implementations of many of the variations of pointer analysis that have been proposed, as well as new variations. This allows the different analyses to be compared within the context of the same framework.

SPARK is a component of the Soot bytecode analysis and optimization framework [Soot, VRGH+00]. The pointer information computed by SPARK can be used by various client analyses within Soot, or it can be encoded in attributes for use by other optimizers, virtual machines, or native compilers. This large collection of possible client analyses provides many different measures of the effectiveness of the pointer analysis.

In addition to describing the SPARK framework itself, this thesis reports the results of a substantial experimental study of Java points-to analyses and the tradeoffs between analysis efficiency and accuracy. These experiments reveal several variations appropriate for Java that provide both precise information and fast analysis times. Furthermore, the experimental results demonstrate that SPARK is not only modular, but its efficiency is very competitive compared to other Java points-to systems described in previously published work.

1.2 Contributions

The work reported in this thesis consists of the design of the SPARK pointer analysis framework, its implementation, and results of experiments performed with it. These three contributions are described in the following subsections.

1.2.1 Design

Pointer Assignment Graph

SPARK introduces the notion of a *pointer assignment graph* (described in detail in Section 4.1), a single model in which very different pointer analyses can be expressed and efficiently implemented. This is in contrast to the many incomparable representations typically used to present different pointer analyses in the literature.

The pointer assignment graph allows the following variations of pointer analyses to be expressed:

- subset-based [And94] or equality-based [Ste96b];
- varying levels of context-sensitivity;[1]
- field and array references merged for all object instances (*field-based* analysis), or considered separately for each instance (*field-sensitive* analysis);
- variables in SSA form [AWZ88], split along UD-DU webs [Muc97, Section 8.10], or as in original source;
- which declared types and casts (if any) are respected;
- whether an initial approximation to the call graph is used, or whether the call graph is constructed as the pointer information is computed; and

[1] Although currently only context-insenstive analyses are implemented, SPARK is designed to facilitate experimentation with context-sensitivity.

- if an initial call graph is used, which approximation (such as class hierarchy analysis [DGC95], rapid type analysis [BS96], or variable type analysis [SHR+00]) is used to compute it.

Staged Analysis

The pointer analysis in Spark proceeds in three stages.

1. The pointer assignment graph is built based on the program being analyzed.
2. The pointer assignment graph is simplified.
3. The simplified pointer assignment graph is used to compute points-to information.

This division into stages is key to the flexibility of Spark. A large number of combinations of different implementations of each stage are possible, leading to many variations in the pointer analysis. The stages of Spark are described in detail in Chapter 4.

1.2.2 Implementation

The current version of Spark includes the following implementations of its components.

- A context-insensitive implementation of the pointer assignment graph builder with many parameters which determine how language features are represented. The pointer assignment graph builder is described in detail in Section 4.2.
- Implementations of simplification algorithms to merge strongly connected components and single-entry subgraphs. Simplification of the pointer assignment graph is described in detail in Section 4.3.
- Five different implementations of points-to set propagation algorithms: a simple iterative algorithm, an efficient worklist-based algorithm, a new, space-efficient

alias edge algorithm, and incremental versions of the worklist and alias edge algorithms. These algorithms are presented in Section 4.4.

- Four different implementations of points-to sets: an implementation based on hash tables, an implementation based on bit vectors, an implementation based on sorted arrays, and a hybrid implementation which represents the elements of small sets explicitly, but switches to bit vectors to represent larger sets. These implementations of points-to sets are described in more detail in Section 4.5.

- Two client analyses that use the results of SPARK have been implemented: a call graph trimmer, and a side-effect analysis. The results of these client analyses are further used by other analyses within Soot, or they can be encoded in attributes for use by other optimizing compilers. These client analyses are described in Chapter 6.

1.2.3 Experiments

The SPARK framework was used for an extensive empirical study of factors affecting precision and efficiency of subset-based Java points-to algorithms. The following factors were studied:

- respecting declared types and casts during the analysis;

- constructing an initial call graph prior to the analysis, or constructing it during the analysis as points-to sets become available;

- modelling of field dereference expressions in a *field-sensitive* or *field-based* manner;

- implementation of points-to set data structures;

- several points-to set propagation algorithms; and

- off-line simplification of the pointer assignment graph prior to propagation.

From the results of these experiments, three analysis variations were selected as appropriate compromises between analysis precision and efficiency. The experiments showed the performance of SPARK on these variations to be very competitive compared to other Java points-to analyses that have been described in the literature.

1.3 Thesis Organization

The rest of this thesis is organized as follows. The next chapter is a survey of related work. Chapter 3 provides an overview of the overall design of SPARK, and of the Soot framework of which it is a part. Chapter 4 gives a detailed description of the design of the pointer analysis engine, the core of SPARK. A description of the pointer assignment graph is given first, followed by descriptions of the stages which SPARK uses to compute pointer information. Results of experiments conducted with SPARK are reported in Chapter 5. Client analyses that use the results computed by SPARK are described in Chapter 6. Finally, Chapter 7 concludes this work, and provides many examples of research to which SPARK could be applied in the future.

Chapter 2
Related Work

This chapter presents previous work on points-to analysis. The first section covers early work leading to points-to analysis. The second section is an overview of the techniques that have been used in the past to improve the efficiency and precision of points-to analyses. The third section explains the work that has been done so far to adapt points-to analyses designed for C to Java. The fourth section discusses the applications for which points-to information has been used, concentrating primarily on applications related to Java. An extensive survey of points-to analysis research, with a particular focus on the problems that remain unsolved, is given by Hind [Hin01].

2.1 Early Work on Alias and Points-To Analysis

The earliest work [Wei80, CR82, Cou86, LR92, CBC93] on estimating the sets of locations to which pointers could point used an alias set representation. This representation encodes the set of pairs of variables which could point to the same memory location. One such set of alias relationships can be computed for the program as a whole, or a separate alias set can be computed for each program point. One difficulty with this representation is that its size can be quadratic in the number of variables in the program. Another drawback is that alias sets do not give information about the objects to which pointers point, such as their type; rather, they only specify which pairs of variables may point to the same objects.

To address these problems, Emami, Ghiya and Hendren [EGH94] introduced *points-to* analysis. A points-to analysis divides memory into concrete locations. Then, for each variable, it computes the set of concrete locations to which that variable may point. Alias sets can be recovered from points-to sets: a pair of variables is aliased whenever their points-to sets have a non-empty intersection. However, for many applications, it is more convenient to use points-to sets without first constructing alias sets.

Emami, Ghiya and Hendren's implementation used a separate concrete location for each stack variable, and modelled the entire heap as a single concrete location. The analysis was context-sensitive and flow-sensitive. For stack-directed pointers, it computed not only may points-to information, but also must points-to information, and used it to improve the precision of the flow-sensitive analysis by removing old points-to relationships when a variable was known to be overwritten. When analyzing C, function pointers present a challenge because they make it difficult to determine the targets of calls through them. The points-to analysis treated each function as a concrete location, so the set of possible targets of a call through a function pointer was simply the points-to set.

Andersen [And94] proposed a flow-insensitive, context-insensitive version of points-to analysis that did not compute must points-to information. However, his analysis modelled the heap more precisely, using a separate concrete location to represent all memory allocated at a given dynamic allocation site. The implementation expressed the analysis using subset constraints, and then solved the constraints.

Solving a system of set constraints such as those generated by Andersen's analysis is equivalent to finding the transitive closure of the constraint graph, and a typical implementation may therefore take time cubic in the size of the program. Steensgaard [Ste96b] proposed a more conservative analysis by replacing each subset constraint with a set equality constraint. The advantage of this approach is that it reduces the problem to one of finding connected components in the constraint graph, which can be done in almost linear time using a fast union-find algorithm [Tar75]. However, the stronger constraints make the analysis much less precise. In fact, for Java programs, the constraint graph is fully connected, because every object is passed to the initializer of `java.lang.Object`, so an unmodified version of Steensgaard's algorithm would produce the worst-case assumption that every variable may point to every object.

2.2 Improving Analysis Efficiency

Since the introduction of subset-based and equality-based points-to analysis, researchers have worked on improving the efficiency of the former, and the precision of the latter.

Wilson and Lam [WL95] implemented a flow-sensitive, context-sensitive subset-based analysis using *partial transfer functions* to summarize the effect of each function on points-to sets. This meant that their analysis did not have to analyze each function for every calling context; rather, it only had to apply the partial transfer function in each calling context. The analysis could therefore be more efficient than the flow-sensitive, context-sensitive analysis of Emami, Ghiya, and Hendren.

Ruf [Ruf95] advocated abandoning context-sensitivity altogether. He implemented both context-insensitive and maximally context-sensitive versions of a subset-based analysis. On his benchmark suite, the context-insensitive version produced only a small number of spurious points-to relationships compared to the context-sensitive version. Moreover, when he applied the points-to results to computing side-effect information, the few spurious points-to relationships introduced even fewer spurious side-effects.

Shapiro and Horwitz [SH97b] studied flow-insensitive, context-insensitive points-to analyses. They presented empirical results demonstrating that an equality-based analysis is considerably less precise than a subset-based analysis, but that the subset-based analysis is much slower on larger programs. In addition, they presented a points-to analysis algorithm with a parameter which could be adjusted to make the analysis faster at the expense of precision. The idea was to separate the variables in the program into k categories. When two variables were in the same category, constraints between them were treated as equality constraints; only variables in different categories could have subset constraints between them. Using a separate category for each variable resulted in a fully subset-based analysis, while assigning all variables to a single category resulted in a fully equality-based analysis. The analysis could be tuned between these two extremes by using an intermediate number of categories.

Hasti and Horwitz [HH98] used static single assignment (SSA) form [AWZ88] to obtain precision comparable to a flow-sensitive points-to analysis from a much faster, flow-insensitive points-to analysis. The main benefit of a flow-sensitive analysis is strong update: when a variable is overwritten, the analysis can infer that after being overwritten, the variable no longer points to the objects it used to point to. A flow-insensitive analysis ignores the order in which assignments are executed; it has no way to distinguish between "before" and "after" the assignment. When a program is converted into SSA form, its variables are split so that each variable is assigned at only one point in the program. This means that in SSA form, no variable is ever overwritten. A variable which is overwritten in the original program is represented by two or more separate variables in SSA form. In SPARK, all analyses are flow-insensitive, but before starting the analysis, SPARK uses the Soot framework to split variables along UD-DU webs [Muc97, Section 8.10], a slight relaxation of SSA form. A Soot transformation to true SSA form has been written, and is expected to soon be merged into the publicly available version of Soot.

Diwan, McKinley, and Moss [DMM98] applied points-to analysis to Modula-3, which enforces declared types, unlike C. They studied three simple alias analyses. The first analysis was to treat variables as possibly aliased whenever the type of one variable is a subtype of the other. The second analysis added the constraint that a field of an object may only be aliased to that same field of another object. Finally, the third was an equality-based analysis similar to Steensgaard's. The results of the alias analysis were used to compute side-effect information, which was used to remove redundant loads. Their analysis was able to remove between 37% and 87% of the redundant loads in the program, resulting in a 1% to 8% speedup. The simplest analysis which considered only declared types managed to detect nearly all of the redundant loads detected by the other two more precise analyses. Experiments conducted using SPARK show that information provided by declared types such as that used by Diwan, McKinley, and Moss can significantly improve analysis precision and efficiency of more complicated analyses.

Aiken, Fähndrich, Foster, and Su [AFFS98, FFSA98, SFA00] developed a framework called BANE for solving general subset constraint problems. In particular, the framework can be used to solve points-to analysis problems that can be expressed using set constraints. Their framework is able to detect and collapse cycles in the constraint graph as it is solving it, improving the efficiency of subset-based analyses.

Rountev and Chandra [RC00] observed that the initial subset constraint graph may contain cycles or subgraphs with a single entry point, and that when analyzing C programs, the points-to sets of all nodes in a cycle or in a single entry subgraph will be equal.[1] They therefore proposed simplifying the graph by merging variables known to have equal points-to sets before starting to solve the constraints. On their C benchmarks, they found that simplifying the constraint graph before solving it improved the solution time and memory requirements by about 50%. Spark includes a similar algorithm to simplify its pointer assignment graph, and empirical results from Spark agree with those of Rountev and Chandra.

Das [Das00] noticed that in C programs, many pointers are only used to implement call-by-reference, and that it is relatively inexpensive to analyze these pointers with a subset-based analysis. He therefore proposed an analysis that uses subset constraints between stack variables that do not have their address taken, and equality constraints between other variables. The pointers used to implement call-by-reference rarely have their address taken, so they are analyzed quickly with great precision by a subset-based analysis. The remaining pointers, which could slow down a subset-based analysis, are analyzed using the imprecise but inexpensive equality constraints. Using this analysis, Das was able to analyze a large program consisting of about two million lines of code.

Heintze and Tardieu [HT01a, HT01b, Hei99] report analyzing huge programs with a fully subset-based analysis. This efficiency appears to be due to three main factors. First, their analysis is demand-driven, producing only those points-to sets needed by

[1] In an analysis for Java, it is not necessarily true that the points-to sets of all nodes in a cycle or single entry subgraph will be equal if declared types are being respected. See Section 4.3 for details.

a client of the analysis, rather than producing the entire solution at once. Second, it uses an algorithm that detects and merges cycles in the constraint graph as the analysis proceeds. Third, their representation of points-to sets has been carefully tuned, and is very efficient. It is not clear which of these three factors contribute most significantly to the speed of their system; however, their work shows that a combination of the three makes it feasible to perform subset-based analyses for very large programs.

2.3 Points-To Analysis for Java

With the exception of the work by Diwan, McKinley, and Moss, the points-to analyses discussed so far were designed to analyze programs written in C. Java has several features not present in C that affect points-to analysis. Specifically, Java disallows only stack-directed pointers, it enforces declared types, and it uses virtual dispatch, so a static call graph is not immediately available, as it is in C in the absence of function pointers. This is especially problematic because Java includes a very large standard class library which cannot be left out of the call graph, making even trivial programs appear very large from the point of view of whole-program analysis. Several researchers have tried to adapt points-to analyses to reflect these features specific to Java.

Liang, Pennings and Harrold [LPH01] performed a comparison of several different analyses adapted to Java. All of their analyses were flow-insensitive and context-insensitive. Because their implementation could not scale to analyzing the complete standard library of version 1.1.8 of the JDK, they used hand-coded summaries of the pointer-related effects of the library. They studied both field-sensitive and field-based analysis of field expressions. In a field-sensitive approach, a separate points-to set is computed for each field of each concrete location, while in a field-based approach, only a single points-to set is computed for each field. A field-sensitive approach can distinguish between the same field of two different objects, while a field-based approach cannot. They also compared both equality-based and subset-based analyses. After noticing that a completely equality-based analysis applied to Java produces the worst-case information that every pointer may point to every object, they modified the equality-based analysis to be subset-based in the areas that degraded precision the most. Finally, they also compared using a call graph precomputed using class hierarchy analysis [DGC95] to constructing a call graph on-the-fly from the points-to information as it was computed. The precision of these analyses was measured by its impact on the precision of the call graph that could be constructed from the points-to information, and the precision of escape information that could be computed. They found the subset-based analysis to be significantly more precise than even their modified equality-based analysis, but they did not notice a significant effect on precision from varying the modelling of field references or the method of call graph construction. In their implementation, the field-based analysis using the call graph computed using CHA was considerably faster than the other variations.

Rountev, Milanova and Ryder [RMR01] modified Soot [Soot, VRGH^{+}00] to output subset constraints to be used as input to BANE [AFFS98], which they used to compute a flow-insensitive, context-insensitive, field-sensitive points-to analysis that computed the call graph on-the-fly. They were unsuccessful in expressing an efficient field-based analysis directly in BANE, so they modified BANE to allow each subset constraint to be annotated with a field. Using these field annotations, their analysis was efficient enough to be able to analyze benchmarks with the whole standard library from version 1.1.8 of the JDK. During the analysis, the declared types of variables were not considered; however, objects of incompatible type were removed from the final points-to sets after the analysis completed. They showed using experimental data that their analysis computed precise side-effect information, a precise approximation to the call graph, and precise escape information.

Whaley, Rinard and Vivien [WR99, VR01] used a demand-driven, subset-based, context-sensitive, flow-sensitive, field-sensitive analysis to compute escape information for deciding which objects could be safely allocated on the stack rather than on the heap. As soon as an object was determined to escape, the analysis for that object terminated. This made it possible for such a precise analysis to scale to reasonably-sized programs. Choi *et al.* [CGS+99] presented a very similar escape analysis. They applied it to eliminating synchronization of thread-local objects, in addition to allocating objects on the stack. Bogda and Hölzle [BH99] also used a points-to analysis to compute escape information for eliminating synchronization. The intra-procedural part of their analysis was equality-based, while the inter-procedural part was subset-based, giving a good compromise between analysis efficiency and precision.

Whaley and Lam [WL02] adapted the demand-driven algorithm of Heintze and Tardieu [HT01a, HT01b] to Java by adding field-sensitivity, making it respect declared types, and computing the call graph on-the-fly. With this analysis, they were able to analyze benchmarks using the standard library from version 1.3.1 of the JDK, which is about three times larger than the library in version 1.1.8. However, their implementation did not come close to matching the scalability of Heintze and Tardieu's implementation for C, suggesting that implementation features other than the demand-driven algorithm affect the efficiency of the analysis.

Recently, Milanova, Rountev and Ryder [MRR02a, MRR02b] proposed object-sensitivity, an adaptation of context-sensitivity designed to precisely model features often present in object-oriented programs, such as encapsulation. They applied a preliminary version of their analysis to constructing object relationship diagrams for program understanding, an application for which a high level of precision is needed.

2.4 Applications of Points-To Analysis

This section describes some of the analyses that have been constructed to make use of points-to information. Some of these clients, such as side-effect analysis, have been studied for both C and Java, while others, such as call graph construction and escape analysis are particularly useful for dealing with features specific to Java.

2.4.1 Side-Effect Information

The purpose of a side-effect analysis is to approximate the set of memory locations read and written by specific instructions, and to summarize this information for larger regions of the program. This information can then be used to improve the effectiveness of a wide variety of dataflow analyses and traditional compiler optimizations in the presence of pointers. The side-effect analysis implemented using SPARK is described in Section 6.2 of this thesis.

Ghiya and Hendren [GH98] used side-effect information to improve precision of common subexpression elimination, loop-invariant hoisting, and redundant load elimination in a C compiler. On their benchmarks, these improvements translated into up to 10% speedups. They also showed how to use side-effect information for array dependence testing, in program understanding tools, and to automatically insert data prefetching hints into code. A similar study was done for C programs by Hind and Pioli [HP00]. They evaluated several points-to analyses by measuring their effects on live variable analysis, reaching definitions, constant propagation, and dead code elimination.

Clausen [Cla97] proposed a simple side-effect analysis for Java which did not use a points-to analysis; it used only information about declared types, and made worst-case assumptions about the possible targets of pointers. The resulting side-effect information was applied to dead code removal, loop invariant hoisting, constant propagation, and common subexpression elimination. On early versions of Java, these optimizations produced speedups of up to 25%.[2]

The precision of side-effect information that can be obtained has become a common metric of the precision of points-to information. Both Shapiro and Horwitz [SH97a], and Rountev, Milanova and Ryder [RMR01, MRR02b] used it as one of their main metrics in comparing the precision of different points-to analyses.

[2]Early Java virtual machines did not have aggressive just-in-time compilers like they do today. Modern just-in-time compilers can perform some of these optimizations based on intraprocedural analysis.

2.4.2 Call Graph Construction

In Java, all instance methods are invoked using virtual calls. This means that whole-program analyses require some approximation of the call graph. Some points-to analyses require such a call graph to be constructed prior to the analysis. The output of a points-to analysis can also be used to construct such a call graph, or to make an existing call graph more precise. The application of SPARK to call graph construction is covered in Section 6.1 of this thesis.

Several methods have been proposed for constructing call graphs without using a complete points-to analysis. Dean, Grove, and Chambers [DGC95] proposed *class hierarchy analysis*, which uses only the subclass relationships in the type hierarchy to resolve method targets. Bacon and Sweeney [BS96] introduced *rapid type analysis*, which restricts class hierarchy analysis to classes which appear in allocation sites in the program. Sundaresan *et al.* [SHR$^+$00] proposed an even more precise method, *variable type analysis*, a technique similar to subset-based points-to analysis in that it uses subset constraints to express the possible sets of run-time types of objects that each variable may hold. All of these methods are available in SPARK. Tip and Palsberg [TP00] studies several other variations of call graph construction algorithms based on subset constraints.

A call graph can be constructed almost directly from precise points-to information. It has become common in studies of points-to analyses [LPH01, RMR01, WL02] to use the precision of the call graph that can be constructed as one measure of the precision of the points-to information.

2.4.3 Escape Analysis

The goal of an escape analysis is to determine which objects can be referenced by pointers in methods or threads other than the method or thread in which they are allocated. Research on escape analysis for Java has focused on two main applications, stack allocation and synchronization elimination, which are discussed in the next two paragraphs.

Java forces programmers to allocate all objects on the heap, rather than on the stack. This can have adverse effects on the performance of Java programs, because these objects need to be freed by the garbage collector. Several researchers [WR99, VR01, CGS+99] used escape analyses inside their compilers to detect which objects could safely be allocated on the stack rather than on the heap.

It is very easy to add synchronization locks to Java programs, so many programs and libraries use them extensively even when they are not necessary. Several approaches [BH99, Ruf00, CGS+99] were independently developed to use escape information to reduce the overhead of these locks. All three approaches use escape analysis to determine which objects cannot be referenced by threads other than the thread in which they are allocated. Any locks on such objects can be removed, because these objects are only used by a single thread. Most modern implementations of Java use thin locks [BKMS98], which are extremely efficient when there is no contention over the lock (as is the case for thread-local objects), so it may appear that synchronization elimination is no longer necessary. However, even thin locks become expensive on multi-processor architectures [KKO02].

Chapter 3
Spark in the Context of Soot

3.1 Soot Overview

SPARK is a component of the Soot framework [Soot, VRGH+00] for analyzing, optimizing, and annotating Java bytecode. The Soot framework defines four different intermediate representations, and includes code to convert between them and Java bytecode.

Baf is a stack-based representation similar to bytecode.

Jimple is a stack-less, three-address, typed intermediate representation suitable for many analyses.

Grimp is a representation similar to Jimple, but with aggregated expressions (that is, statements such as `d = (a + b) * c` are allowed, whereas in Jimple, this computation would be split into two statements, one to do the addition, and the other to do the multiplication).

Dava AST is a high-level, structured representation used for decompilation.

The most common use of Soot is for optimizing and annotating bytecode. Soot reads the bytecode (which may be produced by `javac` or any other compiler targetting bytecode) either for a single class file, or a whole program. Soot successively converts the bytecode to its various intermediate representations, and applies analyses, transformations, and annotation generators designed for each intermediate representation. Soot provides a mechanism [PQVR+01] for attaching attributes with arbitrary analysis results to classes, methods, or individual instructions. Finally, the intermediate representation is translated back to bytecode, annotated with any of the attributes that were attached, and written back to class files.

Of the intermediate representations defined by Soot, Jimple is the most suitable for whole-program points-to analysis. SPARK is therefore based entirely on Jimple. Jimple statements relevant to points-to analysis are explained below.

Assignment statement: An assignment statement has the form `p = q`, and assigns the value of one variable to another. If the variables are of pointer type, a points-to analysis must consider that after this statement, the target of the assignment may point to the object that the source of the assignment points to.

Identity statement: Jimple introduces virtual variables to represent the parameters of methods and the parameter of an exception handler. These variables are present only implicitly in the original bytecode. An identity statement is an assignment statement with one of these virtual variables as its source rather than an ordinary variable. For example, every instance method contains a statement like `p := @this`, which assigns the implicit parameter `this` to the variable `p`. SPARK treats identity statements in the same way as other assignment statements.

Allocation statement: From the point of view of a points-to analysis, an allocation statement is any statement that causes a variable to point to some newly-allocated location. In Jimple, this includes statements that allocate objects and arrays (single and multi-dimensional), and that load string constants. In Jimple, the call to a constructor that is associated with an object being created is not part of the allocation statement; it is represented as a separate invocation statement. Some examples of allocation statements are:

- `p = new java.lang.String`,
- `q = newarray (int)[12]`, and
- `r = "Hello, World!"`.

Field store: A field store has the form `p.f = q`, and stores the value of the variable `q` into the field `f` of the object pointed to by `p`.

Field load: A field load has the form `p = q.f`, and loads the value of the field `f` of the object pointed to by `q` into the variable `p`.

Static field store: A static field store has the form `Class.field = p`, and stores the value of a variable into a static field of a class. Static fields are the Java equivalent of global variables. Each static field is associated with a class, and there is a single instance of each static field in the whole program.

Static field load: A static field load has the form `p = Class.field`, and loads the value of a static field into a variable.

Array store: An array store has the form `p[i] = q`, and stores the value of variable `q` into the `i`th element of the array pointed to by the variable `p`. In SPARK, arrays are treated like objects, with a single virtual field representing all the elements of the array.

Array load: An array load has the form `p = q[i]`, and loads the value of the `i`th element of the array pointed to by `q`.

Cast statement: A cast statement has the form `p = (T) q`, and causes the pointer stored in the variable `q` to be assigned to the variable `p`, provided that the type of the target of the pointer is a subtype of `T`. If it is not, the assignment does not take place, and an exception is thrown. A points-to analysis can treat such a cast statement like an assignment from `q` to `p`, but it can also take advantage of knowing that the pointer that is assigned must be pointing to an object whose type is a subtype of `T`.

Invocation statement: An invocation statement causes a method to be invoked. If the method is static, the invocation statement contains a specification of the method that will be invoked. Otherwise, the invocation statement contains a *signature* of the method to be invoked, as well as a variable pointing to the receiver object of the method. The actual method that will be invoked is resolved from the run-time type of the receiver object and the method signature. If the method accepts parameters, the invocation statement contains variables whose values will be passed to the parameters of the method. If the method returns a value, the invocation statement may optionally contain a target variable to which the return value will be assigned when the method returns. Any of these variables may be of pointer type, so a points-to analysis must consider the resulting flow of pointers. Some examples of invocation statements are:

- `p = staticinvoke <java.lang.String: valueOf(int)>(5)`,
- `i = virtualinvoke s.<java.lang.String: length()>()`,
- `virtualinvoke p.<java.io.PrintStream: close()>()`,
- `specialinvoke this.<java.lang.Object: void <init>()>();`, and
- `i = interfaceinvoke c.<java.util.Collection: int size()>();`.

Return statement: A return statement has the form `return` or `return p`, and causes control to return from a method back to its caller, optionally passing back a value. At the call site, the returned value may be assigned to a variable, or discarded if no target variable is specified. If the value being returned is of pointer type, a points-to analysis should should take the pointer flow into account.

Throw statement: A throw statement has the form `throw p`, and transfers control to an exception handler, passing it a pointer to an exception object (`p`, in this case). Each exception handler contains an identity statement that retrieves the exception object from the implicit parameter variable. A points-to analysis should track the pointer flow from the throw statement to the parameter of the exception handler. In Spark, this is currently done by representing all thrown exceptions as assignments to a single variable holding all thrown exception objects, and by assignments from this variable to the parameters of each exception handler. This method of handling exceptions is based on the conservative assumption that any exception thrown may be caught by any handler in the program. Because exception handlers are usually very short, and because very few objects are usually passed through thrown exceptions, this approximation appears not to degrade the precision of the points-to information. Soot could be extended to provide more precise information about which exception handlers catch which exception, and this information could then be used by Spark.

3.2 Spark within Soot

Figure 3.1 shows how Spark interacts with other components within Soot. The core component of Spark is the *pointer analysis engine*, described in detail in Chapter 4. It takes as its input the Jimple representation of the whole program, optionally a conservative call graph, and a simulated representation of any native methods used by the program. The initial call graph may be created using class hierarchy analysis [DGC95], rapid type analysis [BS96], or variable type analysis [SHR$^+$00]. Spark can also operate without an initial call graph, and generate one on-the-fly based on the points-to information that it computes. The output of the pointer analysis engine is, for each variable of reference type in the program, an abstract set of locations to which the variable may point.

The points-to information is used by client analyses, such as a call graph trimmer, which removes extraneous edges from the call graph, and a side-effect analysis, which computes the locations possibly read or written by the statements and methods of the program. These two client analyses are presented in more detail in Chapter 6. Other analyses, such as escape analysis, could be implemented.

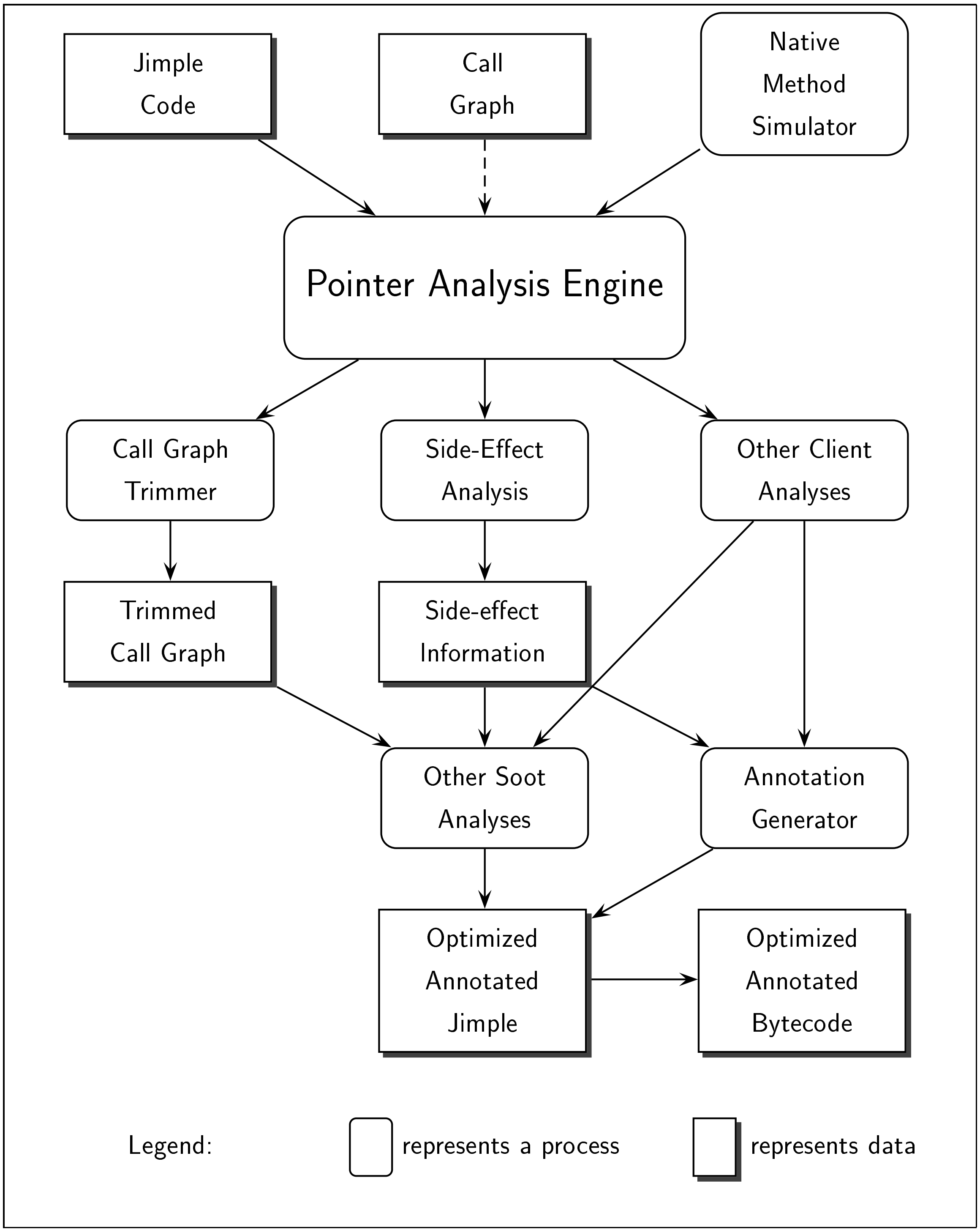


Figure 3.1: How SPARK Interacts with Soot

The results of the client analyses can be used by other analyses and transformations within Soot. For example, the static method binder and static inliner use the trimmed call graph, while the common subexpression eliminator and partial redundancy eliminator use the side-effect information.

The results of the client analyses can also be encoded as attributes in the Jimple code, which are transferred to class file attributes when the Jimple is translated back to Java bytecode. The information in these attributes can be used by another compiler or interpreter reading the resulting bytecode. For example, a just-in-time compiler executing the bytecode could use the side-effect information computed by Soot. This is an important use of SPARK because points-to analysis, and the analyses that depend on it, are generally considered to be too time-consuming to be included in just-in-time compilers.

Chapter 4
Pointer Analysis Engine

This chapter describes the *pointer analysis engine*, the core component of SPARK. Figure 4.1 shows the overall organization of the pointer analysis engine. The analysis consists of three stages: building the pointer assignment graph, simplifying it, and then propagating the points-to sets along it to obtain the final solution. These stages are described in more detail in the rest of this chapter. A pointer assignment graph builder is first used to convert the input Jimple representation into the internal representation used by SPARK, a pointer assignment graph. The graph builder determines how features of the program, such as field references, array element references, and parameters passed to methods are represented. It is described in more detail in Section 4.2. The pointer assignment graph may then be simplified by merging nodes that are known to have the same points-to sets. This simplification reduces the amount of processing required to compute the points-to sets. It is described in more detail in Section 4.3. Finally, the points-to set propagator computes the points-to set for each variable by propagating sets along assignments in the program (which are represented by edges in the pointer assignment graph). The points-to set propagation algorithms implemented in SPARK are described in detail in Section 4.4.

By tuning parameters of the builder, simplifier, and propagator (or by providing alternative implementations), we can control the precision and efficiency of the points-to analysis. For example, to implement a merge-based analysis, we instruct the builder to use bi-directional edges, and the simplifier to merge the nodes connected by these

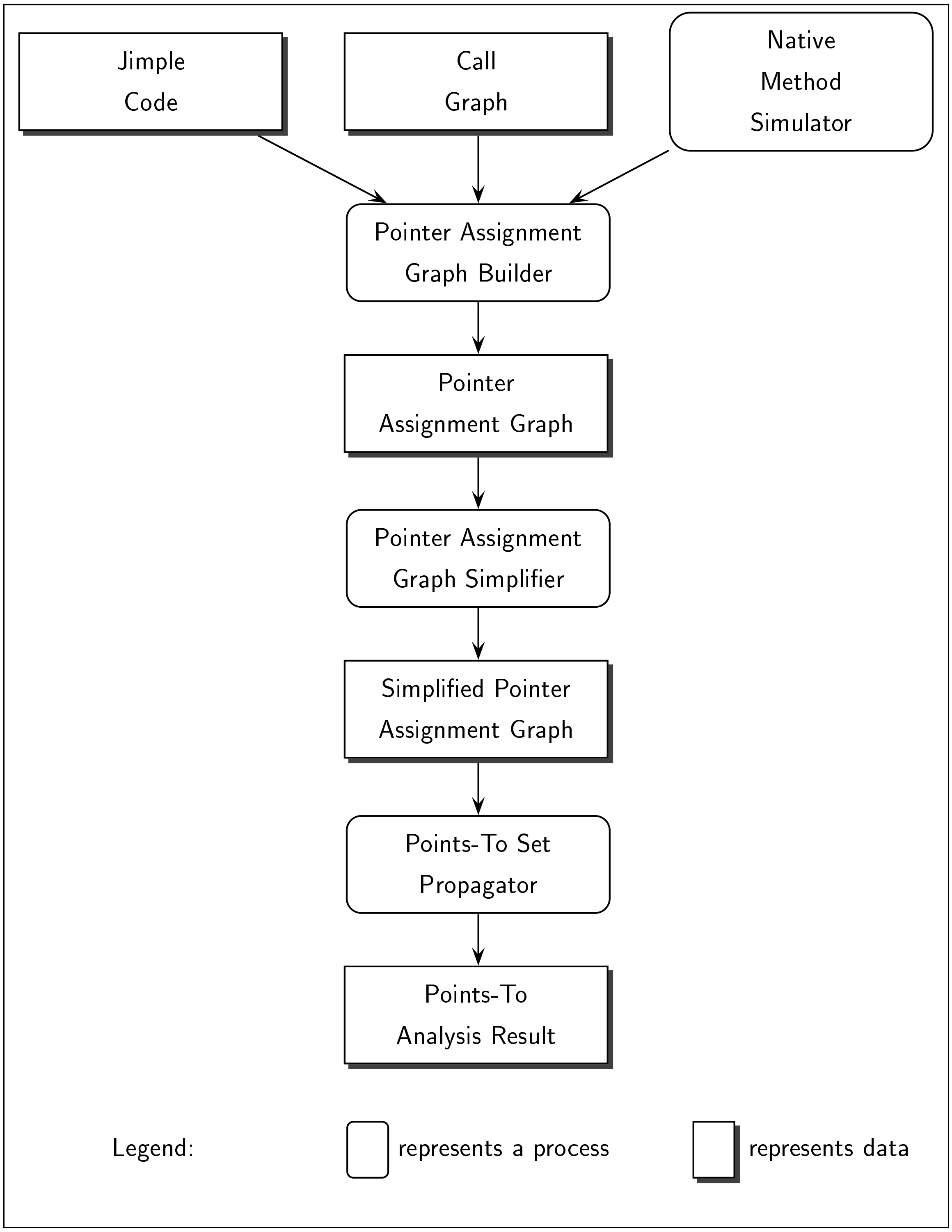


Figure 4.1: Spark Overview

edges, leaving a trivial amount of computation for the propagator. On the other hand, a subset-based analysis would enable little merging in the simplifier, leaving most of the computation to be done by the propagator. Integrating all three components in the Spark framework makes it feasible to implement and compare analyses sharing characteristics of the two extremes.

4.1 Pointer Assignment Graph

Spark uses a *pointer assignment graph* as its internal representation of the program being analyzed. The first stage of Spark, the pointer assignment graph builder, constructs the pointer assignment graph from the Jimple input. Depending on the parameters to the builder, the pointer assignment graph for the same source code can be very different, reflecting varying levels of precision desired of the points-to analysis. For example, the builder may make assignments directed for a subset-based analysis, or bi-directional for an equality-based analysis. Separating the builder from the solver makes it possible to use the same solution algorithms and implementations to solve different variations of the points-to analysis problem.

The pointer assignment graph represents the memory locations used by the program using four different types of nodes, and assignments of pointers using four different types of edges. These are presented in the following subsections.

4.1.1 Allocation Nodes

new 1

An allocation node represents a set of run-time objects to which a pointer could point. The current design of Spark requires the sets of run-time objects represented by allocation nodes to be disjoint; that is, each object at run-time is represented by exactly one allocation node. To satisfy this requirement, the builder may use an allocation node to represent all objects allocated at a given allocation site (since every object is allocated at exactly one allocation site), or to represent all objects with a given run-time type (since every object has exactly one run-time type).

Each allocation node has an associated type, and all objects that it represents are expected to have exactly this type at run-time (not a subtype). For the case of an allocation node representing a set of objects of multiple run-time types, or whose type cannot be determined statically, SPARK introduces a special type `AnyType`. Allocation nodes with this type can represent objects of any run-time type.

4.1.2 Variable Nodes

p

A variable node represents a set of memory locations possibly holding pointers to objects. SPARK eventually computes, for each variable node, a set of allocation nodes representing the set of objects to which a member of the set of memory locations represented by the variable node may point. The most common use of variable nodes is to represent local variables and method parameters, but they are also used to represent static fields, and they may be used to represent instance fields if the instances of a field are being modelled together in a field-based analysis.

Depending on a parameter to the builder, each variable node may have a declared type limiting the set of objects that it may point to to those of compatible run-time type.

4.1.3 Field Reference Nodes

p.f

A field reference node represents a pointer dereference. Each field reference node has an associated variable node as its *base*, and an abstract *field*. The field reference node represents all memory locations used to store the given field of all objects pointed to by the base. The field may be an actual Java field, or the special *elements* field used to represent elements of an array. Note that Java field references need not always be modelled using field reference nodes; if instances are being modelled together, field references are represented by variable nodes.

Like the variable node, each field reference node may have a declared type limiting the set of objects to which it may point.

4.1.4 Concrete Field Nodes

new 1.f

Later, during the propagation of points-to sets, a fourth type of node is created to hold the points-to set of each field of objects created at each allocation site. These nodes are parameterized by allocation site and field. However, they are not part of the initial pointer assignment graph.

4.1.5 Allocation Edges

new 1 → p

An allocation edge is an edge from an allocation node to a variable node, and represents an assignment of pointers to the objects represented by the allocation node to the location represented by the variable node. The presence of an allocation edge constrains the points-to information to include the objects represented by the allocation node in the points-to set of the locations represented by the variable node. Examples of Jimple statements for which allocation edges are generated include allocation statements such as `p = new Object();` and loads of string constants, such as `s = "Hello";`.

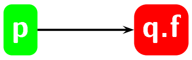
4.1.6 Assignment Edges

p → q

An assignment edge is an edge from a variable node to another variable node, and it represents an assignment from the location represented by the first variable node to the location represented by the second variable node. The presence of an assignment edge from p to q constrains the points-to set of p to be a subset of the points-to set of q. In order to constrain two points-to sets to be equal (for an equality-based analysis, for example), the builder can insert assignment edges in both directions between two nodes. Assignment edges are inserted between nodes whenever the pointers can flow from one variable to another. Examples include explicit assignment statements such as `q = p;`, but also interprocedural flow of parameters to methods.

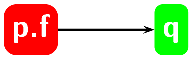
At each call site, assignment edges are added from the nodes representing the actual arguments to the nodes representing the corresponding parameters of all methods that may be targets of the call site, and an assignment edge is added from the return node of each of these methods to the node for the variable that receives the return value (if any) at the call site.

4.1.7 Store Edges



A store edge is an edge from a variable node to a field reference node, and it represents a store from the location represented by the variable node to the appropriate field of some object pointed to by the base of the field reference node. Store edges are added to the pointer assignment graph for store statements in the source, such as `q.f = p;`.

4.1.8 Load Edges



A load edge is an edge from a field reference node to a variable node, and it represents a load from the appropriate field of some object pointed to by the base of the field reference node to the location represented by the variable node. Load edges are added to the pointer assignment graph for load statements in the source, such as `q = p.f;`.

4.1.9 Example

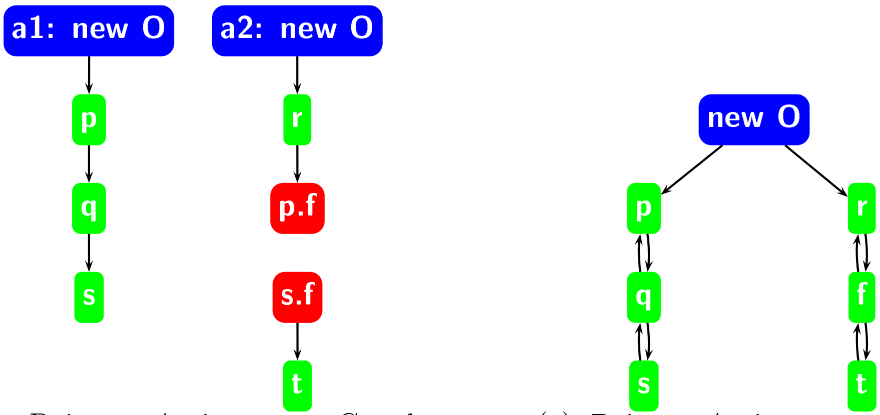
Figure 4.2 shows a small piece of code, and two examples of pointer assignment graphs that could be used to represent it. The code is not intended to do anything specific; it is given only as an example to illustrate how pointer assignment graphs could be built for it.

The first example graph in Figure 4.2(b) would be constructed for a subset-based field-sensitive analysis with separate allocation nodes for objects allocated at each

```
static void foo() {
a1: p = new O();
    q = p;
a2: r = new O();
    p.f = r;
    t = bar( q );
}

static O bar( O s ) {
    return s.f;
}
```

(a) Example Code



(b) Pointer Assignment Graph for Subset-Based, Field-Sensitive Analysis

(c) Pointer Assignment Graph for Equality-Based, Field-Based Analysis

Figure 4.2: Example to Illustrate Pointer Assignment Graphs

allocation site. The edges are therefore only present in the direction of the assignments in the source code. For a field-sensitive analysis, the field references are represented using field reference nodes. Objects allocated at each of the two allocation sites are represented using distinct allocation nodes.

The allocation statements `a1: p = new O()` and `a2: r = new O()` cause the allocation edges from **a1: new O** to **p** and from **a2: new O** to **r**, respectively, to be added. The simple assignment statement `p = q` is modelled by the assignment edge from **p** to **q**. The interprocedural flow from `q` to the parameter `s` of the `bar` method is represented by the assignment edge from **q** to **s**. The store `p.f = r;` is represented by the store edge from **r** to **p.f**, and the flow from `s.f` returned to the variable `t` is represented by the load edge from **s.f** to **t**.

At this point, it is not yet known that `p` and `s` will be aliased, so there are no edges between **p.f** and **s.f**. This flow through aliasing will be handled later, during the points-to set propagation stage, by the propagation algorithms presented in Section 4.4.

The second example graph in Figure 4.2(c) would be constructed by a less precise, equality-based, field-based analysis, with allocation nodes representing all objects of a given run-time type. Because this is an equality-based analysis, all of the assignment edges now go in both directions. Field-based analysis means that the field references are represented by a single variable node **f** not dependent on the base object (`p` or `s`), rather than by field reference nodes **p.f** and **s.f**. In a field-based analysis, we conservatively assume that all objects could be aliased for the purpose of modelling field references; this is reflected by the single variable node representing the field `f` of all objects. Because this analysis represents all objects of a given type by a single node, the objects allocated at the two allocation sites are represented by a single node **new O**, since they are of the same type.

4.2 Building the Graph

The *pointer assignment graph builder* takes as input Jimple intermediate code, a call graph, and simulations of native methods, and produces from them a *pointer assignment graph* containing the same information in a form suitable for performing pointer analysis. This section describes the design of the builder.

4.2.1 Design

The task of the builder can be decomposed into two steps.

1. Iterating through the Jimple input, and determining how the different Jimple features relate to each other. This generally corresponds to determining which edges will be present in the pointer assignment graph.

2. Creating the appropriate pointer assignment graph node to represent each feature in the Jimple input. This is determined by some of the pointer analysis parameters listed in Section 4.2.2, specified as Soot phase options to SPARK.

Each step is represented by an abstract class and its implementation. This makes it possible to change the implementation of each step, without affecting the other. While the current implementation of the first step constructs a graph representing context-insensitive relationships in the Jimple source, SPARK is designed to allow experimenting with context-sensitive implementations in the future. The second step can also have different implementations, for example to change the set of options that determine which types of nodes will be constructed for each feature, or to create an entirely different representation of the pointer assignment graph. Of course, the entire builder could be replaced, so that the pointer assignment graph could be created from a different source representation (such as one based on a language other than Java bytecode), or read in from a file.

The current implementation of the builder constructs variable nodes for local variables and static fields, and a single variable node representing all thrown exceptions. Depending on options given to SPARK, instance fields, method parameters, and return values are represented with either variable or field reference nodes. Array element references are always represented with field reference nodes. Allocation nodes are created for allocation sites and string constants, including command-line parameters to the `main` method.

Edges are created for all pointer-valued assignments including casts, for throw and catch statements, and, unless the call graph is being constructed on the fly, for pointers passed to and returned from methods. In addition, special edges are added for implicit flow of pointers. If a class has a `finalize` method, an edge is added from the allocation node of each allocation site allocating an object of that class to the variable node representing the implicit `this` parameter to the `finalize` method. This models the eventual flow of the object from the allocation site to its `finalize` method when it is garbage collected. Similarly, since the `this` pointer of the `start` method of `java.lang.Thread` implicitly flows to the `this` pointer of the `run` method of any of its subclasses, assignment edges are added to reflect this.

4.2.2 Parameters and Options

Representing Fields

The following three SPARK options affect whether certain features are represented as variable nodes or field reference nodes.

ignoreBaseObjects: When this option is set to true, each reference to an instance field is represented by a variable node, regardless of the object that is the base of the reference (a *field-based* analysis, as compared to a *field-sensitive* analysis). That is, all instances of a given field in all objects are grouped together. This allows for a very fast analysis because pointers can be propagated to variable nodes in a single pass, with no iteration. However, using variable nodes to represent references to instance fields is less precise than using field reference nodes, because it does not distinguish between fields of provably different objects. The default value is false.

parmsAsFields, returnsAsFields: These two options control whether method parameters and return values are represented with variable nodes, or with field reference nodes having the `this` pointer of the method as their base. In combination with respecting declared types during propagation, representing parameters and return values with field reference nodes gives some of the benefits of constructing the call graph on the fly. Pointer flow to and from the targets of a method call is restricted to methods declared in classes reaching the receiver of the call and their superclasses, because the receiver of the call can only be stored in the `this` pointer of these methods. Constructing the call graph on the fly would, in addition, prevent pointer flow to and from methods declared in proper superclasses of classes reaching the receiver. Although these options improve precision, they introduce very large numbers of field reference nodes into the pointer assignment graph, making the analysis very slow, and making it require unreasonable amounts of memory. The default value for both options is false.

The next two options specify which allocation nodes are created to represent allocation sites.

typesForSites: Normally, each allocation site appearing in the program is represented by a unique allocation node. When this option is set to true, however, a single allocation node is used to represent all allocation sites allocating objects with the same type, as in Variable Type Analysis [SHR+00]. This reduces the size of the graph that SPARK has to process, and therefore speeds up the analysis, at the expense of precision (since all objects in the program having a given type are represented together). The default value of this option is false.

mergeStringBuffer: Whenever strings are concatenated using the `+` operator in Java, the corresponding bytecode contains an allocation of a `java.lang.StringBuffer`, and the required operations on it. These operations are implemented in a way that prevents a flow- and context-insensitive analysis from being able to show that the uses of these `java.lang.StringBuffer` objects are not aliased, resulting in large numbers of variables with many aliases. These take a long time to analyze, and also drastically increase the memory requirements for the analysis. Using a single allocation node to represent all allocation sites of type `java.lang.StringBuffer`, like with the typesForSites option, does not affect precision, because the variables storing these objects would all have equal points-to sets anyway. The mergeStringBuffer option has the same effect as the typesForSites option, but only for allocation sites of type `java.lang.StringBuffer`. Its default value is true.

The next option activates the native method simulator.

simulateNatives: Soot includes a framework for simulating the effect on whole-program analyses of the native methods defined in the standard Java library classes. When this option is set to true, SPARK uses this framework to model the effect of these methods. The default value is true.

The next option determines how simple assignment edges are represented.

simpleEdgesBidirectional: Normally, when the Jimple source contains an assignment of the form `a = b`, a directed edge is created from the node representing `b` to the node representing `a`, to reflect the pointer flow. However, a unification-based analysis treats the assignment as bi-directional. When this option is set to true, simple assignment edges are always created in both direction. In combination with merging of strongly-connected components (see Section 4.3), this allows SPARK to perform an analysis like that suggested by Steensgaard [Ste96a]. The default value for this option is false.

The next option specifies whether the call graph should be built on the fly.

onFlyCallGraph: Normally, the builder inserts edges into the pointer assignment graph to represent pointer flow through method parameters and return values, based on the active call graph found in the Soot `Scene` when SPARK is started. When this option is set to true, these edges are not initially added. Instead, the solver adds these edges during the analysis as it propagates points-to sets to the receivers of method calls. The solver accomplishes this by calling back into the builder during solving time. The default value of this option is false.

4.3 Simplifying the Graph

Once the pointer assignment graph has been built, we can proceed directly to propagating the points-to sets. However, it may be possible to prove beforehand that the points-to sets of certain variables will turn out to be equal. In this case, we can simplify the graph by merging the nodes corresponding to variables known to have equal points-to sets. This results in a smaller pointer assignment graph given as input to the points-to set propagation algorithm, hopefully making the analysis run faster and require less memory.

4.3.1 Merging Nodes

SPARK includes support for merging nodes using the fast union-find [Tar75] algorithm at the core of its implementation of a pointer assignment graph. The algorithm is based on successively combining pairs of nodes, and choosing one of the two original nodes as a unique representative for the pair. At any time, for each set of nodes that have been combined, one of the nodes that were combined serves as the unique representative node for the entire set of nodes. The `Node` class contains a `getReplacement()` method, which returns the unique representative node for the set containing the node, as well as a `mergeWith()` method, which merges a node with another.

Merging nodes in a pointer assignment graph is not as simple as applying the union-find algorithm to them, however. Whenever two nodes are merged, the rest of the pointer assignment graph must be updated. In particular, all edges to and from the nodes must be replaced with edges to and from the unique representative of the new combined node. In addition, because each field reference node has a variable node as its base, whenever two variable nodes are merged, all field reference nodes having them as bases must be updated with the unique representative of the new combined node as their base. When this creates multiple field reference nodes with the same base and field, these must in turn be merged. Finally, whenever two nodes are merged, their points-to sets must also be merged. The method used in SPARK to perform this merging of nodes is described next.

Updating the Graph for Merged Nodes

Whenever two nodes are merged, all edges to and from the nodes must be replaced with edges to and from the unique representative of the combined node. This is a slow process, because not only do the adjacency sets of the merged nodes need to be merged, but the adjacency sets of nodes *adjacent to* the merged nodes must be updated as well. Even worse, this must be repeated for each of the many pairs of nodes that are merged.

After experimenting with several methods of updating the edges in the pointer assignment graph to reflect merged nodes, a lazy approach was implemented in SPARK, in keeping with the design of the union-find algorithm. Specifically, when two nodes are merged, their adjacency sets are also merged, but the adjacency sets of nodes adjacent to them are left alone. Instead, whenever the adjacency set of a node is queried, it is checked to ensure that no node in it has already been merged into another node. When a node that has been merged into another node is found, it is replaced with the unique representative of the combined node. This makes each merge operation cheap, delaying the updating of adjacency sets until those sets are iterated over. Updates therefore need not be done to adjacency sets that will never be read, and the updates due to many merges can be done all at once. Moreover, since the updates are done when the adjacency set is being iterated over anyway, the overhead of having to access each adjacency set to update it is avoided.

This approach makes it slightly more expensive to query the adjacency set of a node, which could reduce performance if the sets are accessed frequently. However, determining that an adjacency set does not require any updates is very fast. In addition, SPARK has a global flag that is set whenever nodes are merged. Adjacency sets are only checked when this flag is set, so no checks will be performed unless merges have occurred. In addition, after a period of heavy merging, all the adjacency sets can be updated, and the flag reset, so that SPARK will not have to check for merged nodes until another merge occurs. SPARK does this after the pointer assignment graph is simplified and before propagation begins, so the adjacency sets are not checked unless additional merging occurs during propagation.

Updating Field Reference Nodes When Variable Nodes Are Merged

The updating of field reference nodes when the variable nodes that serve as their base are merged is also done lazily. Specifically, when the unique representative of the combined node containing a field reference node p.f is requested, the following procedure is followed (it is illustrated in Figure 4.3, which shows the union-find pointers after node p.f has been merged into node q.f, and node q has been merged into r).

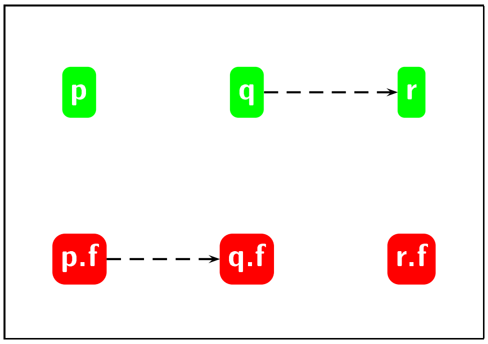


Figure 4.3: Example Illustrating Merging of Field Reference Nodes

1. The union-find pointers are followed to find the unique representative for p.f. Assume that this unique representative is q.f.
2. The base node q of the unique representative q.f is examined. If q is itself the unique representative of the combined node containing it, then q.f is the correct unique representative for p.f, and is returned.
3. Otherwise, the unique representative for q is found. Assume that this unique representative is r. Then the unique representative for the original field reference node p.f is the field reference node with the same field f and base r, namely the node r.f.

After the unique representative has been found, the union-find pointers are updated as in the standard union-find algorithm, so that the next time the unique representative of p.f is requested, the pointer can be followed directly to r.f.

Updating Points-to Sets

Whenever two nodes are merged, the points-to set of the node chosen as the representative for the combined node becomes the union of the two points-to sets.

4.3.2 Strongly Connected Components

When a set of variable nodes forms a strongly connected component in the pointer assignment graph, we have the constraints $\textit{points-to}(n_i) \subseteq \textit{points-to}(n_j) \subseteq \textit{points-to}(n_i)$ for any two nodes n_i, n_j in the set. The points-to graphs of all the nodes in the set are therefore equal, and the nodes can be merged without affecting the result of the points-to analysis. When the option simplifySCCs is set to true, SPARK performs this simplification of the graph before propagation begins. Strongly connected components are found using the well-known, linear-time, depth-first-search-based algorithm described, for example, in [CLR90, Section 23.5]. The default value of the simplifySCCs option is true.

If the declared types of variables are being respected during propagation, the nodes of a strongly connected component may have different points-to sets if they have different declared types. There are two possible ways to handle this case.

1. We can merge the nodes of the strongly connected component anyway, and give the resulting node a declared type that is the nearest common supertype of the declared types of all the nodes. This reduces precision, but allows us to simplify the graph as much as if declared types were not being respected.

2. We can detect only strongly connected components in which the declared types of all the nodes are equivalent. This is done with the normal algorithm for finding strongly connected components, but considering only edges joining nodes with the property that all objects compatible with the declared type of the

source node are also compatible with the declared type of the destination node. In other words, the declared type of the source is a subtype of the declared type of the destination. By merging only the strongly connected components in which the declared types are equivalent, we preserve all precision, but we may simplify the graph less than we could if declared types were not being respected.

The value of the option ignoreTypesForSCCs determines the alternative which is chosen. Because only a small percentage of nodes appear in strongly connected components [RC00], and of those, only a small percentage appear in strongly connected components with multiple declared types, the default value of this option is false.

4.3.3 Single Entry Subgraphs

It is quite common for subgraphs to contain chains of variable nodes, in which each node except the first has only one predecessor. Since the points-to set of the first node will flow to all the other nodes in the chain, the points-to sets of all the nodes will be equal. Therefore, the nodes in the chain could all be merged into a single node, and a single points-to set could be used for all of them together. This idea can be extended to any *single entry subgraph*: any subgraph for which there is a unique "first" node such that the points-to relationships in the points-to sets of any node in the subgraph are also in the points-to set of the "first" node. The idea of merging single entry subgraphs is very similar to the technique that Rountev and Chandra propose for C [RC00].

Definition 1 (Single Entry Subgraph) *A single entry subgraph corresponding to a given* header node *is a subgraph of the pointer assignment graph induced by a set of variable nodes, with each node having the properties:*

1. *that every path to it from a field reference or allocation node passes through the header node, and*

2. *that there exists at least one path from the header node to each node in the subgraph.*

The header node need not necessarily be a variable node. Every variable node is itself a single entry subgraph, with itself as its header node.

Theorem 1 *The points-to set of every node n in a single entry subgraph is equal to the points-to set of the header node h.*

Proof: By definition, there is a path from h to n, so we have the constraint $\textit{points-to}(h) \subseteq \textit{points-to}(n)$. Now, let a be an allocation node in the points-to set of n. This means that there is a path

$$n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow \cdots \rightarrow n_k \rightarrow n$$

with either $n_0 = a$, or n_0 being a field reference node with a in its points-to set, in order to force

$$\{a\} \subseteq \textit{points-to}(n_0) \subseteq \textit{points-to}(n_1) \subseteq \cdots \subseteq \textit{points-to}(n_k) \subseteq \textit{points-to}(n)$$

Since n_0 is a field reference or allocation node, there is at least one field reference node or allocation node on the path of nodes $n_0, n_1, n_2, \ldots, n_k$. Let n_l be the last field reference or allocation node on this path. Then, by the definition of a header node, $h \in \{n_l, n_{l+1}, \ldots, n_k, n\}$. Therefore, $\{a\} \subseteq \textit{points-to}(h)$. Because this argument applies to any $a \in \textit{points-to}(n)$, we have $\textit{points-to}(n) \subseteq \textit{points-to}(h) \subseteq \textit{points-to}(n)$, so the sets are equal, as required. □

In order to simplify the pointer assignment graph as much as possible, we are interested in finding *maximal* single entry subgraphs, and reducing each of them to a single node. However, this is not a required condition; reducing single entry subgraphs that are not maximal will still correctly simplify the pointer assignment graph to some extent.

When the simplifyOffline option is set to true, SPARK uses the algorithm in Figure 4.4 to find and reduce single entry subgraphs. This is a greedy algorithm which repeatedly looks for nodes that are in a common single-entry subgraph and merges them. Whenever a variable node has exactly one predecessor, and the predecessor is a variable node, the two nodes form a single-entry subgraph, since every path to the successor must pass through the predecessor. Therefore, the two nodes can be merged. Similarly, whenever there is a pair of variable nodes with a common predecessor that is a field reference or allocation node, and this predecessor is their *only* predecessor, then every path to each of the variable nodes passes through this predecessor, so the two variable nodes are in the same single-entry subgraph and can be merged. In the absence of cycles and nodes unreachable from any field reference or allocation node, the algorithm finds maximal single-entry subgraphs.

repeat
  while there exists a variable node with exactly one predecessor and this predecessor is a variable node **do**
    merge this variable node with its predecessor
  end while
  while these exists a pair of variable nodes, and a field reference or allocation node such that the field reference or allocation node is the only predecessor of each of the variable nodes **do**
    merge the pair of variable nodes
  end while
until no changes

Figure 4.4: Algorithm for Reducing Single Entry Subgraphs

As in the simplification of strongly connected components, edges where the type of the source node of the edge is not a subtype of the type of the target node of the edge are ignored when declared types are being respected, to prevent nodes which could have unequal points-to sets due to different declared types from being merged.

4.4 Flowing Points-to Sets

The final step of points-to analysis is propagation of points-to sets along edges in the pointer assignment graph to find a fixed point solution of the subset constraints represented by those edges. SPARK currently contains five algorithms[1] for such a computation, and others can be easily added.

4.4.1 Iterative Propagation Algorithm

The algorithm[2] presented in Figure 4.5 is the simplest propagation algorithm in SPARK, used as a baseline, and for testing the correctness of the other, more complicated algorithms. It is a direct extension of the algorithm given by Andersen [And94], extended to distinguish fields in pointer dereference expressions. The algorithm begins by propagating all allocation nodes to the points-to sets of their successors. It then repeatedly propagates points-to sets along the pointer assignment graph until a fixed point is reached. An assignment edge of the form p $\rightarrow$ q indicates that *points-to*(p) $\subseteq$ *points-to*(q), so it is handled by adding the points-to set of p into the points-to set of q. Concrete field nodes are introduced to model the fields of concrete heap objects. Suppose a store edge of the form p $\rightarrow$ q.f is encountered. This means that the field f of the object that q points to can now point to any object that p pointed to. We do not know exactly which object q will point to at run-time; we only know that it will be one of the objects in the points-to set of q. So, for each allocation node a in the points-to set of q, we create a concrete field node a.f to represent the field f of any object created at allocation site a. We then add the points-to set of p into the points-to set of a.f. In a similar way, when we encounter a field load of the form p.f $\rightarrow$ q, we know that for some a in the points-to set of p, pointers flow from a.f to q. So, for each such a, we add the points-to set of a.f into the points-to set of q.

[1] For clarity, algorithms are presented here without support for on-the-fly call graph construction. This support *is* implemented in SPARK, however.

[2] In the algorithms presented in this thesis, the $\cup=$ symbol is used to indicate set union and assignment. That is, $x \cup= y$ indicates that the set $x \cup y$ is assigned to x.

process allocations
repeat
  process every assignment edge
  process every store edge
  process every load edge
until no changes

procedure process allocations ()
for each allocation edge new 1 $\rightarrow$ p **do**
  points-to(p) $\cup=$ {new 1}
end for

procedure process assignment edge (p $\rightarrow$ q)
points-to(q) $\cup=$ *points-to*(p)

procedure process store edge (p $\rightarrow$ q.f)
for each allocation node a $\in$ *points-to*(q) **do**
  points-to(a.f) $\cup=$ *points-to*(p)
end for

procedure process load edge (p.f $\rightarrow$ q)
for each allocation node a $\in$ *points-to*(p) **do**
  points-to(q) $\cup=$ *points-to*(a.f)
end for

Figure 4.5: Iterative Propagation Algorithm

As has been widely noted, this algorithm runs slowly and scales poorly. SPARK includes a slight performance improvement: prior to starting the algorithm, a topological sort is performed on the variable nodes in the pointer assignment graph.[3] Then, the loop between lines 2 and 6 iterates over edges in topological order of their source node. If the pointer assignment graph is cycle-free, this ensures that all points-to sets of variable nodes are propagated on each execution of this loop. Even when the graph contains cycles, considering edges in this order maximizes the length of the path of nodes to which each points-to relationship can flow in each iteration, greatly reducing the number of iterations required and the time to complete the analysis.

This algorithm is selected in SPARK by setting the option propagator to the value iter.

4.4.2 Worklist Propagation Algorithm

For non-trivial benchmarks, the Iterative propagation algorithm is much too slow. A better, but more complex solver based on worklists is also provided as part of SPARK, and is presented in Figures 4.6 and 4.7.

This *worklist propagation algorithm* maintains a worklist of variable nodes. Whenever points-to relationships are added to the points-to set of a variable node, the node is added to the worklist. In the inner loop of the algorithm, nodes are removed from the worklist, and the edges associated with those nodes are processed. As before, variable nodes are removed from the worklist in topological order. First (line 5), any assignment edges originating at the node removed from the worklist (p) are processed, to flow the changes in the points-to set to their successors. Next (line 6), store edges originating at the node removed from the worklist (p) are processed. After that (line 7), the algorithm processes store edges q → p.f whose destination node (p.f) has the node removed from the worklist (p) as its base. This is because the new points-to relationships in the points-to set of p require the points-to set

[3]If the graph contains cycles, the nodes that are part of cycles will obviously not be sorted in topological order; however, all nodes that are not in cycles will be ordered before any of their successors.

of q to be propagated to points-to sets of additional concrete field nodes, to which they were not propagated in previous iterations when the points-to set of p was smaller. Finally (line 8), the algorithm processes any load edges corresponding to fields of objects in the points-to set of p. Since there are new points-to relationships in p, there are new concrete field nodes whose points-to sets need to be propagated to reflect the loads.

This inner loop processing the worklist is not sufficient to obtain a complete solution. Whenever a variable node p appears in the worklist (which means that its points-to set has new nodes in it that need to be propagated), the algorithm propagates along edges that are likely to require propagation: assignment edges of the form p → q, and load and store edges involving p. This is not enough, however. For example, suppose variable p has already been processed with the allocation site a in its points-to set, so it is not in the worklist. Further suppose that a is now added to the points-to set of q. p and q are possible aliases; that is, they may both point to a, and stores to q.f may be loaded from p.f. This means that after processing any store into q.f, we should process all loads from p.f. However, p is not in the worklist, and adding all aliased nodes to the worklist after processing a store edge would be prohibitively expensive. To ensure that stores to q.f are propagated to loads of its alias p.f, the algorithm includes an outer loop. In each iteration of this outer loop, *all* the load and store edges are considered, rather than just those associated with nodes in the worklist, in order to propagate points-to relationships caused by aliasing that may have been missed by the inner loop. To summarize, lines 10 and 11 in the outer loop are necessary for correctness; lines 6 to 8 could be removed, but including them greatly reduces the number of iterations of the outer loop and therefore the analysis time.

This algorithm is selected in SPARK by setting the option propagator to the value worklist.

```
process allocations
repeat
    repeat
        remove first node p from worklist
        process each assignment edge p → q
        process each store edge p → q.f
        process each store edge q → p.f
        process each load edge p.f → q
    until worklist is empty
    process every store edge
    process every load edge
until worklist is empty
```

Figure 4.6: Worklist Propagation Algorithm (part 1 of 2)

procedure process allocations ()
for each allocation edge new 1 → p **do**
    points-to(p) ∪= {new 1}
    worklist ∪= {p}
end for

procedure process assignment edge (p → q)
points-to(q) ∪= *points-to*(p)
if *points-to*(q) was changed **then**
    worklist ∪= {q}
end if

procedure process store edge (p → q.f)
for each allocation node a ∈ *points-to*(q) **do**
    points-to(a.f) ∪= *points-to*(p)
end for

procedure process load edge (p.f → q)
for each allocation node a ∈ $points\text{-}to_{new}$(p) **do**
    points-to(q) ∪= *points-to*(a.f)
    if *points-to*(q) was changed **then**
        worklist ∪= {q}
    end if
end for

Figure 4.7: Worklist Propagation Algorithm (part 2 of 2)

4.4.3 Incremental Worklist Propagation Algorithm

In certain implementations of sets (hash set and sorted array set), each set union operation takes time proportional to the number of elements in the sets being combined. While iterating through an analysis, the contents of one set are repeatedly merged into the contents of another set, often adding only a small number of new elements in each iteration. We can improve the algorithm by noting that the elements that have already been propagated will remain in the destination set in every subsequent iteration, so they need not be propagated again. Instead, we can propagate only the newly-added elements.

Thus, as an optional improvement, SPARK includes versions of the solvers that use incremental sets. Each points-to set is divided into a "new" part and an "old" part. During each iteration, elements are propagated only between the new parts, which are likely to be small. At the end of each iteration, all the new parts are flushed into their corresponding old part. An additional advantage of this is that when constructing the call graph on-the-fly, only the smaller, new part of the points-to set of the receiver of each call site needs to be considered in each iteration.

The worklist propagation algorithm using incremental sets is presented in Figures 4.8 and 4.9. The *points-to* sets have been replaced by $\textit{points-to}_{new}$ and $\textit{points-to}_{old}$. The procedures for processing assignment, store, and load edges have been changed. In general, every propagation between points-to sets has been replaced by a propagation between the new portions of points-to sets. Any elements that already appear in the old points-to set of the destination node are excluded from the propagation, so that the new points-to set of the destination node truly gets only the elements that the node did not have before. For example, occurrences of $\textit{points-to}(\boxed{q}) \cup= \textit{points-to}(\boxed{p})$ in the non-incremental algorithm have been replaced with $\textit{points-to}_{new}(\boxed{q}) \cup= \textit{points-to}_{new}(\boxed{p}) \setminus \textit{points-to}_{old}(\boxed{q})$. This ensures that only new parts of points-to sets are propagated.

There are now two different methods used to process store edges such as $\boxed{p} \rightarrow \boxed{q.f}$, depending on whether it is the source node ($\boxed{p}$) or the base ($\boxed{q}$) of the destination node ($\boxed{q.f}$) which was removed from the worklist (so its points-to set is known to have new elements). When the points-to set of the source node p is known to have new elements, only its new points-to set is propagated to fields of objects in both portions of the points-to set of q, since these new objects in p have not yet been propagated to the field of of any objects pointed to by q, new or old. On the other hand, when it is the points-to set of the base of the destination node that is known to have new elements, both parts of the points-to set of the source node p are propagated to the fields of only the newly added objects of q (that is, to fields of objects in $\mathit{points\text{-}to}_{new}$(q)).

Another difference compared to the original worklist propagation algorithm is the addition of lines 9, 10, 15, and 16, which flush the new portions of points-to sets into the old portions.

As in the non-incremental version of the algorithm, an outer loop is required to process all stores and loads, to account for flow due to aliasing that may have been missed by the inner loop. In the outer loop, both parts of each points-to set are propagated to ensure a complete propagation.

The incremental worklist propagation algorithm is selected in Spark by setting the option propagator to the value worklist, and the option setImpl to the value double.

```
incrementally process allocations
repeat
   repeat
      remove first node p from worklist
      incrementally process each assignment edge p → q
      incrementally process each store edge p → q.f for source
      incrementally process each store edge q → p.f for destination base
      incrementally process each load edge p.f → q
      points-to_old(p) ∪= points-to_new(p)
      points-to_new(p) ← {}
   until worklist is empty
   fully process every store edge
   fully process every load edge
   for each concrete field node a.f do
      points-to_old(a.f) ∪= points-to_new(a.f)
      points-to_new(a.f) ← {}
   end for
until worklist is empty

procedure incrementally process allocations ()
for each allocation edge new 1 → p do
   points-to_new(p) ∪= {new 1}
   worklist ∪= {p}
end for

procedure incrementally process assignment edge ( p → q )
points-to_new(q) ∪= points-to_new(p) \ points-to_old(q)
if points-to_new(q) was changed then
   worklist ∪= {q}
end if
```

Figure 4.8: Incremental Worklist Propagation Algorithm (part 1 of 2)

procedure incrementally process store edge for source ($p \rightarrow q.f$)

for each allocation node $a \in (\mathit{points\text{-}to}_{new}(q) \cup \mathit{points\text{-}to}_{old}(q))$ **do**
    $\mathit{points\text{-}to}_{new}(a.f) \cup= \mathit{points\text{-}to}_{new}(p) \setminus \mathit{points\text{-}to}_{old}(a.f)$
end for

procedure incrementally process store edge for destination base ($p \rightarrow q.f$)

for each allocation node $a \in \mathit{points\text{-}to}_{new}(q)$ **do**
    $\mathit{points\text{-}to}_{new}(a.f) \cup= (\mathit{points\text{-}to}_{new}(p) \cup \mathit{points\text{-}to}_{old}(p)) \setminus \mathit{points\text{-}to}_{old}(a.f)$
end for

procedure incrementally process load edge ($p.f \rightarrow q$)

for each allocation node $a \in \mathit{points\text{-}to}_{new}(p)$ **do**
    $\mathit{points\text{-}to}_{new}(q) \cup= (\mathit{points\text{-}to}_{new}(a.f) \cup \mathit{points\text{-}to}_{old}(a.f)) \setminus \mathit{points\text{-}to}_{old}(q)$
    if $\mathit{points\text{-}to}_{new}(q)$ was changed **then**
        $\mathit{worklist} \cup= \{q\}$
    end if
end for

procedure fully process store edge ($p \rightarrow q.f$)

for each allocation node $a \in (\mathit{points\text{-}to}_{new}(q) \cup \mathit{points\text{-}to}_{old}(q))$ **do**
    $\mathit{points\text{-}to}_{new}(a.f) \cup= (\mathit{points\text{-}to}_{new}(p) \cup \mathit{points\text{-}to}_{old}(p)) \setminus \mathit{points\text{-}to}_{old}(a.f)$
end for

procedure full process load edge ($p.f \rightarrow q$)

for each allocation node $a \in (\mathit{points\text{-}to}_{new}(p) \cup \mathit{points\text{-}to}_{old}(p))$ **do**
    $\mathit{points\text{-}to}_{new}(q) \cup= (\mathit{points\text{-}to}_{new}(a.f) \cup \mathit{points\text{-}to}_{old}(a.f)) \setminus \mathit{points\text{-}to}_{old}(q)$
    if $\mathit{points\text{-}to}_{new}(q)$ was changed **then**
        $\mathit{worklist} \cup= \{q\}$
    end if
end for

Figure 4.9: Incremental Worklist Propagation Algorithm (part 2 of 2)

4.4.4 Alias Edge Propagation Algorithm

Andersen's [And94] algorithm for C uses a separate points-to set for each allocation site to represent pointers stored into objects created at that allocation site. Accordingly, the standard extension [LPH01, RMR01] to Java handles field-sensitivity using a separate points-to set for each field of the objects created at each allocation site. This ensures that aliased field references p.f and q.f are correctly handled, since if p and q both have allocation site a in their points-to sets, stores into them and loads out of them will flow into and out of, respectively, the points-to set for a.f.

Unfortunately, as points-to sets grow large, this representation becomes prohibitively inefficient. If *points-to*(p) = {a1, a2, ..., an}, then any stores to p.f must be propagated to each of the n sets *points-to*(ai.f) (see Figure 4.10(a)). The space and time requirements are quadratic in the size of the sets, since n possibly large sets must be created, where n is the size of the set for p.

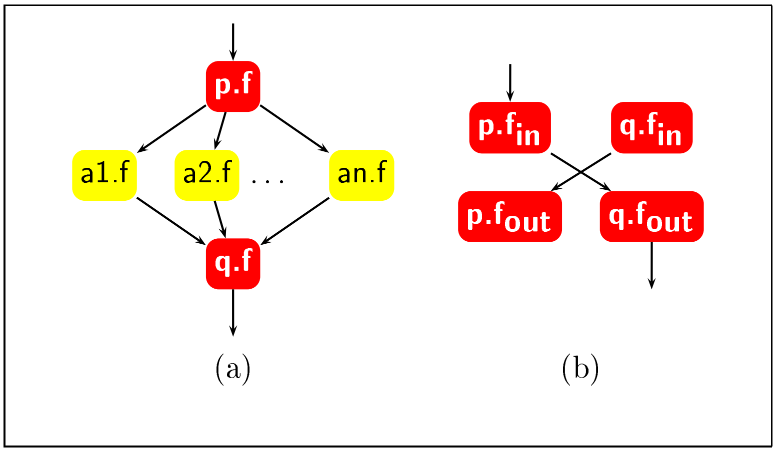


Figure 4.10: Field Representation in Standard (a) and Alias Edge (b) Algorithms

Points-to sets were originally proposed as a compact representation of alias relationships [EGH94]. If the average points-to set is of size n, and there are v variables, a points-to set representation takes $O(nv)$ space, while an alias set representation may take $\Theta(v^2)$ space, since each variable could be aliased to each other variable. When n is much smaller than v, as is usually true when analyzing C, points-to sets are more efficient. However, in handling aliases in Java, we are only interested in aliased variables dereferenced with the same field, because a field in Java can only be accessed by a field expression specifying that field. This is in contrast to C, where one can take the address of a field of an object, use unsafe casts, or even use pointer arithmetic to create other aliases to the field of an object. Most fields in Java are dereferenced few times, and therefore with few variables. Therefore, in Java, for any given field, our n is much greater than v, so the $\Theta(v^2)$ representation based on alias sets can be more efficient.

One way to implement such a representation is to eliminate the concrete field nodes, and add edges directly between field reference nodes that are determined to be aliased. However, the may-alias relationship is not transitive. If p and q are aliased (that is, the intersection of their points-to sets is not empty), we cannot simply add pointer assignment edges in both directions between p.f and q.f, because these two field references may not have the same points-to sets. For example, suppose *points-to*(p) = {a1, a2} and *points-to*(q) = {a1}. Then p and q are possibly aliased, but p.f may point to objects in *points-to*(a2.f) that q.f may not point to. To get around this difficulty, we split all field reference nodes into two halves, an `in` half used as the destination of field stores, and an `out` half used as the source of field loads, and add edges only from the `in` half of a node to the `out` half of other nodes, as shown in Figure 4.10(b). This allows us to represent the alias relationship without making it transitive, while ensuring that anything stored into p.f can be loaded from q.f and vice-versa.

The alias edge propagation algorithm is presented in Figures 4.11 and 4.12. This algorithm uses three worklists:

worklist stores variable nodes whose points-to sets have changed and must be propagated along assignment and store edges, like in the worklist propagation algorithm.

aliasWorklist stores variable nodes after their points-to sets have been propagated so that they can be considered for possible aliasing with other nodes, and the corresponding alias edges can be added.

```
process allocations
repeat
    process worklist
    process aliasWorklist
    process fieldRefWorklist
until worklist is empty

procedure process allocations ()
for each allocation edge new 1 → p do
    points-to(p) ∪= {new 1}
    worklist ∪= {p}
end for

procedure process worklist ()
while worklist is not empty do
    remove first node p from worklist
    aliasWorklist ∪= {p}
    for each assignment edge p → q do
        points-to(q) ∪= points-to(p)
        if points-to(q) was changed then
            worklist ∪= {q}
        end if
    end for
    for each store edge p → q.f do
        points-to(q.f_in) ∪= points-to(p)
        if points-to(q.f_in) was changed then
            fieldRefWorklist ∪= {q.f_in}
        end if
    end for
end while
```

Figure 4.11: Alias Edge Propagation Algorithm (part 1 of 2)

procedure process *aliasWorklist* ()
while *aliasWorklist* is not empty **do**
    remove first node p from *aliasWorklist*
    for each p.f with p as its base **do**
        for each q which is dereferenced with field f as q.f **do**
            if *points-to*(p) $\cap$ *points-to*(q) $\neq \emptyset$ **then**
                aliasEdges $\cup=$ $\{p.f_{in} \rightarrow q.f_{out},\ q.f_{in} \rightarrow p.f_{out}\}$
                fieldRefWorklist $\cup=$ $\{p.f_{in},\ q.f_{in}\}$
            end if
        end for
    end for
end while

procedure process *fieldRefWorklist* ()
while *fieldRefWorklist* is not empty **do**
    remove first node $p.f_{in}$ from *fieldRefWorklist*
    for each edge $p.f_{in} \rightarrow q.f_{out} \in$ *aliasEdges* **do**
        points-to($q.f_{out}$) $\cup=$ *points-to*($p.f_{in}$)
    end for
end while
for each field reference node p.f **do**
    for each load edge $p.f \rightarrow q$ **do**
        points-to(q) $\cup=$ *points-to*($p.f_{out}$)
        if *points-to*(q) was changed **then**
            worklist $\cup=$ $\{q\}$
        end if
    end for
end for

Figure 4.12: Alias Edge Propagation Algorithm (part 2 of 2)

fieldRefWorklist stores field reference nodes whose points-to sets have changed and must be propagated along alias edges.

The points-to sets of nodes removed from *worklist* are propagated along assignment and store edges originating at those nodes. Whenever a points-to relationship is added to the points-to set of a variable node or field reference node, that node is added to the *worklist* or the *fieldRefWorklist*, respectively, so that the new points-to relationship will be propagated further along edges originating at that node. In addition, each node that is removed from the *worklist* is added to the *aliasWorklist*, so that it will later be processed for any new aliasing relationships that may have arisen from the new elements in its points-to set. To find these relationships (in the "process *aliasWorklist*" procedure), for each node p, we find all the fields with which it is dereferenced, and for each such field, we find all other nodes q that are dereferenced with the same field. If the points-to sets of p and q have a non-empty intersection, then their fields are aliased, so we add the appropriate edges between them ($p.f_{in}$ → $q.f_{out}$ and $q.f_{in}$ → $p.f_{out}$), and add the nodes to the *fieldRefWorklist*, so that points-to sets will be propagated along these new edges. The *fieldRefWorklist* keeps track of in field reference nodes whose points-to sets have new elements that must be propagated. When it is processed, these points-to sets are propagated to the points-to sets of out field reference nodes along alias edges. Finally, all load edges are processed, propagating points-to sets of out field reference nodes to the points-to sets of the corresponding variable nodes.

The alias edge propagation algorithm is selected in SPARK by setting the option propagator to the value alias.

4.4.5 Incremental Alias Edge Propagation Algorithm

Like the worklist propagation algorithm, the alias edge propagation algorithm can be made incremental. The incremental version is presented in Figures 4.13 through 4.16. Overall, this algorithm is very similar to the non-incremental version. The main difference is that points-to sets are again divided into two parts, and only the new parts are propagated. After each variable node from the *worklist* has been processed, its new part is flushed into the old part. Similarly, after each in field reference node from the *fieldRefWorklist* is processed, its new points-to set is flushed into its old points-to set. The points-to sets for the out field reference nodes are flushed when all the loads are processed (in the "incrementally process *fieldRefWorklist*" procedure).

The incremental alias edge propagation algorithm is selected in SPARK by setting the option propagator to the value alias, and the option setImpl to the value double.

```
incrementally process allocations
repeat
    incrementally process worklist
    incrementally process aliasWorklist
    incrementally process fieldRefWorklist
until worklist is empty

procedure incrementally process allocations ()
for each allocation edge new 1 → p do
    points-to_new(p) ∪= {new 1}
    worklist ∪= {p}
end for
```

Figure 4.13: Incremental Alias Edge Propagation Algorithm (part 1 of 4)

procedure incrementally process *worklist* ()

while *worklist* is not empty **do**
    remove first node $\boxed{\mathsf{p}}$ from *worklist*
    $\mathit{aliasWorklist} \cup= \{\boxed{\mathsf{p}}\}$
    for each assignment edge $\boxed{\mathsf{p}} \longrightarrow \boxed{\mathsf{q}}$ **do**
        $\textit{points-to}_{new}(\boxed{\mathsf{q}}) \cup= \textit{points-to}_{new}(\boxed{\mathsf{p}}) \setminus \textit{points-to}_{old}(\boxed{\mathsf{q}})$
        if $\textit{points-to}_{new}(\boxed{\mathsf{q}})$ was changed **then**
            $\mathit{worklist} \cup= \{\boxed{\mathsf{q}}\}$
        end if
    end for
    for each store edge $\boxed{\mathsf{p}} \longrightarrow \boxed{\mathsf{q.f}}$ **do**
        $\textit{points-to}_{new}(\boxed{\mathsf{q.f_{in}}}) \cup= \textit{points-to}_{new}(\boxed{\mathsf{p}}) \setminus \textit{points-to}_{old}(\boxed{\mathsf{q.f_{in}}})$
        if $\textit{points-to}_{new}(\boxed{\mathsf{q.f_{in}}})$ was changed **then**
            $\mathit{fieldRefWorklist} \cup= \{\boxed{\mathsf{q.f_{in}}}\}$
        end if
    end for
    $\textit{points-to}_{old}(\boxed{\mathsf{p}}) \cup= \textit{points-to}_{new}(\boxed{\mathsf{p}})$
    $\textit{points-to}_{new}(\boxed{\mathsf{p}}) \leftarrow \{\}$
end while

Figure 4.14: Incremental Alias Edge Propagation Algorithm (part 2 of 4)

procedure incrementally process *aliasWorklist* ()
while *aliasWorklist* is not empty **do**
  remove first node p from *aliasWorklist*
  for each p.f with p as its base **do**
    for each q which is dereferenced with field f as q.f **do**
      if $\textit{points-to}(p) \cap \textit{points-to}(q) \neq \emptyset$ **then**
        $\textit{aliasEdges} \cup= \{p.f_{in} \rightarrow q.f_{out}, q.f_{in} \rightarrow p.f_{out}\}$
        $\textit{fieldRefWorklist} \cup= \{p.f_{in}, q.f_{in}\}$
        $\textit{points-to}_{new}(q.f_{out}) \cup= \textit{points-to}_{old}(p.f_{in}) \setminus \textit{points-to}_{old}(q.f_{out})$
        $\textit{points-to}_{new}(p.f_{out}) \cup= \textit{points-to}_{old}(q.f_{in}) \setminus \textit{points-to}_{old}(p.f_{out})$
      end if
    end for
  end for
end while

Figure 4.15: Incremental Alias Edge Propagation Algorithm (part 3 of 4)

```
procedure incrementally process fieldRefWorklist ()
while fieldRefWorklist is not empty do
    remove first node p.f_in from fieldRefWorklist
    for each edge p.f_in → q.f_out ∈ aliasEdges do
        points-to_new(q.f_out) ∪= points-to_new(p.f_in) \ points-to_old(q.f_out)
    end for
    points-to_old(p.f_in) ∪= points-to_new(p.f_in)
    points-to_new(p.f_in) ← {}
end while
for each field reference node p.f do
    for each load edge p.f → q do
        points-to_new(q) ∪= points-to_new(p.f_out) \ points-to_old(q)
        if points-to_new(q) was changed then
            worklist ∪= {q}
        end if
    end for
    points-to_old(p.f_out) ∪= points-to_new(p.f_out)
    points-to_new(p.f_out) ← {}
end for
```

Figure 4.16: Incremental Alias Edge Propagation Algorithm (part 4 of 4)

4.5 Points-to Set Implementations

One purpose of SPARK is to enable experimentation with different implementations of points-to sets. There are currently four implementations of points-to sets, and more can be added by implementing a subclass of the `PointsToSetInternal` abstract class. This class contains default implementations of the required set operations in terms of three basic operations:

`add` adds an element to the set.

`forall` executes a given method once for each element, passing the element as a parameter.

`contains` returns a boolean value indicating whether a given element is in the set.

This makes it very easy to try out new set representations, since only these three functions must be implemented. However, the set implementations currently included in SPARK implement custom versions of the other set operations for efficiency. These other operations are:

`addAll` adds all elements of one set into another.

`hasNonEmptyIntersection` returns a boolean value indicating whether the intersection of the set with another given set is empty.

`possibleTypes` returns a set of the types of all objects contained in the set.

Each set may optionally have a declared type. In this case, the set ignores insertions of allocation nodes with a type that is not a subtype of the declared type.

4.5.1 Hash Set

The hash set is a simple wrapper around `java.util.HashSet` from the standard class library. It is provided as a baseline against which other set implementations can be compared, and for testing of more complicated implementations.

4.5.2 Sorted Array Set

The sorted array set is a representation of a points-to set using an array which is always kept in sorted order. Membership testing is implemented using a binary search, which executes in time logarithmic in the number of elements in the set. Element insertion takes time linear in the number of elements in the set, because the elements that come after the element being inserted must be shifted to make room for the new element. However, using the merge step of the well-known merge sort algorithm, the very common operation of computing the union of two sets takes time linear in the size of the sets. When the array becomes full, it is copied to a new array twice as large as the original. Merging two sets is always done into a new array large enough to hold both sets, to avoid having to resize the array during this very common operation.

4.5.3 Bit Set

The bit set represents a points-to set as a bit vector. All of the allocation nodes in the pointer assignment graph are numbered sequentially. To insert the node numbered i into the set, we set the ith bit. Both testing membership and inserting an element take constant time. Merging a pair of sets takes time linear in the total number of allocation nodes, rather than the number of elements in the sets. However, the proportionality constant is very small, because the sets are merged 32 bits at a time. In addition, when the set is large, each element takes only a single bit, compared to 32 bits in the sorted array set. The drawback is that sets with few elements use as much memory as sets with many elements.

4.5.4 Hybrid Set

The hybrid set is a hybrid representation of a points-to set. It uses explicit pointers to the set elements in arbitrary order when the set contains 16 elements or fewer. When the set grows larger, this implementation switches to the bit vector representation.

The hybrid set implementation was introduced to reduce memory requirements. In early experiments on large benchmarks, the analysis encountered very large numbers of small sets, along with significant numbers of very large sets. Using the sorted array set implementation, the very large sets used more memory than was available. On the other hand, with the bit set implementation, each of the small sets required as much memory as a large set, and there were so many small sets that, once again, all available memory was exhausted. As we will see from the experimental results, the hybrid sets turned out to be most efficient not only in terms of memory requirements, but also in terms of analysis time.

Chapter 5
Experimental Results

This chapter reports on an extensive empirical study of a variety of subset-based points-to analyses. This study demonstrates that SPARK provides a general and effective means to express different points-to analyses. Many different variations were expressed within the same framework, making it possible to compare both precision and cost of the analyses.

5.1 Benchmarks

SPARK was evaluated on benchmarks from the SPECjvm [Spec] suite, along with `sablecc` and `soot` from the Ashes [Ashe] suite, and `jedit` [Jedi], a full-featured editor written in Java. The last three were selected because they are non-trivial Java applications used in the real world, and they were also used in other points-to analysis studies [RMR01, WL02, LPH01]. All benchmarks were analyzed with the Sun JDK 1.3.1_01 standard class library, on a 1.67 GHz AMD Athlon with 2GB of memory running Linux 2.4.18. In addition, the `javac` benchmark was also evaluated with the Sun JDK 1.1.8 standard class library for comparison with other studies. The complete list of benchmarks appears in the summary in Table 5.1. The first column gives the benchmark name (`javac` is listed twice: once with the 1.3.1_01 JDK class library, and once with the 1.1.8 JDK class library). The next two columns give the number of methods determined to be reachable, and the number of Jimple[1] statements in these methods. Note that because of the large class library, these are the largest Java benchmarks for which a subtype-based points-to analysis has so far been reported. The fourth column gives the number of distinct types encountered by the subtype tester.

Detailed experiments on individual factors affecting precision and efficiency of points-to analysis were performed on a selection of four of the benchmarks. `compress` (Lempel-Ziv compression) was chosen as a small SPECjvm benchmark, `javac` (Java compiler) as a large SPECjvm benchmark, and `sablecc` (parser generator) and `jedit` (text editor) as large non-SPECjvm benchmarks written by distinct groups of people. The other benchmarks exhibited similar trends.

Benchmark	methods (CHA)	stmts (CHA)	types
`compress`	15183	278902	2770
`db`	15185	278954	2763
`jack`	15441	288142	2816
`javac` (1.1.8)	4602	86454	874
`javac` (1.3.1)	16307	301801	2940
`jess`	15794	288831	2917
`mpegaudio`	15385	283482	2782
`raytrace`	15312	281587	2789
`sablecc`	16977	300504	3070
`soot`	17498	310935	3435
`jedit`	19621	367317	3395

Table 5.1: Benchmark Characteristics

[1] Jimple is the three-address typed intermediate representation used by Soot.

5.2 Factors Affecting Precision

This section analyzes three factors that affect not only the efficiency of the analysis, but also the precision of its result. These factors are: (1) how types are used in the analysis, (2) whether the analysis uses a CHA-based call graph or builds the call graph on the fly, and (3) whether the analysis is field-based or field-sensitive.

Table 5.2 gives the results. Each analysis is named by a triple of the form xx-yyy-zz which specifies the setting for each of the three factors (a complete explanation of each factor is given in the subsections below). For each benchmark and points-to analysis combination, the table gives a summary of the precision for dereference sites and call sites.

For dereference sites, the table gives the percentage of field dereference sites of the form `p.f` with 0, 1, 2, 3-10, 11-100, 101-1000 and more than 1000 elements in their points-to sets. Dereference sites with 0 items in the set correspond to statements that cannot be reached (i.e. the CHA call graph conservatively indicates that the dereference was in a reachable method, but no allocation ever flows to the statement).

For call sites, the table reports the percentage of all `invokevirtual` and `invokeinterface` call sites with 0, 1, 2, and more than two target methods, where the target methods are found using the types of the allocation sites pointed to by the receiver of the method call. For example, for a call of the form `o.m()`, the types of allocation sites pointed to by `o` would be used to find the target methods. Calls with 0 targets correspond to unreachable calls, and calls with 1 target are guaranteed to be monomorphic at run-time.

5.2.1 Respecting Declared Types

Unlike in C, variables in Java are strongly-typed, limiting the possible set of objects to which a pointer could point. However, many points-to analyses adapted from C do not take advantage of this. For example, the analyses described in [RMR01, SHR$^+$00] ignore declared types as the analysis proceeds; however, objects of incompatible type are removed after the analysis completes.

Benchmark Analysis	Dereference Sites (% of total) 0	1	2	3-10	11-100	101-1000	1001+	Call Sites (% of total) 0	1	2	3+
`compress`											
nt-otf-fs	35.2	23.4	6.3	14.1	5.9	0.1	14.9	53.8	42.6	1.6	1.9
at-otf-fs	35.3	32.7	8.0	17.4	4.3	2.2	0.0	53.8	42.6	1.6	1.9
ot-otf-fs	36.9	32.1	7.8	17.0	4.3	1.8	0.0	54.6	42.3	1.3	1.8
ot-cha-fs	20.5	39.6	10.1	21.8	6.0	2.1	0.0	40.8	51.7	2.6	4.9
ot-otf-fb	26.3	38.1	9.4	19.2	5.1	1.9	0.0	48.0	47.4	2.0	2.6
ot-cha-fb	16.0	41.6	10.9	22.9	6.4	2.2	0.0	37.5	54.3	2.9	5.2
`javac`											
nt-otf-fs	31.4	22.2	6.0	12.9	5.8	6.4	15.2	50.1	45.3	1.9	2.7
at-otf-fs	31.6	33.9	8.7	17.7	5.7	2.4	0.0	50.1	45.3	1.9	2.7
ot-otf-fs	33.0	33.3	8.6	17.3	5.7	2.0	0.0	50.8	45.2	1.5	2.5
ot-cha-fs	18.4	40.0	10.5	21.5	7.2	2.3	0.0	38.0	53.9	2.6	5.5
ot-otf-fb	23.6	38.6	10.0	19.2	6.5	2.1	0.0	44.6	49.9	2.1	3.3
ot-cha-fb	14.5	41.7	11.3	22.5	7.6	2.4	0.0	34.9	56.3	3.0	5.8
`sablecc`											
nt-otf-fs	31.6	24.2	5.9	12.7	9.5	0.2	15.8	49.9	45.8	2.1	2.2
at-otf-fs	31.7	37.9	7.4	16.2	4.9	2.0	0.0	49.9	45.8	2.1	2.2
ot-otf-fs	33.1	37.4	7.3	15.7	4.9	1.6	0.0	50.8	45.5	1.6	2.0
ot-cha-fs	18.4	44.1	9.2	20.1	6.4	1.9	0.0	37.9	54.2	2.9	5.0
ot-otf-fb	23.6	42.6	8.7	17.7	5.7	1.7	0.0	44.7	50.3	2.2	2.8
ot-cha-fb	14.4	45.8	10.0	21.0	6.8	1.9	0.0	34.9	56.6	3.3	5.2
`jedit`											
nt-otf-fs	25.6	29.6	6.6	12.7	3.8	1.5	20.2	43.8	52.0	1.9	2.2
at-otf-fs	25.7	42.4	9.0	16.3	4.7	2.0	0.0	43.8	52.0	1.9	2.2
ot-otf-fs	27.1	42.0	8.9	15.9	4.3	1.9	0.0	44.6	51.9	1.4	2.1
ot-cha-fs	14.5	47.9	10.7	19.4	5.5	2.1	0.0	33.2	59.3	2.3	5.1
ot-otf-fb	18.9	46.7	10.0	17.6	4.8	2.0	0.0	38.6	56.7	1.9	2.8
ot-cha-fb	12.1	49.0	11.0	20.1	5.7	2.1	0.0	30.7	61.5	2.5	5.3

Table 5.2: Analysis Precision

The first three lines for each benchmark in Table 5.2 show the effect of declared types. The first line shows the precision of an analysis in which declared types are ignored, *notypes* (abbreviated nt). The second line shows the results of the same analysis after objects of incompatible type have been removed after completion of the analysis, *aftertypes* (abbreviated at). This is the method studied in [SHR+00, RMR01]. The third line shows the precision of an analysis in which declared types are respected throughout the analysis, *on-the-fly types* (abbreviated ot).

We see that removing objects based on declared type after completion of the analysis (at) achieves almost the same precision as enforcing the types during the analysis (ot). However, notice that during the analysis (nt), between 15% and 20% of the points-to sets at dereference sites are over 1000 elements in size. These large sets increase memory requirements prohibitively, and slow the analysis considerably. These numbers show that enforcing declared types as the analysis proceeds eliminates almost all of these large sets. Based on this observation, the rest of this chapter focuses on analyses that respect declared types.

Enforcing declared types during the analysis requires fast subtype testing. For this purpose, SPARK precomputes and stores the subtype relationships in a two-dimensional bit array. Although this requires space quadratic in the number of types, for the benchmarks used in this study, the number of types was around 3000 (see Table 5.1), so this table takes slightly over 1MB of memory, which is small compared to all the information that Soot keeps about a 600KLOC program. In addition, other parts of Soot can take advantage of fast subtype testing. More complicated, fast, space-efficient subtype testing mechanisms are evaluated in [VHK97].

Based on these results, respecting declared types during a Java points-to analysis is highly recommended because it improves precision while making the analysis considerably more efficient.

5.2.2 Call Graph Construction

The call graph used for an inter-procedural points-to analysis can be constructed ahead of time using, for example, CHA [DGC95], or on-the-fly as the analysis proceeds [RMR01], for greater precision. In Table 5.2, these variations are abbreviated as cha and otf, respectively. As the third and fourth lines for each benchmark show, computing the call graph on-the-fly increases the number of points-to sets of size zero (dereference sites determined to be unreachable), but has a smaller effect on the size distribution of the remaining sets.

5.2.3 Field Dereference Expressions

A *field-based* (abbreviated fb) analysis ignores the base objects in field dereference expressions, considering only the field, while a *field-sensitive* (abbreviated fs) parameterizes each field dereference expression by its base object for greater precision.

Comparing rows 3 and 5 (on-the-fly call graph), and rows 4 and 6 (CHA call graph), for each benchmark, we see that field-sensitive analysis is more precise than the field-based analysis. Thus, it is probably worthwhile to do field-sensitive analysis if the cost of the analysis is reasonable. Later, in Table 5.4, we will see that with the appropriate solver, the field-sensitive analysis can be made to be quite competitive with the field-based analysis.

5.3 Factors Affecting Performance

5.3.1 Set Implementation

This subsection compares the performance of analyses with the four different implementations of points-to sets described in Section 4.5, namely hash sets, sorted array sets, bit sets, and hybrid sets. Table 5.3 shows the efficiency of the implementations using two of the propagation algorithms: the naive, iterative algorithm, and the incremental worklist algorithm. Both algorithms used a CHA call graph, and the pointer assignment graph was simplified before propagation by collapsing cycles, as well as single-entry subgraphs as described in Section 4.3. Both algorithms respected declared types during the computation. The Graph space column shows the space needed to store the original pointer assignment graph, and the remaining space columns show the space needed to store the points-to sets. The data structure storing the graph is designed for flexibility rather than space efficiency; it could be made smaller if necessary. In any case, its size is linear in the size of the program being analyzed.

(time in seconds, space in MB)

Benchmark	Graph	Hash		Array		Bit		Hybrid	
Algorithm	space	time	space	time	space	time	space	time	space
`compress`									
Iterative	31	3448	311	1206	118	36	75	24	34
Incr. Worklist	31	219	319	62	57	14	155	9	53
`javac`									
Iterative	34	3791	361	1114	139	50	88	33	41
Incr. Worklist	34	252	369	61	68	19	181	13	65
`sablecc`									
Iterative	36	4158	334	1194	132	50	93	32	42
Incr. Worklist	36	244	342	54	62	17	193	11	66
`jedit`									
Iterative	42	6502	583	2233	229	91	168	59	77
Incr. Worklist	42	488	597	135	114	38	349	24	128

Table 5.3: Set Implementation

The terrible performance of the hash set implementation is disappointing, as this is the implementation provided by the language. Clearly, anyone serious about implementing an efficient points-to analysis in Java must write a custom set representation.

The sorted array set implementation is prohibitively expensive using the iterative algorithm, but becomes reasonable using the incremental worklist algorithm, which is designed explicitly to limit the size of the sets that must be propagated. Notice that the memory requirements are also much smaller when the incremental worklist algorithm is used. This is because the implementation of set union creates an array large enough to hold both sets being combined. If these two sets are equal or almost equal, the resulting array ends up being twice as large as it would need to be. In the incremental algorithm, the sets being propagated are kept small, so most union operations involve one large set, and one very small set.

The bit set implementation is much faster still than the sorted array set implementation. However, especially when used with the incremental worklist algorithm, its memory usage is high, because even the many very small sets are represented using the same size bit-vector as large sets. In addition, the incremental worklist algorithm splits each points-to set into two halves, making the bit set use twice the memory.

Finally, the hybrid set implementation is even faster than the bit set implementation, while maintaining modest memory requirements. The hybrid set implementation is consistently the most efficient over a wide variety of settings of the other parameters, and it is therefore used in all the remaining experiments. It is strongly recommended that implementations similar to the hybrid set implementations be used in future points-to analysis research, because they are consistently more efficient than the other implementations.

5.3.2 Points-To Set Propagation Algorithms

Table 5.4 shows the time and space requirements of the propagation algorithms included in SPARK. All measurements in this table were made using the hybrid set implementation, and without any simplification of the pointer assignment graph.[2] Again, the Graph space column shows the space needed to store the original pointer assignment graph, and the remaining space columns show the space needed to store the points-to sets. For each analysis, the best time and space numbers are shown in bold.

The iterative algorithm is consistently slowest, and is given as a baseline only. The

[2] The time and space reported for the hybrid set implementation in Table 5.3 are different than in Table 5.4 because the former were measured with off-line pointer assignment graph simplification, and the latter without.

(time in seconds, space in MB)

Benchmark Analysis	Graph space	Iterative time	Iterative space	Worklist time	Worklist space	Incr. Worklist time	Incr. Worklist space	Alias time	Alias space	Incr. Alias time	Incr. Alias space
`compress`											
nt-otf-fs	32	1628	357	992	365	**399**	605	871	**100**	820	114
ot-otf-fs	37	133	52	58	51	**52**	69	62	**47**	58	61
ot-cha-fs	36	49	68	15	63	**13**	91	20	**62**	26	83
ot-otf-fb	35	158	54	86	**52**	**66**	66	93	53	73	67
ot-cha-fb	34	17	62	**10**	**56**	13	76	19	58	25	77
`javac`											
nt-otf-fs	34	2316	502	1570	512	**715**	856	1225	**142**	1097	160
ot-otf-fs	40	201	69	103	66	**90**	90	103	**65**	97	83
ot-cha-fs	39	64	83	22	**77**	**18**	109	27	78	34	103
ot-otf-fb	37	218	70	123	**66**	**102**	84	142	68	111	85
ot-cha-fb	37	22	75	**11**	**67**	15	90	22	69	30	92
`sablecc`											
nt-otf-fs	35	2190	462	1382	472	**635**	772	3020	**145**	3413	163
ot-otf-fs	41	274	72	104	70	**95**	94	114	**69**	107	87
ot-cha-fs	41	66	88	20	**83**	**18**	117	28	84	36	109
ot-otf-fb	38	255	74	138	**72**	**114**	90	158	73	125	92
ot-cha-fb	38	52	81	**14**	**74**	18	97	27	77	36	99
`jedit`											
nt-otf-fs	oom	oom	oom	oom	oom	oom	oom	2425	**283**	**2042**	307
ot-otf-fs	49	313	121	142	117	**101**	169	151	**102**	112	126
ot-cha-fs	48	107	141	59	131	**38**	196	44	**117**	56	150
ot-otf-fb	47	298	104	178	**99**	**111**	126	225	102	127	127
ot-cha-fb	45	28	109	**21**	**98**	27	128	36	100	49	129

Table 5.4: Propagation Algorithms

worklist algorithm is usually about twice as fast as the iterative algorithm. For the CHA-based, field-based analysis, this algorithm is consistently the fastest, faster even than the incremental worklist algorithm. This is because the incremental worklist algorithm is designed to propagate only the newly-added part of the points-to sets in each iteration, but the CHA-based, field-based analysis requires only a single iteration. Therefore, any benefit from its being incremental is outweighed by the overhead of maintaining two parts of every set.

However, both field-sensitivity and on-the-fly call graph construction require iteration, so for these, the incremental worklist algorithm is consistently fastest. Note that this speedup comes with a cost in the memory required to maintain two parts of every set.

Notice also that while the field-based analysis is faster than the field-sensitive analysis with a CHA call graph, it is slower when the call graph is constructed on the fly (with all propagation algorithms). This is because although a field-based analysis with a CHA call graph completes in one iteration, constructing the call graph on-the-fly requires iterating regardless of the field representation. The less precise field-based representation causes more methods to be found reachable, increasing the number of iterations required.

The nt-otf-fs line shows how much ignoring declared types hurts space efficiency (the "oom" for `jedit` signifies that the analysis exceeded the 1700MB of memory allotted). The alias edge algorithm is the only one that can handle the resulting large sets with reasonable memory requirements. This algorithm spends a significant amount of time building alias edges rather than propagating points-to sets, so the benefit from the incremental version is much smaller. In fact, for the analyses requiring few iterations (ot-cha-fs and ot-cha-fb), the overhead of the incremental version outweighs the reduction in the size of sets to be propagated, and is even slightly slower than the non-incremental version.

In summary, Table 5.4 demonstrates the following key points about the tradeoff between analysis time and space.

- The incremental worklist algorithm is the fastest for most analyses, except for the field-based analysis using a CHA-based call graph, for which the non-incremental worklist algorithm is faster.

- The non-incremental algorithms require less memory than their incremental counterparts.

- For field-based analyses, the space requirements of the non-incremental versions of the worklist and alias edge propagation algorithms are comparable; however, for field-sensitive analyses, especially of the large `jedit` benchmark, the alias edge propagation algorithm requires significantly less memory.

- When declared types are not respected during the analysis, only the alias edge algorithm can complete in a reasonable amount of memory.

5.3.3 Graph Simplification

Rountev and Chandra [RC00] showed that simplifying the pointer assignment graph by merging nodes known to have equal points-to sets speeds up the analysis. The behaviour of Spark agrees with their findings.

When respecting declared types, a cycle can only be merged if all nodes in the cycle have the same declared type, and a single-entry subgraph can only be merged if all its nodes have declared types that are supertypes of the predecessor. Since the experimental results presented earlier suggested that respecting declared types makes the analysis much faster, as well as more precise, it is useful to know how much respecting declared types reduces the opportunities for simplification. These measurements are presented in Table 5.5. On the benchmarks in this study, between 6% and 7% of variable nodes were removed by collapsing cycles, compared to between 5% and 6% when declared types were respected. Between 59% and 62% of variable nodes were removed by collapsing single-entry subgraphs, compared to between 55% and 58% when declared types were respected. Thus, the effect of respecting declared types on simplification is minor.

Benchmark		SCC	SESG	Both
`compress`	nt-cha-fs	6.7%	59.5%	60.7%
	ot-cha-fs	5.3%	55.6%	56.4%
	ot-otf-fs	1.1%	31.5%	31.6%
`javac`	nt-cha-fs	7.1%	59.8%	61.4%
	ot-cha-fs	5.7%	55.8%	57.0%
	ot-otf-fs	1.1%	32.2%	32.3%
`sablecc`	nt-cha-fs	6.4%	60.4%	61.6%
	ot-cha-fs	5.0%	56.3%	57.0%
	ot-otf-fs	1.0%	31.9%	32.0%
`jedit`	nt-cha-fs	7.1%	61.7%	63.0%
	ot-cha-fs	5.6%	57.8%	58.8%
	ot-otf-fs	1.3%	33.3%	33.5%

Table 5.5: Simplification

On the other hand, when constructing the call graph on-the-fly, no interprocedural edges are present before the analysis begins. This means that any cycles spanning multiple methods are broken, and the corresponding nodes cannot be merged. The 6%-7% of nodes removed by collapsing cycles dropped to 1%-1.5% when the call graph was constructed on-the-fly. The 59%-62% of nodes removed by collapsing single-entry subgraphs dropped to 31%-33%. When constructing the call graph on-the-fly, simplifying the pointer assignment graph before the analysis has little effect, and on-the-fly cycle detection methods should be used instead.

5.4 Overall Results

Based on the experimental results reported up to this point, three analyses appear to be good compromises between precision and speed, with reasonable space requirements. Each of the three analyses should be implemented using the hybrid set implementation.

1. ot-otf-fs (declared types, on-the-fly call graph, field-sensitive) is suitable for applications requiring the highest precision. For this analysis, the incremental worklist algorithm works best.

2. ot-cha-fs (declared types, CHA-based call graph, field-sensitive) is much faster, but with a drop in precision as compared to ot-otf-fs (mostly because it includes significantly more call edges). For this analysis, the incremental worklist algorithm works best.

3. ot-cha-fb (declared types, CHA-based call graph, field-based) is the fastest analysis, completing in a single iteration, but it is also the least precise. For this analysis, the non-incremental worklist algorithm works best.

Table 5.6 shows the results of these three analyses on the full set of benchmarks. The first column gives the benchmark name (`javac` is listed twice: once with the 1.3.1_01 JDK class library, and once with the 1.1.8 JDK class library). The remaining columns give the analysis time, total space, and precision for each of the three recommended analyses. The total space includes the space used to store the pointer assignment graph as well as the points-to sets; these were reported separately in previous tables. The precision is measured as the percentage of field dereference sites at which the points-to set of the pointer being dereferenced has size 0 or 1; for a more detailed measurement of precision, see Table 5.2.

(time in seconds, space in MB, precision in precent)

Benchmark	ot-otf-fs			ot-cha-fs			ot-cha-fb		
	time	space	prec.	time	space	prec.	time	space	prec.
`compress`	52	106	69.1	13	127	60.1	10	90	57.6
`db`	52	107	68.9	14	128	59.9	11	90	57.4
`jack`	54	112	68.7	14	132	60.1	11	94	57.6
`javac` (1.1.8)	8	27	63.6	3	24	57.4	1	16	55.1
`javac` (1.3.1)	89	131	66.3	18	148	58.4	11	104	56.2
`jess`	57	115	68.1	15	136	59.2	10	97	56.8
`mpegaudio`	56	112	68.6	16	134	59.7	11	93	57.4
`raytrace`	53	107	68.5	13	129	59.6	11	91	57.1
`sablecc`	95	136	70.5	18	158	62.5	14	112	60.3
`soot`	88	143	68.3	19	162	60.4	18	116	58.4
`jedit`	100	218	69.1	38	244	62.3	21	143	61.1

Table 5.6: Overall Results

Chapter 6
Client Analyses

6.1 Call Graph Construction

In an object-oriented polymorphic language such as Java, the method that is invoked at a virtual call site depends on the run-time type of the receiver object. Any interprocedural program analysis therefore needs some way to approximate the set of target methods that could possibly be invoked at each call site. That is, it needs an approximation of the *call graph*. Making the call graph precise is important because it both improves the precision, and reduces the cost, of subsequent analyses. Also, for applications in embedded systems, where memory is scarce, a precise call graph in which fewer methods are determined to be possibly reachable is useful for reducing the memory footprint of the code.

Constructing a call graph is one natural application of points-to information. The points-to analysis computes a set of objects to which each variable may point. We can deduce the run-time type of each of these objects to obtain a set of possible types of objects pointed-to by each variable. Using the set for the receiver variable at each call site, for each type, the method that will be invoked is identified according to the method dispatch specification of the language. This yields a list of possible target methods for each call site, from which the call graph is constructed.

A call graph builder has been implemented which uses the points-to sets computed by SPARK to compute a call graph. The rest of this section is a study of the effect of the points-to analysis on the precision of the call graph.

Table 6.1 shows measurements of the precision of the call graph constructed using five different analyses on the benchmarks described in Section 5.1. Class Hierarchy Analysis [DGC95] and Variable Type Analysis [SHR+00] are two previously-published call graph construction algorithms. The other three analyses are constructions of the call graph from the points-to information computed by SPARK. As before, ot-cha-fb indicates a field-based points-to analysis starting from a CHA-based call graph, ot-cha-fs indicates a field-sensitive points-to analysis starting from a CHA-based call graph, and ot-otf-fs indicates a field-sensitive points-to analysis in which the call graph is constructed during the analysis. For each analysis, the first column gives the number of methods that were determined to be possibly reachable in the call graph, and the second column gives the percentage of call sites in the CHA-reachable methods that were determined to have receiver sets of zero or one methods. These call sites are significant because their target method is uniquely determined, enabling optimizations such as method inlining or call devirtualization.

The call graph produced from the field-based points-to analysis is very similar to the one produced by VTA, which is to be expected because the analyses are very similar. VTA differs from the field-based points-to analysis only in that all objects of a given run-time type are modelled together, rather than being distinguished by their allocation site. That is, all allocation sites allocating the same type of object are modelled with a single allocation node, while SPARK uses a separate allocation node for every allocation site.

Making the points-to analysis field-sensitive produces a moderate improvement in call graph precision, at the cost of some analysis time. A much more dramatic improvement is obtained by the call graph on-the-fly during the points-to analysis, rather than starting with a CHA-based call graph. Note, however, that such an analysis is significantly more costly than the simpler analyses, like the field-based analysis or VTA, as shown in Table 5.6. This suggests that further research should be done into analyses that build the call graph on-the-fly, to make them competitive

	CHA		VTA		ot-cha-fb		ot-cha-fs		ot-otf-fs	
Benchmark	mthds	sites	mthds	sites	mthds	sites	mthds	sites	mthds	sites
`compress`	15737	71.3	14042	90.2	14015	90.2	13237	90.6	10842	94.9
`db`	15739	71.3	14042	90.2	14015	90.2	13239	90.6	10844	95.0
`jack`	15995	69.8	14298	90.3	14271	90.3	13494	90.8	11099	95.0
`javac`	16872	71.5	15167	89.7	15140	89.7	14374	90.1	11982	94.1
`jess`	16348	71.8	14637	90.5	14610	90.5	13833	90.9	11450	95.1
`mpegaudio`	15947	71.3	14285	90.2	14258	90.2	13489	90.6	11072	94.9
`raytrace`	15866	71.7	14173	90.3	14146	90.3	13362	90.7	10968	95.0
`sablecc`	17530	71.7	15826	90.0	15799	90.0	15023	90.4	12700	94.5
`soot`	18053	71.4	16364	89.7	16337	89.7	15558	90.1	13104	94.1
`jedit`	20199	74.0	18614	90.7	18595	90.7	18456	90.9	16267	94.1

Table 6.1: Call Graph Precision

in efficiency with simpler analyses, and to improve their precision even further.

6.2 Side-effect Analysis

6.2.1 Background

Side-effect analysis is an application of points-to analysis that can aid a compiler to produce more aggressively optimized code. The purpose of this analysis is to approximate the sets of run-time objects which each instruction and each method of the program may read or write. Having such an approximation may allow a compiler to eliminate redundant loads and stores in the presence of method calls. It may also improve precision of other intraprocedural analyses, which may in turn enable many other optimizations.

As an example, consider the code fragment in Figure 6.1. If we knew that `bar()` does not write `this.a`, then we could move the load of `this.a` out of the loop, assuming no concurrent writes by any other threads. We could then recognize `d` as

```
foo() {
    this.a = 2;
    b = 0;
    for( int c = 0; c < 1000000; c++ ) {
        d = this.a;
        e = this.bar();
        b = b + d;
    }
    System.out.println( "b = "+b );
    System.out.println( "e = "+e );
}
```

Figure 6.1: Code Example for Side-Effect Analysis

a compile-time constant `2`, and `b` as an induction variable not used inside the loop. The additions could then be turned into a single multiplication `2 * 1000000` outside the loop, which could be evaluated at compile-time. We could attempt an even more ambitious optimization if we knew that `bar()` performs no writes or native method calls: we could move the call out of the loop. The optimized code resulting from these optimizations is shown in Figure 6.2. Note that all of these optimizations depend on knowing that `bar()` has no side-effects.

```
foo() {
    this.a = 2;
    b = 2000000;
    e = this.bar();
    System.out.println( "b = "+b );
    System.out.println( "e = "+e );
}
```

Figure 6.2: Optimized Version of Code Example

In order to approximate the sets of objects written at various points in the program, a side-effect analysis needs information about which variables point to which objects. That is, a side-effect analysis depends on a points-to analysis. For this reason, a side-effect analysis has been developed based on SPARK. The side-effect analysis obtains the points-to information it requires from SPARK. Its output can either be used directly by optimizations within Soot, or it can be encoded in class file attributes, where it can be used by other systems, such a just-in-time compilers.

This section describes the implementation of the side-effect analysis and the encoding of its results in attributes. It also gives experimental evidence that the analysis produces precise approximations of side-effects compared to the simple heuristics typically used in just-in-time compilers and in Soot, and that the encoding is a sufficiently efficient representation of the side-effect information.

6.2.2 Representation of Side-Effect Information

Side-effect information expresses dependences between instructions. For example, a client might want to know whether a write `p.f = a;` in one instruction may overwrite the value written in another instruction `q.f = b;`. In Java class file attributes, it is difficult to encode an expression such as `p.f`, because the local variable `p` appears in the bytecode as an unlabeled stack location. Moreover, the set of heap locations which an instruction may read or write can be very large. In this case, it could be very costly for the client using the side-effect information to recover the dependences between instructions from the read and write sets.

Instead of encoding the field expressions and read and write sets in attributes, the implementation directly encodes the dependences between instructions. For example, a write to `p.f` overwriting the value written to `q.f` would be encoded as a Write-Write dependence between the two bytecode instructions writing `p.f` and `q.f`. A client reading the attribute can convert this dependence into whatever internal representation it has for `p.f` and `q.f`. For each pair of statements, the attribute specifies whether there is a Write-Write, Write-Read, Read-Write, or Read-Read dependence between them. Although the Read-Read dependences may not be useful to a just-in-time compiler, they are included for completeness; they could be removed if it were necessary to reduce the space required by the attributes.

The size of this representation grows quadratically as the number of inter-dependent instructions in the method being analyzed. Most methods are short, and even longer methods tend to have few instructions that are inter-dependent. However, some methods are like the constructor of `spec.io.TableOfExistingFiles`, a class contained in the harness of all the SPECjvm [Spec] benchmarks. This method consists of 633 calls to the `put` method of `java.util.Hashtable`. Since all of these calls read and write the same locations, they should all have dependences between them encoded, leading to $\binom{633}{2} = 200028$ dependences of each type (Write-Write, Write-Read, Read-Write, and Read-Read). Furthermore, the methods called from each of these call sites possibly call a large number of other methods, so the call sites take a long time and a large amount of memory to analyze.

To limit the growth of the attribute size and amount of computation required, the side-effect analysis uses the following method to reduce the size of the set of dependences as it is being computed. Each instruction is assigned a pair of numbers, representing the sets of locations that the instruction can read and write. Dependences are then computed between these numbered read and write sets, rather than the instructions themselves. The simplest such assignment of numbered locations would assign distinct locations to each instruction, and the resulting dependence graph would be as large as the dependence graph between instructions. However, some sets of instructions can easily be determined to read or write the same locations, and can therefore share the same numbered locations, reducing the effective number of instructions to be considered. Specifically, all method calls with equal sets of possible target methods share read and write locations. Also, all field reference expressions having the same base pointer and the same field share the same location. This reduces the 633 method calls in `spec.io.TableOfExistingFiles` to a single pair of numbered locations, drastically reducing the size of the attribute and the time and memory needed to compute it. However, this approach makes it slightly more difficult for the client to extract the information. In order to determine whether there is a dependence between two instructions, it must look up the numbered locations read and written by the instructions, and then look in the graph for dependences between these locations. This reduced form of the dependence information still has a worst-case size quadratic in the size of each method. However, as the experimental results in Section 6.2.6 show, in practice, the size of this representation is acceptable.

In addition to the relationships between the locations read and written by statements, the side-effect attribute encodes, for each call site, whether a native method may be called from the call site, or transitively from any methods that may be called from it. This information may be useful to clients of the side-effect analysis, and it is trivial to compute while computing the side-effect information.

6.2.3 Implementation of Side-Effect Analysis

The points-to analysis produces, for each local variable of pointer type, an abstract set of the possible locations to which it could point. From this information, the side-effect analysis computes abstract sets of locations read and written by each instruction. These locations include instance fields, static fields, and array elements. The abstract sets for each instruction are combined into larger abstract sets for whole methods. These sets contain all locations accessed within the method, but not those accessed in other methods that it may call. Finally, the sets for each method are combined into even larger sets that encode, for each call site, the set of locations accessed in all the methods possibly called from the call site, and other methods transitively called from them. This yields a read and write set for every instruction, including method invoke instructions. These read and write sets are then used to determine whether dependences exist between them.

A naive implementation of this recursive definition of read and write sets of call sites would be intractable, because many call sites have large numbers of transitive targets, and the sets for each target would have to be recomputed at each call site. A natural optimization would be to use memoization to avoid computing points-to sets of each method and of each call site more than once. Unfortunately, such an implementation has prohibitive memory requirements to store all the read and write sets, even for medium-sized programs. The current implementation therefore makes a compromise between memory requirements and running time: it memoizes the read and write sets accessed by each statement and method, but not the read and write sets accessed by each call site.

6.2.4 Attribute Encoding

The side-effect information is encoded in Java class file attributes using the annotation framework included in Soot [PQVR+01]. This section describes in detail the format of these attributes. The side-effect information for each method is encoded in two attributes: a code attribute with the name SideEffectAttribute, and a method attribute with the name DependenceGraph.

SideEffectAttribute

This attribute maps statements to abstract locations read and written, and also indicates which invoke statements may transitively call native methods.

0	1	0	1	2	3	4	5	6	
record count		bytecode offset		read set		write set		calls native	• • •

The first two bytes of the attribute are a big-endian integer specifying the number of records that follow.

Each record that follows consists of seven bytes:

- The first two bytes are a big-endian integer specifying the bytecode offset of the instruction that this record describes.
- The third and fourth bytes are the number of the numbered location *read* by the instruction that this record describes.
- The fifth and sixth bytes are the number of the numbered location *written* by the instruction that this record describes.
- The least significant bit of the seventh byte is one if the instruction that this record describes invokes a method that may be a native method, and zero otherwise. The remaining bits are reserved for future use.

The special numbered location `0xffff` indicates a non-existent location, and is used to indicate that an instruction does not read or write anything. For example, the record for a `getfield` bytecode instruction will specify the location that the instruction reads, and `0xffff` for the location that it writes, since this instruction performs no writes.

DependenceGraph

This attribute specifies dependences between numbered locations.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td rowspan="2">•••</td></tr>
<tr><td colspan="2">set</td><td colspan="2">set</td></tr>
</table>

It consists of a number of records, each four bytes in length. The first two bytes and the last two bytes of each record each specify a numbered location. If a numbered location may overlap another numbered location, then the two locations will appear as a record in this attribute. Note that each unordered pair of locations is encoded in the attribute only once, with the lower-numbered location listed first, but the relation is symmetric.

6.2.5 Side-Effect Example

The format of the side-effect attributes will now be demonstrated using a more complete example than the one presented in the introduction to this section. First, the Java code for the example is presented in Figure 6.3. Then, the computed side-effect information is presented as comments in a Jimple version of the code for the `main` method in Figure 6.4. Finally, a disassembled representation of the resulting bytecode for the `main` method is presented in Figure 6.5.

```
class Example {
    int x = 0;
    public void bar() {
        this.x = 5;
    }
    public static final void main( String[] argv ) {
        Example s1 = new Example();
        Example s2 = new Example();
        Example s3 = s2;
        int sum = 0;

        s1.x = 1;
        s3.x = 1;
        for( int i = 0; i < 1000000; i++ ) {
            sum += s1.x;
            s2.x = 0;
            s3.bar();
        }
    }
}
```

Figure 6.3: Java Code for Side-Effect Example

After each statement that may read or write to memory, the Jimple representation in Figure 6.4 contains a comment of the form `// SEReads : 1`. These indicate the numbered locations that are read and written by the statement. The two calls to the constructor `<init>` read and write the same locations, `0` and `1`, respectively. The store to field `x` of `r2` writes location `2`, which is then read by the load in the line immediately after `label0:`. At the beginning of the code, the dependence graph comment shows which pairs of locations may overlap. The location `0`, which represents the read set of the constructor overlaps nothing, because the constructor does not read anything. The location `1` representing the write set of the constructor overlaps locations `2`, `3`, `4`, and `5`, because these all refer to the field `x` of some object, and this field is written by the constructor. Locations `2`, `3`, and `4` refer to the field `x` of `s1`, `s3`, and `s2`, respectively, of the original Java program. The dependence graph shows that locations `3` and `4` overlap, because `s2` and `s3` are aliased; however, location `2` does not overlap with locations `3` or `4`, because `s1` is not aliased to either `s3` or `s2`. Similarly, location `5` representing the write set of the `bar()` method overlaps with locations `3` and `4` but not with `2`, because the `bar()` method writes the field `x` of the object that `s3` and `s2` point to, but not the object that `s1` points to.

In the bytecode presented in Figure 6.5, the side-effect information has been encoded in two attributes: `DependenceGraph` at the top of the code, and `SideEffectAttribute` at the bottom. The `DependenceGraph` attribute encodes the pairs that appeared in the dependence graph comment in the Jimple code. The `SideEffectAttribute` encodes the read and write sets of individual statements. The first and second entries correspond to the calls to the `<init>` method at bytecode offsets 4 (`00 04`) and 12 (`00 0c`). They show that each of these statements reads location `0` (`00 00`) and writes location `1` (`00 01`). The field stores (`putfield`) at bytecode offsets 22 (`00 16`), 27 (`00 1b`), and 45 (`00 2d`) read nothing (`ff ff`), and write locations 2 (`00 02`), 3 (`00 03`), and (`00 04`), respectively. The field load (`getfield`) at bytecode offset 38 (`00 26`) reads location 2 (`00 02`) and writes nothing (`ff ff`). Finally, the call to `bar()` at bytecode offset 49 (`00 31`) reads location 0 (`00 00`) and writes location 5 (`00 05`).

```
    public static final void main(java.lang.String[] )
// Dependence Graph
// (1,2), (1,3), (1,4), (1,5), (3,4), (3,5), (4,5)
    {
        java.lang.String[] r0;
        Example $r1, r2, r3, r4, $r5;
        int i0, i1, $i2;

        r0 := @parameter0: java.lang.String[];
        $r1 = new Example;
        specialinvoke $r1.<Example: void <init>()>();
// SEReads : 0
// SEWrites: 1

        r2 = $r1;
        $r5 = new Example;
        specialinvoke $r5.<Example: void <init>()>();
// SEReads : 0
// SEWrites: 1

        r3 = $r5;
        r4 = r3;
        i0 = 0;
        r2.<Example: int x> = 1;
// SEWrites: 2

        r4.<Example: int x> = 1;
// SEWrites: 3

        i1 = 0;
        goto label1;

     label0:
        $i2 = r2.<Example: int x>;
// SEReads : 2

        i0 = i0 + $i2;
        r3.<Example: int x> = 0;
// SEWrites: 4

        virtualinvoke r4.<Example: void bar()>();
// SEReads : 0
// SEWrites: 5

        i1 = i1 + 1;

     label1:
        if i1 < 1000000 goto label0;

        return;
    }
```

Figure 6.4: Jimple Code for Side-Effect Example

```
public static final void main(String[] arg0)
[(attribute DependenceGraph:
00 01 00 02
00 01 00 03
00 01 00 04
00 01 00 05
00 03 00 04
00 03 00 05
00 04 00 05
)]
Code(max_stack = 2, max_locals = 5, code_length = 63)
0:    new              <Example> (21)
3:    dup
4:    invokespecial    Example.<init> ()V (24)
7:    astore_0
8:    new              <Example> (21)
11:   dup
12:   invokespecial    Example.<init> ()V (24)
15:   astore_1
16:   aload_1
17:   astore_2
18:   iconst_0
19:   istore_3
20:   aload_0
21:   iconst_1
22:   putfield         Example.x I (18)
25:   aload_2
26:   iconst_1
27:   putfield         Example.x I (18)
30:   iconst_0
31:   istore           %4
33:   goto             #55
36:   iload_3
37:   aload_0
38:   getfield         Example.x I (18)
41:   iadd
42:   istore_3
43:   aload_1
44:   iconst_0
45:   putfield         Example.x I (18)
48:   aload_2
49:   invokevirtual    Example.bar ()V (20)
52:   iinc             %4      1
55:   iload            %4
57:   ldc              1000000 (23)
59:   if_icmplt        #36
62:   return

Attribute(s) =
(attribute SideEffectAttribute:
00 07 00 04 00 00 00 01 00
      00 0c 00 00 00 01 00
      00 16 ff ff 00 02 00
      00 1b ff ff 00 03 00
      00 26 00 02 ff ff 00
      00 2d ff ff 00 04 00
      00 31 00 00 00 05 00
)
```

Figure 6.5: Bytecode for Side-Effect Example

6.2.6 Experimental Results

The section reports results of experiments that were performed to determine the effectiveness of the side-effect analysis and the attribute encoding. Specifically, the following two quantities were measured:

1. The size of the attributes compared to the size of the original bytecode.

2. The percentage of dependences between instructions within a method ruled out by the side-effect analysis.

These measurements were performed on the same benchmarks as described in Section 5.1.

Attribute Size

Table 6.2 gives the size of the side-effect attributes as a percentage of the size of the original class files. For most of the benchmarks, the attributes are between 25% and 50% of the original class file size, and in no case do they exceed the original size. Considering that the attributes encode all the information available to the side-effect analysis, the size of the encoding is acceptable.

The attributes are very regular, and are therefore likely to be highly compressible with standard compression algorithms. However, the purpose of SPARK is to facilitate experimentation, and use of such an algorithm would increase the burden on the client reading the attributes, which would have to decompress them. Therefore, no such compression algorithm was applied. In a production system, compression would almost certainly be desirable.

Dependences

Many ahead-of-time and just-in-time Java compilers make the following conservative assumptions about the side-effects of instructions:

- Field accesses of the same field of any object may be aliased.

Benchmark	Size increase
`compress`	24.5
`db`	30.1
`jack`	46.0
`javac`	35.7
`jess`	41.2
`mpegaudio`	33.2
`raytrace`	41.0
`sablecc`	96.4
`soot`	49.8
`jedit`	37.5

Table 6.2: Attribute Size as Percentage of Original Class File Size

- Methods other than the method being analyzed may read and write any fields on the heap.

This means that in these systems, for each field, there are dependences between all reads and writes of it, and there are dependences between method invocation instructions and all instructions that access the heap. Table 6.3 presents measurements of the percentage of these dependences that are ruled out by the side-effect analysis. That is, it shows how much precision the side-effect analysis adds to these common conservative assumptions. As before, ot-cha-fb indicates a field-based points-to analysis starting from a CHA-based call graph, ot-cha-fs indicates a field-sensitive points-to analysis starting from a CHA-based call graph, and ot-otf-fs indicates a field-sensitive points-to analysis in which the call graph is constructed during the analysis.

The numbers reflect the relative complexity of the benchmarks. On the very simple benchmarks, such as `compress` and `db`, the conservative assumption is successful in minimizing the number of dependences, leaving little room for the side-effect analysis to show improvement. On the other hand, on the highly object-oriented benchmarks,

Benchmark	ot-cha-fb	ot-cha-fs	ot-otf-fs
`compress`	2.5	2.6	2.6
`db`	2.8	2.9	2.9
`jack`	13.1	13.1	13.1
`javac`	19.4	19.4	19.4
`jess`	14.3	14.4	14.5
`mpegaudio`	5.9	5.9	6.0
`raytrace`	18.8	18.8	18.8
`sablecc`	56.1	56.2	56.2
`soot`	64.8	65.3	65.3
`jedit`	34.1	34.1	35.3

Table 6.3: Percentage of Dependences Ruled Out by Side-Effect Analysis

such as `sablecc` and `soot`, the side-effect analysis manages to rule out more than half of the dependences that the field-based assumption could not. The differences due to varying the precision of the points-to analysis are very small; only for the `jedit` benchmark is the difference between the most precise, field-sensitive on-the-fly call graph analysis and the least precise, field-based CHA call graph analysis more than one percent of the dependences.

Note that the number of dependences ruled out does not tell us whether those dependences that were ruled out are important to optimizations. It is therefore difficult to predict from this data the effect of side-effect analysis on the effectiveness of optimizations. However, the high numbers of dependences ruled out suggest that side-effect analysis could have a significant effect. Also, it appears that the fast, field-based points-to analysis using a CHA-based call graph is precise enough to produce this effect.

6.2.7 Future Work on Side-Effect Analysis

An effective side-effect analysis has been built on top of SPARK. Its output is encoded in class file attributes, where it can be used by other systems. An obvious area for further experimentation is modifying optimizing compilers to make use of this side-effect information, and to study how different points-to analyses affect the optimizations made possible by side-effect analysis.

Chapter 7
Conclusions and Future Work

7.1 Conclusions

This thesis introduced SPARK, a flexible framework for experimenting with points-to analyses of Java programs. It presented the modular design of SPARK, and details of its implementation. SPARK was used to perform a substantial study of factors affecting the precision and efficiency of points-to analyses for Java, and the results of this study were reported. Based on these experiments, three variations of points-to analyses were selected as particularly effective for Java, in light of the high precision of their results combined with efficient execution of the analysis. Two implementations of client analyses using the points-to information were presented: call graph construction and side-effect analysis. Other clients are planned in the future.

The flexibility of SPARK comes from its modular design. Individual implementations of its components are designed to be interchangeable, leading to large numbers of possible combinations of variations. The division of SPARK into three stages, connected using the pointer assignment graph, facilitates the creation of and experimentation with additional modules implementing new points-to algorithms.

SPARK includes several implementations of its main components. A pointer assignment graph builder is used to create a representation of the program being analyzed, to be processed by the rest of SPARK. SPARK includes two simplification algorithms to reduce the size of the pointer assignment graph. The current version of Spark contains five points-to set propagation algorithms, each of which is particularly suited to specific variations of points-to analysis. Four different implementations of data structures for representing points-to sets are included with Spark.

The use of Spark was demonstrated in an extensive study of the factors affecting precision and efficiency of Java points-to analyses. Respect for declared types and casts was shown to be extremely important for both analysis precision and efficiency. Constructing a call graph during the points-to analysis improves precision with a moderate cost in analysis time. The improvement in precision of a field-sensitive analysis over a field-based analysis is moderate, and comes at little additional cost when an efficient points-to set implementation and propagation algorithm is used. The hybrid points-to set implementation was shown to be consistently more efficient than all other implementations studied; it is up to two orders of magnitude more efficient than the implementation based on the `HashSet` class included in the Java standard class library. The worklist-based propagation algorithm was shown to be the most efficient in terms of time, while the alias edge propagation algorithm was the most efficient in terms of space when the points-to sets were allowed to grow very large by not making use of declared type information. The incremental versions of the algorithms were faster than the non-incremental versions when the analysis required many iterations, while for the simpler analyses requiring little iteration, the overhead of the incremental version outweighed the benefit. Off-line simplification of the pointer assignment graph was shown to be compatible with respect for declared types: that is, respecting declared types does not significantly decrease the opportunities for simplification. However, off-line simplification is nearly useless if the call graph is not computed prior to the analysis. Because Spark is already so efficient at analyses for which the call graph *is* computed ahead of time, it is not clear that simplifying the pointer assignment graph ahead of time is worthwhile for Java.

Spark has been used as the basis of two client analyses. The call graph construction based on Spark is more general, more efficient and more precise than VTA [SHR+00], the analysis previously available in the Soot framework. Spark is also the basis of a side-effect analysis whose output is encoded in class file attributes.

This side-effect analysis has been shown to provide significantly more information than the conservative assumptions used in typical just-in-time compilers. It therefore shows promise in improving the optimizations performed by such systems.

SPARK has been demonstrated to be a practical, flexible and efficient framework on which further point-to analysis research can be based.

7.2 Future Work

The purpose of SPARK is to serve as a framework to facilitate experimentation with points-to analyses for Java. This section describes some of the areas in which SPARK could be used.

7.2.1 Precision of Data Flow Analyses

In the absence of accurate points-to information, traditional data flow analyses used for optimization — such as constant propagation, constant subexpression elimination, and partial redundancy elimination — are forced to make conservative assumptions. This reduces the precision of the analyses and the opportunities for optimization.

Soot is a framework for implementing these data flow analyses and related optimizations. Since SPARK is a part of Soot, analyses implemented in Soot can now be improved to take advantage of the points-to information provided by SPARK. The effect of points-to information on these analyses can be the subject of future research.

7.2.2 Using Side-Effect Information in Just-In-Time Compilers

Section 6.2 described a side-effect analysis that has been implemented on top of SPARK, whose results are stored in attributes for the use of other compilers, including just-in-time compilers. An interesting area of future research would be to modify existing just-in-time compilers to make use of this information, and to study the effect that it can have on the effectiveness of their optimizations.

7.2.3 Points-To Analysis Algorithms and Set Implementations

This thesis included a study of the points-to analysis algorithms and points-to set implementations included in SPARK, and they were found to be very effective. However, programs are becoming larger, and points-to information is being used in new areas, such as program understanding and verification. Because of these changes, more efficient and more precise points-to analyses will continue to be needed. The flexibility of SPARK makes it a natural platform on which to experiment with and compare future points-to analysis algorithms.

In particular, implementing a demand-driven analysis like the one designed for C by Heintze and Tardieu [HT01b, HT01a] may further improve the performance of SPARK. Another interesting area to be explored is the use of binary decision diagrams [Bry92] to represent the large points-to relation that must be manipulated [BLQ+02, BLQ+03].

7.2.4 Context-Sensitivity

Context-sensitive points-to analyses can produce much more precise information than context-insensitive ones. In an object-oriented language that encourages encapsulation, such as Java, the information lost due to context-insensitivity is especially significant. Unfortunately, context-sensitive analyses are prohibitively costly to compute for moderately large programs, and, due to the large class library, even trivial Java programs are moderately large.

However, the excellent performance of SPARK may make some context-sensitive algorithms feasible. In addition, SPARK can be used to experiment with new algorithms with only a limited degree of context-sensitivity, specifically designed for analyzing object-oriented languages. For example, SPARK would be an ideal framework in which to implement the *object-sensitive* points-to analysis [MRR02b] proposed by Milanova, Rountev and Ryder.

7.2.5 Precision of Call Graph Construction

The Java language specifies rules with subtle effects on the control flow of a program that must be taken into account by whole-program analyses such as points-to analysis. The following are several examples.

- The first reference to a class causes its static initializer method to execute.
- Finalizer methods are executed automatically by the system without any explicit calls to them.
- Methods related to thread creation can be executed without being explicitly invoked.
- Reflection can be used to create arbitrary objects and execute arbitrary methods that cannot be identified statically.

Most whole-program analyses handle these issues either using very conservative assumptions, leading to large call graphs, or by ignoring them, leading to possibly incorrect analysis results. Although SPARK is already able to produce precise call graphs, even more precise methods of modelling these effects could further improve both the precision and efficiency of SPARK.

Appendix A
Using Spark

A.1 Obtaining Spark

SPARK is a part of the Soot bytecode analysis and transformation framework. Soot is maintained by the Sable Research Group at McGill University, and is freely available under the Lesser General Public Licence.
Soot can be downloaded from the Soot homepage:

- `http://www.sable.mcgill.ca/soot/`

Javadoc documentation for the Soot source is available from:

- `http://www.sable.mcgill.ca/soot/doc/`

This includes documentation for SPARK, which is found in the package `soot.jimple.spark` and its subpackages.
Tutorials on using Soot are available at:

- `http://www.sable.mcgill.ca/soot/tutorial`

Questions, discussions, and comments about Soot and SPARK should be directed to the Soot mailing list. Instructions about subscribing to the list are found on the Soot homepage. Archives of the list are found at:

- `http://www.sable.mcgill.ca/listarchives/soot-list/`

A.2 Spark Options

This section describes the command-line options to SPARK. Values for options are specified on the Soot command-line, following the switch `-p wjtp.Spark`. For example:

```
java soot.Main -a --app -p wjtp.Spark disabled:false,verbose:true Hello
```

For the most current, automatically generated documentation of SPARK options, please see the file `src/soot/jimple/spark/opts.ps` in the Soot distribution.

A.2.1 General Options

Option `verbose`

- Allowed values: `true false`
- Default value: `false`

When this option is set to `true`, SPARK prints detailed information.

Option `ignoreTypesEntirely`

- Allowed values: `true false`
- Default value: `false`

When this option is set to `true`, all parts of SPARK completely ignore declared types of variables and casts.

Option `forceGCs`

- Allowed values: `true false`
- Default value: `false`

When this option is set to `true`, calls to `System.gc()` will be made at various points to allow memory usage to be measured.

A.2.2 Pointer Assignment Graph Building Options

Option `VTA`

- Allowed values: `true` `false`
- Default value: `false`

Setting `VTA` to `true` has the effect of setting `ignoreBaseObjects`, `typesForSites`, and `simplifySCCs` to `true` to simulate Variable Type Analysis [SHR+00]. Note that the algorithm differs from the original VTA in that it handles array elements more precisely. To use the results of the analysis to trim the invoke graph, set the `trimInvokeGraph` option to `true` as well.

Option `RTA`

- Allowed values: `true` `false`
- Default value: `false`

Setting `RTA` to `true` sets `typesForSites` to `true`, and causes SPARK to use a single points-to set for all variables, giving pessimistic Rapid Type Analysis [BS96]. To use the results of the analysis to trim the invoke graph, set the `trimInvokeGraph` option to `true` as well.

Option `ignoreBaseObjects`

- Allowed values: `true` `false`
- Default value: `false`

When this option is set to `true`, fields are represented by variable nodes, and the object that the field belongs to is ignored (all objects are lumped together). This is also referred to as a field-based analysis. Otherwise, fields are represented by field reference nodes, and the objects that they belong to are distinguished, giving a field-sensitive analysis.

Option `typesForSites`

- Allowed values: `true false`
- Default value: `false`

When this option is set to `true`, types rather than allocation sites are used as the elements of the points-to sets.

Option `mergeStringBuffer`

- Allowed values: `true false`
- Default value: `true`

When this option is set to `true`, all allocation sites creating objects of type `java.lang.StringBuffer` are grouped together as a single allocation site.

Option `simulateNatives`

- Allowed values: `true false`
- Default value: `true`

When this option is set to `true`, effects of native methods are simulated.

Option `simpleEdgesBidirectional`

- Allowed values: `true false`
- Default value: `false`

When this option is set to `true`, all edges connecting variable nodes are made bidirectional, as in Steensgaard's analysis [Ste96b].

Option `onFlyCallGraph`

- Allowed values: `true` `false`
- Default value: `false`

When this option is set to `true`, the call graph is computed on-the-fly as points-to information is computed. Otherwise, an initial approximation to the call graph is used.

Option `parmsAsFields`

- Allowed values: `true` `false`
- Default value: `false`

When this option is set to `true`, parameters to methods are represented as fields of the `this` object; otherwise, parameters are represented as variable nodes.

Option `returnsAsFields`

- Allowed values: `true` `false`
- Default value: `false`

When this option is set to `true`, return values from methods are represented as fields of the `this` object; otherwise, return values are represented as variable nodes.

A.2.3 Pointer Assignment Graph Simplification Options

Option `simplifyOffline`

- Allowed values: `true` `false`
- Default value: `false`

When this option is set to `true`, variable nodes in the same single-entry subgraph are merged together (since they must have equal points-to sets).

Option `simplifySCCs`

- Allowed values: `true false`
- Default value: `false`

When this option is set to `true`, variable nodes which form strongly-connected components are merged together (since they must have the same points-to set).

Option `ignoreTypesForSCCs`

- Allowed values: `true false`
- Default value: `false`

When this option is set to `true`, when collapsing strongly-connected components, nodes forming SCCs are collapsed regardless of their type. The collapsed SCC is given the most general type of all the nodes in the component.

When this option is set to `false`, only edges connecting nodes of the same type are considered when detecting SCCs.

This option has no effect unless `simplifySCCs` is `true`.

A.2.4 Points-To Set Flowing Options

Option `propagator`

- Allowed values: `iter worklist alias none`
- Default value: `worklist`

This option tells SPARK which propagation algorithm to use.

`iter` is a simple, iterative algorithm, which propagates everything until the graph does not change.

`worklist` is a worklist-based algorithm that tries to do as little work as possible. This is currently the fastest algorithm.

`alias` is an alias-edge based algorithm. This algorithm tends to require the smallest amount of memory for very large problems, because it does not represent explicitly points-to sets of fields of heap objects.

`none` means that propagation is not done; the pointer assignment graph is only built and simplified. This is useful if an external propagator is to be used later on the pointer assignment graph.

Option `setImpl`

- Allowed values: `hash bit hybrid array double`
- Default value: `double`

Selects an implementation of a points-to set that SPARK should use.

`hash` is an implementation based on Java's built-in hash-set.

`bit` is an implementation using a bit vector.

`hybrid` is an implementation that keeps an explicit list of up to 16 elements, and switches to using a bit-vector when the set gets larger than this.

`array` is an implementation that keeps the elements of the points-to set in an array that is always maintained in sorted order. Set membership is tested using binary search, and set union and intersection are computed using an algorithm based on the merge step from merge sort.

`double` is an implementation that itself uses a pair of sets for each points-to set. The first set in the pair stores new pointed-to objects that have not yet been propagated, while the second set stores old pointed-to objects that have been propagated and need not be reconsidered. This allows the propagation algorithms to be incremental, often speeding them up significantly.

Option `doubleSetOld`

- Allowed values: `hash bit hybrid array`
- Default value: `hybrid`

Selects an implementation for the new points-to sets in the double points-to set implementation.

This option has no effect unless `setImpl` is set to `double`.

Option `doubleSetNew`

- Allowed values: `hash bit hybrid array`
- Default value: `hybrid`

Selects an implementation for the old points-to sets in the double points-to set implementation.

This option has no effect unless `setImpl` is set to `double`.

A.2.5 Output Options

Option `dumpHTML`

- Allowed values: `true false`
- Default value: `false`

When this option is set to `true`, a browseable HTML representation of the pointer assignment graph is output after the analysis completes. Note that this representation is typically very large.

Option `trimInvokeGraph`

- Allowed values: `true false`
- Default value: `false`

When this option is set to `true`, the results of the points-to analysis are used to make the invoke graph more precise after the analysis completes.

Bibliography

[AFFS98] Alexander Aiken, Manuel Fähndrich, Jeffrey S. Foster, and Zhendong 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.

[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).

[Ashe] Ashes Suite Collection.
URL: <`http://www.sable.mcgill.ca/software/`>.

[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.

[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.

[BKMS98] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of* *the ACM SIGPLAN '98 Conference on Programming Language Design and Inplementation*, pages 258–268. 1998.

[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.
URL: <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 Inplementation*. 2003.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341. 1996.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245. 1993.

[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.

[Cla97] Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, November 1997.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Mass., 1990.

[Cou86] Deborah S. Coutant. Retargetable high-level alias analysis. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 110–118. 1986.

[CR82] Anita L. Chow and Andres Rudmik. The design of a data flow analyzer. In *Proceedings of the SIGPLAN '82 Symposium on Compiler construction*, pages 106–113, 1982.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Inplementation*, pages 35–46. 2000.

[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, 7–11 August 1995.

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Inplementation*, pages 106–117. 1998.

[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 Inplementation*, pages 85–96. 1998.

[GH98] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 121–133. 1998.

[Hei99] Nevin Heintze. Analysis of large code bases: the compile-link-analyze model, 1999.
URL: <`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 Inplementation*, 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.

[HP00] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[HT01a] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Inplementation*, pages 24–34. 2001.

[HT01b] 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 Inplementation*, pages 254–263. 2001.

[Jedi] jEdit: Open Source programmer's text editor.
URL: <`http://www.jedit.org/`>.

[KKO02] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings* *of the 17th ACM Conference on Object-oriented programming, systems, languages, and applications*, pages 130–141. 2002.

[Lan92] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[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.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Inplementation*, pages 235–248. 1992.

[MRR02a] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Constructing precise object relation diagrams. In *IEEE International Conference on Software Maintenance (ICSM'02)*. October 2002.

[MRR02b] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'02)*. July 2002.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[PQVR+01] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A Framework for Optimizing Java Using Attributes. In *Compiler Construction, 10th International Conference (CC 2001)*, volume 2027 of *Lecture Notes in Computer Science*, pages 334–554, 2001.

[RC00] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Inplementation*, pages 47–56. 2000.

[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.

[Ruf95] Erik Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the Conference on Programming Language Design and Inplementation*, pages 13–22. 1995.

[Ruf00] Erik Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Inplementation*, pages 208–218. 2000.

[SFA00] Zhendong Su, Manuel Fähndrich, and Alexander Aiken. Projection merging: reducing redundancies in inclusion constraint graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–95, 2000.

[SH97a] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the Fourth International Symposium on Static Analysis (SAS'97)*, volume 1302 of *Lecture Notes in Computer Science*, pages 16–34, 1997.

[SH97b] 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.

[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.

[Soot] Soot: a Java Optimization Framework.
URL: <`http://www.sable.mcgill.ca/soot/`>.

[Spec] SPEC JVM98 Benchmarks.
URL: <`http://www.spec.org/osg/jvm98/`>.

[Ste96a] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Compiler Construction, 6th International Conference*, volume 1060 of *Lecture Notes in Computer Science*, pages 136–150, 24–26 April 1996.

[Ste96b] 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.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[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.

[VHK97] Jan Vitek, R. Nigel Horspool, and Andreas Krall. Efficient type inclusion tests. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 142–157. 1997.

[VR01] Frédéric Vivien and Martin Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Inplementation*, 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.

[Wei80] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–94. 1980.

[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 Inplementation*, pages 1–12. 1995.

[WL02] John Whaley and Monica 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.

[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.