

Pointer Analysis in the Presence of Dynamic Class Loading[★]

Martin Hirzel¹, Amer Diwan¹, and Michael Hind²

¹ University of Colorado, Boulder, CO 80309, USA
{hirzel,diwan}@cs.colorado.edu

² IBM Watson Research Center, Hawthorne, NY 10532, USA
hind@watson.ibm.com

Abstract. Many optimizations need precise pointer analyses to be effective. Unfortunately, some 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 these features or are overly conservative. This paper presents the first non-trivial pointer analysis that deals with all Java language features. This paper identifies all problems in performing Andersen’s pointer analysis for the full Java language, presents solutions to those problems, and uses a full implementation of the solutions in Jikes RVM for validation and performance evaluation. The results from this work should be transferable to other analyses and to other languages.

1 Introduction

Pointer analysis benefits many optimizations, such as inlining, load elimination, code movement, stack allocation, and parallelization. Unfortunately, dynamic class loading, reflection, and native code make ahead-of-time pointer analysis of Java programs impossible.

This paper presents the first non-trivial pointer analysis that works for all of Java. Most prior papers assume that all classes are known and available ahead of time (e.g., [39,40,47,60]). The few papers that deal with dynamic class loading assume restrictions on reflection and native code [7,36,44,45]. Prior work makes these simplifying assumptions because they are acceptable in some contexts, because dealing with the full generality of Java is difficult, and because the advantages of the analyses often outweigh the disadvantages of only handling a subset of Java.

This paper describes how to overcome the restrictions of prior work in the context of Andersen’s pointer analysis [3], so the benefits become available in the general setting of an executing Java virtual machine. This paper:

[★] This work is supported by NSF ITR grant CCR-0085792, an NSF Career Award CCR-0133457, an IBM Ph.D. Fellowship, an IBM faculty partnership award, and an equipment grant from Intel. Any opinions, findings and conclusions or recommendations expressed in this material are the authors’ and do not necessarily reflect those of the sponsors.

- (a) identifies all problems of performing Andersen’s pointer analysis for the full Java language,
- (b) presents a solution for each of the problems,
- (c) reports on a full implementation of the solutions in Jikes RVM, an open-source research virtual machine from IBM [2],
- (d) validates, for our benchmark runs, that the list of problems is complete, the solutions are correct, and the implementation works, and
- (e) evaluates the efficiency of the implementation.

The performance results show that the implementation is efficient enough for stable long-running applications. However, because Andersen’s algorithm has cubic time complexity, and because Jikes RVM, which is itself written in Java, leads to a large code base even for small benchmarks, performance needs improvements for short-running applications. Such improvements are an open challenge; they could be achieved by making Andersen’s implementation in Jikes RVM more efficient, or by using a cheaper analysis.

The contributions from this work should be transferable to

- Other analyses: Andersen’s analysis is a whole-program analysis consisting of two steps: modeling the code and computing a fixed-point on the model. Several other algorithms follow the same pattern, such as VTA [54], XTA [57], or Das’s one level flow algorithm [15]. Algorithms that do not require the second step, such as CHA [16,20] or Steensgaard’s unification-based algorithm [52], are easier to perform in an online setting. Andersen’s analysis is flow-insensitive and context-insensitive. While this paper should also be helpful for performing flow-sensitive or context-sensitive analyses online, these pose additional challenges (multithreading and exceptions, and multiple calling contexts) that need to be addressed.
- Other languages: This paper shows how to deal with dynamic class loading, reflection, and native code in Java. Other languages have similar features, which pose similar problems for pointer analysis.

2 Motivation

Java features such as dynamic class loading, reflection, and native methods prohibit static whole-program analyses. This paper identifies all Java features that create challenges for pointer analysis; this section focuses just on class loading, and discusses why it precludes static analysis.

2.1 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 [56] and Tomcat [55].

2.2 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 invoking `Class.forName(String name)`, where *name* can be computed at runtime. For example, a program may compute the localized calendar class name by reading an environment variable. 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 only in the execution environment, and not in the development environment. Only an online analysis could analyze such a program.

2.3 It Is Not Known Statically When a Given Class Will Be Loaded

If the classes to be analyzed are available only in the execution environment, but `Class.forName` is not used, 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 `Constants.CAPACITY` has the illegal value `-1`. The effect, an `ExceptionInInitializerError`, should only become visible when the class containing the static field is loaded. Furthermore, *hashMap* should be initialized after `CAPACITY`, to ensure that the latter receives the correct value. Loading classes eagerly and still preserving the proper (lazy) class loading semantics is challenging.

2.4 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 executing code 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: }

```

Fig. 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 [30]. At one extreme, exemplified by Steensgaard [52] and type-based analyses [18,25,57], the analyses are fast, but imprecise. At the other extreme, exemplified by shape analyses [29, 49], 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 choose a well-known analysis and to extend it to handle all features of Java. This goal was motivated by our need to build a pointer analysis to support connectivity-based garbage collection, for which type-based analyses are too imprecise [32]. 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. This left us with a choice between Steensgaard’s [52] and Andersen’s [3] analysis. Andersen’s analysis is less efficient, but more precise [31,50]. We decided to use Andersen’s analysis, because it poses a superset of the Java-specific challenges posed by Steensgaard’s analysis, leaving the latter (or points in between) as a fall-back option.

3.2 Online Interprocedural Analyses

An *online* interprocedural analysis is an interprocedural analysis that occurs during execution, and thus, can correctly deal with dynamic class loading.

3.2.1 Demand-driven interprocedural analyses. A number of pointer analyses are demand-driven, but not online [1,9,10,27,38,59]. All of these analyses 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 issues specific to dynamic class loading.

3.2.2 Incremental interprocedural analyses. Another related area of research is incremental interprocedural analysis [8,14,23,24]. 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. This paper differs in that it focuses on the dynamic semantics of the Java programming language, not programmer modifications to the source code.

3.2.3 Extant analysis. Sreedhar, Burke, and Choi [51] 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.

3.2.4 Analyses that deal with dynamic class loading.

Below, we discuss some analyses that deal with dynamic class loading. None of these analyses deals with reflection or JNI, or validate their analysis results. Furthermore, all are less precise than Andersen’s analysis.

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 [7] and King [36] adapt Ruf’s escape analysis [48] to deal with dynamic class loading. Ruf’s analysis is unification-based, and thus less precise than Andersen’s analysis. Escape analysis is a simpler problem than pointer analysis because the impact of a method is independent of its parameters and the problem doesn’t require a unique representation for each heap object [11]. Bogda and Singh discuss tradeoffs of when to trigger the analysis, and whether to make optimistic or pessimistic assumptions for optimization. King 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 [6].

Qian and Hendren [45], in work concurrently with ours, adapt Tip and Palsberg’s XTA [57] to deal with dynamic class loading. The main contribution of their paper is a low-overhead call edge profiler, which yields a precise call graph on which XTA is based. Even though XTA is weaker than Andersen’s analysis, both have separate constraint generation and constraint propagation steps, and thus pose similar problems. Qian and Hendren solve the problems posed by dynamic class loading similarly to the way we solve them; for example, their approach to unresolved references is analogous to our approach in Section 4.5.

3.3 Andersen’s Analysis for Static Java

A number of papers describe how to use Andersen’s analysis for Java [39,40,47,60]. 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 [47] 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 [60] improve the efficiency of Andersen’s analysis by using implementation techniques from CLA [28], 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 [58], 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 4.2.1.

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 [18,37,41].

4 Algorithm

Section 4.1 presents the architecture for performing Andersen’s pointer analysis online. The subsequent sections discuss parts of the architecture that deal with: constraint finding (4.2), call graph building (4.3), constraint propagation (4.4), type resolution (4.5), and other constraint generating events (4.6).

4.1 Architecture

As mentioned in Section 1, Andersen’s algorithm has two steps: finding the constraints that model the code semantics of interest, and propagating these constraints until a fixed point is reached. In an offline setting, the first step requires a scan of the program and its call graph. In an online setting, this step is more complex, because parts of the program are “discovered” during execution of various VM events. Figure 2 shows the architecture for performing Andersen’s pointer analysis online. The events during virtual machine execution (left column) generate inputs to the analysis. The analysis (dotted box) consists of four components (middle column) that operate on shared data structures (right column). Clients (bottom) trigger the constraint propagator component of the analysis, and consume the outputs. The outputs are represented as points-to sets in the constraint graph. In an online setting, the points-to sets conservatively describe the pointers in the program until there is an addition to the constraints.

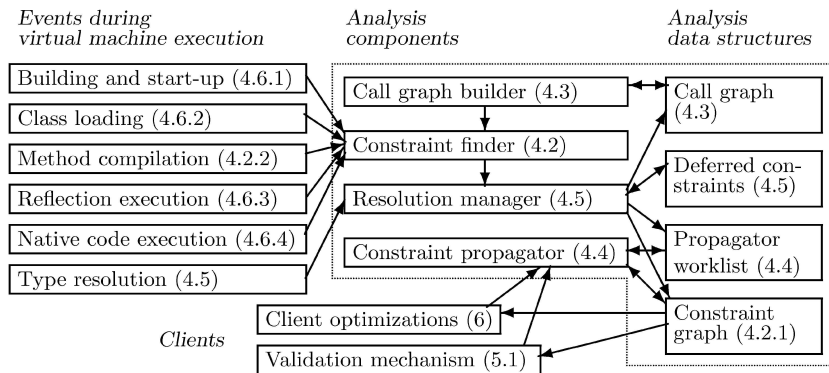


Fig. 2. Architecture for performing Andersen’s pointer analysis online. The numbers in parentheses refer to sections in this paper.

When used offline, Andersen’s analysis requires only a part of the architecture in Figure 2. In an offline setting, the only input comes from method compilation. It is used by the constraint finder and the call graph builder to create a constraint graph. After that, the constraint propagator finds a fixed-point on the constraint graph. The results are consumed by clients.

Four additions to the architecture make Andersen’s analysis work online:

Building the call graph online. Andersen’s analysis relies on a call graph for interprocedural constraints. This paper uses an online version of CHA (class hierarchy analysis [16,20]) for the call graph builder. CHA is an offline whole-program analysis, Section 4.3 describes how to make it work online.

Supporting re-propagation. Method compilation and other constraint-generating events happen throughout the execution. Where an offline analysis can propagate once after all constraints have been found, the online analysis has to propagate whenever a client needs points-to information and new constraints have been created since the last propagation. Section 4.4 describes how the propagator starts with its previous solution and a worklist of changed parts in the constraint graph to avoid incurring the full propagation cost every time.

Supporting unresolved types. The constraint finder may find constraints that involve as-yet unresolved types. But both the call graph builder and the propagator rely on resolved types for precision; for example, the propagator filters points-to sets by types. Section 4.5 describes how the resolution manager defers communicating constraints from the constraint finder to other analysis components until the involved types are resolved.

Capturing more input events. A pointer analysis for Java has to deal with features such as reflection and native code, in addition to dynamic class loading. Section 4.6 describes how to handle all the other events during virtual machine execution that may generate constraints.

4.2 Constraint Finder

Section 4.2.1 describes the constraint graph data structure, which models the data flow of the program. Section 4.2.2 describes how code is translated into constraints at method compilation time. Our approach to representing the constraint graph and analyzing code combines ideas from various earlier papers on offline implementation of Andersen’s analysis.

4.2.1 Constraint graph. The constraint graph has four kinds of nodes that participate in constraints. The constraints are stored as sets at the nodes. Table 1 describes the nodes, introducing the notation that is used in the remainder of this paper, and shows which sets are stored at each node. The node kinds in “[...]” are the kinds of nodes in the set.

Table 1. Constraint graph representation.

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

Flow-to sets (Column 3 of Table 1) 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 $v \in \text{flowFrom}(v'.f)$.

Points-to sets (Column 4 of Table 1) 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. Since it stores points-to sets with $h.f$ -nodes instead of $v.f$ -nodes, the analysis is *field sensitive* [39].

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 has a map from fields f to $h.f$ -nodes (i.e., the nodes that represent the instance fields of the objects represented by the h -node). In addition to language-level fields, each h -node has a special node $h.f_{td}$ that represents the field containing the reference to the type descriptor for the heap node. A type descriptor is implemented as an object in Jikes RVM, and thus, must be modeled by the analysis. For each h -node representing arrays of references, there is a special node $h.f_{elems}$ that represents all of their elements. Thus, the analysis does not distinguish between different elements of an array.

There are many alternatives for storing 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 [22,47]. Thus, our analysis saves space compared to BANE, but may have to perform more work at propagation time. As another example, CLA [28] 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, to obtain efficient propagation, our representation uses redundant flow-from sets.

Finally, there are many choices for how to implement the sets. The SPARK paper evaluates various data structures for representing points-to sets [39], finding that hybrid sets (using lists for small sets, and bit-vectors for large sets) yield the best results. We found the shared bit-vector implementation from CLA [26] to be even more efficient than the hybrid sets used by SPARK.

4.2.2 Method compilation. The left column of Figure 2 shows the various events during virtual machine execution that invoke the constraint finder. This section is only concerned with finding intraprocedural constraints during method compilation; later sections discuss other kinds of events.

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

Column “Actions” in Table 2 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.

Table 2. Intraprocedural constraint finder.

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

In addition to the actions in Table 2, the analysis needs to address some more issues during method compilation.

4.2.2.1 Unoptimized code. The intraprocedural constraint finder is implemented as a pass of the Jikes RVM optimizing compiler. However, Jikes RVM compiles some methods only with a baseline compiler, which does not use a representation that is amenable to constraint finding. We handle such methods by running the constraint finder as part of a truncated optimizing compilation. Other virtual machines, where some code is not compiled at all, but interpreted, can take a similar approach.

4.2.2.2 Recompile of methods. Many JVMs, including Jikes RVM, may recompile a method (at a higher optimization level) if it executes frequently. The recompiled methods may have new variables or code introduced by optimizations (such as inlining). Since each inlining context of an allocation site is modeled by a separate h -node, the analysis generates new constraints for the recompiled methods and integrates them with the constraints for any previously compiled versions of the method.

4.2.2.3 Magic. Jikes RVM has some internal “magic” operations, for example, to allow direct manipulation of pointers. The compilers expand magic in special ways directly into low-level code. Likewise, the analysis expands magic in special ways directly into constraints.

4.3 Call Graph Builder

For each call-edge, the analysis generates constraints that model the data flow through parameters and return values. Parameter passing is modeled as a move

from actuals (at the call-site) to formals (of the callee). Each return statement in a method m is modeled as a move to a special v -node $v_{\text{retval}(m)}$. The data flow of the return value to the call-site is modeled as a move to the v -node that receives the result of the call.

We use CHA (Class Hierarchy Analysis [16,20]) 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]. In work concurrent with ours, Qian and Hendren developed an even more precise alternative based on low-overhead profiling [45].

CHA is a static whole-program analysis, but to support Andersen’s analysis online, CHA must also run online, i.e., deal with dynamic class loading. The key to solving this problem is the observation that for each call-edge, either the call-site is compiled first, or the callee is compiled first. The constraints for the call-edge are added when the second of the two is compiled. This works as follows:

- When encountering a method $m(v_{\text{formal}_1(m)}, \dots, v_{\text{formal}_n(m)})$, the call graph builder
 - creates a tuple $I_m = \langle v_{\text{retval}(m)}, v_{\text{formal}_1(m)}, \dots, v_{\text{formal}_n(m)} \rangle$ for m as a callee,
 - finds 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 $c : v_{\text{retval}(c)} = m(v_{\text{actual}_1(c)}, \dots, v_{\text{actual}_n(c)})$, the call graph builder
 - creates a tuple $I_c = \langle v_{\text{retval}(c)}, v_{\text{actual}_1(c)}, \dots, v_{\text{actual}_n(c)} \rangle$ 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 eliminates many of these possibilities later on. One could easily imagine making this more precise, for example by assuming that throws can only reach catch clauses in the current method or its (transitive) callers.

4.4 Constraint Propagator

The propagator propagates points-to sets following the constraints that are implicit in the flow sets until the points-to sets reach a fixed point. In order to avoid

wasted work, our algorithm maintains two pieces of information, a worklist of v -nodes and isCharged-bits on $h.f$ -nodes, that enable it to propagate only the changed points-to sets at each iteration (rather than propagating all points-to sets). The worklist contains v -nodes whose points-to sets have changed and thus need to be propagated, or whose flow sets have changed and thus the points-to sets need to be propagated to additional nodes. The constraint finder initializes the worklist.

The algorithm in Figure 3, which is a variation of the algorithm from SPARK [39], implements the constraint propagator component of Figure 2.

```

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

```

Fig. 3. Constraint propagator

The propagator puts a v -node on the worklist when its points-to set changes. 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 the algorithm that iterates over all ($v.f$ -) nodes: all other parts of the algorithm attempt to update points-to sets while visiting only nodes that are relevant to the points-to sets being updated.

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

The propagator performs on-the-fly filtering by types: it only adds 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 $v.\text{pointsTo} = \{h_A, h_B, h_C\}$. 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 [46]. They yielded only modest performance improvements compared to shared bit-vectors [26] 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 (à la BANE [19]).

4.5 Resolution Manager

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 the analysis.

First, the CHA (class hierarchy analysis) that implements the call graph builder 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, the call graph builder adds an edge 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. Therefore, the resolution manager defers all flow sets and points-to sets involving nodes of unresolved types, thus hiding them from the propagator:

- 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 are unresolved, it defers the information for later instead. Table 3 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.

Table 3. 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

- 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. Qian and Hendren developed a similar design independently [45].

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 large points-to sets, which in turn slowed down our analysis prohibitively. Our current approach is both more precise and more efficient.

4.6 Other Constraint-Generating Events

This section discusses the remaining events in the left column of Figure 2 that serve as inputs to the constraint finder.

4.6.1 VM building and start-up. Jikes RVM itself is written in Java, and begins execution 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 run-time services. These objects live in the same heap as application objects, so our analysis must model them.

Our analysis models all the *code* in the boot image as usual, with the intraprocedural constraint finder pass from Section 4.2.2 and the call graph builder from Section 4.3. Our analysis models the *data snapshot* of the boot image with special boot image *h*-nodes, and with points-to sets of global *v*-nodes and boot image *h.f*-nodes. The program that creates the boot image does not maintain a mapping from objects in the boot image to their actual allocation site, and thus, the boot image *h*-nodes are not allocation sites, instead they are synthesized at boot image writing time. Finally, the analysis propagates on the combined constraint system. This models how the snapshot of the data in the boot image may be manipulated by future execution of the code in the boot image.

Our techniques for correctly handling the boot image can be extended to form a general hybrid offline/online approach, where parts of the application are analyzed offline (as the VM is now) and the rest of the application is handled by the online analysis presented in this work. Such an approach could be useful for applications where the programmer asserts no use of the dynamic language features in parts of the application.

4.6.2 Class loading. Even though much of this paper revolves around making Andersen’s analysis work for dynamic class loading, most analysis actions actually happen during other events, such as method compilation or type resolution. The only action that does take place exactly at class loading time is that the constraint finder models the `ConstantValue` bytecode attribute of static fields with constraints [42, Section 4.5].

4.6.3 Reflection execution. Java programs can invoke methods, access and modify fields, and instantiate objects using reflection. Although approaches such as String analysis [12] could predict which entities are manipulated in special cases, this problem is undecidable in the general case. Thus, 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 that 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 [39,60].

Our solution is to handle reflection when the code is actually executed. We instrument the virtual machine service that handles reflection with code that adds constraints dynamically. For example, if reflection stores into a field, the constraint finder observes the actual source and target of the store and generates a constraint that captures the semantics of the store at that time.

This strategy for handling reflection introduces new constraints when the reflective code does something new. Fortunately, that does not happen very

often. When reflection has introduced new constraints and a client needs up-to-date points-to results, it must trigger a re-propagation.

4.6.4 Native code execution. 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.

Our approach is to be imprecise, but conservative, for return values from JNI methods, while being precise for data manipulation by JNI methods. If a JNI method returns a heap allocated object, the constraint finder assumes that it could return an object from any allocation site. This is imprecise, but easy to implement. The constraint propagation uses type filtering, and thus, will filter the set of heap nodes returned by a JNI method based on types. If a JNI 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, JNI methods that make calls or manipulate object fields are handled precisely by our mechanism for reflection.

5 Validation

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

5.1 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 4.4) just before GC starts. As there is no memory available to grow points-to sets at that time, 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 add 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. This extra word is only used for validation runs; the pointer analysis does not require any change to the object header.

5.2 Validation Anecdotes

Our validation methodology helped us find many bugs, some of which were quite subtle. Below are two examples. In both cases, 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.

5.2.1 Field reference class. 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 static field, and had 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 superclass was missing some points-to set elements. That resulted in warnings from our validation methodology. Upon investigating those warnings, we became aware of the incorrect assumption and fixed it.

5.2.2 Field initializer attribute. 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 (Section 4.6.2).

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 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 [4,13,43,53] typically use a dual execution strategy, where each method is initially either interpreted or compiled without optimizations. No inlining is performed for such methods. Later, an optimizing compiler that may perform inlining recompiles the minority of frequently executing methods. Because inlining is not performed during the initial execution, our analysis does not need to propagate constraints 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 the left column of Figure 2, an inlining client must be prepared to invalidate inlining decisions. Techniques such as code patching [13] and on-stack replacement [21,34] support invalidation. If instant invalidation is needed, our analysis must repropagate 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 [5,35], that would allow our analysis to be lazy about repropagating after it finds new constraints.

CBGC (connectivity-based garbage collection) is a new garbage collection technique that requires pointer analysis [32]. 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 cross-partition pointers. CBGC exploits the observation that connected objects tend to die together [33], 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 efficiency of our pointer analysis implementation in Jikes RVM 2.2.1. Prior work (e.g., [39]) has evaluated the precision of Andersen’s analysis. 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), a stack walk determines whether the yield-point belongs to analysis or application code, and counts it accordingly. We performed all experiments on a 2.4GHz Pentium 4 with 2GB of memory running Linux, kernel version 2.4.

Since Andersen’s analysis has cubic time complexity and quadratic space complexity (in the size of the code), optimizations that increase the size of the code can dramatically increase the constraint propagation time. In our experi-

ence, aggressive inlining can increase constraint propagation time by up to a factor of 5 for our benchmarks. In default mode, Jikes RVM performs inlining (and optimizations) only inside the hot application methods, but is more aggressive about methods in the boot image. We force Jikes RVM to be more cautious about inlining inside boot image methods by using a `FastAdaptiveMarkSweep` image and disabling inlining at build time. During benchmark execution, Jikes RVM does, however, perform inlining for hot boot image methods when recompiling them.

7.1 Benchmark Characteristics

Table 4 describes our benchmark suite; *null* is a dummy benchmark with an empty main method. Column “Analyzed methods” gives the number of methods analyzed. 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 the benchmark’s methods, library methods called by the benchmark, and methods belonging to Jikes RVM itself. 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. Finally, Column “Run time” gives the run time for our benchmarks using our configuration of the Jikes RVM.

Table 4. Benchmark programs.

Program	Command line arguments	Analyzed methods	Loaded classes	Run time
<i>null</i>	<i>none</i>	15,598	1,363	1s
javalex	<code>qb1.lex</code>	15,728	1,389	37s
compress	<code>-m1 -M1 -s100</code>	15,728	1,391	14s
db	<code>-m1 -M1 -s100</code>	15,746	1,385	28s
mtrt	<code>-m1 -M1 -s100</code>	15,858	1,404	14s
mpegaudio	<code>-m1 -M1 -s100</code>	15,899	1,429	27s
jack	<code>-m1 -M1 -s100</code>	15,962	1,434	21s
richards	<i>none</i>	15,963	1,440	4s
hsqldb	<code>-clients 1 -tpc 50000</code>	15,992	1,424	424s
jess	<code>-m1 -M1 -s100</code>	16,158	1,527	29s
javac	<code>-m1 -M1 -s100</code>	16,464	1,526	66s
xalan	<code>1 1</code>	17,057	1,716	10s

The Jikes RVM methods and classes account for a significant portion of the code in our benchmarks. Thus, our analysis has to deal with much more code than it would have to in a JVM that is not written in Java. On the other hand, writing the analysis itself in Java had significant software engineering benefits;

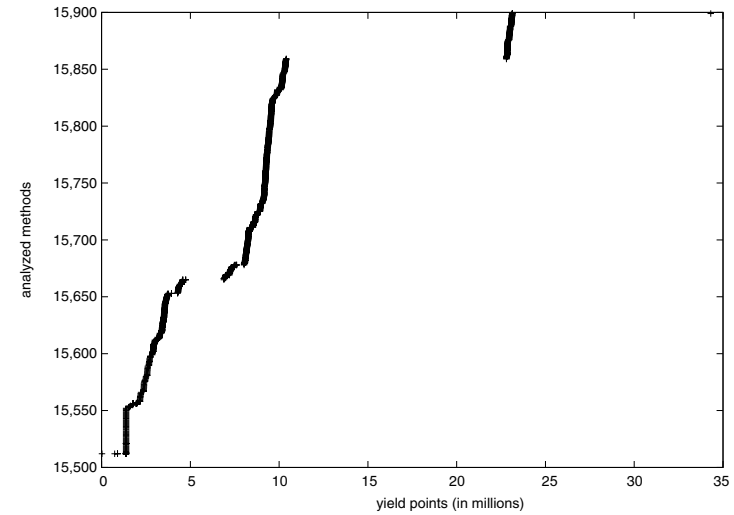


Fig. 4. Yield-points versus analyzed methods for *mpegaudio*. The first shown data point is the `main()` method.

for example, the analysis relies on garbage collection for its data structures. In addition, the absence of artificial boundaries between the analysis, other parts of the runtime system, and the application exposes more opportunities for optimizations. Current trends show that the benefits of writing system code in a high-level, managed, language are gaining wider recognition. For example, Microsoft is pushing towards implementing more of Windows in managed code.

Figure 4 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 15,500: 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.

From Figure 4, we see that there are two significant stages (around the 10 and 25 million yield-point marks) when the application is executing only methods that it has encountered before. At other times, the application encounters new methods as it executes. We expect that for longer running benchmarks (e.g., a webserver that runs for days), the number of analyzed methods stabilizes after a few minutes of run time. 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. We define *eager* propagation to be propagation after every event from the left column of Figure 2, if it generated new constraints. We define *lazy* propagation to be propagation that occurs just once at the end of the program execution.

7.2.1 Cost in space. Table 5 shows the total allocation for our benchmark runs. Column “No analysis” gives the number of megabytes allocated by the program without our analysis. Column “No propagation” gives the allocation when the analysis generates, but does not propagate, constraints. Thus, this column gives the space overhead of just representing the constraints. Columns “Eager”, “Lazy”, and “At GC” give the allocation when using eager, lazy, and at GC propagation. The difference between these and the “No propagation” column represents the overhead of representing the points-to sets. Sometimes we see that doing more work actually reduces the amount of total allocation (e.g., *mpegaudio* allocates more without any analysis than with lazy propagation). This phenomenon occurs because our analysis is interleaved with the execution of the benchmark program, and thus the Jikes RVM adaptive optimizer optimizes different methods with our analysis than without our analysis.

Table 5. Total allocation (in megabytes)

Benchmark	Eager	At GC	Lazy	No propagation	No analysis
<i>null</i>	48.5	48.1	48.8	13.5	9.7
<i>javalex</i>	621.7	104.7	110.6	70.0	111.8
<i>compress</i>	416.2	230.0	167.0	129.3	130.2
<i>db</i>	394.4	213.8	151.0	112.7	113.6
<i>mtrt</i>	721.9	303.8	240.5	201.5	172.9
<i>mpegaudio</i>	755.9	145.8	83.1	42.8	137.0
<i>jack</i>	1,782.4	418.4	354.8	309.2	322.8
<i>richards</i>	1,117.8	61.3	67.7	26.6	12.6
<i>hsq1</i>	4,047.0	3,409.6	3,343.8	3,291.1	3,444.6
<i>jess</i>	4,694.8	458.0	394.4	341.4	398.3
<i>javac</i>	2,023.0	450.4	381.3	328.2	429.3
<i>xalan</i>	6,074.9	166.4	200.4	131.5	37.6

Finally, since the boot image needs to include constraints for the code and data in the boot image, our analysis inflates the boot image size from 31.5 megabytes to 73.4 megabytes.

Table 6. Percent of execution time in constraint finding

Program	Analyzing methods	Resolving classes and arrays
<i>null</i>	69.16%	3.68%
<i>javalex</i>	2.02%	0.39%
<i>compress</i>	5.00%	1.22%
<i>db</i>	1.77%	0.39%
<i>mtrt</i>	7.68%	1.70%
<i>mpegaudio</i>	6.23%	6.04%
<i>jack</i>	6.13%	2.10%
<i>richards</i>	21.98%	5.88%
<i>hsq1</i>	0.29%	0.09%
<i>jess</i>	5.59%	1.24%
<i>javac</i>	3.20%	1.60%
<i>xalan</i>	26.32%	8.66%

7.2.2 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. Table 6 shows that generating constraints for methods is the dominant part of constraint generation. Also, as the benchmark run time increases, the percentage of time spent in constraint generation decreases. For example, the time spent in constraint finding is a negligible percentage of the run time for our longest running benchmark, *hsq1*.

7.2.3 Cost of propagation. Table 7 shows the cost of propagation. Columns “Count” give the number of propagations that occur in our benchmark runs. Columns “Time” give the arithmetic mean \pm 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.

Table 7 shows that the mean pause time due to eager propagation varies between 3.8 and 16.8 seconds for the real benchmarks. In contrast, a full (lazy) propagation is much slower. Thus, 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 propagation cost did not depend on which of the events in the left column of Figure 2 generated new constraints that were the reason for the propagation.

Figure 5 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*, 524 propagations take under 1 second. The remaining propagations are much more expensive (10 seconds or more), thus increasing the average. We

Table 7. Propagation statistics (times in seconds)

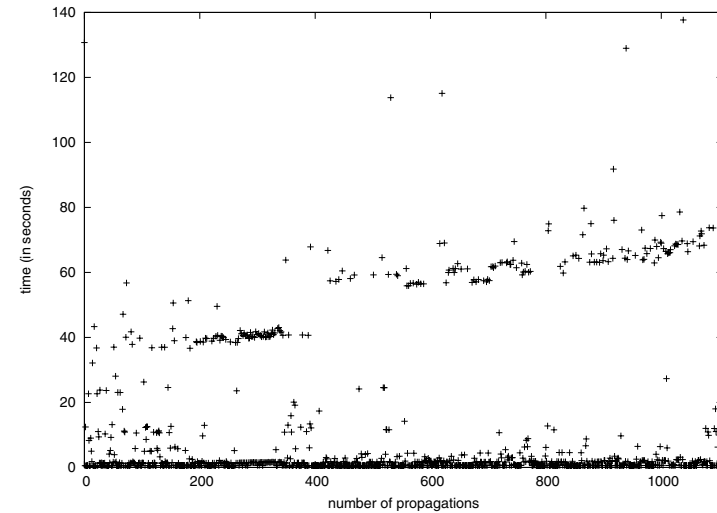
Program	Eager		At GC		Lazy	
	Count	Time	Count	Time	Count	Time
<i>null</i>	1	135.6±0.0	1	120.7±0.0	1	137.8±0
javalex	166	13.6±22.0	1	120.7±0.0	1	158.4±0
compress	127	8.6±18.7	3	104.7±23.6	1	142.8±0
db	140	10.0±20.2	3	106.1±24.8	1	144.5±0
mtrt	262	5.5±14.4	3	106.8±24.7	1	148.0±0
mpegaudio	317	5.5±13.4	3	105.4±24.1	1	144.3±0
jack	392	10.9±17.8	3	114.4±33.8	1	161.8±0
richards	410	3.8±10.9	1	120.8±0.0	1	134.8±0
hsqldb	391	10.1±20.6	6	76.6±94.8	1	426.7±0
jess	734	16.8±20.5	3	117.7±38.5	1	182.4±0
javac	1,103	12.5±22.9	5	114.3±97.6	1	386.7±0
xalan	1,726	11.2±21.4	1	120.5±0.0	1	464.6±0

also discern that more expensive propagations occur later in the execution. The omitted graphs for other benchmarks have a similar shape. Although we present the data for eager propagation, clients of our analysis do not necessarily require eager propagation (Section 6).

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.

7.3 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. On profiling our analysis, we found that the worklist part (lines 2 to 19 in Figure 3) takes up far more of the propagation time than the iterative part (lines 20 to 26 in Figure 3). Thus, in our future work, we will first focus on the worklist part to improve propagator performance.

**Fig. 5.** Propagation times for *javac* (eager).

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.

References

1. G. Agrawal, J. Li, and Q. Su. Evaluating a demand driven technique for call graph construction. In *Internat. Conference on Compiler Construction (CC)*, 2002.
2. 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.
3. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994. DIKU report 94/19.
4. 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.

5. 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 Prog. (ECOOP)*, 2002.
6. 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.
7. J. Bogda and A. Singh. Can a shape analysis work at run-time? In *Java Virtual Machine Research and Technology Symp. (JVM)*, 2001.
8. M. Burke and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *Trans. on Prog. Lang. and Systems (TOPLAS)*, 1993.
9. R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *Principles of Prog. Lang. (POPL)*, 1999.
10. 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.
11. 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.
12. A. S. Christensen, A. Möller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Static Analysis Symposium (SAS)*, 2003.
13. M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Prog. Lang. Design and Impl. (PLDI)*, 2000.
14. K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *Trans. on Prog. Lang. and Systems (TOPLAS)*, 1986.
15. M. Das. Unification-based pointer analysis with directional assignments. In *Prog. Lang. Design and Impl. (PLDI)*, 2000.
16. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference for Object-Oriented Prog. (ECOOP)*, 1995.
17. D. Detlefs and O. Agesen. Inlining of virtual methods. In *European Conference for Object-Oriented Prog. (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. T. Harris. Early storage reclamation in a tracing garbage collector. *ACM SIG-PLAN Notices*, 1999.
26. N. Heintze. Analysis of large code bases: The compile-link-analyze model. <http://cm.bell-labs.com/cm/cs/who/nch/cla.ps>, 1999.
27. N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Prog. Lang. Design and Impl. (PLDI)*, 2001.
28. 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.
29. L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell University, 1990.
30. M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.
31. M. Hind and A. Pioli. Which pointer analysis should I use? In *Internat. Symp. on Software Testing and Analysis (ISSTA)*, 2000.
32. M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *Obj.-Oriented Prog., Systems, Lang., and Applic. (OOPSLA)*, 2003.
33. M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *Internat. Symp. on Memory Management (ISMM)*, 2002.
34. U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Prog. Lang. Design and Impl. (PLDI)*, 1992.
35. U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Prog. Lang. Design and Impl. (PLDI)*, 1994.
36. 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.
37. J. R. Larus and S. Chandra. Using tracing and dynamic slicing to tune compilers. University of Wisconsin Technical Report 1174, Aug. 1993.
38. 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.
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-insensitive 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. F. Qian and L. Hendren. Towards dynamic interprocedural analysis in JVMs. In *Java Virtual Machine Research and Technology Symp. (JVM)*, 2004.
46. A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Prog. Lang. Design and Impl. (PLDI)*, 2000.
47. 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.
48. E. Ruf. Effective synchronization removal for Java. In *Prog. Lang. Design and Impl. (PLDI)*, 2000.

49. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Principles of Prog. Lang. (POPL)*, 1999.
50. M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Static Analysis Symp. (SAS)*, 1997.
51. 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.
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. V. Sundaresan, L. J. Hendren, C. Razafimahefa, V.-R. Raja, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Obj.-Oriented Prog., Systems, Lang., and Applic. (OOPSLA)*, 2000.
55. The Apache Tomcat Project. Apache Tomcat.
<http://jakarta.apache.org/tomcat>.
56. The Eclipse Project. Eclipse. <http://www.eclipse.org>.
57. F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Obj.-Oriented Prog., Systems, Lang., and Applic. (OOPSLA)*, 2000.
58. 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 Prog. (ECOOP)*, 2000.
59. F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Prog. Lang. Design and Impl. (PLDI)*, 2001.
60. J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Static Analysis Symp. (SAS)*, 2002.