Demand-Driven Points-to Analysis for Java

Manu Sridharan
University of California, Berkeley
manu_s@cs.berkeley.edu

Denis Gopan
University of Wisconsin, Madison
gopan@cs.wisc.edu

Lexin Shan
University of California, Berkeley
lshan@berkeley.edu

Rastislav Bodík
University of California, Berkeley
bodik@cs.berkeley.edu

ABSTRACT

We present a points-to analysis technique suitable for environments with small time and memory budgets, such as just-in-time (JIT) compilers and interactive development environments (IDEs). Our technique is demand-driven, performing only the work necessary to answer each query (a request for a variable's points-to information) issued by a client. In cases where even the demand-driven approach exceeds the time budget for a query, we employ *early termination*, *i.e.*, stopping the analysis prematurely and returning an over-approximated result to the client. Our technique improves on previous demand-driven points-to analysis algorithms [17, 33] by achieving much higher precision under small time budgets and early termination.

We formulate Andersen's analysis [5] for Java as a CFL-reachability problem [33]. This formulation shows that Andersen's analysis for Java is a *balanced-parentheses problem*, an insight that enables our new techniques. We exploit the balanced parentheses structure to approximate Andersen's analysis by *regularizing* the CFL-reachability problem, yielding an asymptotically cheaper algorithm. We also show how to regain most of the precision lost in the regular approximation as needed through *refinement*. Our evaluation shows that our regularization and refinement approach achieves nearly the precision of field-sensitive Andersen's analysis in time budgets as small as 2ms per query. Our technique can yield speedups of up to 16x over computing an exhaustive Andersen's analysis for some clients, with little to no precision loss.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Optimization*; D.2.6 [**Software Engineering**]: Programming Environments—*Interactive Environments*

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

General Terms

Algorithms, Languages, Performance

Keywords

Demand-driven analysis, points-to analysis, context-free language reachability, refinement

1. INTRODUCTION

Motivation Points-to analysis is an increasingly important prerequisite for analysis and optimization of object-oriented programs, since mainstream languages like Java and C# encourage pervasive use of pointers. Accordingly, much recent work has focused on creating scalable and precise points-to analyses [7, 10, 12, 18, 25, 26, 29–31, 36, 37, 39, 44, 45]. While impressive progress has been made, certain analysis environments with the following properties still cannot use existing algorithms:

Extreme resource constraints The environment needs the points-to analysis to run in a fraction of a second and consume very little memory.

Changing code The code being analyzed can change, and the points-to analysis must be able to quickly recompute its results after such a change.

One such environment is a just-in-time (JIT) compiler, the optimizer of choice for modern object-oriented languages. JIT optimization could benefit greatly from precise points-to information, for example by handling virtual calls more precisely. Many virtual calls in Java programs only have one possible call target, and precise points-to analysis is often necessary to find the target [26, 31]. Knowing this target allows a JIT compiler to inline the call, which in turn facilitates other optimizations. A JIT compiler has extreme resource constraints since any time and space overhead of compilation is counted against the running application. Code analyzed by a JIT compiler can change due to dynamic class loading.

Interactive development environments (IDEs), which increasingly incorporate sophisticated analyses to aid in program development, are another environment in need of better points-to analysis. Precise points-to information would be useful in an IDE both for program understanding and possibly for enabling new automatic refactorings [15]. IDEs

have extreme resource constraints, as they run while the developer is interacting with the tool, and changing code, since the developer is editing the target program.

To handle extreme resource constraints and changing code, a points-to analysis must ideally run in a fraction of a second and consume a very small amount of memory, and efficiently handle changes in the analyzed program. The most scalable recent analyzers for Java [26, 45] can take tens of seconds and tens of megabytes of memory to compute flow-insensitive context-insensitive Andersen's points-to analysis [5], and therefore are not suitable for use with extreme resource constraints. While recent work on incremental points-to analysis is promising [21, 24], building a high-performance points-to analysis that maintains precision and efficiency with arbitrary code changes is still an open problem.

Both JIT compilers and IDEs tend to only care about points-to information for a small subset of variables in the program: those in frequently-executing code for a JIT compiler, and those in the code being inspected by the developer for an IDE. This behavior naturally leads to the consideration of *demand-driven* points-to analysis algorithms, which perform only the work necessary to find points-to information for a set of variables specified by a client (the client's *queries*). Most points-to analysis algorithms are *exhaustive*, computing points-to information for all variables in the program[1].

While demand-driven algorithms seem promising, doing only the necessary work for a query may still be too expensive for our target environments. We have found that a demand version of field-sensitive Andersen's analysis for Java (adapted from an algorithm for C [17]) can take several seconds to find points-to information for a single variable, too long for an environment with extreme resource constraints. Our technique for handling such cases is *early termination*, in which, after some resource budget is exhausted, the points-to analysis prematurely terminates and returns a sound, easily computed, but possibly very imprecise result to the client. For example, when terminated early, a points-to analysis could say that a queried variable can point to all possible heap objects.

While using early termination can lead to a loss of precision, the severity of this loss depends on the client of the analysis, as suggested in [17]. If queries that require long running times to complete typically return results that are not useful to the client, then early termination of such queries do not lead to precision loss. For example, if a client issues a query to determine if a virtual call can be resolved to a single target, and the result from running the query to completion indicates multiple possible targets, the result from early termination is no worse for the client.

We found when applying existing demand-driven algorithms [17, 33] to Java, early termination and small time budgets lead to a large precision loss for our clients, as many queries that would have produced useful results if run to completion were terminated early. The key contribution of our work is a new demand-driven points-to analysis technique that retains nearly all of its precision when used with early termination.

Our Approach Our goal was to develop a demand-driven version of flow-insensitive, context-insensitive, field-sensitive Andersen's analysis, as presented in previous work [7, 26, 37, 44, 45]. Our technique was developed through insights gained from formulating this analysis as a context-free language reachability (CFL-reachability) problem [33], an extended form of graph reachability in which legal paths must be labelled with a string in a specified context-free language. To perform points-to analysis with CFL-reachability, a graph representation of the program is constructed, with nodes representing variables and objects and edges representing statements relevant to points-to analysis. Determining whether a variable x can point to an object o requires finding a path p between x and o's nodes in the graph, such that p's label is in a context-free language that ensures the corresponding program statements can cause x to point to o.

Our CFL-reachability formulation of Andersen's analysis for Java exposed a key difference with a similar formulation for C [33], namely that the Java analysis is a *balanced-parentheses problem*, similar to other program analysis problems formulated in CFL-reachability [33, 34]. Performing field-sensitive Andersen's analysis in CFL-reachability requires a context-free language that ensures that on any path between x and o indicating that x can point to o, field write edges (open parentheses) and field read edges (closed parentheses) are balanced.

To obtain precise results under a tight time budget, our points-to analysis technique uses *approximation* and *refinement*, both of which exploit the balanced-parentheses structure of Andersen's analysis for Java. Our approximation is based on *regularizing* the CFL-reachability problem of ensuring that field access parentheses are balanced. By assuming that an appropriate path always exists from the source of an open parenthesis edge to the target of a matching closed parenthesis edge on the same field, we can approximate field-sensitive Andersen's analysis with *regular reachability*, which is asymptotically less expensive than CFL-reachability. The RegularPT algorithm computes this regular reachability using simple depth-first search. Our regularization corresponds to a field-based handling of field accesses, which previous work has shown to have nearly the same precision as a field-sensitive handling [26, 28]. Our experiments show that RegularPT can achieve nearly 90% of the precision of field-sensitive Andersen's analysis for our clients, with a time budget of only 2ms per query.

In cases where an answer to a query using regular reachability is insufficiently precise for a client's needs, we can *iteratively refine* our approximation to regain precision. We refine by checking for the existence of a connecting path between parenthesis edges that our regularization assumed was present. The connecting paths are discovered using approximate regular reachability, and may therefore require further refinement, leading to an iterative refinement process that can regain nearly all the precision lost through approximation. The RefinedRegularPT algorithm extends RegularPT to perform this iterative refinement. Given a time budget of 20ms per query, our evaluation shows that RefinedRegularPT is up to 3.8% more precise than RegularPT (relative to field-sensitive Andersen's) and still achieves much higher precision than previous techniques.

[1]Some pointer analyses can a be viewed as answering queries on demand after performing some preprocessing work on the entire program, e.g. [10, 12]. Given our extreme resource constraints, we restrict our definition of demand-driven algorithms to techniques that only *examine* the statements required for a particular query [17, 22].

Contributions Our main contributions are:

- **Balanced Parentheses:** We formulate Andersen's analysis for Java as a CFL-reachability problem and observe that the context-free grammar that governs the analysis has the balanced parentheses property, with reads and writes of instance fields as the parentheses (Section 2).
- **Regular Approximation:** We show how to use the balanced-parentheses property to approximate the Andersen's analysis CFL-reachability problem with regular reachability, yielding a fast algorithm RegularPT that essentially performs depth-first search (Section 3).
- **Refinement:** We give a technique for iteratively refining our regular approximation using recursive reachability queries, and present an algorithm RefinedRegularPT that employs this technique (Section 4).
- **Evaluation:** We thoroughly evaluate our algorithms, and show that for our experimental clients they are precise, retain most of their precision under tight time budgets and early termination, and yield large performance improvements over available exhaustive and demand-driven algorithms (Section 5).

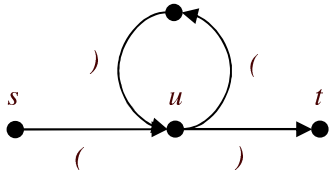
2. ANDERSEN'S ANALYSIS VIA CFL-REACHABILITY

In this section, we present CFL-reachability [33] (Section 2.1) and Andersen's points-to analysis [5] (Section 2.2), and then give a formulation of Andersen's analysis for Java in terms of CFL-reachability (Section 2.3).

2.1 CFL-reachability

Context-free language reachability (CFL-reachability) is an extension of traditional graph reachability. We are given a directed graph G whose edges are labelled with letters in some alphabet Σ and a context-free language L over Σ. Each path p in G has a corresponding string $s(p)$ in Σ^*, constructed by concatenating in order the labels of edges in p. We say p is an *L-path* iff $s(p) \in L$. Given nodes v and w in G, w is *L-reachable* from v iff there exists an L-path in G from v to w.

As an example, let Σ be the letters '(' and ')', and L be the set of strings with balanced parentheses generated by the grammar $S \rightarrow SS \mid (\,S\,) \mid \epsilon$. Consider the following graph G, adapted from [33]:



There is exactly one L-path p from s to t, with $s(p) =$ "(())". We say p is a *nonTerm-path* if and only if nonterminal *nonTerm* generates $s(p)$, and n_1 *nonTerm* n_2 means a *nonTerm*-path from n_1 to n_2 exists. So, in our example, we have an S-path from s to t, or $s\ S\ t$. While node t is L-reachable from node s, node u is not.

CFL-reachability is often used to ensure that some set of parentheses are properly matched. For example, in interprocedural dataflow analysis [34], CFL-reachability is used for context-sensitivity, with the entry and exit edges of each method invocation as the matched parentheses. We will use CFL-reachability to match field read and write parentheses in our points-to analysis. Note that not all analyses formulated in CFL-reachability use the context-free language to check for matched parentheses, for example Andersen's analysis for C [33].

```
1  x = new Obj(); // o1
2  z = new Obj(); // o2
3  w = x;
4  y = x;
5  y.f = z;
6  v = w.f;
```

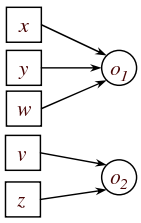


Figure 1: A small code example and the points-to relation computed by Andersen's analysis.

Determining CFL-reachability is computationally more expensive than standard graph reachability. Consider the *single-source L-path problem*, which requires finding all nodes L-reachable from some source node n. The best known algorithm solves the single-source L-path problem in worst-case $O(\Gamma^3 N^3)$ time [33], where N is the number of nodes in G and Γ is the size of a normalized grammar for L. In contrast, a single-source standard reachability problem can be solved by depth-first search in $O(E)$ time. Note that when L is a regular language, the single-source L-path problem can be solved in $O(SE)$ time, where S is the number of states in a deterministic finite automaton for L [47]. Our technique uses regular reachability for points-to analysis through approximation of a CFL-reachability problem.

2.2 Andersen's Analysis

Points-to analysis is traditionally presented as the problem of computing a *points-to relation* that conservatively maps each pointer variable to the heap objects it can point to at runtime. Figure 1 gives a small Java code example and shows the points-to relation computed by Andersen's points-to analysis [5] for the example. Since statically determining the set of objects that a program will allocate at runtime is undecidable, dynamic heap objects are modelled with a finite set of *abstract locations*. Andersen's analysis uses a single abstract location to represent the objects created by all executions of a single allocation statement (e.g. new in Java). In Figure 1, abstract locations o_1 and o_2 respectively model the objects allocated at lines 1 and 2 of the code. Typically, the points-to relation is represented with *points-to sets*; for each variable x, the points-to set $pt(x)$ contains all abstract locations o such that (x, o) is in the points-to relation.

Andersen's analysis [5] is a *subset-based*, *flow-insensitive*, *context-insensitive* points-to analysis. A subset-based points-to analysis models the effects of a statement `x = y` with the subset constraint $pt(y) \subseteq pt(x)$, precisely modelling the copying of values from y to x; other pointer-manipulating statements are also modelled with subset constraints. Flow-insensitivity means that the analysis disregards the control-flow of the program, treating its statements as if they could execute in any order. Finally, since the analysis is context-insensitive, it merges information from different calls of a procedure, rather than reasoning about each call separately.

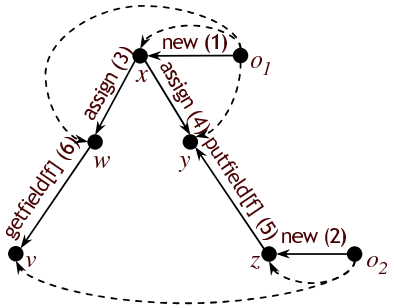


Figure 2: The graph representation of the code of Figure 1. The line number of the statement from Figure 1 corresponding to an edge is indicated in parentheses. Dashed edges indicate L_{FT}-reachability.

The *flows-to* relation is the inverse of the points-to relation; (o, x) is in the flows-to relation iff $o \in pt(x)$. For ease of presentation, our formulation of Andersen's analysis in CFL-reachability will focus on how to compute the flows-to relation rather than points-to.

2.3 CFL-reachability formulation

Our CFL-reachability formulation of Andersen's analysis for Java is based on a previous CFL-reachability formulation of Andersen's analysis for C [33]; our contribution is to adapt the previous formulation to Java[2]. We first define a graph representation G of a Java program P, with nodes for variables and abstract locations and edges for the different pointer-manipulating statements of P. Then, we develop a context-free language L_{FT} (FT for flows-to) with the following property: if a heap object represented by abstract location o can flow to variable x during the execution of P, then x will be L_{FT}-reachable from o in G. Hence, computing L_{FT}-reachability is equivalent to computing the flows-to relation.

Figure 2 shows the graph representation of the code of Figure 1. A dotted edge from o to x in the figure indicates that x is L_{FT}-reachable from o. We indicate in parentheses the line number of the statement from Figure 1 corresponding to each edge; these line numbers are not part of the edge labels.

We include edges for the following types of canonical statements in our graph. A statement `x = new Obj()` is modelled with a new edge $o \rightarrow x$, where o is the abstract location node for the statement. We include an assign edge $x \rightarrow y$ for a statement `y = x`. We have a putfield[f] edge $x \rightarrow y$ for a statement `y.f = x`, and a getfield[f] edge $x \rightarrow y$ for a program statement `y = x.f`; the f in the edge label is the field being accessed. We define the *base variable* of a field access edge to be the variable being dereferenced, the source of a getfield[f] edge and the target of a putfield[f] edge. Loads and stores to arrays are modelled by representing all elements of an array as a field *arr*, so a read (write) access to any array element becomes a getfield[arr] (putfield[arr]) statement.

Let us first see how to define L_{FT} given a graph with only new and assign edges; in this restricted case, the language is regular. Let the start symbol of L_{FT} be *flowsTo*: if we have o *flowsTo* x, then x is L_{FT}-reachable from o. A new statement sets its assigned variable to point to a new heap object, creating a *flowsTo*-path from the source of the new edge to its target, like the one from o_1 to x in Figure 2. An assignment `p = q` causes p to point to the object that q points to, so if there is a *flowsTo*-path from o to q, there should also be a *flowsTo*-path from o to p. In Figure 2, the edges o_1 new x and x assign w comprise a *flowsTo*-path from o_1 to w. We can write a simple regular expression for L_{FT} over the assign and new terminals:

$$\mathit{flowsTo} \rightarrow \mathsf{new}\ (\ \mathsf{assign}\ )^*$$

Now we must reason about field reads and writes. We need to extend L_{FT} to (conservatively) handle the case where an object o flows to a variable x by first being written into a field f of some object o', and then being read from the f field of o' into x. Points-to analyses handle fields with various degrees of precision. A *field-sensitive* handling of fields is defined as follows:

DEFINITION 1. *Given $o \in pt(d)$, some field write `c.f = d`, and some field read `a = b.g`, a* field-sensitive analysis *concludes $o \in pt(a)$ iff the following conditions hold.*

1. *Fields f and g correspond to the same memory offset in an object.*
2. *b and c are may-aliased, i.e., $pt(b) \cap pt(c) \neq \emptyset$.*

For Java programs, the first condition reduces to checking that $f = g$, since Java guarantees that each instance field of an object names a distinct offset[3]. The second condition checks for *may-aliasing* between the base variables b and c of the field accesses, *i.e*, that b and c may point to the same object at runtime. If this condition does not hold, then b and c *always* point to different heap objects, and hence the field write and read cannot induce a flow of objects from d to a.

As an example, consider statements 5 and 6 in Figure 1, `y.f = z` and `v = w.f`. These statements satisfy condition 1 of Definition 1 since they access the same instance field f. Furthermore, y and w can both point to o_1, and are therefore may-aliased, satisfying condition 2. So, a field-sensitive analysis will conclude that these statements can cause a flow of objects through the f field. For example, since z can point to o_2, statements 5 and 6 imply that v can also point to o_2.

The may-aliasing relation between variables given in Definition 1 cannot be formulated in CFL-reachability with our graph representation as defined thus far. For example, while variables w and y from Figure 1 are may-aliased, w is unreachable from y in the graph of Figure 2 even by standard graph reachability. w and y are both L_{FT}-reachable from o_1 in the graph, which implies that they are may-aliased. But, because of the direction of the edges in the graph, we cannot combine the corresponding *flowsTo*-paths from o_1 to w and y into a single path from w to y.

To overcome this limitation, we introduce reversed edges, or *barred edges*, to the graph [33]. For each edge $n_1 \rightarrow n_2$ labelled `t` in the graph, we now also have an edge $n_2 \rightarrow n_1$

[2]Our CFL-reachability formulation of Andersen's analysis for Java is equivalent to previously presented set constraints formulations [7, 26, 37, 44, 45]; we elide the details of the correspondence, which are straightforward.

[3]This property does not always hold for structure fields in C because of casts; see [48] for details.

labelled $\bar{\mathsf{t}}$. Note that in all of our example graphs, we omit the barred edges for clarity. Given a graph with barred edges, a reverse path $\overline{p}$ can be constructed for any path p by reversing the order of the edges in p and replacing each edge in p with its inverse, substituting barred edges for standard edges and vice-versa. In our example from Figure 2, barred edges allow us to reverse the *flowsTo*-path from o_1 to w, yielding a $\overline{flowsTo}$-path from w to o_1. This $\overline{flowsTo}$-path can be combined with the *flowsTo*-path from o_1 to y to obtain the desired path from w to y.

We characterize paths that show variables to be may-aliased with the following production:

$$alias \rightarrow \overline{flowsTo}\ \ flowsTo$$

To show that this production for *alias* captures the may-aliasing definition in Condition 2 of Definition 1, we must show that variables b and c are may-aliased iff they are connected by an *alias*-path; here we argue the forward direction. If b and c are may-aliased, then by Definition 1 we have $pt(b) \cap pt(c) \neq \emptyset$. Since the flows-to relation is the inverse of points-to, this means that there exists some o' such that o' *flowsTo* b and o' *flowsTo* c. Given o' *flowsTo* b, we can construct a reverse $\overline{flowsTo}$-path from b to o' using barred edges. The *flowsTo*-path from o' to c can be appended to this $\overline{flowsTo}$-path to yield the desired *alias*-path from b to c.

We are now ready to add field-sensitive handling of fields to L_{FT} by updating the *flowsTo* production as follows:

$$flowsTo \rightarrow \mathsf{new}\ \ (\ \mathsf{assign}\ \mid\ \mathsf{putfield}[f]\ \ alias\ \ \mathsf{getfield}[f]\ )^*$$

This production checks for putfield[f] / getfield[f] statement pairs that meet the conditions of Definition 1. The L_{FT} language is no longer regular, but a context-free language of balanced parentheses, with putfield[f] and getfield[f] as the parens.

The CFL-reachability formulation of Andersen's analysis for C in [33] does not have the balanced-parentheses property, primarily because unlike Java, C allows the address of a pointer variable to be taken (using the & operator). Our algorithms rely on the balanced-parentheses property, and hence are not immediately applicable to C. A detailed discussion of points-to analysis for C vs. Java is beyond the scope of this paper.

Figure 3 gives a context-free grammar for L_{FT}. The terminals are the labels for statement edges and their inverse edges (e.g. assign and $\overline{\mathsf{assign}}$). Each $\overline{flowsTo}$ production reverses the *flowsTo* production to its left and inverts the edges. We use *alias* in the last $\overline{flowsTo}$ production where $\overline{alias}$ would be expected since the two have identical productions: $\overline{alias} \rightarrow \overline{\overline{flowsTo}\ flowsTo} = \overline{flowsTo}\ \overline{\overline{flowsTo}} = \overline{flowsTo}\ flowsTo$. Note that we can easily write a production for a points-to non-terminal, $pointsTo \rightarrow \overline{flowsTo}$, showing that the grammar could be re-written to emphasize *pointsTo*-paths rather than *flowsTo*-paths.

Going back to our example in Figure 2, let us see how to derive the *flowsTo* path from $o2$ to v. First, we derive y *alias* w as follows:

$$\begin{aligned} & y\ \ \overline{\mathsf{assign}}\ \ x\ \ \overline{\mathsf{new}}\ \ o1\ \ \mathsf{new}\ \ x\ \ \mathsf{assign}\ \ w \\ \rightarrow\ & y\ \ \overline{flowsTo}\ \ o1\ \ flowsTo\ \ w \\ \rightarrow\ & y\ \ alias\ \ w \end{aligned}$$

With this *alias* path and statements 5 and 6, we can now derive $o2$ *flowsTo* v:

$$\begin{aligned} & o2\ \ \mathsf{new}\ \ z\ \ \mathsf{putfield}[f]\ \ y\ \ alias\ \ w\ \ \mathsf{getfield}[f]\ \ v \\ \rightarrow\ & o2\ \ flowsTo\ \ z\ \ \mathsf{putfield}[f]\ \ y\ \ alias\ \ w\ \ \mathsf{getfield}[f]\ \ v \\ \rightarrow\ & o2\ \ flowsTo\ \ v \end{aligned}$$

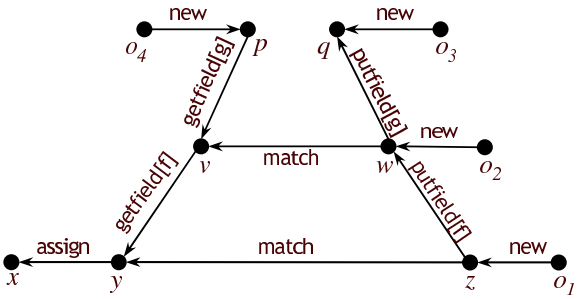


Figure 4: A graph illustrating match edges.

3. REGULAR APPROXIMATION

In this section, we show how to *regularize* L_{FT}-reachability by approximating with reachability over a regular language R_{FT}, taking advantage of L_{FT}'s balanced-parentheses structure. This regularization is useful because, as discussed in Section 2.1, reachability over a regular language is asymptotically less expensive than CFL-reachability. We give a simple and efficient demand-driven algorithm RegularPT, essentially depth-first search, for finding points-to information based on R_{FT}-reachability. Section 5 will show that RegularPT achieves most of the precision of L_{FT}-reachability within a time budget of only 2ms per query.

3.1 Regular Reachability

Recall our field-sensitive production for L_{FT} from Section 2.3:

$$flowsTo \rightarrow \mathsf{new}\ \ (\ \mathsf{assign}\ \mid\ \mathsf{putfield}[f]\ \ alias\ \ \mathsf{getfield}[f]\ )^*$$

The context-free aspect of L_{FT}-reachability is checking putfield[f] and getfield[f] edges, the balanced parentheses of L_{FT}, for the field-sensitivity conditions specified in Definition 1. The most expensive part of field-sensitivity is checking for an *alias*-path between the base variables of putfield[f] and getfield[f] edges, since the path may be complex and must itself have balanced parentheses.

To make determining L_{FT}-reachability less costly, we eliminate the search for *alias*-paths between matched parentheses by conservatively assuming that such a path always exists. With this assumption, handling of putfield[f] / getfield[f] statement pairs reduces to checking that the accessed fields are identical. We perform this check for identical fields ahead of time by adding *match edges* to our graph, from the source of each putfield[f] edge to the target of each getfield[f] edge on the same field f. Figure 4 shows an example graph with all match edges included. Given a graph with all possible match edges, we can over-approximate L_{FT}-reachability with *regular reachability* using language R_{FT}, defined as follows:

$$flowsToReg \rightarrow \mathsf{new}\ \ (\ \mathsf{assign}\ \mid\ \mathsf{match}\ )^*$$

Given graph G, let G_R be G with match edges added from the source of each putfield[f] edge to the target of

$flowsTo$	$\rightarrow$ new	$\overline{flowsTo}$	$\rightarrow$ $\overline{\text{new}}$
$flowsTo$	$\rightarrow flowsTo$ assign	$\overline{flowsTo}$	$\rightarrow$ $\overline{\text{assign}}$ $\overline{flowsTo}$
$flowsTo$	$\rightarrow flowsTo$ putfield[f] $alias$ getfield[f]	$\overline{flowsTo}$	$\rightarrow$ $\overline{\text{getfield}}[f]$ $alias$ $\overline{\text{putfield}}[f]$ $\overline{flowsTo}$
$alias$	$\rightarrow \overline{flowsTo}$ $flowsTo$		

Figure 3: A context-free grammar for L_{FT}. The terminals putfield[f] and getfield[f], and their barred versions, are the parentheses of the language.

each getfield[f] edge on the same field f. To show that R_{FT}-reachability over-approximates L_{FT}-reachability and is sound, We must show that if x is L_{FT}-reachable from o in G, then x is R_{FT}-reachable from o in G_R. Note that the *flowsToReg* production differs from the *flowsTo* production only in that "putfield[f] *alias* getfield[f]" has been replaced by match. So, the soundness proof reduces to showing that whenever nodes x and y in G are connected by a path labelled putfield[f] *alias* getfield[f], G_R includes a match edge from x to y. This clearly holds by construction, since we add match edges between putfield[f] and getfield[f] edges in G_R regardless of whether their base variables are connected by an *alias*-path.

Since R_{FT}-reachability over-approximates L_{FT}-reachability, node x may be R_{FT}-reachable from node o in G_R but not L_{FT}-reachable from o. For example, in Figure 4, v is R_{FT}-reachable from o_2 but not L_{FT}-reachable, since there is no *alias*-path from q to p. In general, precision is lost in cases where an R_{FT}-path includes an *invalid* match edge m, where the base variables of the field accesses corresponding to m are not connected by an *alias*-path. A points-to analysis technique that, like R_{FT}-reachability, handles field accesses by checking only for matching fields is known as a *field-based analysis*. This technique has been shown to have precision relatively close to that of a field-sensitive analysis in previous work [26, 28]. In Section 4, we show how refinement can be used to recover most of the precision lost by using match edges.

Since R_{FT} is regular, answering the single-source R_{FT}-problem is asymptotically cheaper than the single-source L_{FT}-problem, as discussed in Section 2.1. Note the simplicity of this regular expression for *flowsToReg*, relative to the grammar of Figure 3. One consequence of our use of match edges is that we no longer need to consider both standard and barred edges when determining R_{FT}-reachability; this leads to both conceptual simplicity and a simple depth-first search algorithm.

3.2 RegularPT

Figure 5 gives pseudocode for an algorithm RegularPT, which determines points-to information on demand using R_{FT}-reachability. The `regularPT` procedure takes a node x and returns the set of all nodes o such that x is R_{FT}-reachable from o. This returned set is a points-to set for x, since R_{FT}-reachability over-approximates the flows-to relation. The `doTraversal` procedure is a standard worklist-based depth-first search, traversing incoming assign and match edges to find reachable new edges and their abstract locations.

The worst-case complexity of RegularPT is $O(E + M)$, where E is the number of edges in G and M is the number of match edges in G_R. The derivation of this bound is straightforward, as G_R has $E + M$ edges and RegularPT essentially performs depth-first search on G_R. In real-world Java programs, E is typically $O(N)$ (where N is the number of nodes in G), since variables are typically assigned very few times. While M can be $O(N^2)$ in the worst-case, we have not observed a large number of match edges to be a scalability bottleneck in practice.

```
procedure regularPT(x: Node): Set of Node
  pointsTo: Set of Node
  marked: Set of Node
  worklist: List of Node
  propagate(source, marked, worklist)
  while (worklist is non-empty) do
    remove w from front of worklist
    foreach NEW edge o -> w do
      add o to pointsTo
    end
    foreach ASSIGN and MATCH edge y -> w do
      propagate(y, marked, worklist)
    end
  end
  return pointsTo
end
procedure propagate(x: Node,
                    marked: Set of Node,
                    worklist: Set of Node)
  if (x not in marked) then
    add x to marked; add x to front of worklist
  end
end
```

Figure 5: Pseudocode for the RegularPT algorithm.

3.3 Improving Precision with Types

The Java type system can be used to improve the precision of RegularPT in two ways. We say types `A` and `B` are *incompatible* if `A` is not a subtype of `B` and `B` is not a subtype of `A`[4]. If variable x has declared type `A`, and variable y has incompatible declared type `B`, then any *flowsToReg*-path ending at x that passes through y can safely be ignored, since the type system prohibits a flow of objects from y to x. Such *flowsToReg*-paths can exist in the graph because of downcasts. For example, given statements `Object o = a; b = (B)o`, we have a *flowsToReg* b, even if a and b have incompatible declared types. When answering a query for variable x with declared type `A`, RegularPT does not add nodes whose declared types are incompatible with `A` to the worklist.

We can also decrease the number of added match edges using types. It is possible for the base variables of a getfield[f] edge and putfield[f] edge (e.g. nodes v and w in Figure 4) to have incompatible types, even though the f field is accessed on both variables. For example, if we have class `A` with field

[4]If `A` is an interface, we must check that all classes implementing `A` are incompatible.

f, class `B extends A`, and class `C extends A`, then `B` and `C` are incompatible in spite of both having field f. When the base variables of a getfield[f] edge $v \rightarrow y$ and a putfield[f] edge $z \rightarrow w$ have incompatible types, we can safely avoid adding a match edge from z to y, since there will *never* be an *alias*-path between v and w. Empirically, these type-based tests considerably improved precision on our benchmarks, consistent with results in past work on Java points-to analysis that used similar techniques [26].

4. REFINEMENT

Here we show how a simple refinement technique allows us to recover most of the precision lost by approximating L_{FT}-reachability with R_{FT}-reachability. Our evaluation shows the precision of R_{FT}-reachability to be relatively close to that of L_{FT}-reachability for the clients we tested (see Section 5). However, in cases where the precision of R_{FT}-reachability is insufficient and time constraints are tight, our experiments show that our refinement technique is more precise than computing fully field-sensitive L_{FT}-reachability.

In this section, we first describe how a match edge can be *refined* to check if it has introduced imprecision relative to L_{FT}-reachability. We then give an algorithm RefinedRegularPT that iteratively refines match edges. In each iteration, RefinedRegularPT adds precision by refining more match edges, terminating when either the points-to analysis client is satisfied or all relevant match edges have been refined. When refining all match edges, RefinedRegularPT can provide nearly the same precision as L_{FT}-reachability.

4.1 Refining match edges

When match edges introduce imprecision relative to computing L_{FT}-reachability, the match edges can be *refined* to recover (most) lost precision. Recall that the soundness of R_{FT}-reachability relies on the fact that whenever there is a "putfield[f] *alias* getfield[f]"-path from x to y in the original graph G, there is a match edge from x to y in G_R. Our technique refines a match edge m by verifying that no *alias*-path exists between the base variables of the corresponding putfield[f] and getfield[f] edges; if no *alias*-path exists, m can be removed from G_R without affecting the soundness of R_{FT}-reachability.

We use R_{FT}-reachability to approximate the search for an *alias*-path during refinement. The *alias* production $\overline{flowsTo}$ *flowsTo*, given in Figure 3, requires using potentially expensive L_{FT}-reachability, as an *alias*-path cannot contain match edges. Instead of searching for *alias*-paths, we refine a match edge by looking for *aliasReg*-paths, defined with a simple extension of the R_{FT} grammar (see Section 3.1):

$$aliasReg \rightarrow \overline{flowsToReg}\ flowsToReg$$
$$\overline{flowsToReg} \rightarrow (\ \overline{\mathsf{assign}}\ \mid\ \overline{\mathsf{match}}\ )^*\ \ \overline{\mathsf{new}}$$

Finding *aliasReg*-paths instead of *alias*-paths during refinement is clearly sound, by an argument similar to the soundness argument for R_{FT}-reachability. An *aliasReg*-path may itself contain match edges, and could therefore connect two nodes that are not connected by an *alias*-path. However, we can once again regain precision using refinement, this time refining the match edges on the *aliasReg*-path. Our iterative refinement algorithm is based on repeatedly discovering *aliasReg*-paths while refining match edges, and then refining the match edges on those *aliasReg*-paths.

As an example, let us consider refining the match edge $z \rightarrow y$ in Figure 4. To do so, we search for an *aliasReg*-path from w to v, and we find one via o_2 labelled $\overline{\mathsf{new}}$ new match. This path includes the match edge $w \rightarrow v$, which we can in turn refine by searching for an *aliasReg*-path from q to p. No such path exists, so the match edge $w \rightarrow v$ can be safely removed from the graph. This removal eliminates the only *aliasReg*-path from w to v, meaning the match edge $z \rightarrow y$ can also be safely removed.

4.2 RefinedRegularPT

The RefinedRegularPT algorithm is an extension of the RegularPT algorithm that performs iterative refinement of match edges based on the needs of the points-to analysis client. A client of a points-to analysis typically aims to perform some transformation or verification of a program that relies on some pointer-related program property. We say that a points-to query is *positively answered* when the points-to information computed by the analysis allows the client to prove the relevant program property. The refinement loop of RefinedRegularPT iterates until the given query is positively answered, or until no further refinement is possible.

For example, consider a points-to analysis client that tries to resolve virtual calls for the purpose of inlining. Given a virtual call `x.foo()`, this client tries to use points-to information for x to show that only one implementation of `foo()` can be invoked by this call at runtime, allowing for inlining of that implementation at the call site. The client issues a query for the receiver x of the call to `foo` to RefinedRegularPT, and considers the query positively answered when the points-to information for x shows that the call to `foo` has only one possible target.

Figure 6 gives pseudocode for the RefinedRegularPT algorithm. The `refinedRegularPT` procedure takes as input a node x and returns `true` if the query has been positively answered, and `false` otherwise[5]. The `doTraversal` procedure is similar in function to the `regularPT` procedure of RegularPT (seen in Figure 5), computing a points-to set for x through a depth-first traversal of the graph; each call to `doTraversal` computes a new points-to set. The `propagate` procedure is identical to that of Figure 5.

Refining match edges Lines 24-30 of Figure 6 perform refinement of match edges. The pseudocode checks for an *aliasReg*-path between a getfield[f] edge $p \rightarrow w$ and a putfield[f] edge $y \rightarrow q$ by finding points-to sets $pt(p)$ and $pt(q)$ (lines 24 and 26), and then checking if $pt(p) \cap pt(q) \neq \emptyset$. The points-to sets include nodes that are reachable along $\overline{flowsToReg}$-paths from p and q. Therefore, for any $o \in pt(p) \cap pt(q)$, an *aliasReg*-path from p to q through o can be constructed. For a given getfield[f] edge $p \rightarrow w$, there may be many putfield[f] edges on the same field, and therefore many incoming match edges to w. Instead of refining each such match edge individually, RefinedRegularPT refines them together, thereby avoiding redundant computation of $pt(p)$ for each edge.

We have found an alternate strategy for refining match edges to be empirically more efficient in certain cases. The alternate strategy first finds $pt(p)$, but then finds the set of nodes Q that are reachable along *flowsToReg*-paths from nodes in $pt(p)$; if $q \in Q$, an *aliasReg*-path from p to q clearly

[5]For positively answered queries, our implementation also makes the computed points-to set available to the client.

```
1  getfieldsToRefine: Set of GETFIELD[f] edges
2  getfieldsSeen: Set of GETFIELD[f] edges
3  procedure doTraversal(x: Node): Set of Node
4    pointsTo: Set of Node
5    marked: Set of Node
6    worklist: List of Node
7    propagate(x, marked, worklist)
8    while (worklist is non-empty) do
9      remove w from front of worklist
10     foreach NEW edge o -> w do
11       add o to pointsTo
12     end
13     foreach ASSIGN edge y -> w do
14       propagate(y, marked, worklist)
15     end
16     foreach GETFIELD[f] edge e = p -> w do
17       if (e !in getfieldsToRefine)
18         add e to getfieldsSeen
19         foreach PUTFIELD[f] edge y -> q do
20           propagate(y, marked, worklist)
21         end
22       else
23         remove e from getfieldsToRefine
24         ptOfP = doTraversal(p)
25         foreach PUTFIELD[f] edge y -> q do
26           ptOfQ = doTraversal(q)
27           if (ptOfP intersects ptOfQ)
28             propagate(y, marked, worklist)
29           end
30         end
31         add e to getfieldsToRefine
32       end
33     end
34   end
35   return pointsTo
36 end
37 procedure refinedRegularPT(x: Node): bool
38   clear getfieldsToRefine
39   while true do
40     clear getfieldsSeen
41     pointsTo := doTraversal(x)
42     if (positivelyAnswered(pointsTo))
43       return true
44     else
45       if (getfieldsSeen
46            contained in getfieldsToRefine)
47         return false
48       else
49         add getfieldsSeen
50           to getfieldsToRefine
51       end
52     end
53   end
54 end
```

Figure 6: Pseudocode for the RefinedRegularPT algorithm.

exists. We observed that in practice this alternate strategy traverses fewer nodes when there are more than two matching putfield[f] edges for a getfield[f] edge, since it does not compute a points-to set for the target of each putfield[f] edge. Our implementation employs the appropriate strategy based on the number of matching putfield[f] edges.

Choosing match edges to refine RefinedRegularPT focuses analysis effort on match edges that have already been shown to possibly cause imprecision. RefinedRegularPT first tries to answer a query on some variable x without any refinement, just using R_{FT}-reachability. Consider the case where the points-to set for x found using R_{FT}-reachability cannot positively answer the query. Let K be the set of match edges on any R_{FT}-path from some abstract location o to x. If using R_{FT}-reachability to find x's points-to set is less precise than L_{FT}-reachability, then K must contain at least one invalid match edge (*i.e.*, a match edge with no corresponding *alias*-path). In its next pass, RefinedRegularPT only refines match edges in K, aiming to positively answer the query with this small amount of refinement; we have found this amount of refinement to often be sufficient in practice. Proving that some match edge in K is invalid may however require refinement of match edges outside of K, leading to an iterative refinement process.

In the pseudocode of Figure 6, we maintain a set `getfieldsToRefine`, containing the getfield[f] edges whose corresponding match edges should be refined, and a set `getfieldsSeen`, containing getfield[f] edges whose match edges were traversed but not refined in the current iteration of the algorithm. `getfieldsToRefine` is maintained across iterations of the algorithm, while `getfieldsSeen` is cleared on each iteration (line 40). If the points-to result computed by an iteration of the algorithm is sufficient for a positively answered query, we terminate and return `true` (lines 42-43); the `positivelyAnswered` procedure is provided by the client. Otherwise, we add the getfield[f] edges in `getfieldsSeen` to `getfieldsToRefine` (lines 49-50), and begin a new iteration. When we cannot add any new getfield[f] edges to `getfieldsToRefine`, we give up on positively answering the query and return `false` (lines 45-47).

While refining match edges corresponding to a getfield[f] edge e, we remove e from `getfieldsToRefine` (lines 23 and 31 of Figure 6). To see why, consider refining a match edge m for a getfield[next] edge $e = x \rightarrow x$, corresponding to the statement `x = x.next`. The recursive call to `doTraversal` at line 24 will pass x as its argument, and if e remained in `getfieldsToRefine`, RefinedRegularPT would again try to refine m, leading to an infinite loop.

Removing a getfield[f] edge from `getfieldsToRefine` during refinement of a corresponding match edge m can lead to imprecision. With the getfield[f] edge removed, RefinedRegularPT may find *aliasReg*-paths during refinement of m that include m itself, as m will not be refined again when encountered. If all *aliasReg*-paths discovered during refinement of m include m, then m is still invalid, as m cannot be used to justify its own existence. However, in such cases RefinedRegularPT is unable to show that m is invalid, losing precision relative to a fully field-sensitive analysis. In general, if RefinedRegularPT refines all match edges, it may compute a less precise result than L_{FT}-reachability in cases where G_R contains cyclic paths that include field dereferences, e.g. the cyclic getfield[next] edge $x \rightarrow x$ for `x = x.next`. We have not found the precision loss due to this aspect of our algorithm to be significant in practice.

In the worst-case, a single iteration of RefinedRegularPT may require $O(M^M E)$ time, with E and M defined as in Section 3.2. This worst case occurs when all match edges are being refined, and refining one match edge requires refining $M-1$ other match edges, each of which requires refining $M-2$ match edges, and so on. We have not encountered this worst-case behavior in practice, and since we envision clients using strict time budgets and early termination with our algorithms, it is not a practical concern.

5. EVALUATION

We evaluate the behavior of RegularPT and RefinedRegularPT with two clients and several benchmarks. Our evaluation validates the following experimental hypotheses about the algorithms:

The algorithms are precise We show that RegularPT has precision close to that of field-sensitive Andersen's analysis. It resolves more than 89% of the virtual calls that field-sensitive Andersen's analysis can across our benchmarks, and more than 96% of those virtual calls that are not in dead code. RefinedRegularPT provides more precision than RegularPT, resolving nearly all of the virtual calls in live code that field-sensitive Andersen's can. We also show that an intraprocedural version of RegularPT resolves far fewer calls, indicating that our results cannot be obtained with purely intraprocedural analysis.

Precision retained under early termination We show that RegularPT and RefinedRegularPT retain almost all their precision when run with small time budgets and early termination. For nearly all benchmarks, the two algorithms can resolve 90% of the virtual calls that field-sensitive Andersen's can within a 50 node traversal limit (2ms / query). We show that an adaptation of a previously presented demand-driven algorithm [17] that uses full field-sensitivity does not perform nearly as well within a small budget. RegularPT and RefinedRegularPT also answer *all* virtual call and aliasing queries in hot methods of the SPEC benchmarks as precisely as field-sensitive Andersen's analysis, requiring less than 108 nodes of traversal per query.

The algorithms meet our performance goals Since our algorithms perform well with small time budgets, timeouts can be used to ensure good performance while maintaining precise results. For example, we can answer all virtual call queries in hot methods of the `javac` benchmark 16x faster than exhaustive field-based Andersen's analysis and 34x faster than exhaustive field-sensitive Andersen's analysis. The memory consumption of RegularPT and RefinedRegularPT is also much less than that of an exhaustive algorithm.

5.1 Experimental Configuration

Implementation We implement our analyses using the Soot 2.2.1 [42] and SPARK [26] frameworks. We re-use the pointer assignment graph built by SPARK, thereby leveraging their existing analyses for determining reachable code. To handle method calls, we configure SPARK to build a conservative call graph using a class-hierarchy analysis [6, 11]. For each method m, we have a node for each formal parameter of m of reference type, and if necessary a node ret_m for its return value (`return` statements in m are modelled with an assign edge to ret_m). We model a call of method m with assign edges from each actual parameter to the appropriate formal parameter node and from the ret_m node to the appropriate variable at the caller.

Our strategy for handling native methods and reflection is as follows. For native methods in the libraries, we use models provided by Soot. Soot also builds a call graph that soundly handles many reflection calls, assuming that all code is present for analysis. For example, a call to `Class.newInstance()` is modelled by adding edges to all constructors in the program with no arguments. To be conservative, native methods in application code and reflective calls whose targets cannot be soundly determined are handled by telling the client that the query cannot be positively answered.

Algorithm	Description
RegularPT	See Section 3.2
RefinedRegularPT	See Section 4.2
FullFS	Adaptation of algorithm in [17]; See Appendix A
ExhaustiveFB	Exhaustive field-based Andersen's from SPARK [26]
ExhaustiveFS	Exhaustive field-sensitive Andersen's from SPARK [26]

Table 1: Descriptions of points-to analysis algorithms used in our experiments.

To compare against the state-of-the-art, we also implemented a demand-driven algorithm FullFS that uses the same techniques as the algorithm in [17], but works for Java pointer constructs. The basic idea of the algorithm is to find points-to sets for only the variables necessary to answer a top-level points-to query. The points-to sets are found by iterating over relevant statements, applying inference rules to introduce new points-to queries for relevant variables and to propagate abstract locations to queried variables. The algorithm is similar to exhaustive propagation algorithms for Java [26, 45], except that it only propagates the abstract locations relevant to the query. We chose to make FullFS treat fields with full field-sensitivity, thereby computing L_{FT}-reachability. The C algorithm in [17] is field-sensitive for the unnamed field accessed by the C `*` operator, but is field-based for structure fields. Since the `*` operator is so frequently used in C programs, full field-sensitivity seemed to be the analogous handling of Java fields. Details of FullFS appear in Appendix A.

Table 1 lists all points-to analysis algorithms used in our experiments. ExhaustiveFB and ExhaustiveFS are efficient implementations of exhaustive field-based and field-sensitive Andersen's analysis respectively, as provided by SPARK [26]; to the best of our knowledge, their speed is competitive with any published implementation of Andersen's analysis for Java.

All experiments are run on a machine with a Pentium 4 Xeon 2.4GHz processor and 2GB RAM, running Redhat Linux 9. We use the Java 1.4.2 JVM as the underlying VM for our experiments, but we analyze the 1.3.1 libraries, to be consistent with [26] and because Soot provides models for the native methods in the 1.3.1 libraries.

Benchmarks and Clients The characteristics of our benchmarks are presented in Table 2. We use the SPEC JVM98 benchmark suite, two benchmarks from the Ashes suite [1], `soot` and `sablecc`, and `jedit` [2], an open-source text editor. Subsets of these benchmarks were also utilized in previous Java pointer analysis studies [26, 28, 37, 44, 45].

The "# Methods" column reports the number of methods found reachable by SPARK's class-hierarchy analysis (these numbers differ from those in [26] due to improvements in the handling of reflection in Soot 2.2.1). "# Vars" is the number of variables (locals or static fields) in the program, and "#

Benchmark	# Methods	# Vars	# Stmts
soot	6089	51853	146292
compress	12244	95463	269289
jess	12878	101332	289514
raytrace	12378	96873	271980
db	12249	95665	270571
javac	13385	107753	318411
mpeg	12456	98458	276062
jack	12502	98579	278965
sablecc	14065	110292	352338
jedit	17510	144062	412835

Table 2: Information about our benchmarks.

Stmts" is the number of assignment statements (the number includes the temporary variables and assignments introduced to make each assignment one of our simple forms). Our largest benchmark `jedit` is comparable in size with the largest benchmarks used in other pointer analysis studies [7, 45].

We evaluate our analyses using two clients, *virtcall* and *localalias*. *virtcall* attempts to resolve virtual calls to a single target by finding the points-to set of the call's receiver. We only consider calls where cheaper type-based techniques cannot resolve the call. *localalias* attempts to disambiguate pairs of local variables in methods that are potentially involve in conflicting field reads / writes. For variables `x` and `y` and field `f`, we will query `x` and `y` if we see writes to both `x.f` and `y.f` or a write (read) of `x.f` and a read (write) of `y.f`. This information can be useful for a variety of optimizations, including eliminating redundant loads and dead stores [13].

For the SPEC benchmarks, we check *virtcall* and *localalias* queries for the *hot methods* of each benchmark, those methods that execute frequently at runtime. We found the hot methods by running the benchmarks through Jikes RVM [4], and observing which methods get recompiled with the optimizing compiler (at any optimization level) in its adaptive optimization system. This experiment reflects the queries likely to be raised by an optimizing JIT compiler. To simulate how inlining may affect our analysis results, we modified our graph to inline all getter (e.g. `Obj getFoo() { return this.foo; }`) and setter methods. This transformation essentially adds context-sensitivity for getter and setter methods, and possibly makes analysis more difficult for RegularPT, since there are more putfield and getfield statements and potentially more invalid match edges. We also run the *virtcall* client for all virtual calls in the program (including the Java libraries) with no inlining, to reflect an IDE client where a developer wishes to navigate to the invoked method for some virtual call.

For the *virtcall* client, a query is positively answered (see Section 4.2) when the points-to analysis shows that the call has 0 or 1 targets. A virtual call can have 0 targets if it resides in a method that is included in the initial call graph but that the points-to analysis can prove is dead. The *localalias* client actually raises two points-to queries, as it is checking if two variables can point to some common object. Together, these queries are positively answered if they show that the queried variables cannot be aliased. Handling the paired queries of *localalias* in RefinedRegularPT requires minor modifications to its refinement loop, which we elide.

Benchmark	Virt	FeasVirt
soot	2812	1051
compress	5428	1801
jess	5540	1861
raytrace	5438	1803
db	5450	1819
javac	6334	1952
mpeg	5451	1800
jack	6022	2370
sablecc	6101	1898
jedit	7480	2612

Table 3: Number of virtual calls unresolvable by types in each benchmark (the Virt column), and the number of such calls resolvable by field-sensitive Andersen's (the FeasVirt column).

When measuring the precision of our algorithms, we use field-sensitive Andersen's analysis as a "gold standard," *i.e.*, we measure how much of the precision of computing field-sensitive Andersen's can be obtained quickly by our demand-driven algorithms. In the rest of this section, we refer to the set of queries positively answered by field-sensitive Andersen's as the *feasible queries*, since these are the only queries that our demand algorithms can hope to positively answer. Table 3 shows the total number of virtual calls in our benchmark (excluding those resolvable with types alone), and the number of those that are feasible queries (virtual calls that field-sensitive Andersen's resolved). The exclusion of calls resolvable with types alone makes the field-sensitive Andersen's analysis look less precise than in previous work [26], since we exclude many easy queries. Also note that some of the calls unresolved by field-sensitive Andersen's actually have multiple targets at runtime; these calls are unresolvable by any analysis without an extra program transformation such as cloning [45].

Benchmark	Hot	Virt	FeasVirt	Alias	FeasAlias
compress	7	0	0	0	0
jess	28	9	3	5	0
raytrace	23	4	0	6	4
db	4	5	5	9	0
javac	95	115	30	68	10
mpeg	45	2	0	1	0
jack	22	12	9	3	0

Table 4: Information on *virtcall* and *localalias* queries in hot methods. Hot gives the number of hot methods. Virt gives the number of virtual calls in hot methods, and FeasVirt gives the number of those calls that can be resolved by field-sensitive Andersen's (the number of feasible queries). Alias and FeasAlias are analogous, but for *localalias* queries. We did not collect hot method information for the other three benchmarks because our experimental infrastructure did not support it.

Table 4 gives data on our queries in hot methods. Although the number of queries raised is small, their importance is potentially very high since they all occur in hot methods. For example, if an alias query is positively an-

Benchmark	Intra (Live)	Reg (Live)	RefReg (Live)
soot	18.4 (16.0)	94.1 (98.5)	96.9 (99.8)
compress	26.0 (23.1)	89.1 (96.4)	93.7 (98.9)
jess	25.4 (22.5)	89.4 (96.6)	93.9 (99.0)
raytrace	26.1 (23.1)	89.1 (96.4)	93.7 (98.9)
db	25.7 (22.7)	89.3 (96.5)	93.7 (98.9)
javac	25.3 (22.3)	89.9 (96.7)	94.1 (98.8)
mpeg	26.0 (23.1)	89.1 (96.4)	94.4 (98.9)
jack	27.0 (25.1)	91.8 (97.5)	95.2 (99.2)
sablecc	23.9 (20.6)	89.7 (96.3)	93.9 (98.8)
jedit	21.7 (19.0)	92.7 (99.1)	97.2 (99.9)

Table 5: RegularPT and RefinedRegularPT have nearly the precision of field-sensitive Andersen's. The table gives the percentage of *virtcall* queries positively answered by an intraprocedural field-based analysis (the Intra column), RegularPT (the Reg column), and RefinedRegularPT with a 5 second time limit per query (the RefReg column), as a percentage of those answered positively by field-sensitive Andersen's. The parenthesized Live numbers indicate the result if limited to queries in code that cannot be proven dead by the points-to analysis.

swered, it may allow for a load to be eliminated in frequently executing code, which could have a significant impact on performance.

5.2 Experimental Results

Precision Table 5 shows the results of measuring the precision of our algorithms for the *virtcall* client. The table shows the percentage of feasible *virtcall* queries that an intraprocedural field-based analysis, RegularPT (a field-based analysis), and RefinedRegularPT also positively answer. RefinedRegularPT can take very long time to answer some queries, so we time each query out at 5 seconds, well above the tolerable time budgets of our target clients. RegularPT positively answers more than 89% of feasible queries in all cases, and more than 96% if restricted to code that cannot be proven dead by the analysis (since it contains a virtual call with 0 targets). These results are consistent with previous work studying field-based analysis [26,28]. RefinedRegularPT can answer nearly all feasible queries. We show below that nearly all of the precision of RegularPT and RefinedRegularPT is preserved under early termination of queries. The purely intraprocedural field-based analysis does much worse than the RegularPT, showing that virtual calls that cannot be resolved with the type system usually cannot be resolved with a purely local analysis.

Figure 7 shows some code adapted from the `jedit` benchmark that illustrates why RegularPT has nearly the precision of field-sensitive Andersen's. Consider a query to resolve the call to `remove()` on the `propTable` variable in `setProperty()`; possible targets are in `Hashtable` or one of its subclasses. RegularPT handles the query by immediately traversing across the incoming match edge from the source of the putfield[properties] edge corresponding to the field write in the `Buffer` constructor. It then finds a new edge from a `Hashtable` abstract location and resolves the call to the implementation of `remove()` in the `Hashtable` class.

This pattern of a field being written only once in a constructor or other initialization method occurs frequently in Java programs, and RegularPT handles it well, as it immediately traverses to the write upon encountering any read of the field. In general, the precision of RegularPT for resolving virtual calls is less than that of a field-sensitive algorithm only when the algorithm encounters a field read `x = y.f` such that objects of multiple types are written into the f field (and such types lead to multiple targets for the call), and not all such objects can be written into y's f field. Such polymorphic uses of fields occur relatively rarely, and hence the precision of RegularPT for resolving virtual calls is close to that of field-sensitive Andersen's.

```
class Buffer {
  private Hashtable properties;

  public Buffer() {
    this.properties = new Hashtable();
  }

  public void setProperty(String name,
                          Object val) {
    Hashtable propTable = this.properties;
    propTable.remove(name);
    ...
    propTable.put(name,val);
  }
}
```

Figure 7: A typical example where RegularPT succeeds, but FullFS does too much work, derived from code in the `jedit` benchmark.

Precision under early termination We evaluated the precision of RegularPT, RefinedRegularPT, and FullFS under early termination with varying time budgets, to simulate the strict time constraints of IDEs and JIT compilers. Recall from Section 1 that under early termination, if a points-to analysis exceeds some time budget for answering a query for variable x, the analysis is terminated and the client is told that x can point to any location. To simplify implementation, instead of using actual timeouts to enforce budgets, we limit the number of nodes that can be traversed by a particular query. Note that we count a node as traversed each time it is removed from the worklists seen in Figures 5 and 6; RefinedRegularPT can visit a node multiple times, and each visit is counted against the traversal budget.

To study behavior under early termination, we computed the cumulative distribution of positively answered queries vs. node traversal budget for each of our clients, benchmarks, and algorithms. Figure 8 shows the cumulative distributions for `jedit`, using the *virtcall* client to query all virtual calls in the program and library; the distributions for other benchmarks look very similar. The vertical axis is the percentage of feasible queries that were positively answered, and the horizontal axis is the amount of allowed traversal. Recall from Table 5 that the maximum possible percentage that RegularPT can reach is 92.7%, and 97.2% for RefinedRegularPT. FullFS should reach 100% if allowed enough traversal.

RegularPT and RefinedRegularPT both perform very well under early termination, retaining most of their precision

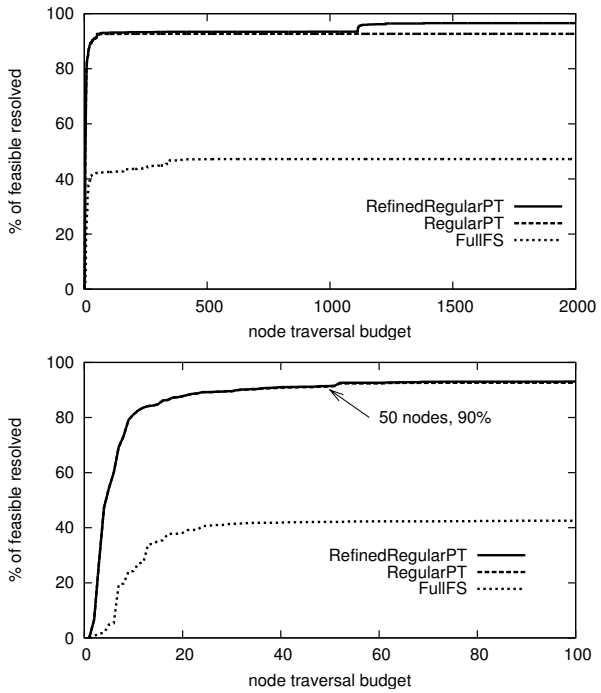


Figure 8: RegularPT and RefinedRegularPT perform very well under early termination. We give cumulative distribution of percentage of feasible queries positively answered vs. node traversal budget for the *virtcall* client on `jedit`, for all three algorithms. The top graph shows the distribution from 0 to 2000 nodes traversed, while the bottom graph focuses on 0 to 100 nodes traversed. The distributions for other benchmarks look very similar.

Budget	Benchmark	Reg	RefReg	FullFS
50 nodes	`soot`	93.7	93.7	46.6
	`compress`	88.7	89.7	41.6
	`jess`	89.1	89.9	41.0
	`raytrace`	88.8	89.6	41.8
	`db`	88.9	89.7	42.3
	`javac`	88.9	89.3	42.5
	`mpeg`	88.8	89.0	41.7
	`jack`	91.5	92.2	44.2
	`sablecc`	89.4	89.6	41.0
	`jedit`	91.4	91.6	42.1
1250 nodes	`soot`	94.1	95.9	55.7
	`compress`	89.1	92.9	50.3
	`jess`	89.4	93.2	49.3
	`raytrace`	89.1	93.0	50.5
	`db`	89.3	93.0	50.9
	`javac`	89.9	93.3	51.9
	`mpeg`	89.1	92.9	50.3
	`jack`	91.8	94.6	59.3
	`sablecc`	89.7	92.1	49.9
	`jedit`	92.7	96.4	47.2

Table 6: Precision of the demand-driven algorithms with traversal budgets of 50 nodes and 1250 nodes. The columns give the percentage of feasible *virtcall* queries positively answered by RegularPT (the Reg column), RefinedRegularPT (the RefReg column), and FullFS (the FullFS column).

even under tight time budgets. Figure 8 shows that even with a 50 node traversal limit (where queries take 2ms or less), both algorithms positively answer more than 90% of feasible queries. RegularPT and RefinedRegularPT behave similarly since the first iteration of RefinedRegularPT is exactly RegularPT, and it therefore positively answers all queries that RegularPT does with the same amount of traversal. RegularPT reaches its maximum percentage of 92.7% with 182 nodes of traversal; for `jedit`, traversing more than this amount is pointless for RegularPT. A similar result is seen across benchmarks, with the largest amount of traversal required to get all positive answers with RegularPT being 259 nodes for `javac`. Traversing 250 nodes takes under 5ms with our untuned implementation.

RefinedRegularPT resolves many of the queries that RegularPT cannot as the traversal limits grow, reaching 96.5% of feasible queries with a traversal budget of 1250 nodes. Traversing 1250 nodes takes 20ms or less with our implementation. ExhaustiveFS (described in Table 1) takes almost 30 seconds to analyze `jedit`, in which time RefinedRegularPT could answer 1500 queries with a 1250 node budget. Therefore, RefinedRegularPT could be very useful in an application that required more precision than RegularPT and only needed pointer information for a subset of the program, perhaps constructing a call graph for part of the libraries to aid in program understanding.

FullFS does not perform well under early termination. The plateau for FullFS is reached at a 522 node limit, and at this point it only positively answers 47.2% of feasible queries. The algorithm requires more than 30000 nodes to positively answer any more queries, and many queries require several hundred thousand nodes of traversal, taking more than 10 seconds of analysis time (sometimes longer than the time required to run ExhaustiveFS).

Table 6 shows the precision of RegularPT, RefinedRegularPT, and FullFS for the *virtcall* client on all our benchmarks, with traversal budgets of 50 nodes (2ms per query) and 1250 nodes (20ms per query). With a 50 node traversal budget, the precision of RegularPT and RefinedRegularPT are almost identical, 88.8-93.7% of field-sensitive Andersen's. A traversal budget of 1250 nodes allows RefinedRegularPT to positively answer 1.8-3.8% more queries than RegularPT, relative to field-sensitive Andersen's. FullFS cannot answer more than 59.3% of feasible queries with a 1250 node traversal budget, and as with `jedit`, a much larger traversal budget (30000 nodes or more) is required on all benchmarks to substantially improve this precision, well beyond the constraints of our target environments.

Tables 7 and 8 give results for *virtcall* and *localalias* queries in hot methods; benchmarks with 0 queries are not listed. For this experiment, we ran RegularPT with a traversal budget of 250 nodes, and FullFS with a traversal budget of 500 nodes. We gave FullFS a larger budget since in our implementation it seems to process nodes faster, and with these budgets the time allowed for each query is roughly even for the two algorithms (about 5ms per query). RegularPT positively answers all feasible queries in the hot methods; there is no need to show RefinedRegularPT since its results are exactly the same. FullFS does well on some benchmarks, but

Benchmark	FeasVirt	Reg	FullFS
jess	3	3	2
db	5	5	5
javac	30	30	15
jack	9	9	9

Table 7: Results for *virtcall* queries in hot methods, showing that RegularPT positively answers the same number of queries as field-sensitive Andersen's. The FeasVirt column gives the number of feasible queries (repeated from Table 4), the Reg column the number resolved by RegularPT with a 250 node traversal budget, and the FullFS column the number resolved by FullFS with a 500 node traversal budget.

Benchmark	FeasAlias	Reg	FullFS
raytrace	4	4	4
javac	10	10	1

Table 8: Results for *localalias* queries in hot methods, showing RegularPT matching field-sensitive Andersen's. FeasAlias gives the number of queries resolved by field-sensitive Andersen's (repeated from Table 4). Reg gives the number resolved by RegularPT, and FullFS the number resolved by FullFS, with the same traversal budgets used for Table 7.

even with the larger traversal budget, it cannot positively answer many queries in javac, the largest of the benchmarks and the one with the most queries. RegularPT traverses 108 nodes or less for all positively answered queries, and therefore could have been run with a smaller traversal budget with no precision penalty.

Our results lead to the slightly counter-intuitive conclusion that within tight time budgets, greater precision can be obtained with an overall less precise algorithm. Consider again the example of Figure 7. Answering the *virtcall* query on the propTable.remove() call with full field-sensitivity, as FullFS does, leads to extra work, since the object written to the properties field in the constructor of Buffer will certainly flow to any read of the field, and hence all match edges involving properties cannot be removed by refinement. Furthermore, this extra work can increase costs substantially, as RegularPT only requires traversing 3 nodes to answer this query. The example reflects a common case, and illustrates that the greater precision of our algorithms is due to both the relatively small difference in precision between field-based and field-sensitive analysis *and* to the fact that RegularPT often requires very little traversal to answer a query.

We inspected several of the feasible virtual call queries that RefinedRegularPT could not quickly answer by hand, and found that the reasons for their difficulty were independent of field-sensitivity. Some queries involved a parameter of a function nested deeply in the libraries that gets called from many places; avoiding long traversals in these cases would be difficult, and the likelihood of resolving such calls to a single target is lower. In other cases, imprecision in the conservative call graph lead to traversals of excess methods; refining this call graph on-the-fly is possible, but it's unclear if the extra work involved would be worth-

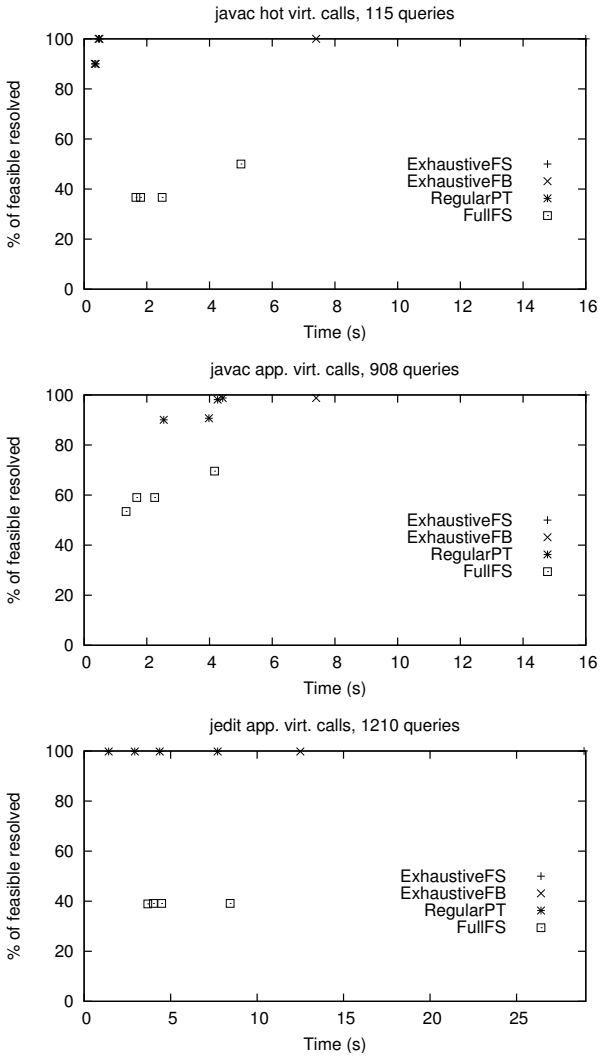


Figure 9: Complete time/precision comparison of several demand algorithm configurations and exhaustive algorithms on *virtcall* queries. The data show that RegularPT can get close to the same results as field-sensitive Andersen's with much less cost than running an exhaustive analysis. The x axis is analysis time in seconds, and the y axis is the percentage of feasible queries that were positively answered. For RegularPT and FullFS the data points left to right are for traversal budgets of 50 nodes, 100 nodes, 200 nodes, and 500 nodes. Times are also give for ExhaustiveFB and ExhaustiveFS (described in Table 1). From top to bottom, the graphs show virtual calls in hot methods in javac, all virtual calls in javac application code, and all virtual calls in jedit application code. These graphs are representative; other graphs are very similar.

while with strict time constraints. Context-insensitivity also causes excessive traversal; when the traversal enters a call to `ArrayList.get()`, for example, it exits at all of the many call sites of the method. It may also be possible to restrict traversal using context-sensitivity, but again, this will involve extra work that may not be very beneficial with tight time budgets.

Performance Since we intend to run our demand algorithms with early termination, time performance can be adjusted depending on how much precision is necessary for the client. Figure 9 shows that high precision *and* performance can be obtained with RegularPT through aggressive early termination. We ran experiments where we measured the total time required for RegularPT and FullFS to process all *virtcall* queries in hot methods and all *virtcall* queries in application code (*i.e.*, excluding the Java libraries). The latter experiment simulates some potential program understanding functionality in an IDE where a call graph is built for the application. We compared these total running times to the time required to run ExhaustiveFB and ExhaustiveFS, described in Table 1. We show graphs of running time vs. percentage of feasible queries resolved for three representative benchmarks and clients: calls in hot methods in `javac`, calls in application code in `javac`, and calls in application code in `jedit`. In all cases, the precision of a field-based analysis was nearly exactly that of a field-sensitive analysis, so we excluded RefinedRegularPT from the graphs (its performance is almost identical to that of RegularPT).

RegularPT gives significant speedups over ExhaustiveFB in all cases without sacrificing precision. For `javac` virtual calls in hot methods, RegularPT has the same precision as a field-sensitive analysis with a running time of .46 seconds, a 34x speedup over the 16 seconds for ExhaustiveFB and a 16x speedup over the 7.4 seconds for ExhaustiveFS. For virtual calls in application code, speedups over ExhaustiveFS (ExhaustiveFB) range from 3.62x (1.68x) for `javac` (the smallest speedup in our benchmarks) to 20.4x (8.8x) for `jedit`. FullFS fails to provide nearly the same precision within the same time budgets given to RegularPT.

The memory consumption of our algorithms is quite reasonable. Given a budget of 250 nodes of traversal, neither RegularPT nor RefinedRegularPT allocates more than 50 kilobytes of memory across our benchmarks, and our implementation could be more memory efficient. In contrast, even with an efficient BDD representation, exhaustive field-sensitive Andersen's analysis takes 23 MB for `javac` and 28 MB for `jedit` [7], since all points-to sets need to be represented.

Other Factors We have evaluated our algorithms in a static environment, where all of the benchmark code is available and the graph representation of pointer assignments is built up-front. In a JIT compiler or IDE, such representations may not be readily available. Since our graph representation essentially matches the assignment statements in the program, it can be constructed efficiently. In an IDE, the representation for a method can simply be rebuilt from scratch after its code changes; no complex incremental update is required. Such rebuilding can occur in the background while the user continues working. In a JIT compiler, the graph can be constructed immediately from an intermediate representation. If a method has no intermediate representation because it has only been interpreted, its graph can be constructed at query time, and the cost of building the graph can be factored into early termination heuristics.

Because new code may become available after analysis on our environments (through editing in an IDE or dynamic class loading in a JIT compiler), analysis results may become invalid. If any analysis results are cached in an IDE, they can simply be flushed and the corresponding queries re-run. A JIT compiler presents a greater challenge, since some (possibly running) code may have already been optimized based on previous analysis results. There are three ways in which dynamic loading of class C can invalidate the results of some query q:

1. C provides a new target for some method invocation that was traversed in answering q. This could affect analysis results by returning some new value that was not previously possible, for example.
2. C calls some existing method m whose parameters were traversed in answering q (since new values could now be passed into m).
3. C has a putfield to a field f, and a getfield on f was encountered when answering q. This new putfield could lead to new match edges that must be considered.

Since our algorithms traverse a representation close to the statements of the program, they could potentially keep track of which statements could be affected by dynamic class loading, and then add appropriate guards to such statements. If a guard later failed due to dynamic class loading, the optimized code could be invalidated, using on-stack replacement [8, 14] if necessary. See [20] for an enumeration of the issues related to running pointer analysis in a JIT compiler.

6. RELATED WORK

Points-to analysis has been an active area of research for many years. We limit our discussion to work that shares one or more of the key features of our work: use of CFL-reachability, demand-driven algorithms, and analysis of Java. See [19] for more on past work in points-to analysis and [38] for a discussion of various analyses for object-oriented programs.

Our use of CFL-reachability is based on the work of Reps et al. on developing and utilizing the CFL-reachability framework [33–35]. Our key insight was to recognize that Andersen's analysis for Java is a balanced-parentheses problem when expressed in terms of CFL-reachability, a structure we exploit in both RegularPT and RefinedRegularPT. Our match edges are related to the summary edges used by the efficient CFL-reachability algorithm for balanced parentheses languages [32,34,35]. Summary edges are computed bottom-up as paths between parentheses are found, while match edges are added exhaustively and then refined by checking for paths between parentheses.

CFL-reachability formulations lead directly to demand-driven algorithms through the use of the magic-sets transformation [32]. The inference rules of FullFS (presented in Appendix A), an adaptation of the demand-driven algorithm of Heintze et al. [17], correspond exactly to a magic-sets transformation of the grammar in Figure 3, rewritten to find *pointsTo*-paths. As shown in Section 5, FullFS does not have our desired performance characteristics under early termination.

Much work has been done on whole-program points-to analysis for Java. A variety of points-to analyses are implemented and explored in the SPARK framework [26], based on Soot [42]. Their incremental worklist algorithm achieves excellent scalability for Andersen's analysis on large Java programs, partially due to an efficient points-to set data structure and exploitation of declared type information in the program. Recently, BDDs have been employed to reduce the space required for storing points-to sets [7], and a higher-level language was designed to specify BDD-based analyses more concisely [27]. Their field-based program representation is similar to our graph with conservative match edges; instead of a match edge, they create a node for each field, and represent getfields and putfields to assignments from and to the field node. This representation works well for exhaustive propagation of points-to sets, but the match edge representation is more suitable for our refinement techniques.

Some recent work makes promising advances in performing incremental points-to analysis. Kodumal and Aiken show how to perform a limited form of incremental analysis in a set constraints solver using backtracking [24]. Their technique is most effective in cases where code changes are primarily limited to a small set of source files, which they show is a typical development pattern. Hirzel et al. present a points-to analysis implemented in a JIT compiler that handles all Java language features, which can quickly update its computed results after program changes [21]. Our approach to handling code changes is to recompute from scratch points-to queries that are affected by a program change. In cases where the number of queried variables is moderate, our approach has the advantages of simplicity, as no engineering for incrementality is needed, and of not needing to cache intermediate analysis results, which can consume significant amounts of memory,

Choi et al. [9] and Whaley and Rinard [46] define flow and context-sensitive points-to and escape analyses for Java. Vivien and Rinard extend the analysis in [46] to be incremental [43], focusing analysis effort on code deemed to be likely to yield profitable results. Their results show that their incremental analysis obtains most of the effect of an exhaustive analysis in much less time. Our positive results for early termination may be explained by similar underlying principles. It is unclear how long their analysis takes to answer individual queries.

Liang et al. experimented with a variety of points-to analysis algorithms for Java [28]. They conclude that field-sensitivity yields little benefit over field-based analysis for the extra required effort. Our precision results in the demand-driven setting support this conclusion. Annotated set constraints are utilized by Rountev et. al. to perform field-sensitive Andersen's analysis for Java [37], and their analysis seems to scale well. Recently, Kodumal and Aiken have shown an efficient translation from balanced-parentheses CFL-reachability problems to set constraints [23]. Their translation could be applied to our formulation of Andersen's analysis to give an alternate set-constraints based implementation. Whaley and Lam [44] adapt the fast points-to analysis algorithm of Heintze and Tardieu [18] for Java, also with good scalability results. Their work adds field-sensitivity and flow-sensitivity for local variables. Recently, they have utilized BDDs to produce a scalable flow and context-sensitive exhaustive analysis [45]. BDDs were also used for exhaustive context-sensitive points-to analysis in [49]. Our focus on demand-driven algorithms distinguish the current work from previous efforts.

Analyses other than Andersen's have been employed to perform points-to analysis and/or call graph construction for Java programs. A variety of cheaper analyses for computing call graphs have been proposed, based on analyzing the class hierarchy [11], reachable code [6], and further efficiently analyzable information [40, 41]. O'Callahan presents a context-sensitive, field-sensitive, unification-based points-to analysis based on type inference techniques [31]. Milanova et al. propose object-sensitivity as an alternative to context-sensitivity for more precisely analyzing features of object-oriented programs [30]. We plan to explore greater context and flow-sensitivity in our analyses in the near future to help both precision and performance.

A large body of past work aimed to improve the performance of alias analysis for C programs. Cycle elimination [12] and projection merging [39] dramatically improved the scalability of set-constraint solvers in doing Andersen's analysis; the same set-constraint solver [3] was utilized in the Java analysis in [37]. Das adds a small amount of directionality to a unification-based analysis in [10], creating an analysis with precision close to Andersen's while remaining highly scalable; it is not immediately clear how to adapt such a technique to Java. Heintze and Tardieu present a highly scalable implementation of Andersen's analysis in [18], based on online cycle elimination, an efficient program representation, and an efficient set data structure. This algorithm was also successfully applied to Java programs [44].

Guyer and Lin [16] present a client-driven alias analysis that detects which statements cause imprecision for a given client, and then analyzes that part of the code with greater flow- and context-sensitivity; their results are promising. We could perhaps adapt some of their techniques to add greater sensitivity to our algorithms.

7. CONCLUSIONS

We have developed novel demand-driven points-to analysis algorithms that yields much higher precision than previous techniques within small time budgets. The RegularPT algorithm is very simple to implement and achieves nearly 90% of the precision of field-sensitive Andersen's analysis within 2ms per query. Given a slightly larger time budget, RefinedRegularPT improves on the precision of RegularPT and provides much more precision than a fully field-sensitive approach. Our algorithms are especially suitable for JIT compilers and IDEs, which have extreme resource constraints and must handle changes in the analyzed code. However, the algorithms are generally suitable for any client that that only requires points-to information for a subset of program variables. Given the relative ease of implementing the algorithms, they provide a compelling alternative to engineering an exhaustive points-to analysis for such clients.

Acknowledgements

This work is supported in part by the National Science Foundation, with grants CCF-0085949, CCR-0105721, CCR-0243657, CNS-0225610, CCR-0326577, an award from University of California MICRO program, the Okawa Research Award, donations from IBM and Intel, and a National Defense Science and Engineering Graduate Fellowship. This

work has also been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCHC020056. The views expressed herein are not necessarily those of DARPA or IBM.

We thank Bill McCloskey, Simon Goldsmith, John Kodumal, V. C. Sreedhar, and Bill Thies for carefully reading and commenting on drafts of this paper. We also thank the anonymous reviewers for their helpful comments.

8. REFERENCES

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

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

[3] A. Aiken, M. Fändrich, J. S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Workshop on Types in Compilation, Kyoto, Japan*, March 1998.

[4] B. Alpern, D. Attanasio, J. Barton, M. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, T. Ngo, M. Mergen, V. Sarkar, M. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno virtual machine. *IBM Systems Journal, Java Performance Issue*, 39(1), 2000.

[5] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[6] D. Bacon and P. Sweeney. Fast static analysis of c++ virtual function calls. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), San Jose, CA*, October 1996.

[7] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Diego, CA*, June 2003.

[8] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 26, pages 1–15, New York, NY, 1991. ACM Press.

[9] J.-D. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Denver, Colorado*, November 1999.

[10] M. Das. Unification-based pointer analysis with directional assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, British Columbia, Canada*, June 2000.

[11] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, August 1995.

[12] M. Fändrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Conference on Programming Language Design and Implementation (PLDI)*, June 1998.

[13] S. Fink, K. Knobe, and V. Sarkar. Unified analysis of array and object reference in strongly typed languages. In *Proceedings of the 2000 Static Analysis Symposium*, June 2000.

[14] S. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement, 2003.

[15] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[16] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *International Static Analysis Symposium (SAS), San Diego, CA*, June 2003.

[17] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah*, June 2001.

[18] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah*, June 2001.

[19] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE), Snowbird, Utah*, June 2001.

[20] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the presence of dynamic class loading. In *European Conference on Object-Oriented Programming (ECOOP)*, June 2004.

[21] M. Hirzel, D. von Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. Technical report, IBM Research RC23638, June 2005.

[22] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *SIGSOFT'95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995.

[23] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2004.

[24] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS '05: Proceedings of the 12th International Static Analysis Symposium*. London, United Kingdom, September 2005.

[25] O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[26] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction (CC), Warsaw, Poland*, April 2003.

[27] O. Lhoták and L. Hendren. Jedd: a BDD-based relational extension of Java. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, 2004.

[28] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE), Snowbird, Utah*, June 2001.

[29] D. Liang, M. Pennings, and M. J. Harrold. Evaluating the precision of static reference analysis using profiling. In *Internal Symposium of Software Testing and Analysis (ISSTA), Rome, Italy*, July 2002.

[30] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *International Symposium on Software Testing and Analysis (ISSTA), Rome, Italy*, July 2002.

[31] R. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie Mellon University, November 2000.

[32] T. Reps. Solving demand versions of interprocedural analysis problems. In *International Conference on Compiler Construction (CC), Edinburgh, Scotland*, April 1994.

[33] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, November/December 1998.

[34] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, CA*, pages 49–61, January 1995.

[35] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), New Orleans, LA*, December 1994.

[36] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, British Columbia, Canada*, June 2000.

[37] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Tampa Bay, Florida*, October 2001.

[38] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction (CC), Warsaw, Poland*, April 2003.

[39] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *ACM Symposium on Principles of Programming Languages (POPL), Boston, Massachusetts*, pages 81–95, January 2000.

[40] V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Minneapolis, MN*, October 2000.

[41] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Minneapolis, MN*, October 2000.

[42] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[43] F. Vivien and M. C. Rinard. Incrementalized pointer and escape analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah*, June 2001.

[44] J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *International Static Analysis Symposium (SAS), Madrid, Spain*, September 2002.

[45] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM Press, June 2004.

[46] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Denver, Colorado*, November 1999.

[47] M. Yannakakis. Graph-theoretic methods in database theory. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, April 1990.

[48] S. H. Yong, S. Horwitz, and T. W. Reps. Pointer analysis for programs with structures and casting. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 91–103, 1999.

[49] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, 2004.

APPENDIX

A. FULLFS DETAILS

Here we give the details of FullFS, an algorithm employing the techniques used by the demand-driven points-to analysis algorithm of Heintze et al. [17]. We present inference rules for FullFS in Figure 10, showing which points-to queries must be raised, given an initial set of queries and the statements in a particular program. The left column presents the statement types relevant to Java points-to analysis, and the right column gives the inference rules that can be instantiated when the corresponding statement is present. We adopt the notation of [17], where $x \hookrightarrow .$ means that a query has been raised to find what x points to. So, rule (1) states that if there is a points-to query for p and a statement `p = new T()`, then add $p \hookrightarrow o_a$ to the points-to relation (`a` is some label for the statement). Rule (3) states that given statement `p = r` and a query $p \hookrightarrow .$, we must add the query $r \hookrightarrow .$ to see what r points to. Once we discover $r \hookrightarrow o_1$, rule (4) will add $p \hookrightarrow o_1$.

Handling getfield and putfield statements is more complex. Given a getfield statement `p = r.f` and a query $p \hookrightarrow .$, we must find the values written into the f field of abstract locations that r can point to. Rule (6) introduces the query $r \hookrightarrow .$ to find what r points to. Given that $r \hookrightarrow o_1$, introduces the pointed-to-by query $. \hookrightarrow o_1(f_w)$ to find what variables point to o_1. In [17], there are queries of the form $. \hookrightarrow x$ for variables x, since there can be pointers to variables in C through the `&` operator. In Java, we only have such pointed-to-by queries for allocation sites. The parenthesized f_w gives the *reason* for this query; the form is a

statement	FullFS inference rules
`a: p = new T`	$\dfrac{p \hookrightarrow .}{p \hookrightarrow o_a}$ (1) $\dfrac{. \hookrightarrow o_a(any)}{p \hookrightarrow o_a}$ (2)
`p = r`	$\dfrac{p \hookrightarrow .}{r \hookrightarrow .}$ (3) $\dfrac{p \hookrightarrow . \qquad r \hookrightarrow o_1}{p \hookrightarrow o_1}$ (4) $\dfrac{r \hookrightarrow o_1 \qquad . \hookrightarrow o_1(any)}{p \hookrightarrow o_1}$ (5)
`p = r.f`	$\dfrac{p \hookrightarrow .}{r \hookrightarrow .}$ (6) $\dfrac{p \hookrightarrow . \qquad r \hookrightarrow o_1}{. \hookrightarrow o_1(f_w)}$ (7) $\dfrac{p \hookrightarrow . \qquad r \hookrightarrow o_1 \quad o_1.f \hookrightarrow o_2}{p \hookrightarrow o_2}$ (8) $\dfrac{r \hookrightarrow o_1 \qquad . \hookrightarrow o_1(f_r) \quad o_1.f \hookrightarrow o_2 \qquad . \hookrightarrow o_2(any)}{p \hookrightarrow o_2}$ (9)
`p.f = r`	$\dfrac{p \hookrightarrow o_1 \qquad . \hookrightarrow o_1(f_w)}{r \hookrightarrow .}$ (10) $\dfrac{p \hookrightarrow o_1 \qquad . \hookrightarrow o_1(f_w) \quad r \hookrightarrow o_2}{o_1.f \hookrightarrow o_2}$ (11) $\dfrac{r \hookrightarrow o_2 \qquad . \hookrightarrow o_2(any)}{p \hookrightarrow .}$ (12) $\dfrac{r \hookrightarrow o_2 \qquad . \hookrightarrow o_2(any) \quad p \hookrightarrow o_1}{o_1.f \hookrightarrow o_2 \qquad . \hookrightarrow o_1(f_r)}$ (13)

Figure 10: Inference rules for FullFS.

field name f with subscript w for write and r for read. Here, we are looking for writes to field f, or f_w, since the getfield statement reads from field f. We keep these reasons to avoid unnecessary work (see below). Rule (8) says that once the analysis has discovered that $r \hookrightarrow o_1$ and $o_1.f \hookrightarrow o_2$, we can add the fact $p \hookrightarrow o_2$.

Putfield statements `p.f = r` are handled as follows. If we have $p \hookrightarrow o_1$ and $. \hookrightarrow o_1(f_w)$, the statement is relevant since we must discover what is written into the f field of o_1. In such a case, rule (10) introduces the query $r \hookrightarrow .$, and once we find $r \hookrightarrow o_2$, rule (11) adds $o_1.f \hookrightarrow o_2$. This latest points-to fact allows rule (8) to trigger for the appropriate getfield statements, flowing o_2 across the field. Notice that if we did not have the reason f_w and just had pointed-to-by queries of the form $. \hookrightarrow o_1$, then when we encounter a statement `p.g = r`, we would still introduce $r \hookrightarrow .$ to be sound, since a read from the g field may have been encountered.

```
1: x = y.f;
2: w = new T();
3: y = w;
4: z = w;
5: z.f = p;
6: p = new T();
```

$$\dfrac{\dfrac{x \hookrightarrow .}{x \hookrightarrow .}\,1 \qquad \dfrac{\dfrac{\dfrac{\dfrac{x \hookrightarrow .}{y \hookrightarrow .}\,1 \qquad \dfrac{y \hookrightarrow .}{w \hookrightarrow .}\,3}{y \hookrightarrow o_2}\,3 \qquad \dfrac{w \hookrightarrow o_2}{x \hookrightarrow .}\,2}{x \hookrightarrow o_6} \qquad \dfrac{\vdots}{o_2.f \hookrightarrow o_6}}{x \hookrightarrow o_6}\,1$$

Figure 11: Example code and partial derivation for FullFS.

Keeping the reason ensures we only do work for relevant field writes.

Rules (2), (5), (9), (12), and (13) concern properly handling the pointed-to-by queries. Rule (2) says that if there is a pointed-to-by query $. \hookrightarrow o_a(any)$ (where (any) means for any reason), we must add $p \hookrightarrow o_a$; rule (5) similarly handles assignments. Rules (9), (12), and (13) handle the case when we have $. \hookrightarrow o_2(any)$ and o_2 is written into a field. For statement `p.f = r`, rule (12) adds the query $p \hookrightarrow .$, since we must which abstract locations have o_2 written into their f field. When we find such an abstract location o_1, rule (13) adds the fact $o_1.f \hookrightarrow o_2$, and introduces the pointed-to-by query $. \hookrightarrow o_1(f_r)$ since we are looking for reads of f. Rule (9) handles the case when we find a statement `p = r.f` that reads f from o_1, adding the fact $p \hookrightarrow o_2$.

Figure 11 gives some sample code and partial derivation of the fact $x \hookrightarrow o_6$, assuming $x \hookrightarrow .$ is the initial query. Uses of inference rules are labelled with the corresponding statement. Notice that all leaves of the derivation will be $x \hookrightarrow .$, as expected.

FullFS runs the two-set algorithm described in [17], adapted for these inference rules. We maintain a points-to set for each queried variable, and a second set containing the allocation sites in the points-to set which are referenced in pointed-to-by queries. The algorithm iterates over all relevant statements applying inference rules until the sets reach a fixed point. We re-use SPARK's hybrid points-to set representation [26], use a smart iteration order for the statements, and incrementalize the algorithm, only propagating newly added abstract locations in each iteration.