Pointer Analysis in the Presence of Dynamic Class Loading

Martin Hirzel
University of Colorado
Boulder, CO 80309
hirzel@cs.colorado.edu

Amer Diwan
University of Colorado
Boulder, CO 80309
diwan@cs.colorado.edu

Michael Hind
IBM Watson Research Center
Hawthorne, NY 10532
hind@watson.ibm.com

ABSTRACT

Many compiler optimizations and software engineering tools need precise pointer analyses to be effective. Unfortunately, many Java features, such as dynamic class loading, reflection, and native methods, make pointer analyses difficult to develop. Hence, prior pointer analyses for Java either ignore those features or are overly conservative. We describe and evaluate a pointer analysis that deals with all Java language features.

Our analysis is based on Andersen's inclusion-constraint based pointer analysis. For statically-linked languages, Andersen's analysis finds constraints based on a static whole-program analysis of the code, and then propagates the constraints to obtain points-to information. Our version of Andersen's analysis handles features such as dynamic class loading by generating constraints at runtime and propagating the constraints whenever a client of the analysis needs points-to information. We evaluate the correctness of our analysis by comparing its results with pointers created at runtime. We evaluate its performance by running it on a number of Java programs and measuring metrics such as propagation time.

1. INTRODUCTION

Pointer analyses benefit many optimizations, such as virtual method resolution, and software engineering tools, such as program slicers. Despite its benefits, we are not aware of a pointer analysis that works for all of Java. Prior work on pointer analysis for Java [46, 40, 57, 39] handles only a subset of the language, ignoring features such as dynamic class loading, native methods, or reflection. These features create challenges for program analyses because: (i) a program analysis cannot assume that it has seen the whole program: new code may be loaded at any time during program execution (dynamic class loading); [1] and (ii) some code may not be easily amenable to analysis because it is written in a low-level language (native methods called via JNI, the Java Native Interface) or circumvents the type system (reflection). This paper describes, to our knowledge, the first pointer analysis that works for all of Java. Our analysis is based on Andersen's pointer analysis [5] and handles the complete Java language, including dynamic class loading, reflection, and native methods.

Andersen's analysis [5] has two parts: constraint generation and constraint propagation. Our algorithm separates these two parts: (i) When certain events occur (e.g., executing a method for the first time) our analysis generates new constraints; (ii) When a client (e.g., an optimization) needs points-to information, it propagates constraints to arrive at the points-to sets. These points-to sets are conservative until another event generates new constraints. Each constraint propagation uses the previous solution as a starting point, which is cheaper than doing a full propagation.

We have implemented our analysis in the context of Jikes RVM, an open source research virtual machine from IBM [4]. Because Jikes RVM is itself written in Java, any pointer analysis in this system must consider not just the application code, but also Jikes RVM's own code for the JIT compilers, garbage collector, and other runtime services. Our analysis handles all the intricacies of Java and Jikes RVM.

Since pointer analyses are difficult to write and debug, we carefully validated the results of our analysis. Each garbage collection uses information about actual pointers in memory to check the points-to sets generated by our analysis. A pointer that exists between actual objects, but is not represented in the points-to sets, indicates a bug in our analysis. This validation methodology was essential in reaching a high level of confidence in the correctness of our pointer analysis. We evaluated the performance of our analysis using a number of Java programs, including the SPECjvm98 benchmark suite. We found that our analysis yields propagation times of a few seconds, making it practical, particularly for long running applications.

2. MOTIVATION

Static whole-program analyses are not applicable to the full Java language because of dynamic features, such as class loading, reflection, and native methods. It is hard or even impossible to determine the impact of these runtime features statically. This section illustrates the difficulties that class loading poses for static whole-program analyses.

[1] This challenge is not unique to Java. Microsoft's CLI (Common Language Infrastructure) also has this feature, and DLLs and shared libraries pose similar problems.

University of Colorado at Boulder
Technical Report CU-CS-966-03
This work is suported by NSF ITR grant CCR-0085792, NSF Career award CCR-0133457, an IBM faculty partnership award, and an IBM graduate student fellowship. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.
December 10, 2003.

It is not known statically where *a class will be loaded from.*

Java allows user-defined class loaders, which may have their own rules for where to look for the bytecode, or even generate it on the fly. A static analysis cannot analyze those classes. User-defined class loaders are widely used in production-strength commercial applications, such as Eclipse and Tomcat.

It is not known statically which *class will be loaded.*

Even an analysis that restricts itself to the subset of Java without *user-defined* class loaders cannot be fully static, because code may still load statically unknown classes with the *system* class loader. This is done by `Class.forName(String name)`, where `name` can be computed at runtime. For example, a program may compute the calendar class name by reading an environment variable that specifies its location.

One approach to dealing with this issue would be to assume that all calendar classes may be loaded. This would result in a less precise solution, if, for example, at each customer's site, only one calendar class is loaded. Even worse, the relevant classes may be available on the executing machine, but not in the development environment. Only an online analysis could analyze such a program.

It is not known statically when *a given class will be loaded.*

If the classes to be analyzed are only available in the execution environment, one could imagine avoiding *static* analysis by attempting a whole-program analysis during JVM *start-up*, long before the analyzed classes will be needed. The Java specification says it should appear to the user as if class loading is lazy, but a JVM could just pretend to be lazy by showing only the effects of lazy loading, while actually being eager. This is difficult to engineer in practice, however. One would need a deferral mechanism for various visible effects of class loading. An example for such a visible effect would be a static field initialization of the form

```
static HashMap hashMap =
    new HashMap(Constants.CAPACITY);
```

Suppose that for some reason, Constants.CAPACITY has an illegal value, such as -1. The effect (IllegalArgumentException) should only become visible to the user when the class containing the static field is loaded. Furthermore, the class should be loaded after the class Constants, to ensure that CAPACITY already has the right value. It is difficult to be eager about class loading while pretending to be lazy and getting the order of loaded classes right.

It is not known statically whether *a given class will be loaded.*

Even if one ignores the order of class loading, and handles only a subset of Java without *explicit* class loading, *implicit* class loading still poses problems for static analyses. A JVM implicitly loads a class the first time it executes code that refers to it, for example, by creating an instance of the class. Whether a program will load a given class is undecidable, as Figure 1 illustrates.

A run of "`java Main`" does not load class C; a run of "`java Main anArgument`" loads class C, because line 5 creates an instance of C. We can observe this by whether line 10 in the static initializer prints its message. In this example, a static analysis would have to conservatively assume that class C will be loaded, and to analyze it. In general, a static whole-program analysis would have to analyze many more classes than necessary, making it inefficient (analyzing more classes costs time and space) and less precise (the code in those classes may exhibit behavior never encountered at runtime).

```
 1: class Main {
 2:     public static void main(String[] argv) {
 3:         C v = null;
 4:         if (argv.length > 0)
 5:             v = new C();
 6:     }
 7: }
 8: class C {
 9:     static {
10:         System.out.println("loaded class C");
11:     }
12: }
```

Figure 1: Class loading example.

3. RELATED WORK

This paper shows how to enhance Andersen's pointer analysis to analyze the full Java programming language. Section 3.1 puts Andersen's pointer analysis in context. Section 3.2 discusses related work on online, interprocedural analyses. Section 3.3 discusses related work on using Andersen's analysis for Java. Finally, Section 3.4 discusses work related to our validation methodology.

3.1 Static Pointer Analyses

The body of literature on pointer analyses is vast [29]. At one extreme, exemplified by Steensgaard [52] and type-based analyses [18], the analyses are fast, but imprecise. At the other extreme, exemplified by the work on shape analysis [28, 48], the analyses are slow, but precise enough to discover the shapes of many data structures. In between these two extremes there are many pointer analyses, offering different cost-precision tradeoffs.

The goal of our research was to pick an existing analysis and to extend it to handle features of Java. This goal was motivated by our need to build a pointer analysis to support CBGC (connectivity-based garbage collection) [31]. To pick an existing analysis that would be suitable for our needs, we took advice from prior work. First, our own prior work indicated that a type-based analysis would be inadequate for CBGC [31]. Second, Shapiro and Horwitz [49] and Hind and Pioli [30] found that subset-based pointer analyses, such as Andersen's [5], produce results superior to unification-based pointer analyses, such as Steensgaard's [52]. Third, Liang et al. [41] report that it would be very hard to significantly improve the precision of Andersen's analysis without biting into the much more expensive shape analysis. Fourth, Lhoták and Hendren [39] report that they could improve the speed of Andersen's analysis by using filtering based on programming language types. Thus, we decided to base our analysis on Andersen's analysis and to employ type filtering to improve performance.

3.2 Online Interprocedural Analyses

An *online* interprocedural analysis is an interprocedural analysis that deals correctly with dynamic class loading.

Demand-driven interprocedural analyses. A number of pointer analyses are demand-driven, but not online [11, 12, 26, 55, 3, 37]. All of these build a representation of the static whole program, but then compute exact solutions only for parts of it, which makes them more scalable. None of these papers discuss specific issues arising with online analyses that deal with dynamic class loading.

Incremental interprocedural analyses. Another related area of research is incremental interprocedural analysis [15, 10, 24, 23]. The goal of this line of research is to avoid a reanalysis of the complete program when a change is made after an interprocedural analysis has been performed. Our work differs from these works in that we focus on the dynamic semantics of the Java programming language, not programmer modifications to the source code.

Sreedhar, Burke, and Choi [50] describe extant analysis, which finds parts of the static whole program that can be safely optimized ahead of time, even when new classes may be loaded later. It is not an online analysis, but reduces the need for one in settings where much of the program is available statically.

Pechtchanski and Sarkar [44] present a framework for interprocedural whole-program analysis and optimistic optimization. They discuss how the analysis is triggered (when newly loaded methods are compiled), and how to keep track of what to de-optimize (when optimistic assumptions are invalidated). They also present an example online interprocedural type analysis. Their analysis does not model value flow through parameters, which makes it less precise, as well as easier to implement, than Andersen's analysis.

Bogda and Singh [9] adapt Ruf's escape analysis [47] to deal with dynamic class loading. They discuss tradeoffs of when to trigger the analysis, and whether to make optimistic or pessimistic assumptions for optimization. Ruf's analysis is unification-based, similar to Steensgaard's analysis, and thus, less precise than Andersen's analysis.

King [35] also adapts Ruf's escape analysis [47] to deal with dynamic class loading. He focuses on a specific client, a garbage collector with thread-local heaps, where local collections require no synchronization. Whereas Bogda and Singh use a call-graph based on capturing call edges at their first dynamic execution, King uses a call-graph based on rapid type analysis [8].

All of the online analyses discussed above [44, 9, 35] deal with dynamic class loading, but are weaker than Andersen's analysis. Moreover, many of them are escape analyses and not pointer analyses. Choi et al. [13] note that escape analysis is an easier problem than pointer analysis. In addition, none of them handle reflection or JNI (Java Native Interface). Our contribution is to make a stronger analysis support dynamic class loading, as well as reflection and JNI.

3.3 Andersen's Analysis for Static Java

A number of papers describe how to use Andersen's analysis for Java [46, 40, 57, 39]. None of these deal with dynamic class loading. Nevertheless, they do present solutions for various other features of Java that make pointer analyses difficult (object fields, virtual method invocations, etc.).

Rountev, Milanova, and Ryder [46] formalize Andersen's analysis for Java using set constraints, which enables them to solve it with BANE (Berkeley ANalysis Engine) [19]. Liang, Pennings, and Harrold [40] compare both Steensgaard's and Andersen's analysis for Java, and evaluate trade-offs for handling fields and the call graph. Whaley and Lam [57] improve the efficiency of Andersen's analysis by using implementation techniques from CLA [27], and improve the precision by adding flow-sensitivity for local variables. Lhoták and Hendren [39] present SPARK (Soot Pointer Analysis Research Kit), an implementation of Andersen's analysis in Soot [54], which provides precision and efficiency tradeoffs for various components.

Prior work on implementing Andersen's analysis differs in how it represents constraint graphs. There are many alternatives, and each one has different cost/benefit tradeoffs. We will discuss these in Section 4.1.

3.4 Validation Methodology

Our validation methodology compares points-to sets computed by our analysis to actual pointers at runtime. This is similar to limit studies that other researchers have used to evaluate and debug various compiler analyses [36, 18, 41].

4. ALGORITHM

To analyze Java programs, analyses need to handle features such as dynamic class loading, reflection, and native methods. Our algorithm extends Andersen's pointer analysis [5] and handles all the challenges introduced by Java. We implemented and evaluated our analysis in Jikes RVM, an open source research virtual machine for Java developed at IBM [4]. Section 4.1 describes how we use techniques from prior work to implement Andersen's analysis. Section 4.2 describes our own contributions that make it work for all of Java.

4.1 Overview

At a high level, inclusion-constraint based pointer analysis consists of two steps. First, the *constraint finder* analyzes the static program code, building a *representation* that models the constraints inherent in the data flow of the program. Second, the *propagator* processes the representation, producing points-to sets. In an online setting, the points-to sets conservatively describe the pointers in the program until there is an addition to the constraints (Section 4.2).

Representation.

Our representation has four kinds of nodes that participate in the constraints. The constraints themselves are stored as sets at the nodes. Table 1 describes the nodes, introducing the notation that we use in the remainder of the paper, and shows which sets are stored at each node. The node kinds in "$[\cdots]$" are the kinds of nodes in the set.

The constraint finder models program code by v-nodes, $v.f$-nodes, and their flow sets. Based on these, the propagator computes the points-to sets of v-nodes and $h.f$-nodes. For example, if a client of the pointer analysis is interested in whether a variable p may point to objects allocated at an allocation site a, it checks whether the h-node for a is an element of the points-to set of the v-node for p.

Each h-node can be efficiently mapped to its $h.f$-nodes (i.e., the nodes that represent the instance fields of the objects represented by the h-node). In addition to language-level fields, for each h-node, there is a special node $h.f_{\text{td}}$ that represents the field that contains the reference to the type descriptor for the heap node. A type descriptor is just

Table 1: Constraint graph representation.

Node kind	Represents concrete entities	Flow sets	Points-to sets
h-node	Set of heap objects, e.g., all objects allocated at a particular allocation site	none	none
v-node	Set of program variables, e.g., a static variable, or all occurrences of a local variable	flowTo[v], flowTo[$v.f$]	pointsTo[h]
$h.f$-node	Instance field f of all heap objects represented by h	none	pointsTo[h]
$v.f$-node	Instance field f of all h-nodes pointed to by v	flowFrom[v], flowTo[v]	none

like any other object in our system, and thus, must be modeled by our analysis. For each h-node representing arrays of references, there is a special node $h.f_{\text{elems}}$ that represents all of their elements. Thus, our analysis does not distinguish between different elements of an array.

Points-to sets represent the set of objects (r-values) that a pointer (l-value) may point to, and are stored with v-nodes and $h.f$-nodes. Storing points-to sets with $h.f$-nodes instead of $v.f$-nodes makes our analysis *field sensitive* [39].

Flow-to sets represent a flow of values (assignments, parameter passing, etc.), and are stored with v-nodes and $v.f$-nodes. For example, if $v'.f \in \text{flowTo}(v)$, then v's pointer r-value may flow to $v'.f$. Flow-from sets are the inverse of flow-to sets. In the example, we would have $v \in \text{flowFrom}(v'.f)$.

There are many alternatives for where to store the flow and points-to sets. For example, we represent the data flow between v-nodes and $h.f$-nodes implicitly, whereas Bane represents it explicitly [22, 46]. Thus, our analysis saves space compared to Bane, but may have to do some more work at propagation time. As another example, CLA [27] stores reverse points-to sets at h-nodes, instead of storing forward points-to sets at v-nodes and $h.f$-nodes. The forward points-to sets are implicit in CLA and must therefore be computed after propagation to obtain the final analysis results. These choices affect both the time and space complexity of the propagator. As long as it can infer the needed sets during propagation, an implementation can decide which sets to represent explicitly. In fact, a representation may even store some sets redundantly: for example, the flow-from sets are redundant in our representation, to help efficient propagation.

Finally, there are many choices for how to implement the sets. The Spark paper evaluates various data structures for representing points-to sets [39]. In their paper, hybrid sets (using lists for small sets, and bit-vectors for large sets) yielded the best results. We found the shared bit-vector implementation from CLA [25] to be even more efficient than the hybrid sets used by Spark.

Constraint finder.

Table 2 shows the various places in the runtime system where our online analysis finds constraints. Here, we look only at intraprocedural constraint finding during JIT compilation of a method; Section 4.2 discusses interprocedural constraint finding, as well as constraint finding for the various other places in the runtime system.

The intraprocedural constraint finder analyzes the code of a method, and models it in the constraint representation. We implemented it as a flow-insensitive pass of the optimizing compiler of Jikes RVM, operating on the high-level register based intermediate representation (HIR). HIR breaks down access paths by introducing temporaries so that no access path contains more than one pointer dereference.

Table 2: Points in the runtime system where new constraints may arise, ordered from least dynamic to most dynamic.

Runtime system action	Page
Virtual machine build and start-up	7
Class loading	8
Type resolution	6
JIT compilation of a method	4
Reflection	7
Execution of native code	7

Column "Actions" in Table 3 gives the actions of the constraint finder when it encounters the statement in Column "Statement". Column "Represent constraints" shows the constraints implicit in the actions of the constraint finder using mathematical notation.

Propagator.

The propagator propagates points-to sets following the constraints implicit in the flow sets until nothing changes anymore. In formal terminology, it finds the least fixed-point of a system of subset constraints. We implemented this step with the algorithm in Figure 2, which is a modified version of the worklist propagation algorithm from Spark [39].

The propagator puts a v-node on the worklist when its points-to set changes (the constraint finder also puts a v-node on the worklist when its points-to or flow sets change, so when the propagator starts, the worklist already contains all relevant nodes). Lines 4-10 propagate the v-node's points-to set to nodes in its flow-to sets. Lines 11-19 update the points-to set for all fields of objects pointed to by the v-node. This is necessary because for the h-nodes that have been newly added to v's points-to set, the flow to and from $v.f$ carries over to the corresponding $h.f$-nodes. Line 12 relies on the redundant flow-from sets.

The propagator sets the isCharged-bit of an $h.f$-node to true when its points-to set changes. To discharge an $h.f$-node, the algorithm needs to consider all flow-to edges from all $v.f$-nodes that represent it (lines 20-24). This is why it does not keep a worklist of charged $h.f$-nodes: to find their flow-to targets, it needs to iterate over $v.f$-nodes anyway. This is the only part of our algorithm that iterates over all ($v.f$-) nodes: all other parts of our algorithm attempt to update points-to sets while visiting only nodes that are

Table 3: Intraprocedural constraint finder

Statement		Actions	Represent constraints
$v' = v$	(move $v \to v'$)	flowTo(v).add(v')	pointsTo(v) $\subseteq$ pointsTo(v')
$v' = v.f$	(load $v.f \to v'$)	flowTo($v.f$).add(v')	$\forall h \in$ pointsTo(v) : pointsTo($h.f$) $\subseteq$ pointsTo(v')
$v'.f = v$	(store $v \to v'.f$)	flowTo(v).add($v'.f$), flowFrom($v'.f$).add(v)	$\forall h \in$ pointsTo(v') : pointsTo(v) $\subseteq$ pointsTo($h.f$)
ℓ: $v = new \ldots$	(alloc $h_\ell \to v$)	pointsTo(v).add(h_ℓ)	$\{h_\ell\} \subseteq$ pointsTo(v)

```
 1: while worklist not empty, or isCharged(h.f) for any h.f-node
 2:     while worklist not empty
 3:         remove first node v from worklist
 4:         for each v' ∈ flowTo(v)                                  // move v → v'
 5:             pointsTo(v').add(pointsTo(v))
 6:             if pointsTo(v') changed, add v' to worklist
 7:         for each v'.f ∈ flowTo(v)                                // store v → v'.f
 8:             for each h ∈ pointsTo(v')
 9:                 pointsTo(h.f).add(pointsTo(v))
10:                 if pointsTo(h.f) changed, isCharged(h.f) ← true
11:         for each field f of v,
12:             for each v' ∈ flowFrom(v.f)                          // store v' → v.f
13:                 for each h ∈ pointsTo(v)
14:                     pointsTo(h.f).add(pointsTo(v'))
15:                     if pointsTo(h.f) changed, isCharged(h.f) ← true
16:             for each v' ∈ flowTo(v.f)                            // load v.f → v'
17:                 for each h ∈ pointsTo(v)
18:                     pointsTo(v').add(pointsTo(h.f))
19:                     if pointsTo(v') changed, add v' to worklist
20:     for each v.f
21:         for each h ∈ pointsTo(v), if isCharged(h.f)
22:             for each v' ∈ flowTo(v.f)                            // load v.f → v'
23:                 pointsTo(v').add(pointsTo(h.f))
24:                 if pointsTo(v') changed, add v' to worklist
25:     for each h.f
26:         isCharged(h.f) ← false
```

Figure 2: Constraint propagator

relevant to the points-to sets being updated.

To improve the efficiency of this iterative part, we use a cache that remembers the charged nodes in shared points-to sets (and thus speeds up Line 21). The cache speeds up the loops at Lines 20 and 21 by an order of magnitude.

During propagation, we apply on-the-fly filtering by types: we only add an h-node to a points-to set of a v-node or $h.f$-node if it represents heap objects of a subtype of the declared type of the variable or field. Lhoták and Hendren found that this helps keep the points-to sets small, improving both precision and efficiency of the analysis [39]. Our experiences confirm this observation.

The propagator creates $h.f$-nodes lazily the first time it adds elements to their points-to sets, in lines 9 and 14. It only creates $h.f$-nodes if instances of the type of h have the field f. This is not always the case, as the following example illustrates. Let A, B, C be three classes such that C is a subclass of B, and B is a subclass of A. Class B declares a field f. Let h_A, h_B, h_C be h-nodes of type A, B, C, respectively. Let v be a v-node of declared type A, and let v.pointsTo $= \{h_A, h_B, h_C\}$. Now, data flow to $v.f$ should add to the points-to sets of nodes $h_B.f$ and $h_C.f$, but there is no node $h_A.f$.

We also experimented with the optimizations partial online cycle elimination [19] and collapsing of single-entry subgraphs [45]. They yielded only modest performance improvements compared to shared bit-vectors [25] and type filtering [39]. Part of the reason for the small payoff may be that our data structures do not put $h.f$ nodes in flow-to sets (á la Bane [19]).

4.2 Details

Section 4.1 presented an overview of our algorithm, which is closely based on analyses in prior work. We now describe our main contributions: how we solved issues arising from handling all of Java, including dynamic class loading.

Our basic approach to handling dynamic class loading is to run the intraprocedural constraint finder whenever Jikes RVM compiles a method. Besides method compilation, the other events from Table 2 may also lead to new constraints. Whenever a client needs points-to information, we run the propagator to obtain the points-to sets. When propagating the constraints, we use the previous solution as a starting point, and thus, do not need to solve the entire constraint system from scratch every time.

Points-to sets are conservative at the time of the propa-

gation. In other words, if there is a pointer between concrete objects, then the points-to sets represent that pointer. These points-to sets remain conservative until we find new constraints. Since Java supports dynamic class loading, it is not possible to have a non-trivial pointer analysis for Java whose results will be conservative at all times (Section 2). Prior pointer analyses get around this issue by analyzing only a subset of Java, being less precise, or both.

Interprocedural constraints.

For each call-edge, our analysis generates constraints that model the data flow through parameters and return values. Parameter passing is modeled like a move from actuals (at the call-site) to formals (of the callee). Each return statement in a method m is modeled like a move to a special v-node $v_{\text{retval}(m)}$. The data flow of the return value to the call-site is modeled like a move to the v-node that receives the result of the call.

We use CHA (Class Hierarchy Analysis) [20, 16] to find call-edges. A more precise alternative to CHA is to construct the call graph on-the-fly based on the results of the pointer analysis. We decided against that approach because prior work indicated that the modest improvement in precision does not justify the cost in efficiency [39].

CHA is a static whole-program analysis, but we must perform CHA online to deal with dynamic class loading. The key to our solution for this problem is the observation that for each call-edge, either the call-site is compiled first, or the callee is compiled first. We add the constraints for the call-edge when the second of the two is compiled. This works as follows:

- When encountering a method $m(v_{\text{formal}_1(m)}, \ldots, v_{\text{formal}_n(m)})$, the constraint finder
  - creates a tuple $I_m = \langle v_{\text{retval}(m)}, v_{\text{formal}_1(m)}, \ldots, v_{\text{formal}_n(m)} \rangle$ for m as a callee;
  - looks up all corresponding tuples for matching call-sites that have been compiled in the past, and adds constraints to model the moves between the corresponding v-nodes in the tuples; and
  - stores the tuple I_m for lookup on behalf of call-sites that will be compiled in the future.
- When encountering a call-site $c : v_{\text{retval}(c)} = m(v_{\text{actual}_1(c)}, \ldots, v_{\text{actual}_n(c)})$, the constraint finder
  - creates a tuple $I_c = \langle v_{\text{retval}(c)}, v_{\text{actual}_1(c)}, \ldots, v_{\text{actual}_n(c)} \rangle$ for call-site c;
  - looks up all corresponding tuples for matching callees that have been compiled in the past, and adds constraints to model the moves between the corresponding v-nodes in the tuples; and
  - stores the tuple I_c for lookup on behalf of callees that will be compiled in the future.

Besides parameter passing and return values, there is one more kind of interprocedural data flow that our analysis needs to model: exception handling. Exceptions lead to flow of values (the exception object) between the site that throws an exception and the catch clause that catches the exception. For simplicity, our initial prototype assumes that any throws can reach any catch clause. Type filtering will eliminate many of these possibilities later on.

Unresolved References.

The JVM specification allows a Java method to have unresolved references to fields, methods, and classes [42]. A class reference is resolved when the class is instantiated, when a static field in the class is used, or when a static method in the class is called.

The unresolved references in the code (some of which may never get resolved) create two main difficulties for our analysis.

First, the CHA (class hierarchy analysis) that we use to map call-sites to callees and vice versa, does not work when the class hierarchy of the involved classes is not yet known. Our current approach to this is to be conservative: if, due to unresolved classes, CHA cannot yet decide whether a call edge exists, we assume it exists if the signatures match.

Second, the propagator uses types to perform type filtering and also for deciding which $h.f$-nodes belong to a given $v.f$-node. If the involved types are not yet resolved, this does not work. We solve this problem by deferring the representation of all flow sets and points-to sets involving nodes of unresolved types, thus hiding them from the propagator. This task is the responsibility of the *resolution manager*:

- When the constraint finder creates an unresolved node, it registers the node with the resolution manager. A node is unresolved if it refers to an unresolved type. An h-node refers to the type of its objects; a v-node refers to its declared type; and a $v.f$-node refers to the type of v, the type of f, and the type in which f is declared.
- When the constraint finder would usually add a node to a flow set or points-to set of another node, but one or both of them is unresolved, it defers the information for later instead. Table 4 shows the deferred sets stored at unresolved nodes. For example, if the constraint finder finds that v should point to h, but v is unresolved, it adds h to v's deferred pointsTo set. Conversely, if h is unresolved, it adds v to h's deferred pointedToBy set. If both are unresolved, the points-to information is stored twice.
- When a type is resolved, the resolution manager notifies all unresolved nodes that have registered for it. When an unresolved node is resolved, it iterates over all deferred sets stored at it, and attempts to add the information to the real model that is visible to the propagator. If a node stored in a deferred set is not resolved yet itself, the information will be added in the future when that node gets resolved.

With this design, some constraints will never be added to the model, if their types never get resolved. This saves unnecessary propagator work.

Before becoming aware of the subtleties of the problems with unresolved references, we used an overly conservative approach: we added the constraints eagerly even when we had incomplete information. This imprecision led to very

Table 4: Deferred sets stored at unresolved nodes.

Node kind	Flow	Points-to
h-node	none	pointedToBy[v]
v-node	flowFrom[v], flowFrom[$v.f$], flowTo[v], flowTo[$v.f$]	pointsTo[h]
$h.f$-node	there are no unresolved $h.f$-nodes	
$v.f$-node	flowFrom[v], flowTo[v]	none

large points-to sets, which in turn slowed down our analysis prohibitively. Our current approach is both more precise and more efficient. On the other hand, with a less precise approach, the points-to sets may remain correct for longer, since resolving a class cannot cause them to go out of date.

Reflection.

Java programs can invoke methods, access and modify fields, and instantiate objects using reflection. When compiling code that uses reflection, there is no way of determining which methods will be called, which fields manipulated, or which classes instantiated at runtime.

One solution is to assume the worst case. We felt that this was too conservative and would introduce significant imprecision into the analysis for the sake of a few operations which were rarely executed. Other pointer analyses for Java side-step this problem by requiring users of the analysis to provide hand-coded models describing the effect of the reflective actions [57, 39].

Our approach for handling reflective actions is to instrument the virtual machine service that handles reflection with code that adds constraints when they execute. For example, if reflection is used to store into a field, we observe the actual source and target of the store and generate a constraint that captures the semantics of the store.

This strategy for handling reflection introduces new constraints when the reflective code does something new. Fortunately, this does not happen very often. When reflection has introduced new constraints and we need up-to-date points-to results, we must rerun the propagator.

Native Code.

The Java Native Interface (JNI) allows Java code to interact with dynamically loaded native code. Usually, a JVM cannot analyze that code. Thus, an analysis does not know (i) what values may be returned by JNI methods and (ii) how JNI methods may manipulate data structures of the program.

An elegant solution to this would be to analyze the native code as it gets loaded into the virtual machine. In order to do this, the JVM needs to include an infrastructure for analyzing machine code. To date, the only virtual machine we are aware of that offers this functionality is Whaley's Joeq [56], and even Whaley and Lam do not use it to deal with native code in their pointer analysis [57].

Our approach is to be imprecise, but conservative, for return values from JNI methods, while being precise for data manipulation by native methods. If a native method returns a heap allocated object, we assume that it could return an object from any allocation site. This is imprecise, but easy to implement. Recall that the constraint propagation uses type filtering, and thus, will filter the set of heap nodes returned by a native method based on types. If a native method manipulates data structures of the program, the manipulations must go through the JNI API, which Jikes RVM implements by calling Java methods that use reflection. Thus, native methods that make calls or manipulate object fields are handled precisely by our mechanism for reflection.

Other Challenges.

In addition to the challenges mentioned above, there were a number of other, smaller challenges that we had to handle:

- Pre-allocated data structures. Jikes RVM itself is written in Java, and starts up by loading a *boot image* (a file-based image of a fully initialized VM) of pre-allocated Java objects for the JIT compilers, GC, and other runtime services. These objects live in the same heap as application objects, so our analysis must model them. Modeling them by analyzing the Java code that builds the boot image is insufficient, since that risks missing build-time reflection and native code. Instead, our analysis directly models the snapshot of the objects in the boot image.

- Non-JIT-compiled Code. Our intraprocedural constraint finder is a pass of the Jikes RVM optimizing compiler. However, Jikes RVM compiles some methods only with a baseline compiler, and other virtual machines may even not compile them at all, but interpret them instead. The baseline compiler does not use a representation that is amenable to constraint finding, thus we just fire up a truncated optimizing compilation to run the intraprocedural constraint finder for baseline-compiled methods.

- Recompilation of methods. Many JVMs, including Jikes RVM, may recompile a method, for example based on its execution frequency. The recompiled methods may have new variables or code introduced by optimizations (such as inlining). Since we model each inlining context of an allocation site by a separate h-node, we need to generate the constraints for the recompiled methods and integrate the constraints with the constraints for any previously compiled versions of the method.

- Magic. Jikes RVM has some internal operations (collectively called "Magic"), which are expanded in special ways directly into low level code (e.g., write barriers to support garbage collection). We added models for these magic methods that generate the appropriate constraints.

5. VALIDATION

A pointer analysis for a complicated language and environment such as Java and Jikes RVM is a difficult task

indeed: the pointer analysis has to handle numerous corner cases, and missing any of the cases results in incorrect points-to sets. To help us debug our pointer analysis (to a high confidence level) we built a validation mechanism.

5.0.1 Validation Mechanism.

We validate the pointer analysis results at GC (garbage collection) time. As GC traverses each pointer, we check whether the points-to set captures the pointer: (i) When GC finds a static variable p holding a pointer to an object o, our validation code finds the nodes v for p and h for o. Then, it checks whether the points-to set of v includes h. (ii) When GC finds a field f of an object o holding a pointer to an object o', our validation code finds the nodes h for o and h' for o'. Then, it checks whether the points-to set of $h.f$ includes h'. If either check fails, it prints a warning message.

To make the points-to sets correct at GC time, we propagate the constraints (Section 4.1) just before GC starts. However, at this time, there is no memory available to grow points-to sets, so we modified Jikes RVM's garbage collector to set aside some extra space for this purpose.

Our validation methodology relies on the ability to map concrete heap objects to h-nodes in the constraint graph. To facilitate this, we added an extra header word to each heap object that maps it to its corresponding h-node in the constraint graph. For h-nodes representing allocation sites, we install this header word at allocation time.

5.0.2 Validation Anecdotes.

Our validation methodology helped us find many bugs, some of which were quite subtle. Here are two examples.

Anecdote 1. In Java bytecode, a field reference consists of the name and type of the field, as well as a class reference to the class or interface "in which the field is to be found" ([42, Section 5.1]). Even for a static field, this may not be the class that declared the field, but a subclass of that class. Originally, we had assumed that it must be the exact class that declared the field, and written our analysis accordingly to maintain separate v-nodes for static fields with distinct declaring classes. When the bytecode wrote to a field using a field reference that mentions the subclass, the v-node for the field that mentions the super-class ended up missing some points-to set elements. That showed up as warnings from our validation methodology. Upon investigating those warnings, we became aware of the incorrect assumption and fixed it.

Anecdote 2. In Java source code, a static field declaration has an optional initialization, for example, "`final static String s = "abc";`". In Java bytecode, this usually translates into initialization code in the class initializer method `<clinit>()` of the class that declares the field. But sometimes, it translates into a ConstantValue attribute of the field instead ([42, Section 4.5]). Originally, we had assumed that class initializers are the only mechanism for initializing static fields, and that we would find these constraints when running the constraint finder on the `<clinit>()` method. But our validation methodology warned us about v-nodes for static fields whose points-to sets were too small. Knowing exactly for which fields that happened, we looked at the bytecode, and were surprised to see that the `<clinit>()` methods didn't initialize the fields. Thus, we found out about the ConstantValue bytecode attribute, and added constraints when class loading parses and executes that attribute.

In both of these examples, there was more than one way in which bytecode could represent a Java-level construct. Both times, our analysis dealt correctly with the more common case, and the other case was obscure, yet legal. Our validation methodology showed us where we missed something; without it, we might not even have suspected that something was wrong.

6. CLIENTS

This section investigates two example clients of our analysis, and how they can deal with the dynamic nature of our analysis.

Method inlining, which is perhaps the most important optimization for object-oriented programs, can benefit from pointer analysis: if the points-to set elements of v all have the same implementation of a method m, the call $v.m()$ has only one possible target. Modern JVMs [43, 53, 6, 14] typically use a dual execution strategy, where each method is initially executed in an unoptimized manner (either interpreter or non-optimizing compiler). For such methods, inlining is not performed. Later, an optimizing compiler is used for the minority of methods that are determined to be executing frequently. In some systems methods may be re-optimized if an additional inlining candidate is found. Because inlining is not performed during the initial execution, our analysis does not need to perform constraint propagation until the optimizing compiler needs to make an inlining decision.

Since the results of our pointer analysis may be invalidated by any of the events in Table 2, we need to be prepared to invalidate inlining decisions. There are many techniques for doing this in prior work, e.g., code patching [14] and on-stack replacement [33, 21]. If instant invalidation is needed, our analysis must re-propagate every time it finds new constraints. There are also techniques for avoiding invalidation of inlining decisions, such as pre-existence based inlining [17] and guards [34, 7]. These would allow our analysis to be lazy about re-propagating after it finds new constraints.

CBGC (connectivity-based garbage collection) is a new garbage collection technique that requires pointer analysis [31]. CBGC uses pointer analysis results to partition heap objects such that connected objects are in the same partition, and the pointer analysis can guarantee the absence of certain inter-partition pointers. CBGC exploits the observation that connected objects tend to die together [32], and certain subsets of partitions can be collected while completely ignoring the rest of the heap.

CBGC must know the partition of an object at allocation time. However, CBGC can easily combine partitions later if the pointer analysis finds that they are strongly connected by pointers. Thus, there is no need to perform a full propagation at object allocation time. However, CBGC does need full conservative points-to information when performing a garbage collection; thus, CBGC needs to request a full propagation before collecting. Between collections, CBGC does not need conservative points-to information.

7. PERFORMANCE

This section evaluates the time efficiency of our pointer analysis. We implemented our analysis in Jikes RVM 2.2.1.

Table 5: Benchmark programs.

Program	Analyzed Methods	Loaded Classes
null	14,782	1,244
compress	14,912	1,273
db	14,930	1,267
mtrt	15,037	1,286
mpegaudio	15,084	1,311
jack	15,146	1,316
jess	15,342	1,409
javac	15,648	1,408
xalan	16,241	1,597

In addition to the analysis itself, our modified version of Jikes RVM includes the validation mechanism from Section 5. Besides the analysis and validation code, we also added a number of profilers and tracers to collect the results presented in this section. For example, at each yield-point (method prologue or loop back-edge), we walk up the stack to determine whether the yield-point belongs to analysis or application code, and count it accordingly. We performed all experiments on a 2.4GHz Pentium 4 with 2GB of memory running Linux, kernel version 2.4.

7.1 Benchmark Characteristics

Our benchmark suite (Table 5) includes the SPECjvm98 benchmarks [51], *null* (a benchmark with an empty main method), and *xalan*, an XSLT processor [2, 1]. We use the size "100" inputs for the SPECjvm98 benchmarks and the input "1 1" for xalan. To gather data for this table, we enabled aggressive compiler optimizations (-O2) except for inlining.

Column "Analyzed Methods" gives the total number of methods we analyze in these benchmark programs. We analyze a method when it is part of the boot image, or when the program executes it for the first time. The analyzed methods include not just methods of the benchmark, but any library methods called by the benchmarks, and any methods called as part of optimizing compilation, analysis (including our pointer analysis), and system bootup. The *null* benchmark provides a baseline: its data represents approximately the amount that Jikes RVM adds to the size of the application. This data is approximate because, for example, some of the methods called by the optimizing compiler may also be used by the application (e.g., methods on container classes). Column "Loaded Classes" gives the number of classes loaded by the benchmarks. Once again, the number of loaded classes for the *null* benchmark provides a baseline. Since our analysis also analyzes the runtime system code, our benchmarks are potentially much larger than the same benchmarks in prior work.

Figure 3 shows how the number of analyzed method increase over a run of *mpegaudio*. The x-axis represents time measured by the number of thread yield points encountered in a run. There is a thread yield point in the prologue of every method and in every loop. We ignore yield points that occur in our analysis code (this would be hard to do if we used real time for the x-axis). The y-axis starts at 14,750: all methods analyzed before the first method in this graph are in the boot image and are thus analyzed once for all benchmarks. The graphs for other benchmarks have a similar shape, and therefore we omit them.

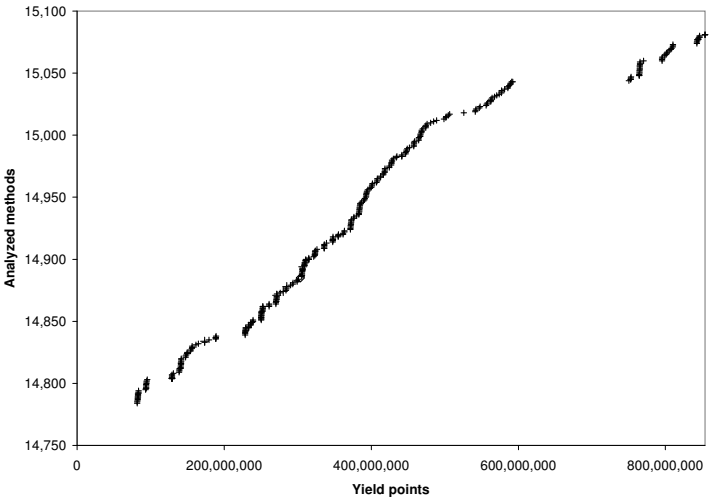

Figure 3: Yield points versus analyzed methods for *mpegaudio*. The first shown data point is the `main()` method.

From Figure 3, we see that the analysis encounters new methods throughout program execution (except for mostly brief periods, most notably around the 600,000,000 yield point mark, where the application is executing only methods that it has encountered before). We were expecting the number of analyzed methods to stabilize early in the run, so we were surprised to see the number climb until the very end of the program run. The reason for this is that the SPECjvm98 benchmarks are all relatively short running: for example, if we had a webserver that ran for days, we would indeed see the number of analyzed methods stabilize after a few minutes. That point may be an ideal time to propagate the constraints and use the results to perform optimizations.

7.2 Analysis Cost

Our analysis has two main costs: constraint finding and constraint propagation. Constraint finding happens whenever we analyze a new method, load a new class, etc. Constraint propagation happens whenever a client of the pointer analysis needs points-to information. The remainder of this section presents data on the performance of our pointer analysis. Since our analysis is based on Andersen's analysis, whose precision has been evaluated in prior work (e.g., [39]), we focus only on the cost of our analysis. Unless otherwise indicated, all our experiments used the -O2 optimization flag of Jikes RVM with inlining explicitly disabled.

7.2.1 Cost of Constraint Finding.

Table 6 gives the percentage of overall execution time spent in generating constraints from methods (Column "Analyzing methods") and from resolution events (Column "Resolving classes and arrays"). For these executions we did not run any propagations. From this table, we see that while the cost of generating constraints for methods is up to 11.3%, the cost for generating constraints due to resolution of classes and arrays is insignificant. When a long-running application, such as a web server, reaches its steady state, no new methods will be analyzed, and thus the cost of the constraint finder would go down to unnoticeable levels. It is still significant for our benchmarks, since even the longest

Table 6: Percent of execution time in constraint finding

Program	Analyzing methods	Resolving classes and arrays
compress	11.3%	0.2%
db	10.0%	0.2%
mtrt	8.9%	0.2%
mpegaudio	6.7%	0.6%
jack	4.7%	0.2%
jess	3.3%	0.2%
javac	2.3%	0.2%
xalan	1.6%	0.2%

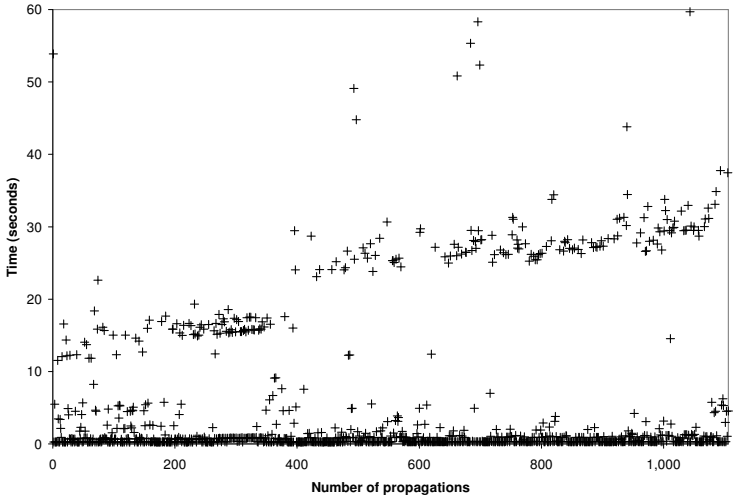

Figure 4: Propagation times for *javac* (eager).

running benchmark (*javac*) runs for merely 33 seconds.

7.2.2 Cost of Propagation.

Table 7 shows the cost of eager propagation (after every event from Table 2, if it generated new constraints), propagation at GC (before every garbage collection), and lazy propagation (just once at the end of the program execution).

Columns "Count" give the number of propagations that occur in our benchmark runs. Columns "Time" give the arithmetic mean ± standard deviation of the time (in seconds) it takes to perform each propagation. We included the lazy propagation data to give an approximate sense for how long the propagation would take if we were to use a static pointer analysis. Recall, however, that these numbers are still not comparable to static analysis numbers of these benchmarks in prior work, since, unlike them, we also analyze the Jikes RVM compiler and other system services.

From Table 7, we see that with eager propagation, the mean pause time due to propagation varies between 2.1 seconds and 6.8 seconds. In contrast, a full (lazy) propagation is much slower. In other words, our algorithm is effective in avoiding work on parts of the program that have not changed since the last propagation.

Our results (omitted for space considerations) showed that the cost of a propagation did not depend on which of the events in Table 2 generated new constraints that were the reason for the propagation.

Figure 4 presents the spread of propagation times for *javac*. A point (x,y) in this graph says that propagation "x" took "y" seconds. Out of 1,107 propagations in *javac*, 706 propagations take under 1 second. The remaining propagations are much more expensive (10 seconds or more), thus driving the average up. We can also discern an upward trend in the more expensive propagations: they are more expensive later in the execution than earlier in the execution. We explore this more in Section 7.3. The omitted graphs for other benchmarks have a similar shape.

While we present the data for eager propagation, we do not expect clients of our analysis to use eager propagation. For example, connectivity-based garbage collection [31] does not need to propagate every time the constraint set grows. It is adequate to propagate only periodically, for example at garbage collections.

As expected, the columns for propagation at GC in Table 7 show that if we propagate less frequently, the individual propagations are more expensive; they are still on average cheaper than performing a single full propagation at the end of the program run. Recall that, for Java programs, performing a static analysis of the entire program is not possible because what constitutes the "entire program" is not known until it executes to completion.

7.2.3 Interactions of Propagation Cost with Inlining.

So far, we have reported our analysis time with -O2 optimizations, but with inlining disabled. Inlining is a double-edged sword: it is both the most effective optimization for Java programs and also has the greatest potential for slowing down other analyses [38]. Inlining increases the size of the code and can thus slow down subsequent analyses, most notably ones that are not linear-time (e.g., the SSA analyses in Jikes RVM). Since Andersen's analysis is also not linear (it is cubic time), we would expect inlining to interact poorly with it. Indeed, when we enabled inlining, we found that the time for propagation at garbage collections went up by a factor of 2 to 5. In other words, if one is employing any non-linear-time algorithms (such as our pointer analysis or SSA-based analyses), one must be selective about applying inlining.[2]

7.3 Understanding the Costs of our Constraint Propagation

The speed of our constraint propagator (a few seconds to update points-to information) may be adequate for long-running clients, but may not be feasible for short-running clients. For example, a web server that does not touch new methods after a few minutes of running can benefit from our current analysis: once the web server stops touching new methods, the propagation time of our analysis goes down to zero. Since we did not have a server application in our suite, we confirmed this behavior by running two benchmarks (*javac* and *mpegaudio*) multiple times in a loop: after the first run, there was little to no overhead from constraint finding or constraint propagation (well under 1%). On the other hand, an application that only runs for a few minutes may find our analysis to be prohibitively slow. In this section, we analyze the performance of our analysis in order to identify how best to focus our efforts in improving it.

Broadly speaking, our propagator has two parts: the worklist driven part (lines 2 to 19 in Figure 2) and the iter-

[2]By utilizing a dual-execution strategy as described in Section 6, modern JVMs satisfy this property.

Table 7: Propagation statistics (times in seconds)

	Eager		At GC		Lazy	
Program	Count	Time	Count	Time	Count	Time
null	0		0		1	53.2±0
compress	130	3.2±6.5	5	40.4±27.5	1	67.4±0
db	143	3.6±6.7	5	42.9±23.7	1	71.4±0
mtrt	265	2.1±5.0	5	46.2±10.5	1	68.1±0
mpegaudio	319	2.2±4.9	5	46.1±10.5	1	66.6±0
jack	397	4.2±6.4	7	49.0±13.3	1	78.2±0
jess	733	6.8±7.8	8	49.7±19.4	1	85.7±0
javac	1,107	5.9±10.5	10	87.4±39.2	1	187.6±0
xalan	1,728	4.9±9.2	8	85.7±28.9	1	215.7±0

Table 8: Arithmetic mean of time spent in the worklist and iterative parts at a propagation. All times in seconds.

Program	Worklist time	Iterative time
compress	2.6	0.4
db	2.9	0.4
mtrt	1.6	0.2
mpegaudio	1.6	0.2
jack	3.5	0.4
jess	5.8	0.7
javac	4.7	0.9
xalan	4.0	0.5

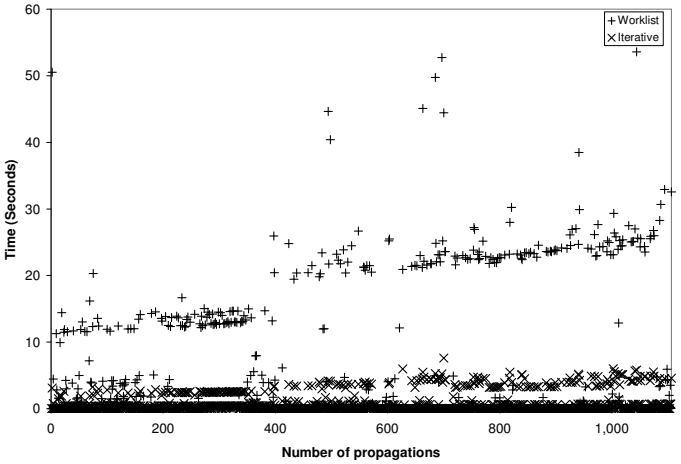

Figure 5: Breakdown of iteration time for *javac*.

ative part (lines 20 to 26 in Figure 2). The worklist driven part only performs as much work as necessary, whereas the iterative part is a little wasteful, since it iterates over all $v.f$-nodes and the corresponding $h.f$-nodes.

Table 8 gives the arithmetic mean of the time spent in the worklist and iterative parts during a propagation. For these experiments, we perform eager propagation (propagate every time new constraints are found). The results are surprising: we expected the iterative part to be the bottleneck, but instead, the worklist part takes more time.

Figure 5 presents the same data graphically for *javac*. A "Worklist" point (x,y) says that, for propagation "x", the worklist part of the algorithm took "y" seconds. An "Iterative" point (x,y) says that, for propagation "x", the iterative part of the algorithm took "y" seconds. We see that as *javac* executes, both the worklist and iterative parts get more expensive. We expect the iterative part to get more expensive, since it has more nodes to iterate through later in the execution compared to earlier in the execution. However, the fact that the worklist part gets more expensive as well suggests that as more and more classes are loaded and methods analyzed, the points-to sets do get larger, increasing the propagation time for the worklist part. The omitted graphs for other benchmarks exhibit similar trends. To summarize, our results indicate that we should focus our efforts for improving propagator performance on the worklist part.

8. CONCLUSIONS

We describe and evaluate the first non-trivial pointer analysis that handles all of Java. Java features such as dynamic class loading, reflection, and native methods introduce many challenges for pointer analyses. Some of these prohibit the use of static pointer analyses.

We validate the output of our analysis against actual pointers created during program runs. We evaluate our analysis by measuring many aspects of its performance, including the amount of work our analysis must do at run time. Our results show that our analysis is feasible and fast enough for server applications.

9. REFERENCES

[1] The Apache XML project. `http://xml.apache.org`.

[2] Colorado benchmark suite. `http://systems.cs.colorado.edu/colorado_bench`.

[3] G. Agrawal, J. Li, and Q. Su. Evaluating a demand driven technique for call graph construction. In *Internat. Conference on Compiler Construction (CC)*, 2002.

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

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

[6] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM.

In *Obj.-Oriented Prog., Systems, Lang., and Applic. (OOPSLA)*, 2000.

[7] M. Arnold and B. G. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In *European Conference for Object-Oriented Programming (ECOOP)*, 2002.

[8] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Obj.-Oriented Prog., Systems, Lang., and Applic. (OOPSLA)*, 1996.

[9] J. Bogda and A. Singh. Can a shape analysis work at run-time? In *Java Virtual Machine Research and Technology Symp. (JVM)*, 2001.

[10] M. Burke and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *Trans. on Prog. Lang. and Systems (TOPLAS)*, 1993.

[11] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *Principles of Prog. Lang. (POPL)*, 1999.

[12] B.-C. Cheng and W.-m. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Prog. Lang. Design and Impl. (PLDI)*, 2000.

[13] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Obj.-Oriented Prog., Systems, Lang., and Applic. (OOPSLA)*, 1999.

[14] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Prog. Lang. Design and Impl. (PLDI)*, 2000.

[15] K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *Trans. on Prog. Lang. and Systems (TOPLAS)*, 1986.

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

[17] D. Detlefs and O. Agesen. Inlining of virtual methods. In *European Conference for Object-Oriented Programming (ECOOP)*, 1999.

[18] A. Diwan, K. S. McKinley, and J. E. B. Moss. Using types to analyze and optimize object-oriented programs. *Trans. on Prog. Lang. and Systems (TOPLAS)*, 2001.

[19] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Prog. Lang. Design and Impl. (PLDI)*, 1998.

[20] M. F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Prog. Lang. Design and Impl. (PLDI)*, 1995.

[21] S. J. Fink and F. Qian. Design, implementation, and evaluation of adaptive recompilation with on-stack replacement. In *Code Gen. and Optimization (CGO)*, 2003.

[22] J. S. Foster, M. Fähndrich, and A. Aiken. Flow-insensitive points-to analysis with term and set constraints. Technical report, University of California at Berkeley, 1997.

[23] D. P. Grove. *Effective Interprocedural Optimization of Object-Oriented Languages*. PhD thesis, University of Washington, 1998.

[24] M. W. Hall, J. M. Mellor-Crummey, A. Carle, and R. G. Rodriguez. Fiat: A framework for interprocedural analysis and transformations. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 1993.

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

[26] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Prog. Lang. Design and Impl. (PLDI)*, 2001.

[27] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Prog. Lang. Design and Impl. (PLDI)*, 2001.

[28] L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell University, 1990.

[29] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.

[30] M. Hind and A. Pioli. Which pointer analysis should I use? In *Internat. Symp. on Software Testing and Analysis (ISSTA)*, 2000.

[31] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *Obj.-Oriented Prog., Systems, Lang., and Applic. (OOPSLA)*, 2003.

[32] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *Internat. Symp. on Memory Management (ISMM)*, 2002.

[33] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Prog. Lang. Design and Impl. (PLDI)*, 1992.

[34] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Prog. Lang. Design and Impl. (PLDI)*, 1994.

[35] A. C. King. Removing GC synchronization (extended version). `http://www.acm.org/src/subpages/AndyKing/overview.html`, 2003. Winner (Graduate Division) ACM Student Research Competition.

[36] J. R. Larus and S. Chandra. Using tracing and dynamic slicing to tune compilers. University of Wisconsin Technical Report 1174, Aug. 1993.

[37] C. Lattner and V. Adve. Data Structure Analysis: An Efficient Context-Sensitive Heap Analysis. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Apr 2003.

[38] H. Lee, D. von Dincklage, A. Diwan, and J. Moss. Understanding the behavior of compiler optimizations. Submitted, November 2003.

[39] O. Lhoták and L. Hendren. Scaling Java points-to analysis using SPARK. In *Internat. Conference on Compiler Construction (CC)*, 2003.

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

[41] D. Liang, M. Pennings, and M. J. Harrold. Evaluating the precision of static reference analysis using profiling. In *Internat. Symp. on Software Testing and Analysis (ISSTA)*, 2002.

[42] T. Lindholm and F. Yellin. *The Java virtual machine specification*. Addison-Wesley, second edition, 1999.

[43] M. Paleczny, C. Vick, and C. Click. The Java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symp. (JVM)*, 2001.

[44] I. Pechtchanski and V. Sarkar. Dynamic optimistic interprocedural analysis: a framework and an application. In *Obj.-Oriented Prog., Systems, Lang., and Applic. (OOPSLA)*, 2001.

[45] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Prog. Lang. Design and Impl. (PLDI)*, 2000.

[46] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Obj.-Oriented Prog., Systems, Lang., and Applic. (OOPSLA)*, 2001.

[47] E. Ruf. Effective synchronization removal for Java. In *Prog. Lang. Design and Impl. (PLDI)*, 2000.

[48] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Principles of Prog. Lang. (POPL)*, 1999.

[49] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Static Analysis Symp. (SAS)*, 1997.

[50] V. C. Sreedhar, M. Burke, and J.-D. Choi. A framework for interprocedural analysis and optimization in the presence of dynamic class loading. In *Prog. Lang. Design and Impl. (PLDI)*, 2000.

[51] Standard Performance Evaluation Corporation (SPEC). SPECjvm98 benchmarks. `http://www.specbench.org/osg/jvm98`.

[52] B. Steensgaard. Points-to analysis in almost linear time. In *Principles of Prog. Lang. (POPL)*, 1996.

[53] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *Obj.-Oriented Prog., Systems, Lang., and Applic. (OOPSLA)*, 2001.

[54] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *European Conference for Object-Oriented Programming (ECOOP)*, 2000.

[55] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Prog. Lang. Design and Impl. (PLDI)*, 2001.

[56] J. Whaley. Joeq: A virtual machine and compiler infrastructure. In *Workshop on Interpreters, Virtual Machines, and Emulators (IVME)*, 2003.

[57] J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Static Analysis Symp. (SAS)*, 2002.