McGill University
School of Computer Science
Sable Research Group

# Scaling Java Points-To Analysis using Spark

Sable Technical Report No. 2002-9

Ondřej Lhoták and Laurie Hendren

October 24, 2002

# Scaling Java Points-To Analysis using Spark

Ondřej Lhoták and Laurie Hendren

23 October 2002

**Abstract**

Most points-to analysis research has been done on different systems by different groups, making it difficult to compare results, and to understand interactions between individual factors each group studied. Furthermore, points-to analysis for Java has been studied much less thoroughly than for C, and the tradeoffs appear very different.

We introduce Spark, a flexible framework for experimenting with points-to analyses for Java. 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 solving algorithms. Spark is composed of building blocks on which new analyses can be based.

We demonstrate Spark in a substantial study of factors affecting precision and efficiency of subset-based points-to analyses, including interactions between these factors. Our results show that Spark is not only flexible and modular, but also offers superior time/space performance when compared to other points-to analysis implementations.

## 1 Introduction

Many compiler analyses and optimizations, as well as program understanding and verification tools, require information about which objects each pointer in a program may point to at run-time. The problem of approximating these points-to sets has been the subject of much research; however, many questions remain unanswered [16].

As with many compiler analyses, a precision vs. time trade-off exists for points-to analysis. For analyzing programs written in C, many points between the extremes of high-precision, slow and low-precision, fast have been explored [6, 8, 11, 15, 19, 21–23]. These analyses have been implemented as parts of distinct systems, so it is difficult to compare and combine their unique features. The design tradeoffs for doing points-to analysis for Java appear to be different than for C, and recently, several different approaches to points-to analysis for Java have been suggested [17, 20, 29]. However, once again, it is hard to compare the results since each group has implemented their analysis in a different system, and has made very different assumptions about how to handle the large Java class libraries and Java native methods.

To address these issues, we have developed the Soot Pointer Analysis Research Kit (Spark), a flexible framework for experimenting with points-to analyses for Java. Spark is very modular: the pointer assignment graph that it produces and simplifies can be used as input to other solvers, including those being developed by other researchers. We hope that this will make it easier for researchers to compare results. In addition, the correctness of new analyses can be verified by comparing their results to those computed by the basic analyses provided in Spark.

In order to demonstrate the usefulness of the framework, we have also performed a substantial empirical study of a variety of subset-based points-to analyses using the framework. We studied a wide variety of factors that affect both precision and time/space costs. Our results show that Spark is not only flexible and modular, but also offers very good time/space performance when compared to other points-to analysis implementations.

Specific new contributions of this paper are:

- The SPARK framework itself, available as part of Soot 1.2.4 [3] and later releases, under the LGPL for the use of all researchers.
- A study of a variety of representations for points-to sets and the presentation of a variety of solving strategies, including an incremental *worklist-based*, field-sensitive algorithm which appears to scale well to larger benchmarks.
- An empirical evaluation of many factors affecting the precision, speed, and memory requirements of subset-based points-to analysis algorithms. We focus on methods to improve the speed of the analysis without significant loss of precision.
- Recommendations to allow analyses to scale to programs on the order of a million lines of code. Even trivial Java programs are becoming this large as the standard class library grows.

The structure of this paper is as follows. In Section 2 we examine some of the challenges and factors to consider when designing an effective points-to analysis for Java. In Section 3 we introduce the SPARK framework and discuss the important components. Section 4 shows SPARK in action via a large empirical study of a variety of subset-based pointer analyses. In Section 5 we discuss related work and in Section 6 we provide our conclusions and discuss future work.

## 2 Points-to Analysis for Java

Although some of the techniques developed for C have been adapted to Java, there are significant differences between the two languages that affect points-to analysis. In C, points-to analysis can be viewed as two separate problems: analysis of stack-directed pointers, and analysis of heap-directed pointers. Most C programs have many more occurrences of the address-of (`&`) operator, which creates stack-directed pointers, than dynamic allocation sites, which create heap-directed pointers. It is therefore important for C points-to analyses to deal well with stack-directed pointers. Java, on the other hand, allows no stack-directed pointers whatsoever, and Java programs usually have many more dynamic allocation sites than C programs of similar size. Java analyses therefore have to handle heap-directed pointers well.

Another important difference is the strong type checking in Java, which limits the sets of objects that a pointer could point to, and can therefore be used to improve analysis speed and precision. Diwan et. al. have shown the benefits of type-based alias analysis for Modula-3 [10]. Our study shows that using types in Java is very useful for improving efficiency, and also results in a small improvement in precision.

The object-oriented nature of Java also introduces new complexities in dealing with any whole program analysis. In order to build a call graph, some approximation of the targets of virtual method calls must be used. There are two basic approaches. For the first approach, a pointer analysis may rely on an approximation of the call graph as built by another analysis. In the second approach, the call graph may be constructed, on-the-fly, as the pointer analysis proceeds. In our empirical study, Section 4, we compare the two approaches.

Related to the problem of finding a call graph is finding the set of methods that must be analyzed. In sequential C programs there is one entry point, `main`, and a whole program analysis can start at this entry point and then incrementally (either ahead-of-time or during analysis) add all called methods. In Java the situation is much more complicated as there are many potential entry points including static initializers, finalizers, thread start methods, and methods called using reflection. Further complicating matters are native methods which may impact points-to analysis, but for which we do not have the code to analyze. Our SPARK framework addresses these points.

Another very important point is the large size of the Java libraries. Even small application programs may touch, or appear to touch, a large part of the Java library. This means that a whole program analysis must be able to handle large problem sizes. Existing points-to analyses for Java have been successfully tested with the 1.1.8 version of the Java standard libraries [17, 20], consisting of 148 thousand lines of code (KLOC). However, current versions of the standard library are over three times larger (eg. 1.3.1_01 is 574 KLOC), dwarfing most application programs that use them, so it is not clear that existing analyses would scale to

such large programs. Our framework has been designed to provide the tools to develop efficient and scalable analyses which can effectively handle large benchmarks using the large libraries.

# 3 SPARK framework

## 3.1 Overview

The Soot Pointer Analysis Research Kit (SPARK) is a flexible framework for experimenting with points-to analyses for Java. Although SPARK is very competitive in efficiency with other points-to analysis systems, the main design goal was not raw speed, but rather the flexibility to make implementing a wide variety of analyses as easy as possible, to facilitate comparison of existing analyses and development of new ones.

SPARK supports both subset-based [6] and unification-based [23] analyses, as well as variations that lie between these two extremes. In this paper, we focus on the more precise, subset-based analyses.

SPARK is implemented as part of the Soot bytecode analysis, optimization, and annotation framework [27]. Soot accepts Java bytecode as input, converts it to one of several intermediate representations, applies analyses and transformations, and converts the results back to bytecode. SPARK uses as its input the Jimple intermediate representation, a three-address representation in which local (stack) variables have been split according to DU-UD webs, and declared types have been inferred for them. The results of SPARK can be used by other analyses and transformations in Soot. Soot also provides an annotation framework that can be used to encode the results in classfile annotations for use by other tools or runtime systems [18].

The execution of SPARK can be divided into three stages: pointer assignment graph construction, pointer assignment graph simplification, and points-to set propagation. These stages are described in the following subsections.

## 3.2 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. 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 consists of three types of nodes. Allocation site nodes (denoted $new_i$) represent allocation sites in the source program, and are used to model heap locations. Simple variable nodes (denoted $l_i$) represent local variables, method parameters and return values, and static fields. Field dereference nodes (denoted $l_i.f$) represent field access expressions in the source program; each is parametrized by a variable node representing the variable being dereferenced by the field access. The nodes in the pointer assignment graph are connected with four types of edges reflecting the pointer flow, corresponding to the four types of constraints imposed by the pointer-related instructions in the source program (Table 1).

Table 1: The four types pointer assignment graph edges.

| | Allocation | Assignment | Field store | Field load |
|---|---|---|---|---|
| Instruction | $l := new\ C$ | $l_i := l_j$ | $l_i.f := l_j$ | $l_i := l_j.f$ |
| Edge | $new\ C \rightarrow l$ | $l_j \rightarrow l_i$ | $l_j \rightarrow l_i.f$ | $l_j.f \rightarrow l_i$ |
| Rules | $\dfrac{new_i \rightarrow l}{new_i \in pt(l)}$ | $\dfrac{l_j \rightarrow l_i \quad new_i \in pt(l_j)}{new_i \in pt(l_i)}$ | $\dfrac{l_j \rightarrow l_i.f \quad new_i \in pt(l_j) \quad new_j \in pt(l_i)}{new_i \in pt(new_j.f)}$ | $\dfrac{l_j.f \rightarrow l_i \quad new_i \in pt(l_j) \quad new_j \in pt(new_i.f)}{new_j \in pt(l_i)}$ |

Later, during the propagation of points-to sets, a fourth type of node (denoted $new_i.f$) 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.

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. As an example, the builder may make assignments directed for a subset-based analysis, or bi-directional for a merge-based analysis. Another example is the representation of field dereference expressions in the graph, as discussed next.

#### 3.2.1 Field Dereference Expressions:

A field expression $p.f$ refers to the field $f$ of the object pointed to by $p$. There are three standard ways of dealing with fields, as summarized in Table 2.

Table 2: Possible representations of field expressions.

| | |
|---|---|
| p.f | field-sensitive |
| ?.f | field-based |
| p.? | field-independent |

A *field-sensitive* interpretation, which is the most precise, considers $p.f$ to represent only the field $f$ of only objects in the points-to set of $p$.

A less precise, *field-based* interpretation approximates each field $f$ of all objects using a single set, ignoring the $p$. This approach is implemented in SPARK by representing the field expression with a simple variable node $f$, rather than by a field dereference node parametrized by $p$. The key advantage of this is that points-to sets can be propagated along a pointer assignment graph of only simple variable nodes *in one single iteration*, by first merging strongly-connected components of nodes, then propagating in topological order.

Many C points-to analyses use a *field-independent* interpretation, which ignores the $f$, and approximates all the fields of objects in the points-to set of $p$ as a single location. In Java, the field information is readily available, and different fields are guaranteed not to be aliased, so a field-independent interpretation makes little sense; we do not discuss it further in this paper.

SPARK supports field-sensitive and field-based analyses, and field-independent analyses would be trivial to implement.

### 3.3 Call Graph Construction

An interprocedural points-to analysis requires an approximation of the call graph. This can be constructed in advance using a technique such as CHA [9], RTA [7] or VTA [25], or it can be constructed on-the-fly as the points-to sets of call site receivers are computed. The latter approach gives somewhat higher precision, but requires more iteration as edges are added to the pointer assignment graph.

SPARK supports all of these variations, but in this paper, our empirical study focuses on CHA and on-the-fly call graph construction. In all cases, SPARK uses the CHA-based call graph builder included in Soot to determine which methods are reachable for the purposes of *building* the pointer assignment graph. However, on-the-fly call graph construction can be achieved during *solving* time by excluding interprocedural edges from the initial graph, and then adding only the reachable edges as the points-to sets are propagated.

In theory, determining which methods are possibly reachable at run-time is simple: start with a root set containing the main method, and transitively add all methods which are called from methods in the set. Java is not this simple, however; execution can also start at static initializers, finalizers, thread start methods, and dynamic call sites using reflection. Soot considers all these factors in determining which methods are reachable. For the many call sites using reflection inside the standard class library, we have compiled, by hand, a list of their possible targets, and they are automatically added to the root set.

In addition, native methods may affect the flow of pointers in a Java program. SPARK therefore includes a native method simulation framework. The effects of each native method are described in the framework using abstract Java code, and SPARK then creates the corresponding pointer flow edges. The native method simulation framework was designed to be independent of SPARK, so the simulations of native methods should be usable by other analyses.

## 3.4 Points-to Assignment Graph Simplification

Before points-to sets are propagated, the pointer assignment graph can be simplified by merging nodes that are known to have the same points-to set. Specifically, all the nodes in a strongly-connected component (cycle) will have equal points-to sets, so they can be merged to a single node. A version of the off-line variable substitution algorithm given in [19] is also used to merge equivalence sets of nodes that have a single common predecessor.[1]

SPARK uses a fast union-find algorithm [26] to merge nodes in time almost linear in the number of nodes. This is the same algorithm used for unification-based [23] analyses. Therefore, by making all edges bidirectional and merging nodes forming strongly-connected components, we can implement a unification-based analysis in SPARK. In fact, we can easily implement a hybrid analysis which is partly unification-based and partly subset-based by making only *some* of the edges bidirectional. One instance of a similarly hybrid analysis is described in [8]. Even when performing a fully subset-based analysis, we can use the same unification code to simplify the pointer assignment graph.

One of the design objectives of SPARK was to allow nodes to be merged during the analysis, rather than only before the analysis starts.[2] This requires not only merging the nodes, but also updating the sets of pointer assignment edges between nodes that are merged, as well as merging the corresponding field dereference nodes. SPARK solves this problem lazily: whenever it iterates through the adjacency list of a node, each of the adjacent nodes is checked, and replaced with a merged node if appropriate. Delaying the update of the adjacency lists until they are needed makes it possible to amortize the cost of updating the adjacency list over many merges.

## 3.5 Set Implementations

Choosing an appropriate set representation for the points-to sets is a key part of designing an effective analysis. The following implementations are currently included as part of SPARK; others should be easy to add.

### 3.5.1 Hash Set:

is a wrapper for the `HashSet` implementation from the standard class library. It is provided as a baseline against which the other set implementations can be compared.

### 3.5.2 Sorted Array Set:

implements a set using an array which is always kept in sorted order. This makes it possible to compute the union of two sets in linear time, like in a merge sort.

### 3.5.3 Bit Set:

implements a set as a bit vector. This makes set operations very fast regardless of how large the sets get (as long as the size of the universal set stays constant). The drawback is that the many sparse sets use a large

[1] If types are being used, then only nodes with compatible types can be merged, the interaction of types and graph simplication is examined in Section 4.

[2] Merging nodes during the analysis would be useful for on-the-fly cycle simplification. However, none of the solving algorithms currently implemented use this feature.

amount of memory.

#### 3.5.4 Hybrid Set:

represents small sets (up to 16 elements) explicitly using pointers to the elements themselves, but switches to a bit vector representation when the sets grow larger, thus allowing both small and large sets to be represented efficiently.

## 3.6 Points-to Set Propagation

After the pointer assignment graph has been built and simplified, the final step is propagating the points-to sets along its edges according to the rules shown in Table 1. SPARK provides several different algorithms to implement these rules.

#### 3.6.1 Iterative Algorithm:

SPARK includes a naive, baseline, iterative algorithm which can be used to check the correctness of the results of the more complicated algorithms. This algorithm is presented in Algorithm 1. Note that for efficiency, all the propagation algorithms in SPARK consider variable nodes in topological order (or pseudo-topological order, if cycles have not been simplified).

**Algorithm 1** Iterative Propagation

```
1: initialize sets according to allocation edges
2: repeat
3:    propagate sets along each assignment edge p → q
4:    for each load edge p.f → q do
5:       for each a ∈ pt(p) do
6:          propagate sets pt(a.f) → pt(q)
7:    for each store edge p → q.f do
8:       for each a ∈ pt(q) do
9:          propagate sets pt(p) → pt(a.f)
10: until no changes
```

#### 3.6.2 Worklist Algorithm:

For some of our benchmarks, the iterative algorithm performs over 60 iterations. After the first few iterations, the points-to sets grow very little, yet each iteration is as expensive as the first few.

A better, but more complex solver based on worklists is also provided as part of SPARK as outlined in Algorithm 2[3]. This solver maintains a worklist of variable nodes whose points-to sets need to be propagated to their successors, so that only those nodes are considered for propagation. As before, variable nodes are removed from the worklist in topological (or pseudo-topological) order.

In the presence of field-sensitivity, however, the worklist algorithm is not so trivial. 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 nodes of the form $p \rightarrow q$, but also along loads and stores involving $p$ (those of the form $p \rightarrow q.f$, $q \rightarrow p.f$, and $p.f \rightarrow q$), since they are likely to require propagation. However, this is not sufficient to obtain the complete solution. For example, suppose variable $p$ has already been processed with the allocation site $a_1$ in its points-to set, and $a_1$ is just added to the points-to set of $q$. This adds $q$ to the worklist, but we also need to re-process edges of the form $p.f \rightarrow r$. For this reason, the

[3]For clarity, the algorithms are presented here without support for on-the-fly call graph construction. Both variations are implemented in SPARK

**Algorithm 2** Worklist Propagation

```
1: for each allocation edge o1 → p do
2:    pt(p) ∪= {o1}
3:    add p to worklist
4: repeat
5:    repeat
6:       remove first node p from worklist
7:       propagate sets along each assignment edge p → q,
         adding q to worklist whenever pt(q) changes
8:       for each store edge q → r.f where p = q or p = r
         do
9:          for each a ∈ pt(r) do
10:            propagate sets pt(q) → pt(a.f)
11:      for each load edge p.f → q do
12:         for each a ∈ pt(p) do
13:            propagate sets pt(a.f) → q
14:            add q to worklist if pt(q) changed
15:   until worklist is empty
16:   for each store edge q → r.f do
17:      for each a ∈ pt(r) do
18:         propagate sets pt(q) → pt(a.f)
19:   for each load edge p.f → q do
20:      for each a ∈ pt(p) do
21:         propagate sets pt(a.f) → q
22:         add q to worklist if pt(q) changed
23: until worklist is empty
```

algorithm must still include an outer iteration over *all* the load and store edges. To summarize, lines 16 to 22 in the outer loop are necessary for correctness; lines 8 to 14 could be removed, but including them greatly reduces the number of iterations of the outer loop and therefore the analysis time.

#### 3.6.3 Incremental Sets:

In certain implementations of sets (hash set and sorted array set), each set union operation takes time proportional to the size of 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 must be found in the set in every subsequent iteration.

Thus, as an optional improvement, SPARK includes versions of the solvers that use incremental sets. Each 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.

## 4 Using SPARK for Subset-based Points-to Analysis

In order to demonstrate that SPARK provides a general and effective means to express different points-to analyses, we have done an extensive empirical study of a variety of subset-based points-to analyses. By expressing many different variations within the same framework we can measure both precision and cost of the analyses. In Section 4.1 we introduce our benchmarks, in Section 4.2 we present measurements of the precision of the analyses, in Section 4.3 we show the factors that affect performance and in Section 4.4 we summarize by showing our results on the three configurations that appear to be the most useful design points.

### 4.1 Benchmarks

We tested SPARK on benchmarks from the SPECjvm [4] suite, along with `sablecc` and `soot` from the Ashes [1] suite, and `jedit` [2], 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 [17, 20, 29]. The complete list of benchmarks appears in the summary in Table 6 at the end of this section, along with some characteristics of the benchmarks, and measurements of the effectiveness of SPARK on them. 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, we also tested the `javac` benchmark with the Sun JDK 1.1.8 standard class library for comparison with other studies.

We chose four representative benchmarks for which to present the detailed results of our experiments on individual factors affecting precision and efficiency of points-to analysis. We chose `compress` (Lempel-Ziv compression) 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. We observed similar trends on the other benchmarks.

### 4.2 Factors Affecting Precision

We now discuss 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 we use a CHA-based call graph or build the call graph on the fly, and (3) whether the analysis is field-sensitive or field-based.

Table 3 gives the results. For each benchmark we experiment with five different points-to analyses, where each analysis is named by a triple of the form xx-yyy-zz which specifies the setting for each of the three factors

Table 3: Analysis precision.

| | | Dereference Sites (% of total) | | | | | | | Call Sites (% of total) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3-10 | 11-100 | 101-1000 | 1001+ | 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 | | | | |
| | 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 | | | | |
| | 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 | | | | |
| | 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 | | | | |
| | 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 |

(a complete explanation of each factor is given in the subsections below). For each benchmark/points-to analysis combination, we give a summary of the precision for dereference sites and call sites.

For dereference sites we consider all occurrences of references of the form `p.f` and we give the percent of dereference sites 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 indicated that it was in a reachable method, but no allocation ever flows to that statement).

For call sites we consider all `invokevirtual` and `invokeinterface` calls and report the percentage of such call sites with 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.

Note that since the required level of precision required is highly dependent on the application of the points-to results, this table is not intended to be an absolute measure of precision; rather, we present it only to give some idea of the relative precision of different analysis variations, and to give basic insight into the effect that different levels of precision have on the analysis.

#### 4.2.1 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 [20, 25] ignore declared types as the analysis proceeds; however, objects of incompatible type are removed after the analysis completes.

The first three lines of each benchmark in Table 3 show the effect of 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`). 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. We therefore recommend enforcing declared types as the analysis proceeds, which eliminates almost all of these large sets. Further, based on this observation, we focus on analyses that respect declared types for the remainder of this paper.

Enforcing declared types during the analysis requires fast subtype testing. For this purpose, we precompute and store the subtype relationships in a two-dimensional bit array. Although this requires space quadratic in the number of types, for benchmarks of the size that we used, the number of types was around 3000 (see Table 6), so this table takes slightly over 1MB of memory, which is small compared to all the information that a whole-program analyzer keeps about a 600KLOC program. In addition, other parts of a whole-program analyzer can take advantage of fast subtype testing. More complicated fast space-efficient subtype testing mechanisms are evaluated in [28].

#### 4.2.2 Call graph construction:

As we have already mentioned, the call graph used for an inter-procedural points-to analysis can be constructed ahead of time using, for example, CHA [9], or on-the-fly as the analysis proceeds [20], for greater precision. We abbreviate these variations as `cha` and `otf`, respectively. As the third and fourth lines for each benchmark in Table 3 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.

#### 4.2.3 Field Dereference Expressions:

We study the handling of field dereference expressions in a field-based (abbreviated fb) and field-sensitive (fs) manner.

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 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. As we will see later, in Table 5, with the appropriate solver, the field-sensitive analysis can be made to be quite competitive to the field-based analysis.

### 4.3 Factors Affecting Performance

#### 4.3.1 Set Implementation:

We evaluated the analyses with the four different implementations of points-to sets described in Section 3. Table 4 shows the efficiency of the implementations using two of the propagation algorithms: the naive, iterative algorithm, and the incremental worklist algorithm. For both algorithms, we used a CHA call graph, and simplified the pointer assignment graph by collapsing cycles, as well as variables with common predecessors as described in [19]. 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.

Table 4: Set Implementation (time in seconds, space in MB).

| | | 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 |
| | Incremental Worklist | 31 | 219 | 319 | 62 | 57 | 14 | 155 | 9 | 53 |
| javac | Iterative | 34 | 3791 | 361 | 1114 | 139 | 50 | 88 | 33 | 41 |
| | Incremental Worklist | 34 | 252 | 369 | 61 | 68 | 19 | 181 | 13 | 65 |
| sablecc | Iterative | 36 | 4158 | 334 | 1194 | 132 | 50 | 93 | 32 | 42 |
| | Incremental Worklist | 36 | 244 | 342 | 54 | 62 | 17 | 193 | 11 | 66 |
| jedit | Iterative | 42 | 6502 | 583 | 2233 | 229 | 91 | 168 | 59 | 77 |
| | Incremental Worklist | 42 | 488 | 597 | 135 | 114 | 38 | 349 | 24 | 128 |

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 iterative 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. We have found the hybrid set implementation to be consistently the most efficient over a wide variety of settings of the other parameters, and therefore recommend that it always be used. We encourage experimentation with the threshold size at which a hybrid set begins to use a bit-vector representation.

#### 4.3.2 Points-To Set Propagation Algorithms

Table 5 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.[4] 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.

Table 5: Propagation Algorithms (time in seconds, space in MB).

| | | Graph | Iterative | | Worklist | | Incr. Worklist | |
|---|---|---|---|---|---|---|---|---|
| | | space | time | space | time | space | time | space |
| compress | nt-otf-fs | 32 | 1628 | 357 | 992 | 365 | 399 | 605 |
| | ot-otf-fs | 37 | 133 | 52 | 58 | 51 | **52** | **69** |
| | ot-cha-fs | 36 | 49 | 68 | 15 | 63 | **13** | **91** |
| | ot-otf-fb | 35 | 158 | 54 | 86 | 52 | **66** | **66** |
| | ot-cha-fb | 34 | 17 | 62 | **10** | **56** | 13 | 76 |
| javac | nt-otf-fs | 34 | 2316 | 502 | 1570 | 512 | 715 | 856 |
| | ot-otf-fs | 40 | 201 | 69 | 103 | 66 | **90** | **90** |
| | ot-cha-fs | 39 | 64 | 83 | 22 | 77 | **18** | **109** |
| | ot-otf-fb | 37 | 218 | 70 | 123 | 66 | **102** | **84** |
| | ot-cha-fb | 37 | 22 | 75 | **11** | **67** | 15 | 90 |
| sablecc | nt-otf-fs | 35 | 2190 | 462 | 1382 | 472 | 635 | 772 |
| | ot-otf-fs | 41 | 274 | 72 | 104 | 70 | **95** | **94** |
| | ot-cha-fs | 41 | 66 | 88 | 20 | 83 | **18** | **117** |
| | ot-otf-fb | 38 | 255 | 74 | 138 | 72 | **114** | **90** |
| | ot-cha-fb | 38 | 52 | 81 | **14** | **74** | 18 | 97 |
| jedit | nt-otf-fs | oom | oom | oom | oom | oom | oom | oom |
| | ot-otf-fs | 49 | 313 | 121 | 142 | 117 | **101** | **169** |
| | ot-cha-fs | 48 | 107 | 141 | 59 | 131 | **38** | **196** |
| | ot-otf-fb | 47 | 298 | 104 | 178 | 99 | **111** | **126** |
| | ot-cha-fb | 45 | 28 | 109 | **21** | **98** | 27 | 128 |

The nt-otf-fs line shows how much ignoring declared types hurts efficiency (the "oom" for `jedit` signifies that the analysis exceeded the 1700MB of memory allotted); we recommend that declared types be respected. Results from the recommended algorithms are in bold.

The iterative algorithm is consistently slowest, and is given as a baseline only. The worklist algorithm is usually about twice as fast as the iterative algorithm. For the CHA 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 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.

[4]The time and space reported for the hybrid set implementation in Table 4 are different than in Table 5 because the former were measured with off-line pointer assignment graph simplification, and the latter without.

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

Note also that while the field-based analysis is faster than field-sensitive 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.

#### 4.3.3 Graph Simplification:

Rountev and Chandra [19] showed that simplifying the pointer assignment graph by merging nodes known to have equal points-to sets speeds up the analysis. Our experience 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 subgraph with a unique predecessor can only be merged if all its nodes have declared types that are supertypes of the predecessor. On our benchmarks, 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 subgraphs with a unique predecessor, compared to between 55% and 58% when declared types were respected. Thus, the effect of respecting declared types on simplification is minor.

On the other hand, when constructing the call graph on-the-fly, no inter-procedural 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 subgraphs with a unique predecessor 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.

### 4.4 Overall Results

Based on our experiments, we have selected three analyses that we recommend as good compromises between precision and speed, with reasonable space requirements:

1. ot-otf-fs is suitable for applications requiring the highest precision. For this analysis, the incremental worklist algorithm works best.
2. ot-cha-fs 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 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.

Each of the three analyses should be implemented using the hybrid sets.

Table 6 shows the results of these three analyses on our 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 next two columns give the number of methods determined to be reachable, and the number of Jimple[5] 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 been reported, to our knowledge. The fourth column gives the number of distinct types encountered by the subtype tester. 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;

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

Table 6: Overall Results (time in seconds, space in MB, precision in precent).

| Benchmark | methods (CHA) | stmts (CHA) | types | ot-otf-fs time | ot-otf-fs space | ot-otf-fs prec. | ot-cha-fs time | ot-cha-fs space | ot-cha-fs prec. | ot-cha-fb time | ot-cha-fb space | ot-cha-fb prec. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| compress | 15183 | 278902 | 2770 | 52 | 106 | 69.1 | 13 | 127 | 60.1 | 10 | 90 | 57.6 |
| db | 15185 | 278954 | 2763 | 52 | 107 | 68.9 | 14 | 128 | 59.9 | 11 | 90 | 57.4 |
| jack | 15441 | 288142 | 2816 | 54 | 112 | 68.7 | 14 | 132 | 60.1 | 11 | 94 | 57.6 |
| javac (1.1.8) | 4602 | 86454 | 874 | 8 | 27 | 63.6 | 3 | 24 | 57.4 | 1 | 16 | 55.1 |
| javac | 16307 | 301801 | 2940 | 89 | 131 | 66.3 | 18 | 148 | 58.4 | 11 | 104 | 56.2 |
| jess | 15794 | 288831 | 2917 | 57 | 115 | 68.1 | 15 | 136 | 59.2 | 10 | 97 | 56.8 |
| mpegaudio | 15385 | 283482 | 2782 | 56 | 112 | 68.6 | 16 | 134 | 59.7 | 11 | 93 | 57.4 |
| raytrace | 15312 | 281587 | 2789 | 53 | 107 | 68.5 | 13 | 129 | 59.6 | 11 | 91 | 57.1 |
| sablecc | 16977 | 300504 | 3070 | 95 | 136 | 70.5 | 18 | 158 | 62.5 | 14 | 112 | 60.3 |
| soot | 17498 | 310935 | 3435 | 88 | 143 | 68.3 | 19 | 162 | 60.4 | 18 | 116 | 58.4 |
| jedit | 19621 | 367317 | 3395 | 100 | 218 | 69.1 | 38 | 244 | 62.3 | 21 | 143 | 61.1 |

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 3. Due to space constraints, we omit the measurements of memory requirements of the analyses; however, `jedit` required the most memory of all the benchmarks, and detailed memory requirements for it are listed in Table 5.

# 5 Related Work

The most closely related work are the various adaptations of points-to analyses for C to Java.

Rountev, Milanova and Ryder [20] based their field-sensitive analysis for Java on Soot [27] and the BANE [5] constraint solving toolkit, on which further points-to analysis work has been done [12, 24]. Their analysis was field-sensitive, constructed the call graph on-the-fly, and ignored declared types until after the analysis completed. They reported empirical results on many benchmarks using the JDK 1.1.8 standard class library. Since they do not handle declared types during the analysis, their implementation suffers from having to represent large points-to sets, and is unlikely to scale well to large class libraries. They do not report results for the JDK 1.3.1 library, but their results for `javac (1.1.8)` show 350 seconds and 125.5 MB of memory (360 MHz Sun Ultra-60 machine with 512 MB of memory, BANE solver written in ML), compared to 8 seconds and 27 MB of memory (1.67 GHz AMD Athlon with 2GB memory, solver written in Java) for our best result for this same benchmark (solving the ot-otf-fs analysis using the incremental worklist solver and the hybrid set representation). The precision of our results should be very slightly better, since the Rountev et. al. method is equivalent to our at-otf-fs analysis, which we showed to be slightly less precise that the ot-otf-fs analysis.

Whaley and Lam's [29] approach is interesting in that it adapts the demand-driven algorithm of Heintze and Tardieu [14, 15] (see below) to Java. The intermediate representation on which their analysis operates is different from Jimple (on which our and Rountev, Milanova and Ryder's analyses are based) in that it does not split stack locations based on DU-UD webs; instead, it uses intra-method flow-sensitivity to achieve a similar effect. In contrast with other work that used a conservative (safe) approximation of reachable methods which to analyze, Whaley and Lam's experiments used optimistic assumptions (not safe) about which methods need to be analyzed. In particular, the results presented in their paper [29] are for a variation of the analysis that does not analyze class initializers and assumes that all native methods have no effect on the points-to analysis. Their optimistic assumptions about which methods are reachable lead to reachable method counts almost an order of magnitude lower than reported in other related work, such as the present

paper, and [20, 25]; in fact, they analyze significantly fewer methods than can be observed to be executed at run-time in a standard run of the benchmarks. As a result of the artificially small number of methods that they analyze, they get fast execution times. Even so, when looking at the `jedit` benchmark, the only benchmark for which they analyze at least half of the number of methods analyzed by SPARK, their analysis runs in 614 seconds and 472 MB of memory (2GHz Pentium 4 with 2GB of memory, solver written in Java), compared to 100 seconds and 218 MB for the most precise analysis in SPARK (1.67 GHz AMD Athlon with 2GB memory, solver written in Java).

Our comparison with these two other previous works for points-to analysis for Java illustrates two important things. First, it would be nice if we could compare the analyses head to head, on the same system, with the same assumptions about what code needs to be analyzed. Second, it appears thatSPARK allows one to develop efficient analyses that compare very favourably with previous work.

Liang, Pennings and Harrold [17] tested several variations of Java points-to analyses, including subset-based and unification based variations, field-based and field-sensitive variations, and constructing the call graph using CHA [9] and RTA [7]. Instead of analyzing benchmarks with the standard class library, they hand-coded a model of the most commonly used JDK 1.1.8 standard classes. Thus, we cannot make direct comparisons, since our results include all the library code.

Rountev and Chandra [19] described off-line variable substitution as a method of reducing pointer assignment graphs before a points-to analysis is performed on them. In SPARK, we have adapted their approach to Java. Our findings of its effectiveness generally agree with their results.

Heintze and Tardieu [14, 15] reported very fast analysis times using their analysis for C. The main factor making it fast was a demand-driven algorithm that also collapsed cycles in the constraint graph on-the-fly. Such a demand-driven algorithm is particularly useful when the points-to sets of only a subset of pointer variables are required; we plan to implement it in a future version of SPARK for such applications. In addition, in an unpublished report [13], Heintze discusses an implementation of sets using bit-vectors which are shared, so that copies of an identical set are only stored once. We are also considering implementing this set representation in SPARK.

Since points-to analysis in general has been an active area of research for many years, we can only list the work most closely related to ours. A more complete survey appears in [16].

# 6 Conclusions and Future Work

We have presented SPARK, a flexible framework for experimenting with points-to analysis for Java. Our empirical results have shown that SPARK is not only flexible, but also competitive with points-to analyses that have been implemented in other frameworks. Using SPARK, we studied various factors affecting the precision and efficiency of points-to analysis. Our study led us to recommend three specific analyses, and we showed that they compare favourably to other analyses that have been described in the literature.

We plan several improvements to SPARK. First, we would like to create an on-the-fly pointer assignment graph builder, so that the entire pointer assignment graph need not be built for an on-the-fly call graph analysis. Second, we would like to add Heintze and Tardieu's demand-driven propagation algorithm to SPARK.

We have several studies in mind that we would like to perform using SPARK. First, we are attempting to implement points-to analysis using Reduced Ordered Binary Decision Diagrams to store the large, often duplicated sets. Second, we plan to study the effects of various levels of context-sensitivity on Java points-to analysis. Third, we will experiment with various clients of the points-to analysis.

# 7 Acknowledgements

We are grateful to Feng Qian for work on the native method simulator, to Navindra Umanee for help producing the list of methods called using reflection by the standard class library, to Marc Berndl and John Jorgensen for helpful discussions, to Atanas Rountev, Ana Milanova, and Barbara Ryder for providing details

about how their points-to analysis determines the reachable call graph, and to John Whaley for answering our questions about the assumptions made by his analysis, and the settings he used in his experiments.

## References

[1] Ashes suite collection. `http://www.sable.mcgill.ca/software/`.

[2] jEdit: Open source programmer's text editor. `http://www.jedit.org/`.

[3] Soot: a Java optimization framework. `http://www.spec.org/osg/jvm98/`.

[4] SPEC JVM98 benchmarks. `http://www.spec.org/osg/jvm98/`.

[5] A. Aiken, M. Fähndrich, J. S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Types in Compilation, Second International Workshop, TIC '98*, volume 1473 of *LNCS*, pages 78–96, 1998.

[6] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, May 1994. (DIKU report 94/19).

[7] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 1996 OOPSLA*, pages 324–341, 1996.

[8] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of PLDI'00*, volume 35.5 of *ACM Sigplan Notices*, pages 35–46, June 2000.

[9] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *LNCS*, pages 77–101, Aug. 1995.

[10] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proceedings of PLDI'98*, pages 106–117, 1998.

[11] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of PLDI'94*, pages 242–256, 1994.

[12] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of PLDI'98*, pages 85–96, June 1998.

[13] N. Heintze. Analysis of large code bases: The compile-link-analyze model. `http://cm.bell-labs.com/cm/cs/who/nch/cla.ps`, 1999.

[14] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Proceedings of PLDI'01*, pages 24–34, 2001.

[15] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of PLDI'01*, volume 36.5 of *ACM SIGPLAN Notices*, pages 254–263, June 2001.

[16] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of PASTE'01*, pages 54–61, June 2001.

[17] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Proceedings of PASTE'01*, pages 73–79, 2001.

[18] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing java using attributes. In *Compiler Construction (CC 2001)*, volume 2027 of *LNCS*, pages 334–554, 2001.

[19] A. Rountev and S. Chandra. Off-line Variable Substitution for Scaling Points-to Analysis. In *Proceedings of PLDI'00*, pages 47 – 56, Jun 2000.

[20] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of the 2001 OOPSLA*, pages 43–55, 2001.

[21] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. *LNCS*, 2027:20–36, 2001.

[22] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of 24th POPL '97*, pages 1–14, Jan. 1997.

[23] B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of 23rd POPL'96*, pages 32–41, Jan. 1996.

[24] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: reducing redundancies in inclusion constraint graphs. In *Proceedings of the 27th POPL'00*, pages 81–95, 2000.

[25] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Proceedings of the 2000 OOPSLA*, pages 264–280, 2000.

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

[27] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction (CC 2000)*, volume 1781 of *LNCS*, pages 18–34, 2000.

[28] J. Vitek, N. Horspool, and A. Krall. Efficient type inclusion tests. In *Proceedings of the 1997 OOPSLA*, pages 142–157, Oct. 1997.

[29] J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Static Analysis 9th International Symposium, SAS 2002*, volume 2477 of *LNCS*, 2002.