**RC23638 (W0506-058) June 10, 2005**
**Computer Science**

# IBM Research Report

## Fast Online Pointer Analysis

**Martin Hirzel, Daniel von Dincklage*, Amer Diwan*, Michael Hind**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

*University of Colorado
Boulder, CO 80309

**Research Division**
**Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich**

# Fast Online Pointer Analysis

MARTIN HIRZEL
IBM Research
DANIEL VON DINCKLAGE and AMER DIWAN
University of Colorado
MICHAEL HIND
IBM Research

Many optimizations need information about points-to relationships to be effective. Pointer analysis infers such information from program code. Unfortunately, some common programming language features, such as dynamic linking, reflection, and foreign function interfaces, make pointer analyses difficult. For example, prior pointer analyses for the Java language either ignore these features or are overly conservative. To deal with dynamic linking, pointer analysis must run online, as the program is executing. This paper presents the first non-trivial online pointer analysis.

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. Our analysis is fast: on average over our benchmark suite, if the analysis recomputes points-to results upon each program change, most analysis pauses take under 0.1 seconds.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Pointer analysis, class loading, reflection, native interface

## 1. INTRODUCTION

The results of a pointer analysis can make many optimizations more effective. Specifically, any optimization that depends on how heap objects are referenced, such as inlining, load elimination, code movement, stack allocation, and parallelization, can benefit from pointer analysis results.

However, modern programming languages, like Java, have features, like dynamic class loading, that inhibit traditional pointer analysis. All Java programs use dynamic class loading, and many depend on it for program extensibility at runtime. For example, Eclipse [The Eclipse Project ] uses a plug-in architecture that uses dynamic class loading to load features as needed. Dynamic class loading is a widely used feature that can not be ignored.

All previous non-trivial pointer analyses are offline analyses. They are not applicable to general Java applications, because they assume that the entire code base is available ahead of time. Instead, they only handle a subset of Java, and are unsound for some Java features. Unsoundness is acceptable in situations where the advantages of the analyses outweigh the disadvantages of incorrect analysis results. For example, many software engineering tools can benefit from pointer analysis results even when they are unsound. But unsound pointer analyses can not support optimizations that rely on sound points-to information.

This paper presents the first non-trivial pointer analysis that works for all of Java. It describes how to perform Andersen's pointer analysis [1994] online in the general setting of an executing Java virtual machine. Thus, the benefits of points-to information become available to optimizations in the JIT compilers and other components of the language runtime system. This paper

—identifies all problems of performing Andersen's pointer analysis for the full Java language,

—presents a solution for each of the problems,

—reports on a full implementation of the solutions in Jikes RVM, an open-source research virtual machine from IBM [Alpern et al. 2000],

—discusses how to optimize the implementation to make it fast,

—validates, for our benchmark runs, that the analysis yields correct results, and

—evaluates the efficiency of the implementation.

In a previous paper on our analysis [Hirzel et al. 2004], we found that the implementation was efficient enough for stable long-running applications, but too inefficient for the general case. Because Jikes RVM, which is itself written in Java, leads to a large code base even for small benchmarks, and because Andersen's analysis has time complexity cubic in the code size, obtaining fast pointer analysis in Jikes RVM is challenging. This paper improves analysis time by almost two orders of magnitude compared to our previous paper. On average over our benchmark suite, if the analysis recomputes points-to results upon each program change, most analysis pauses take under 0.1 seconds.

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 [Sundaresan et al. 2000], XTA [Tip and Palsberg 2000], or Das's one level flow algorithm [2000]. Algorithms that do not require the second step, such as CHA [Fernández 1995; Dean et al. 1995] or Steensgaard's unification-based algorithm [1996], are easier to perform in an online setting. Andersen's analysis is flow-insensitive and context-insensitive. Although this paper should also be helpful for performing flow-sensitive or context-sensitive analyses online, these analyses pose additional

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 preliminary version of parts of this paper appeared in *ECOOP '04*.
Authors' addresses: M. Hirzel and M. Hind, IBM T. J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA; {hirzel,hindm}@us.ibm.com. D. von Dincklage and A. Diwan, University of Colorado, Boulder, CO 80309, USA; {danielvd,diwan}@colorado.edu.

challenges that need to be addressed. For example, correctly dealing with exceptions is more difficult in a flow-sensitive analysis than in a flow-insensitive analysis [Choi et al. 1999].

—Other languages: This paper shows how to deal with dynamic class loading, reflection, and native code in Java. Dynamic class loading is a form of dynamic linking, and we expect our solutions to be useful for other forms of dynamic linking, such as DLLs. Reflection is becoming a commonplace language feature, and we expect our solutions for Java reflection to be useful for reflection in other languages. The Java native interface is a form of foreign function interface, and we expect our solutions to be useful for foreign function interfaces of other languages.

Section 2 motivates the need for online analysis. Section 3 introduces abstractions and terminology. Section 4 describes an offline version of Andersen's analysis that we use as the starting point for our online analysis. Section 5 explains how to turn it into an online analysis, and Section 6 shows how to optimize its performance. Section 7 discusses implementation issues, including how to validate that the analysis yields sound results, and how to use the analysis results. Section 8 evaluates the performance of our analysis experimentally. Section 9 discusses related work, and Section 10 concludes.

## 2. MOTIVATION

Dynamic linking prevents static interprocedural analysis. This section explains the problem for the case of dynamic class loading in Java. An analysis running before the program starts executing does not know where classes will be loaded from (2.1), which classes will be loaded (2.2), when they will be loaded (2.3), or even whether they will be loaded (2.4).

### 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 find 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 [The Eclipse Project ] and Tomcat [The Apache Tomcat Project ], to support flexible composition of software components.

### 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 the reflection call 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 the application does not use Class.*forName*, one could imagine avoiding *static* analysis by attempting a whole-program analysis during VM *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 (loading classes as they are needed), but a VM could just pretend to be lazy by showing only the effects of lazy loading, while actually being eager (loading classes before they are really needed). Therefore, one could suggest a "link-time step" for analyzing a program written in the subset of Java without user-defined class loaders or Class.*forName*. This is difficult to engineer in practice, however [Serrano et al. 2000]. Because there is no single point in time where linking happens, one would need a deferral mechanism for various visible effects of class loading. An example for such a visible effect is a static field initialization of the form

$$\textbf{static final int } companySize = SimulationWorld.company.size();$$

Suppose that SimulationWorld.*company* is **null**. The effect, a NullPointerException that materializes as an ExceptionInInitializerError, should only become visible when the class containing the static field is loaded. In fact, if SimulationWorld.*company* is itself a final static variable, then circular dependencies can lead to a situation where the order of class loading determines whether the expression SimulationWorld.*company* is or is not null. 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 Fig. 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 wastes time and space) and less precise (the code in those classes may exhibit behavior never encountered at runtime). In situations like these, online analysis can be both more efficient and more precise.

## 3. BACKGROUND

Section 3.1 describes ways in which pointer analyses abstract data flow and points-to relationships in programs. Section 3.2 gives a concrete example for how pointer

```
 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.

analysis manipulates these abstractions. Section 3.3 defines what online and incremental analyses are.

### 3.1 Abstractions

The goal of pointer analysis is to find all possible targets of all pointer variables and fields. Because the number of pointer variables, fields, and pointer targets is unbounded during execution, a pointer analysis operates on a finite abstraction. This section defines this abstraction. This section elaborates on *what* the analysis finds, subsequent sections give details for *how* it works. While this section enumerates several alternative abstractions, the lists are not intended to be exhaustive.

3.1.1 *Variables.* Variables are locals, parameters, or stack slots of methods, or static fields of classes. A pointer analysis tracks the possible values of pointer variables. The Java type system distinguishes which variables are of reference (i.e. pointer) type: a pointer can not hide in a non-pointer variable or a union. Multiple instances of the locals, parameters, and stack slots of a method can coexist at runtime, one per activation of the method. Pointer analysis abstracts this unbounded number of concrete variables with a finite number of $v$-nodes. Alternatives for variable abstraction include:

(a) For each type, represent all variables of all activations of all methods of that type by one $v$-node (e.g., FTA and CTA in [Tip and Palsberg 2000]).
(b) For each method, represent all variables of all activations of that method by one $v$-node (e.g., MTA and XTA in [Tip and Palsberg 2000]).
(c) For each variable, represent all instances of that variable in all activations of its method by one $v$-node (this is one of the ingredients of a context-insensitive analysis).
(d) Represent each variable by different $v$-nodes based on the context of the activation of its method (this is one of the ways to achieve context-sensitive analysis).

Most of this paper assumes context-insensitive analysis, using Option (c).

3.1.2 *Pointer targets.* Pointer targets are heap objects, in other words, instances of classes or arrays. In Java, there are no pointers to other entities such as stack-allocated objects, and there are no pointers to the interior of objects. Multiple instances of objects of a given type or from a given allocation site can coexist at runtime. Pointer analysis abstracts this unbounded number of concrete heap objects with a finite number of $h$-nodes. Alternatives for heap object abstraction include:

(a) For each type, represent all instances of that type by one $h$-node (type-based analysis).
(b) For each allocation site, represent all heap objects allocated at that allocation site by one $h$-node (allocation-site based analysis).
(c) For each allocation site, represent the heap objects allocated there by different $h$-nodes based on the context of the activation that executes the allocation site (allocation-context based analysis).

Most of this paper assumes allocation-site based analysis, thus using Option (b).

3.1.3 *Fields.* Fields are instance variables or array elements. A field is like a variable, but it is stored inside of a heap object. Just like the analysis is only concerned with variables of reference type, it is only concerned with fields of reference type, and ignores non-pointer fields. Multiple instances of a field can coexist at runtime, because multiple instances of the object it resides in can coexist. Pointer analysis abstracts this unbounded number of fields with a bounded number of nodes. Ignoring arrays for the moment, alternatives for field abstraction for an $h$-node, $h$, and a field, $f$, include:

(a) For a type T, represent all fields of all $h$-nodes of type T by one node (e.g., MTA and CTA in [Tip and Palsberg 2000]).
(b) Ignore the field $f$, and represent all fields of the $h$-node by the $h$-node itself (field-independent analysis).
(c) Ignore the $h$-node $h$, and represent the field $f$ for all $h$-nodes of the type that declares the field by one $f$-node (field-based analysis, e.g., XTA and FTA in [Tip and Palsberg 2000]).
(d) Use one $h.f$-node for each combination of $h$ and $f$ (field-sensitive analysis).

For array elements, the analysis ignores the index, and represents all elements of an array by the same $h.f$-node $h.f_{\text{elems}}$. Most of this paper assumes field-sensitive analysis, thus using Option (d).

3.1.4 *PointsTo sets.* PointsTo sets are sets of $h$-nodes, which abstract the may-point-to relation. Alternatives for where to attach pointsTo sets include:

(a) Attach one pointsTo set to each $v$-node, and one pointsTo set to each $h.f$-node (flow-insensitive analysis).
(b) Have separate pointsTo sets for the same node at different program points (flow-sensitive analysis).

This paper assumes flow-insensitive analysis, thus using Option (a). When the analysis finds that $h \in \text{pointsTo}(v)$, then any of the variables represented by $v$ may point to any of the heap objects represented by $h$. Likewise, when the analysis finds that $h \in \text{pointsTo}(h'.f)$, then the field $f$ of any of the heap objects represented by $h'$ may point to any of the heap objects represented by $h$.

### 3.2 Example

Consider a program with just three statements:

$$1 : x = \textbf{new C}(); \quad 2 : y = \textbf{new C}(); \quad 3 : x = y;$$

In the terminology of Section 3.1, there are two $v$-nodes $v_x$ and $v_y$, and two $h$-nodes $h_1$ and $h_2$. Node $h_1$ represents all objects allocated by Statement 1, and Node $h_2$ represents all objects allocated by Statement 2. From Statements 1 and 2, the analysis produces initial pointsTo sets:

$$\text{pointsTo}(v_x) = \{h_1\}, \quad \text{pointsTo}(v_y) = \{h_2\}$$

But this is not the correct result yet. Statement 3 causes the value of variable $y$ to flow into variable $x$. To model this situation, the analysis also produces flowTo sets:

$$\text{flowTo}(v_x) = \{\}, \quad \text{flowTo}(v_y) = \{v_x\}$$

Formally, these flowTo sets represent subset constraints:

$$v_x \in \text{flowTo}(v_y) \quad \text{represents} \quad \text{pointsTo}(v_y) \subseteq \text{pointsTo}(v_x)$$

In this example, the constraints mean that $x$ points to all targets that $y$ points to (due to Statement 3), but possibly more.

An analysis component called "constraint propagator" propagates pointsTo sets from the subset side to the superset side of constraints until it reaches a fixed-point solution that satisfies the subset constraints. In the running example, it finds:

$$\text{pointsTo}(v_x) = \{h_1, h_2\}, \quad \text{pointsTo}(v_y) = \{h_2\}$$

Now, consider adding a fourth statement, resulting in the program:

$$1 : x = \textbf{new C}(); \quad 2 : y = \textbf{new C}(); \quad 3 : x = y; \quad 4 : x.f = y;$$

Again, the analysis represents statements with flowTo sets:

$$\text{flowTo}(v_x) = \{\}, \quad \text{flowTo}(v_y) = \{v_x, v_x.f\}$$

The node $v_x.f$ represents the contents of fields $f$ of all objects that variable $x$ points to. Due to Statement 4, values flow from variable $y$ to those fields. Therefore, those fields can point to anything that variable $y$ points to. In other words, there are multiple subset constraints:

$$\textbf{for each } h \in \text{pointsTo}(v_x) : \ \text{pointsTo}(v_y) \subseteq \text{pointsTo}(h.f)$$

Since $\text{pointsTo}(v_x) = \{h_1, h_2\}$, the **for each** expands to two new concrete constraints:

$$\text{pointsTo}(v_y) \subseteq \text{pointsTo}(h_1.f), \quad \text{pointsTo}(v_y) \subseteq \text{pointsTo}(h_2.f)$$

For the full set of constraints, the constraint propagator finds the fixed point:

$$\begin{aligned} \text{pointsTo}(v_x) &= \{h_1, h_2\}, & \text{pointsTo}(v_y) &= \{h_2\}, \\ \text{pointsTo}(h_1.f) &= \{h_2\}, & \text{pointsTo}(h_2.f) &= \{h_2\} \end{aligned}$$

### 3.3 Kinds of interprocedural analysis

An *incremental* interprocedural analysis recomputes its results efficiently when the program changes [Cooper et al. 1986; Burke and Torczon 1993; Hall et al. 1993; Grove 1998]. By "efficiently", we mean that computing new results based on previous results must be orders of magnitude cheaper than computing them from scratch. Also, the computational complexity of recomputation must be no worse than the computational complexity of analyzing from scratch. Because incrementality is defined by this efficiency difference, it is not a binary property; rather, there is a continuum of more or less incremental analyses. However, for each incremental analysis, all analysis stages (constraint finding, call graph construction, etc.) must deal with program changes. Prior work on incremental interprocedural analysis focused on programmer modifications to the source code. This paper differs in that it uses incrementality to deal with the dynamic semantics of the Java programming language.

An *online* interprocedural analysis occurs during program execution. In the presence of dynamic linking, incrementality is necessary, but not sufficient, for online analysis. It is necessary because the inputs to the analysis only materialize incrementally, as the program is running. It is not sufficient because online analysis must also interact differently with the runtime system than offline analysis. For example, for languages like Java, an online analysis must deal with reflection, with foreign function interfaces, and with other runtime system issues.

A *modular* interprocedural analysis (e.g., [Chatterjee et al. 1999; Liang and Harrold 1999; Cheng and Hwu 2000]) performs most of its work in an intra-module step, and less work in an inter-module step. By "most" and "less", we mean that the intra-module step must be orders of magnitude more expensive than the inter-module step. Also, the computational complexity of the intra-module step must be no better than the computational complexity of the inter-module step. Modularity is neither necessary nor sufficient for online interprocedural analysis. A modular analysis must also be incremental and interface correctly with the runtime system to be used online.

A *demand-driven* interprocedural analysis (e.g., [Duesterwald et al. 1997; Heintze and Tardieu 2001a; Vivien and Rinard 2001; Agrawal et al. 2002; Lattner and Adve 2003]) attempts to compute just the part of the solution that the client is interested in, rather than the exhaustive solution. Being demand-driven is neither necessary nor sufficient for online interprocedural analysis. A demand-driven analysis must also be incremental and interface correctly with the runtime system to be used online.

If the pointer analysis is to be used for optimizations, then its results must be sound; otherwise the optimizations may change the semantics of the program. If the pointer analysis is to be used for programmer productivity tools, then unsoundness may be tolerable provided the user of the system is aware of it. Since we wish to support both kinds of clients of pointer analysis, we will assume in this paper that the analysis must be sound.

## 4. OFFLINE ANALYSIS

This section describes the offline parts of our version of Andersen's pointer analysis. Due to dynamic class loading, offline analysis does not work for Java, hence subsequent sections describe how to perform it online, in the context of a Java virtual machine.

### 4.1 Offline architecture

Fig. 2 shows the architecture for performing Andersen's pointer analysis offline. In an offline setting, all methods are compiled ahead of time. The call graph builder uses the intermediate representation (IR) from method compilation as input, and creates a call graph. The constraint finder uses the IR as input for creating intraprocedural constraints, and uses the call graph as input for creating interprocedural constraints. The output consists of constraints in the constraint graph that model the code of the program. When the constraint finder is done, the constraint propagator determines the least fixed point of the pointsTo sets of the constraint graph. The propagator uses a worklist to keep track of its progress. The final pointsTo sets in the constraint graph are the output of the analysis to clients.

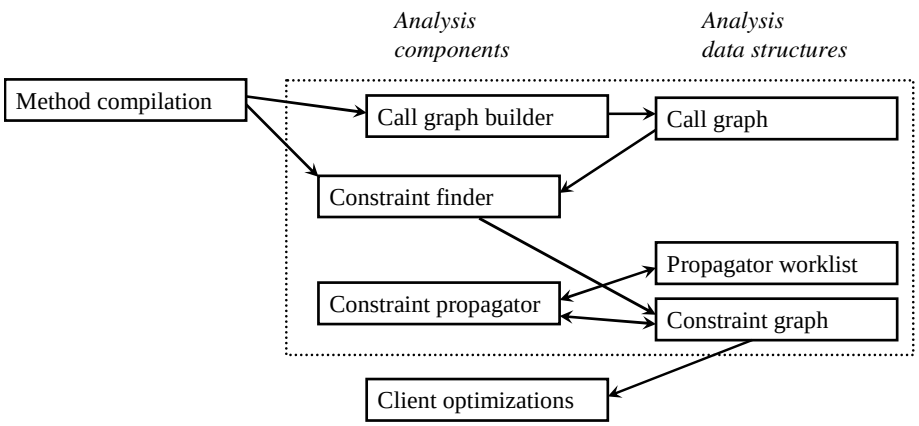



Fig. 2. Offline architecture

### 4.2 Analysis data structures

The call graph models possible method calls (Section 4.2.1). The propagator worklist keeps track of possibly violated constraints (Section 4.2.2). The constraint graph models the effect of program code on pointsTo sets (Section 4.2.3). Our analysis represents pointsTo sets with Heintze's shared bit sets (Section 4.2.4).

4.2.1 *Call graph.* The nodes of the call graph are call sites and callee methods. Each edge of the call graph is a pair of a call site and a callee method that represents a may-call relation. For example, for call site $s$ and method $m$, the call edge $(s, m)$ means that $s$ may call $m$. Due to virtual methods and interface methods, a given call site may have edges to multiple potential callee methods.

Table I. Constraint graph.

| Node kind | Represents concrete entities | PointsTo sets | Flow 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 | pointsTo[$h$] | flowTo[$v$], flowTo[$v.f$] |
| $h.f$-node | Instance field $f$ of all heap objects represented by $h$ | pointsTo[$h$] | none |
| $v.f$-node | Instance field $f$ of all $h$-nodes pointed to by $v$ | none | flowFrom[$v$], flowTo[$v$] |

4.2.2 *Propagator worklist.* The worklist is a list of $v$-nodes that may appear on the left side of unresolved constraints. In other words, if $v$ is in the worklist, then there may be a constraint pointsTo($v$) $\subseteq$ pointsTo($v'$) or a constraint pointsTo($v$) $\subseteq$ pointsTo($v'.f$) that does not hold for the current pointsTo set solution. If that is the case, the propagator has some more work to do to find a fixed point. The work list is a priority queue, so elements are retrieved in topological order. For example, if there is a constraint pointsTo($v$) $\subseteq$ pointsTo($v'$), and no transitive constraints cycle back from $v'$ to $v$, then the worklist returns $v$ before $v'$.

4.2.3 *Constraint graph.* The constraint graph has four kinds of nodes, all of which participate in constraints. The constraints are stored as sets at the nodes. Table I describes the nodes. The reader is already familiar with $h$-nodes, $v$-nodes, and $h.f$-nodes from Sections 3.1.1-3.1.3. A $v.f$-node represents the instance field $f$ accessed from variable $v$; the analysis uses $v.f$-nodes to model loads and stores.

Table I also shows sets stored at each node. The generic parameters in "[$\cdots$]" are the kinds of nodes in the set. The reader is already familiar with pointsTo sets (Column 3 of Table I) from Section 3.1.4.

FlowTo sets (Column 4 of Table I) represent a flow of values (assignments, parameter passing, etc.), and are stored with $v$-nodes and $v.f$-nodes. For example, if $v' \in$ flowTo($v$), then the pointer r-value of $v$ may flow to $v'$. As discussed in Section 3.2, this flow-to relation $v' \in$ flowTo($v$) can also be viewed as a subset constraint pointsTo($v$) $\subseteq$ pointsTo($v'$). FlowFrom sets are the inverse of flowTo sets. For example, $v'.f \in$ flowTo($v$) implies $v \in$ flowFrom($v'.f$).

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). For each $h$-node representing arrays of references, there is a special node $h.f_{\text{elems}}$ that represents all of their elements.

The constraint graph plays a dual role: it models the effect of code on pointers with flow sets, and it models the pointers themselves with pointsTo sets. Clients are interested in the pointsTo sets. The analysis only uses flow sets internally: the constraint finder produces flow sets (also known as constraints), and the constraint propagator iterates over them.

4.2.4 *Heintze's pointsTo set representation.* Heintze's shared bit sets compactly represent pointsTo sets by exploiting the observation that many pointsTo sets are similar or identical. Each set consists of a bit vector ("base bit vector") and a list ("overflow list"). Two sets that are nearly identical may share the same base bit

vector; any elements in these sets that are not in the base bit vector go in their respective overflow lists.

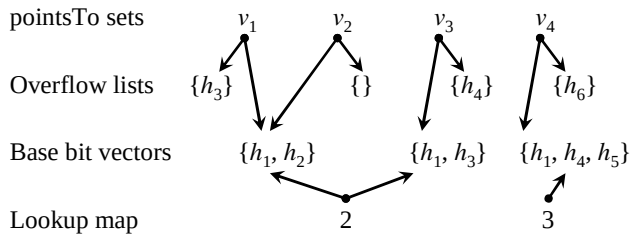



Fig. 3. Example for Heintze's pointsTo set representation

For example, imagine that pointsTo($v_1$) is $\{h_1, h_2, h_3\}$ and pointsTo($v_2$) is $\{h_1, h_2\}$. Then the two pointsTo sets may share the base bit vector containing $\{h_1, h_2\}$. However, since pointsTo($v_1$) also includes $h_3$, the overflow list for pointsTo($v_1$) contains the single element $h_3$; by similar reasoning pointsTo($v_2$)'s overflow list is empty. Fig. 3 shows this pictorially.

```
 1: if h ∉ base and h ∉ overflow
 2:     if |overflow| < overflowLimit
 3:         overflow.add(h)
 4:     else
 5:         desiredSet ← base ∪ overflow ∪ {h}
 6:         for overflowSize ← 0 to newOverflowThreshold
 7:             for each candidate ∈ lookupMap[|desiredSet| − overflowSize]
 8:                 if candidate ⊆ desiredSet
 9:                     base ← candidate
10:                     overflow ← desiredSet − candidate
11:                     return
12:         // we get here only if there was no suitable candidate
13:         base ← desiredSet
14:         overflow ← {}
15:         lookupMap[|base|].add(base)
```

Fig. 4. Adding a new element $h$ to a pointsTo set in Heintze's representation

Fig. 4 gives the algorithm for inserting a new element $h$ into a pointsTo set in Heintze's representation. For example, consider the task of inserting $h_4$ into pointsTo($v_1$). The algorithm adds $h_4$ to the overflow list of pointsTo($v_1$) if the overflow list is smaller than *overflowLimit* (a configuration parameter). If the overflow list is already of size *overflowLimit*, then instead of adding $h_4$ to the overflow list, the algorithm tries to find an existing base bit vector that will enable the overflow list to be smaller than *overflowLimit*, and indeed as small as possible (Lines 6-11). If no such base bit vector exists, then the algorithm creates a new perfectly matching base bit vector and makes it available for subsequent sharing (Lines 13-15).

### 4.3 Offline call graph builder

We use CHA (Class Hierarchy Analysis [Fernández 1995; Dean et al. 1995]) to find call edges. CHA finds call edges based on the subtyping relationship between the receiver variable and the class that declares the callee method. In other words, it only considers the class hierarchy, and ignores what values are assigned to the receiver variable. A more precise alternative than 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 [Lhoták and Hendren 2003]. In work concurrent with ours, Qian and Hendren developed an even more precise alternative based on low-overhead exhaustive profiling [Qian and Hendren 2004].

### 4.4 Offline constraint finder

The constraint finder analyzes the data flow of the program, and models it in the constraint graph.

4.4.1 *Assignments.* The intraprocedural part of the constraint finder analyzes the code of a method and models it in the constraint graph. It is a flow-insensitive pass of the just-in-time compiler. In our implementation, it operates on the high-level register-based intermediate representation (HIR) of Jikes RVM [Alpern et al. 2000]. HIR decomposes access paths by introducing temporaries, so that no access path contains more than one pointer dereference. Column "Actions" in Table II gives the actions of the constraint finder when it encounters the statement in Column "Statement". For example, the assignment $v' = v$ moves values from $v$ to $v'$, and the analysis models that by adding $v'$ to the set of variables to which $v$ flows. Column "Represent constraints" shows the constraints implicit in the actions of the constraint finder using mathematical notation. For example, because pointer values flow from $v$ to $v'$, the possible targets pointsTo($v$) of $v$ are a subset of the targets possible pointsTo($v'$) of $v'$.

Table II. Finding constraints for assignments

| Statement | | Actions | Represent constraints |
|---|---|---|---|
| $v' = v$ | (move $v \to v'$) | flowTo($v$).$add(v')$ | pointsTo($v$) ⊆ pointsTo($v'$) |
| $v' = v.f$ | (load $v.f \to v'$) | flowTo($v.f$).$add(v')$ | **for each** $h \in$ pointsTo($v$) : pointsTo($h.f$) ⊆ pointsTo($v'$) |
| $v'.f = v$ | (store $v \to v'.f$) | flowTo($v$).$add(v'.f)$, flowFrom($v'.f$).$add(v)$ | **for each** $h \in$ pointsTo($v'$) : pointsTo($v$) ⊆ pointsTo($h.f$) |
| $a$: $v =$ **new** ... | (alloc $h_a \to v$) | pointsTo($v$).$add(h_a)$ | $\{h_a\} \subseteq$ pointsTo($v$) |

4.4.2 *Parameters and return values.* The interprocedural part of the constraint finder analyzes call graph edges, and models the data flow through parameters and return values as constraints. It models parameter passing as a move from actuals (at the call site) to formals (of the callee). In other words, it treats parameter passing just like an assignment between the $v$-nodes for actuals and formals (first row of Table II). The interprocedural part of the constraint finder models each return statement in a method $m$ as a move to a special $v$-node $v_{\text{retval}(m)}$. It models the

data flow of the return value to the call site as a move to the $v$-node that receives the result of the call. Fig. 5 shows these flow-to relations as arrows. For example, the arrow $d \rightarrow \text{retval}(\text{A}::m)$ denotes the analysis action $\text{flowTo}(v_d).\text{add}(v_{\text{retval}(\text{A}::m)})$.

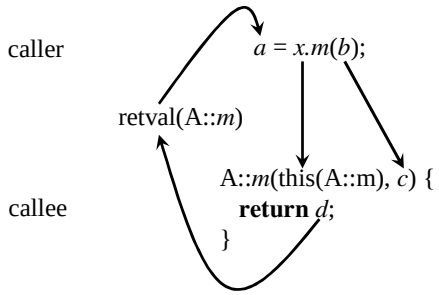



Fig. 5. Finding constraints for parameters and return values

4.4.3 *Exceptions.* Exception handling leads to flow of values (the exception object) between the site that throws an exception and the catch clause that catches the exception. The throw and catch may happen in different methods. For simplicity, our current prototype assumes that any throws can reach any catch clause of matching type. 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.

### 4.5 Offline constraint propagator

The constraint propagator finds a least fixed point of the pointsTo sets in the constraint graph, so they conservatively model the may-point-to relationship.

4.5.1 *Lhoták-Hendren worklist propagator.* Fig. 6 shows the worklist propagator that Lhoták and Hendren present as Algorithm 2 for their offline implementation of Andersen's analysis for Java [2003]. We adopted it as the starting point for our online version of Andersen's analysis because it is efficient and the worklist makes it amenable for incrementalization. The Lhoták-Hendren propagator has two parts: Lines 2-17 are driven by a worklist of $v$-nodes, whereas Lines 18-22 iterate over pointsTo sets of $h.f$-nodes.

The worklist-driven Lines 2-17 consider each $v$-node whose pointsTo set has changed (Line 3). Lines 4-9 propagate the elements along all flow edges from $v$ to $v'$-nodes or $v'.f$-nodes. In addition, Lines 10-17 propagate along loads and stores, where $v$ is the base node of a field access $v.f$, because new pointer targets $h$ of $v$ mean that $v.f$ represents new heap locations $h.f$.

The iterative Lines 18-22 are necessary because processing a store $v \rightarrow v'.f$ in Lines 7-9 can change $\text{pointsTo}(h.f)$ for some $h \in \text{pointsTo}(v')$. When that occurs, simply putting $v'$ on the worklist is insufficient, because the stored value can be retrieved via an unrelated variable. For example, assume

```
1: while worklist not empty
2:     while worklist not empty
3:         remove 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:        for each field f of v
11:            for each v' ∈ flowFrom(v.f)                  // store v' → v.f
12:                for each h ∈ pointsTo(v)
13:                    pointsTo(h.f).add(pointsTo(v'))
14:            for each v' ∈ flowTo(v.f)                    // load v.f → v'
15:                for each h ∈ pointsTo(v)
16:                    pointsTo(v').add(pointsTo(h.f))
17:                    if pointsTo(v') changed, add v' to worklist
18:    for each v.f
19:        for each v' ∈ flowTo(v.f)                        // load v.f → v'
20:            for each h ∈ pointsTo(v)
21:                pointsTo(v').add(pointsTo(h.f))
22:                if pointsTo(v') changed, add v' to worklist
```

Fig. 6. Lhoták-Hendren worklist propagator

$$\begin{aligned} \text{flowTo}(v) &= \{v'.f\}, & \text{flowTo}(v_3.f) &= \{v_4\}, \\ \text{pointsTo}(v) &= \{h\}, & & \\ \text{pointsTo}(v') &= \{h'\}, & \text{pointsTo}(v_3) &= \{h'\}, \\ \text{pointsTo}(v_4) &= \{\}, & \text{pointsTo}(h'.f) &= \{\} \end{aligned}$$

Assume $v$ is on the worklist. Processing the store $v \rightarrow v'.f$ propagates $h$ from $\text{pointsTo}(v)$ to $\text{pointsTo}(h'.f)$. Because $\text{flowTo}(v_3.f) = \{v_4\}$, there is a constraint $\text{pointsTo}(h'.f) \subseteq \text{pointsTo}(v_4)$, but it is violated, since $\text{pointsTo}(h'.f) = \{h\} \not\subseteq \{\} = \text{pointsTo}(v_4)$. Processing the load $v_3.f \rightarrow v_4$ would repair this constraint by propagating $h$ from $\text{pointsTo}(h'.f)$ to $\text{pointsTo}(v_4)$. But the algorithm does not know that it has to consider that load edge, since neither $v_3$ nor $v_4$ are on the worklist. Therefore, Lines 18-22 conservatively propagate along all load edges.

4.5.2 *Type filtering.* Consider a subset constraint $\text{pointsTo}(a) \subseteq \text{pointsTo}(b)$, where $a$ and $b$ may be $v$-nodes or $h.f$-nodes. While the constraint is not yet satisfied, the propagator propagates the left-hand side onto the right-hand side by performing the operation $\text{pointsTo}(b).\text{add}(\text{pointsTo}(a))$. Now let $A$ and $B$ be the declared Java types of $a$ and $b$, respectively. When the constraint involves a type cast that performs a narrowing conversion, $A$ is not a subtype of $B$, and the set $\text{pointsTo}(a)$ may contain $h$-nodes of a type incompatible with $B$. Those should not be propagated into $\text{pointsTo}(b)$, because $b$ can not possibly point to them. Instead, each propagation step filters by the type of the right-hand side set: the operation

$$\text{pointsTo}(b).add(\text{pointsTo}(a))$$

is really implemented as

**for each** $h \in \text{pointsTo}(a)$
  **if** typeOf($h$) is a subtype of typeOf($b$)
    $\text{pointsTo}(b) \leftarrow \text{pointsTo}(b) \cup \{h\}$

This technique is called on-the-fly type filtering. Lhoták and Hendren [Lhoták and Hendren 2003] had demonstrated that it helps keep pointsTo sets small and improves both performance and precision of the analysis. Our experiences confirm this observation.

4.5.3 *Propagator issues.* The propagator creates $h.f$-nodes lazily the first time it adds elements to their pointsTo sets, in lines 9 and 13. 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 $\text{pointsTo}(v) = \{h_A, h_B, h_C\}$. Now, data flow to $v.f$ should add to the pointsTo 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 [Fähndrich et al. 1998] and collapsing of single-entry subgraphs [Rountev and Chandra 2000]. They yielded only modest performance improvements compared to shared bit-vectors [Heintze 1999] and type filtering [Lhoták and Hendren 2003]. Part of the reason for the small payoff may be that our data structures do not put $h.f$-nodes in flowTo sets (á la BANE [Fähndrich et al. 1998]).

## 5. ONLINE ANALYSIS

This section describes how to change the offline analysis from Section 4 to obtain an online analysis. It is organized around the online architecture in Section 5.1.

### 5.1 Online architecture

Fig. 7 shows the architecture for performing Andersen's pointer analysis online. The online architecture subsumes the offline architecture from Fig. 2, and adds additional functionality (shown shaded) for dealing with dynamically loaded code and other dynamic program behavior that can not be analyzed prior to execution.

The left column shows the events during virtual machine execution that generate inputs to the analysis. The dotted box contains the analysis: the middle column shows analysis components, and the right column shows shared data structures that the components operate on. At the bottom, there are clients that trigger the constraint propagator component of the analysis, and consume the outputs (i.e., the pointsTo sets). Arrows mean triggering an action and/or transmitting information. We will discuss the online architecture in detail as we discuss its components in the subsequent sections.



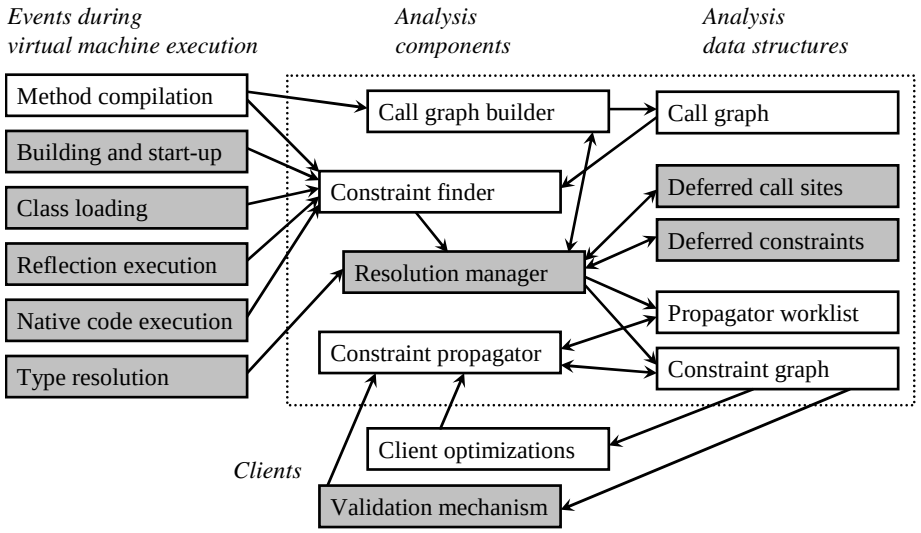



Fig. 7. Online architecture. Components absent from the offline architecture (Fig. 2) are shaded.

### 5.2 Online call graph builder

As described in Section 4.3, we use CHA (Class Hierarchy Analysis) to find call edges. Compared to offline CHA, online CHA has to satisfy two additional requirements: it has to incorporate more call sites and potential callee methods as they are encountered at runtime, and it has to work in the presence of unresolved types.

5.2.1 *Incorporating call sites and callees as they are encountered.* For each call edge, either the call site is compiled first, or the callee is compiled first. The call edge is added when the second of the two is compiled. More precisely,

—When the JIT compiler compiles a new method (Fig. 7, arrow from method compilation to call graph builder), the call graph builder retrieves all call sites of matching signature that it has seen in the past, and adds a call edge if the types are compatible (Fig. 7, arrow from call graph builder to call graph). Then, it stores the new callee method node in a map keyed by its signature, for future retrieval when new call sites are encountered.

—Analogously, when the JIT compiler compiles a new call site (Fig. 7, arrow from call graph builder to call graph), the call graph builder retrieves all callee methods of matching signature that it has seen in the past, and adds a call edge if the types are compatible (Fig. 7, arrow from call graph builder to call graph). Then, it stores the new call site node in a map keyed by its signature, for future retrieval when new callees are encountered.

5.2.2 *Dealing with unresolved types.* The JVM specification allows a Java method to have references to unresolved entities [Lindholm and Yellin 1999]. For example, the type of a local variable may be a class that has not been loaded yet. Furthermore, Java semantics prohibit eager loading of classes to resolve unresolved references: as discussed in Section 2.3, eager loading would require hiding any vis-

ible effects until the class actually is used. Unfortunately, when a variable is of an unresolved type, the subtyping relationships of that type are not yet known, prohibiting CHA. When the call graph builder encounters a call site $v.m()$ where the type of the receiver variable $v$ is unresolved, it refuses to deal with it at this time, and sends it to the resolution manager instead (Fig. 7, arrow from call graph builder to resolution manager).

### 5.3 Resolution manager

The resolution manager enables lazy analysis: it allows the analysis to defer dealing with information that it can not yet handle precisely and that it does not yet need for obtaining sound results. The arrows from the call graph builder and the constraint finder to the resolution manager in Fig. 7 represent call sites and constraints, respectively, that may refer to unresolved types. The arrows from the resolution manager to the deferred call sites and deferred constraints data structures in Fig. 7 represent "shelved" information: this information does indeed involve unresolved types, so the analysis can and should be lazy about dealing with it.

When the VM resolves a type (arrow from type resolution to resolution manager in Fig. 7), the resolution manager reconsiders the deferred information. For each call site and constraint that now involves only resolved types, the resolution manager removes it from the deferred data structure, and sends it back to the call graph builder (in the case of a call site), or on to later analysis stages (in the case of a constraint).

With this design, the analysis will shelve some information forever, if the involved types never get resolved. This saves unnecessary analysis work. Qian and Hendren [2004] developed a similar design independently. Before becoming aware of the subtleties of the problems with unresolved references, we used an overly conservative approach: we added analysis information eagerly even when we had incomplete information. This imprecision led to large pointsTo sets, which, in turn, led to a prohibitive slow-down of our analysis. In addition, it complicated the analysis, because it had to treat unresolved types as a special case throughout the code. Using the resolution manager is simpler, more precise, and more efficient.

### 5.4 Online constraint finder

Compared to the offline constraint finder (Section 4.4), the online constraint finder has to satisfy three additional requirements: it has to capture more input events, it has to find interprocedural constraints whenever more call edges are encountered, and it has to work in the presence of unresolved types.

5.4.1 *Capturing more input events.* In the offline architecture (Fig. 2), the only input to the constraint finder is the compiler's intermediate representation of methods. In the online architecture (Fig. 7) there are several inputs to the constraint finder:

*Method compilation.* Offline pointer analysis assumes that code for all methods is available simultaneously for analysis. Due to dynamic class loading, this is not true in Java. Instead, code for each method becomes available for analysis when the JIT compiler compiles it. This does not necessarily coincide with the time that the class is loaded: the method may get compiled at any time after the enclosing class is loaded and before the method gets executed for the first time. (This assumes a compile-only approach; we will discuss interpreters in Section 7.4.1.)

*Building and start-up.* A Java virtual machine often supports system libraries with classes that interact directly with the runtime system. Support for these is built into the VM, and initialized during VM start-up. This system behavior does not consist of normal Java code, and the pointer analysis must treat it in a VM-dependent way. We will discuss how we handle it for Jikes RVM in Section 7.4.5.

*Class loading.* Methods (including the class initializer method) are handled as usual at method compilation time. 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 [Lindholm and Yellin 1999, Section 4.5]. This attribute initializes fields, but it is not represented as code, but rather as meta-data in the class file.

*Reflection execution.* Section 5.6 describes how the online setting allows the analysis to tackle reflection in a sound and usable way.

*Native code execution.* Section 5.7 describes how the online setting allows the analysis to tackle native code in a robust and usable way.

5.4.2 *Capturing call edges as they are encountered.* Whenever the online call graph builder (Section 5.2) adds a new call edge to the call graph (Fig. 7, arrow from call graph builder to call graph), the constraint finder is notified (Fig. 7, arrow from call graph to constraint finder), so it can model the data flow for parameter passing and return values. This works as follows:

—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 $c$, and stores it for future use.

—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, and stores it for future use.

—When encountering a call edge $c \rightarrow m$, the constraint finder retrieves the corresponding tuples $I_c$ and $I_m$, and adds constraints to model the moves between the corresponding $v$-nodes in the tuples.

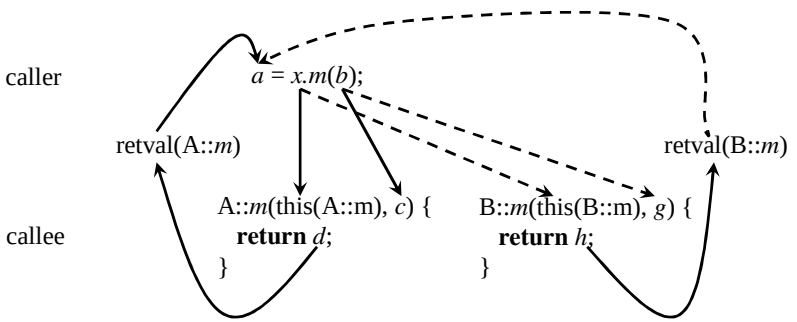



Fig. 8. Call constraints when adding a new callee.

For the example in Fig. 5, the tuple for call site $a = x.m(b)$ is $I_{a=x.m(b)} = \langle v_a, v_x, v_b \rangle$. Suppose the analysis just encountered a new potential callee B::$m$ with the tuple $I_{\text{B::}m} = \langle v_{\text{retval(B::}m)}, v_{\text{this(B::}m)}, v_g \rangle$. The call graph builder uses CHA to decide whether the call-site and callee match, based on the type of the receiver variable $x$ and class B. If they are compatible, the constraint finder models the moves $x \rightarrow \text{this(B::}m)$, $b \rightarrow g$, and $\text{retval(B::}m) \rightarrow a$, shown as dashed arrows in Fig. 8.

5.4.3 *Dealing with unresolved types.* As discussed in Section 5.2.2, Java bytecode can refer to unresolved types. This prohibits type filtering in the propagator (Section 4.5.2), which relies on knowing the subtype relations between the declared types of nodes involved in flow sets. Furthermore, the propagator also requires resolved types for mapping between $h$-nodes and their corresponding $h.f$-nodes. In this case, it needs to know the subtyping relationship between the type of objects represented by the $h$-node, and the class in which the field $f$ was declared.

To allow type filtering and field mapping, the propagator must not see constraints involving unresolved types. The online architecture supports dealing with unresolved types by a layer of indirection: where the offline constraint finder directly sent its outputs to later analysis phases (Fig. 2), the online constraint finder sends its outputs to the resolution manager first (Fig. 7), which only reveals them to the propagator when they become resolved. This works as follows:

—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 pointsTo set of another node, but one or both of them are unresolved, it defers the information for later instead. Table III 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 III. Deferred constraints stored at unresolved nodes.

| Node kind | Flow | PointsTo |
|---|---|---|
| $h$-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.

### 5.5 Online constraint propagator

Method compilation and other constraint-generating events happen throughout program 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, if new constraints have been created since the last propagation. After constraint propagation, the pointsTo sets conservatively describe the pointers in the program until one of the events in the left column if Fig. 7 causes the analysis to add new constraints to the constraint graph.

The propagator propagates pointsTo sets following the constraints represented in the flow sets until the pointsTo sets reach the least fixed point. This problem is monotonous: newly added constraints may not hold initially, but a repropagation starting from the previous fixed point suffices to find the least fixed point of the new constraint set. To accommodate this, the propagator starts with its previous solution and a worklist (Section 4.2.2) of changed parts in the constraint graph to avoid incurring the full propagation cost every time.

Our online constraint propagator thus required a fundamental change in the architecture. Whereas in the offline architecture (Fig. 7), the worklist is private to the constraint propagator, in the online architecture (Fig. 7), the worklist is exposed to the resolution manager, and thus indirectly to the constraint finder.

Exposing the worklist was the only "external" change to the propagator to support online analysis. In addition, we found that "internal" changes that make the constraint propagator more incremental are invaluable for achieving good overall analysis performance. We will describe those incrementalizations in Section 6.1.

### 5.6 Reflection

Java allows code to access methods, fields, and types by name based on strings computed at runtime. For example, in Fig. 9, method $m$ may be any method of class Vector that takes one argument of type Object, such as Vector.*add*, Vector.*contains*, or Vector.*equals*. A static analysis can not determine what actions reflection will perform. This reduces analysis precision; in the example, the analysis would have to conservatively assume that any of the Vector methods with a matching signature can get called. If the command line argument *argv*[0] is `"add"`, then Line 7 calls *add*, adding the vector $v$ into itself. Hence, the analysis would have to model a points-to relation from the vector to itself, even if the program only calls method *contains* or *equals*, which do not install such a pointer. In practice, the issue is usually even more complicated: the analysis can not always determine the signature of the callee method statically (in the example, this relies on knowing the exact contents of the array *paramTypes*), and often does not know the involved types (in the example, Vector.**class** might be obtained from a class loader instead).

One solution would be to use an approach such as String analysis [Christensen et al. 2003] to predict which entities reflection manipulates. However, String analysis is an offline analysis, and while it can solve the problem in special cases, this problem is undecidable in the general case. Another solution would be to assume the worst case. We felt that this was too conservative and would introduce signifi-

```
 1: class Main {
 2:     public static void main(String[] argv) throws Exception {
 3:         Class[] paramTypes = new Class[] { Object.class };
 4:         Method m = Vector.class.getMethod(argv[0], paramTypes);
 5:         Vector v = new Vector();
 6:         Object[] params = new Object[] { v };
 7:         Object result = m.invoke(v, params);
 9:     }
10: }
```

Fig. 9. Reflection example.

cant 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 [Whaley and Lam 2002; Lhoták and Hendren 2003].

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.

### 5.7 Native code

The Java Native Interface (JNI) allows Java code to interact with dynamically loaded native code. Usually, a JVM cannot analyze that code.

When Java code calls a JNI method, and that method returns a value, a static analysis can not determine what that return value might be.

When a JNI method manipulates Java data structures or calls a Java method, it does so by using an API similar to that for reflection, but from native code. This is conceptually analogous to Fig. 9 in that JNI may obtain a method $m$ and invoke it on an object $v$. However, the case is harder than for reflection, because an analysis for Java can not even analyze the code surrounding the reflective call, since it is compiled architecture-specific machine code.

Offline pointer analyses for Java usually require the user to specify a model for the effect of native code on pointers. This approach is error-prone and does not scale well.

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. Type filtering (Section 4.5.2) alleviates this problem by reducing the set of $h$-nodes returned by a JNI method based on its declared return type. If a JNI method manipulates data structures of the program, the manipulations must go through the JNI API, which is equivalent to Java reflection (in fact, Jikes RVM implements it by calling Java methods that use reflection). Therefore, our analysis handles JNI methods that make calls or manipulate object fields by the same mechanism as reflection.

## 6. OPTIMIZATIONS

The first implementation of our online pointer analysis was slow [Hirzel et al. 2004]. Thus, we empirically evaluated where the analysis was spending its time. Besides timing various tasks performed by the analysis, we visualized the constraint graphs to discover bottlenecks. Based on our findings, we decided to further incrementalize the propagator (Section 6.1) and to manually fine-tune the precision of the analysis along various dimensions (Section 6.2).

### 6.1 Incrementalizing the propagator

As discussed in Section 3.3, incrementality is necessary, but not sufficient, for online program analysis. Section 5 described those interactions, and Section 5.5 described some steps towards making the propagator incremental. However, incrementality is not an absolute property; rather, there is a continuum of more and more incremental algorithms. Sections 6.1.1 to 6.1.4 describe several such algorithms, each more incremental than the previous one. None of these algorithms affect precision: they all compute the same fixed point on the constraint set.

6.1.1 *Lhoták-Hendren propagator, and propagating from new constraints only.* This section briefly reviews the starting point of our propagator incrementalization. Section 4.5 describes the propagator that Lhoták and Hendren presented in their initial paper about the SPARK offline pointer analysis framework for Java [2003]. Fig. 6 gives the pseudo-code, which has two parts: Lines 2-17 are driven by a worklist of $v$-nodes, whereas Lines 18-22 iterate over pointsTo sets of $h.f$-nodes. Section 4.2.2 describes the worklist data structure: it is a list of $v$-nodes that may appear on the left side of unresolved constraints. Section 5.5 describes how to use the worklist for incrementalization: essentially, the constraint finder produces work on the worklist, and the constraint propagator consumes that work. When the constraint propagator starts, the worklist tells it exactly which constraints require propagation, making Lines 2-17 of Fig. 6 efficient.

6.1.2 *IsCharged bit for $h.f$-nodes.* We found that after some iterations of the outer loop (Line 1) of the Lhoták-Hendren propagator in Fig. 6, the iterative part (Lines 18-22) dominates runtime. Because the algorithm still has to consider all load edges even when the constraint graph changes only locally, it does not yet work well incrementally. One remedy we investigated is making flowTo sets of $h.f$-nodes explicit, similar to BANE [Fähndrich et al. 1998], as that would allow maintaining a worklist for $h.f$-nodes. We found the space overhead for this redundant flowTo information prohibitive. Therefore, we took a different approach to further incrementalize the algorithm.

Fig. 10 shows the algorithm that we presented as Fig. 3 in our previous paper about this analysis [2004], which maintains isCharged bits on $h.f$-nodes to speed up the iterative part of Fig. 6 (Lines 18-22). The purpose of the iterative part is to propagate new elements from pointsTo sets of $h.f$-nodes. This is only necessary for those $h.f$-nodes whose pointsTo sets changed in Lines 2-17 of Fig. 6. We say an $h.f$-node is charged if processing a load might propagate new pointsTo set elements

```
1: while worklist not empty, or isCharged(h.f) for any h.f-node      // c.f. Fig. 6 L. 1
2:     while worklist not empty
3:         remove 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   // new in Fig. 10
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 // new in Fig. 10
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 v' ∈ flowTo(v.f)                                 // load v.f → v'
22:            for each h ∈ pointsTo(v), if isCharged(h.f)            // c.f. Fig. 6 L. 20
23:                pointsTo(v').add(pointsTo(h.f))
24:                if pointsTo(v') changed, add v' to worklist
25:    for each h.f, isCharged(h.f) ← false                          // new in Fig. 10
```

Fig. 10. IsCharged bit for $h.f$-nodes. Changes compared to Fig. 6 are annotated with comments.

```
1-19: identical with Lines 1-19 in Fig. 10
20: for each v.f
21:     for each v' ∈ flowTo(v.f)                                     // load v.f → v'
22:         if (pointsTo(v), f) ∈ chargedHFCache
23:             chargedHF ← chargedHFCache[pointsTo(v), f]
24:         else
25:             chargedHF ← {h.f for h ∈ pointsTo(v), if isCharged(h.f)}
26:             chargedHFCache[pointsTo(v), f] ← chargedHF
27:         for each h.f ∈ chargedHF
28:             pointsTo(v').add(pointsTo(h.f))
29:             if pointsTo(v') changed, add v' to worklist
30: for each h.f, isCharged(h.f) ← false
```

Fig. 11. Caching charged $h.f$-node sets

from $h.f$. In Fig. 10, when pointsTo($h.f$) changes, Lines 10 and 15 set its isCharged bit. This enables Line 22 to perform the inner loop body for only the few $h.f$-nodes that need discharging. Line 25 resets the isCharged bits for the next iteration.

6.1.3 *Caching charged $h.f$-node sets.* Lines 20-22 of Fig. 10 still iterate over all $h.f$-nodes, even if they can often skip the body of Lines 23-24. Profiling found that much time is spent determining the subset of nodes $h \in$ pointsTo($v$) for which $h.f$ is charged. In Line 22, when pointsTo($v$) is almost the same as pointsTo($v'$) for some other variable $v'$, it is wasteful to compute the subset twice. Instead, the propagator in Fig. 11 caches the set of charged $h.f$-nodes, keyed by the set of base $h$-nodes and the field $f$.

In a naive implementation, a cache lookup would have to compare pointsTo sets for equality. This would take time linear in the size of the sets, which would be no better than performing the iteration in Line 22 of Fig. 10. Instead, our implementation exploits the fact that the pointsTo sets are already shared (see Heintze's representation, Section 4.2.4). Each cache entry is a triple $(b, f, c)$ of a shared bit vector $b$, a field $f$, and a cached set $c$. The set $c \subseteq b$ is the set of $h$-nodes in $b$ for which $h.f$ is charged. Given a pointsTo set, the lookup in Line 23 works as follows:

—Given (pointsTo($v$), $f$), find a cache entry $(b, f', c)$ such that the base bit vector of pointsTo($v$) is $b$, and the fields are the same ($f = f'$).
—For each $h \in c$, put the node $h.f$ in the resulting set chargedHF.
—In addition, iterate over the elements $h$ of the overflow list of pointsTo($v$), and determine for each of them whether $h.f$ is charged.

This lookup is faster than iterating over the elements of the base bit vector individually.

6.1.4 *Caching the charged $h$-node set.* The iterative part of the Algorithm in Fig. 11 still loops over all load edges $v.f \rightarrow v'$ in Lines 20+21. Therefore, its best-case execution time is proportional to the number of assignments of the form $v' = v.f$. This bottleneck limited peak responsiveness of our online pointer analysis. Therefore, we used a final optimization to reduce the time of the iterative part to be proportional to the number of $v.f$-nodes, instead of $v.f \rightarrow v'$-edges. The Algorithm in Fig. 12 uses a set of charged $h$-nodes: an $h$-node is charged if there is at least one field $f$, such that isCharged($h.f$) = **true**.

Lines 12 and 19 in Fig. 12 remember $h$-nodes for which at least one $h.f$-node became charged. Line 26 uses these $h$-nodes to decide whether to consider loads from a given $v.f$-node: Lines 27-36 are necessary only if at least one of the corresponding $h.f$-nodes is charged. The check in Line 26 is a fast bit vector operation. After Line 37 clears all isCharged bits, Line 38 resets chargedH.

The chargedH set is not only useful for reducing the number of outer loop iterations in the iterative part. It also speeds up the computation of charged $h.f$-nodes in Line 25 of Fig. 11 (c.f. Lines 31-32 in Fig. 12). Again, the set intersection is a fast bit-set operation. These optimizations for handling loads in the iterative part (Lines 25-36) also apply to loads in the worklist part (Lines 20-24).

## 6.2 Varying analysis precision

Section 3.1 states that our analysis is context insensitive, allocation-site based, and field sensitive. These choices yield a reasonable overall precision/speed tradeoff. But in fact, it helps to manually revise them in a few places. Section 8.3 will evaluate the effects of these optimizations.

6.2.1 *Selectively use context sensitivity.* Our context-insensitive analysis (Section 3.1.1(c)) gives imprecise results particularly for methods that are called from many places and modify or return their argument objects. In such situations, a

```
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' ∈ 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
11:                    isCharged(h.f) ← true
12:                    chargedH.add(h)
13:        for each field f of v
14:            for each v' ∈ flowFrom(v.f)                          // store v' → v.f
15:                for each h ∈ pointsTo(v)
16:                    pointsTo(h.f).add(pointsTo(v'))
17:                    if pointsTo(h.f) changed
18:                        isCharged(h.f) ← true
19:                        chargedH.add(h)
20:            if pointsTo(v) ∩ chargedH ≠ {}
21:                for each v' ∈ flowTo(v.f)                        // load v.f → v'
22:                    for each h ∈ (pointsTo(v) ∩ chargedH)
23:                        pointsTo(v').add(pointsTo(h.f))
24:                        if pointsTo(v') changed, add v' to worklist
25:    for each v.f
26:        if pointsTo(v) ∩ chargedH ≠ {}
27:            for each v' ∈ flowTo(v.f)                            // load v.f → v'
28:                if (pointsTo(v), h) ∈ chargedHFCache
29:                    chargedHF ← chargedHFCache[pointsTo(v), f]
30:                else
31:                    ch ← pointsTo(v) ∩ chargedH
32:                    chargedHF ← {h.f for h ∈ ch, if isCharged(h.f)}
33:                    chargedHFCache[pointsTo(v), f] ← chargedHF
34:                for each h.f ∈ chargedHF
35:                    pointsTo(v').add(pointsTo(h.f))
36:                    if pointsTo(v') changed, add v' to worklist
37:    for each h.f, isCharged(h.f) ← false
38:    chargedH ← {}
```

Fig. 12. Caching the charged $h$-node set

context-insensitive analysis will propagate information from all call sites into the method and then back out to all the call sites (via the modification or return value). When context sensitivity (Section 3.1.1(d)) improves precision, it can also improve performance, since the analysis has to propagate smaller pointsTo sets with context sensitivity.

We found enabling context sensitivity for all methods prohibitively expensive, and thus we extended our analysis with selective context sensitivity. It is context sensitive only in those cases where the efficiency gain from smaller pointsTo sets outweighs the efficiency loss from more pointsTo sets. We manually picked the following methods to analyze context-sensitively wherever they are called: StringBuffer.*append*(...) (where ... is String, StringBuffer, or char), StringBuffer.*toString*(), and VM_Assembler.*setMachineCodes*(). Both *append*() and *toString*() are Java standard library methods, and *setMachineCodes*() is a Jikes RVM method. Plevyak and Chien [1994] and Guyer and Lin [2003] describe mechanisms for automatically selecting methods for context sensitivity. While those mechanisms happen in an offline, whole-world analysis and thus do not immediately apply to online analysis, we believe they can inspire approaches to automate online context sensitivity selection.

6.2.2 *Selectively relax allocation site sensitivity.* Creating a new $h$-node for each allocation site is more precise than creating an $h$-node for each type. (In the words of Section 3.1.2, allocation-site based treatment of pointer targets (d) is more precise than type-based treatment of pointer targets (c).) However, the precision is useful only if the instances created at two sites are actually stored in separate locations. For example, if a program creates instances of a type T at 100 allocation sites but puts references to them all only in one variable then there is no benefit in distinguishing between the different allocation sites. Thus, rather than creating 100 $h$-nodes, which all need to be propagated, the pointer analysis can create just a single $h$-node that represents all the allocation sites of that type. Using a single $h$-node not only saves memory (fewer $h$-nodes and thus $h.f$-nodes) but also propagation effort (since the pointsTo sets are smaller).

We extended our analysis to selectively merge $h$ and $v$-nodes. We used manual investigation to determine three types whose nodes should be selectively merged: classes OPT_Operand, OPT_Operator, and OPT_Instruction, which are all part of the Jikes RVM optimizing compiler. We believe that deciding which nodes to merge on the fly is good idea; we will explore possible approaches in future work.

6.2.3 *Selectively relax field sensitivity.* If there are many stores into a field accessed via different $v$-nodes, but they all store similar pointsTo sets, then field-sensitivity (Section 3.1.3(d)) is harmful: it degrades performance without improving precision. Context sensitivity creates such situations. For example, when we analyze StringBuffer.*append*() context-sensitively, we effectively create many copies of its body and thus many copies of references to the StringBuffer.*value* instance variable. Since StringBuffer.*value* is initialized with arrays allocated at one of a few sites in the StringBuffer class, there is no point in treating StringBuffer.*value* field sensitively: all $h.f_{\text{StringBuffer.}value}$ point to the same few instances.

We have extended our analysis to selectively fall back to being field-based (Section 3.1.3(c)). In our runs we apply field sensitivity everywhere except for fields affected as described above by context sensitivity, namely StringBuffer.*value*, String.*value*, and VM_Assembler.*machineCodes*. We believe that deciding on whether or not to use field sensitivity on the fly is a good idea; we will investigate possible approaches in future work.

## 7. IMPLEMENTATION ISSUES

Section 7.1 describes how we convinced ourselves that our analysis implementation is sound. Section 7.2 discusses how clients can use the analysis results. Section 7.3

mentions some performance bugs that we found and fixed. Finally, Section 7.4 shows how our implementation tackles the challenges of a concrete VM.

## 7.1 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 pointsTo sets. To help us debug our pointer analysis (to a high confidence level) we built a validation mechanism.

7.1.1 *Validation mechanism.* We validate the pointer analysis results at GC (garbage collection) time (Fig. 7, arrow from constraint graph to validation mechanism). As GC traverses each pointer, it checks whether the pointsTo 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 pointsTo 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 pointsTo set of $h.f$ includes $h'$. If either check fails, it prints a warning message.

Since it checks the correctness of pointsTo sets during GC, the validation mechanisms triggers constraint propagation just before GC starts (Fig. 7, arrow from validation mechanism to constraint propagator). As there is no memory available to grow pointsTo 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, and the extra header word is absent in production runs.

7.1.2 *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.

*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" ([Lindholm and Yellin 1999, 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 pointsTo set elements. That resulted in warnings from our validation methodology. Upon investigating those warnings, we became aware of the incorrect assumption and fixed it.

*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 ([Lindholm and Yellin 1999, 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 pointsTo 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 did not initialize the fields. Thus, we found out about the ConstantValue bytecode attribute, and added constraints when class loading parses and executes that attribute (Section 5.4.1).

## 7.2 Clients

This section investigates example clients of our analysis, and how they can deal with the dynamic nature of our analysis. In general, each client triggers constraint propagation when it requires sound analysis results (Fig. 7, arrow from client optimizations to constraint propagator), and then consumes the resulting pointsTo sets (Fig. 7, arrow from constraint graph to client optimizations).

7.2.1 *Method inlining.* The method inlining optimization can benefit from pointer analysis: if the pointsTo set elements of $v$ all have the same implementation of a method $m$, the call $v.m()$ has only one possible target. Modern JVMs [Cierniak et al. 2000; Arnold et al. 2000; Paleczny et al. 2001; Suganuma et al. 2001] 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 Fig. 7, an inlining client must be prepared to invalidate inlining decisions. Techniques such as code patching [Cierniak et al. 2000] and on-stack replacement [Hölzle et al. 1992; Fink and Qian 2003] support invalidation. If instant invalidation is needed, our analysis must repropagate every time it finds new constraints. There are also techniques for deferring or avoiding invalidation of inlining decisions (pre-existence based inlining [Detlefs and Agesen 1999] and guards [Hölzle and Ungar 1994; Arnold and Ryder 2002], respectively) that would allow our analysis to be lazy about repropagating after it finds new constraints. Qian and Hendren [Qian and Hendren 2005] study the usefulness of pointer analysis for inlining in a VM, and survey invalidation techniques.

7.2.2 *Side-effect analysis.* Side-effect analysis enables various JIT compiler optimizations. Le et al. [2005] show how pointer analysis enables side-effect information, and that that in turn could help optimizations in a JVM achieve higher

speed-ups if pointer analysis information were available in the JVM. Invalidation would proceed similarly to that for method inlining.

7.2.3 *Connectivity-based garbage collection.* CBGC (connectivity-based garbage collection) is a new garbage collection technique that requires pointer analysis [Hirzel et al. 2003]. 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 [Hirzel et al. 2003], 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.

7.2.4 *Other clients.* Pointer analysis could help make many optimizations in a virtual machine more aggressive. This includes compiler optimizations that involve pointers, as well as optimizations related to heap memory management or to parallelization. The concrete examples above show that Java's dynamic class loading forces clients to be speculative, and we presented mechanisms for invalidating optimistic assumptions in a variety of clients.

### 7.3 Fixing performance bugs

We took a "correctness first, performance later" approach in demonstrating the first pointer analysis that works for all of Java [2004]. This led to various performance bugs: situations where the analysis is still sound, but takes excessive time and space. Mentioning them here may help readers avoid pit-falls in implementing similar analyses.

7.3.1 *Use a bit mask for type filtering.* Section 4.5.2 describes type filtering: when propagating from pointsTo($a$) into pointsTo($b$), where $b$ is a $v$-node or an $h.f$-node, the analysis filters the propagated $h$-nodes, only adding $h$-nodes of a subtype of the declared type of $b$ to pointsTo($b$). Type filtering keeps pointsTo sets smaller, improving both the precision and the efficiency of the analysis. However, in our implementation in [Hirzel et al. 2004], it was surprisingly expensive. Our implementation iterated over all the $h$-nodes in pointsTo($a$) one at a time and added them to pointsTo($b$) if the type check succeeded. To avoid the cost of that inner loop, we changed our implementation to filter by doing a logical "and" of bit-vectors. This operation is much faster than our original implementation at the cost of minimal space overhead. As it turns out, Lhoták and Hendren's implementation of type filtering for bit sets also uses the bit mask approach, but they do not describe it in their paper [Lhoták and Hendren 2003].

One subtlety with using a bit mask for type filtering is that Heintze's pointsTo set representation uses a base bit vector and an overflow list (Section 4.2.4). Therefore, the operation

$$\text{pointsTo}(b).\mathit{add}(\text{pointsTo}(a))$$

still involves a loop over the bounded-size overflow list:

$$\begin{array}{l}\text{pointsTo}(b) \leftarrow \text{pointsTo}(b) \cup \Big(\text{pointsTo}(a).\mathit{base} \cap \text{typeMask}(\text{typeOf}(b))\Big)\\ \textbf{for each } h \in \text{pointsTo}(a).\mathit{overflow}\\ \quad \textbf{if } h \in \text{typeMask}(\text{typeOf}(b))\\ \quad\quad \text{pointsTo}(b) \leftarrow \text{pointsTo}(b) \cup \{h\}\end{array}$$

7.3.2 *Use information on private/final/constructor methods for call graph construction.* Class hierarchy analysis (CHA) constructs a call graph based on the canonical definition of virtual method dispatch. But implementing plain CHA was needlessly imprecise. In Java, there is often more information available to disambiguate calls. Private methods are invisible to subclasses; final methods can not be overridden; and constructor calls have a known exact receiver type. We improved both precision and efficiency of the analysis by pruning call edges using this additional information.

7.3.3 *Restrict the base variable type for field accesses.* In Java source code, each local variable $v$ has a declared type $T$. A field access $v.f$ is only legal if $T$ or one of its superclasses declares $f$. However, in Java bytecode, a local variable can have different, conflicting types at different program points. A field access $v.f$ may be legal even if at other points, the variable has a type that is incompatible with the field. The constraint finder represents such a variable with a $v$-node of a type that is general enough to be legal at all program points. But this means that the constraint propagator has to check to which $h$-nodes in pointsTo($v$) the field applies whenever it processes a load or a store. This check is costly, doing it naively was too slow. We fixed it by introducing helper variables of more precise types in some cases where it helps performance.

### 7.4 VM interactions

So far, the description of our pointer analysis was general with respect to the virtual machine in which it is implemented. The following sections describe how to deal with VM-specific features. We use Jikes RVM as a case study, but the approaches generalize to other VMs with similar features.

7.4.1 *Interpreters and 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. Alternatively, they can take the more dynamic approach of finding constraints at interpretation time.

7.4.2 *Recompilation.* Many VMs, including Jikes RVM, may recompile a method (at a higher optimization level) if it executes frequently. Optimizations such as inlining may introduce new variables or code into the recompiled methods.

Since the analysis models each inlining context of an allocation site by a separate $h$-node, it generates new constraints for the recompiled methods and integrates them with the constraints for any previously compiled versions of the method.

7.4.3 *Type descriptor pointers.* In addition to language-level fields, each $h$-node has a special node $h.f_{\text{td}}$ that represents the field containing the reference to the type descriptor for the object. A type descriptor contains support for runtime system mechanisms such as virtual method dispatch, casts, reflection, and type-accurate garbage collection. Jikes RVM implements type descriptors as ordinary Java objects, and thus, our analysis must model them as such.

7.4.4 *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. The analysis treats uses of magic on a case-by-case basis: for example, when magic stores a type descriptor from a variable $v$ to an object header, the analysis adds the appropriate $v'.f_{\text{td}}$ to flowTo($v$). As another example, when magic manipulates raw memory in the garbage collector, the analysis ignores those operations.

7.4.5 *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 and pre-compiled methods 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.

Our analysis models all the *code* in the boot image as usual, with the constraint finder from Sections 4.4 and 5.4. Our analysis models the *data snapshot* of the boot image with special boot image $h$-nodes, and with pointsTo 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.

## 8. RESULTS

Section 8.1 introduces the environment in which our pointer analysis operates, Section 8.2 evaluates the performance of the analysis with all optimizations, and Section 8.3 evaluates the effects of optimizations individually and in groups. We conducted all experiments for this paper using Jikes RVM 2.2.1 running on a 2.4GHz Pentium 4 with 2GB of memory running Linux, kernel version 2.4.

### 8.1 Environment

Section 8.1.1 introduces the benchmarks. Pointer analysis for Java must happen online, dealing with new code as new methods get compiled; Section 8.1.2 characterizes this behavior over time. Finally, Section 8.1.3 describes the infrastructure underlying the rest of the results section.

8.1.1 *Benchmarks.* Table IV describes the benchmark suite. Each row shows a benchmark; *null* is the empty main method, and *average* is the arithmetic mean of all benchmarks except for *null*. The suite includes all SPECjvm98 benchmarks and several other Java programs. Column "Workload" shows the command line arguments passed to the main method. For *pseudojbb*, the workload consists of 1 warehouse and 70,000 transactions.

Table IV. Benchmarks.

| Program | Workload [command line arguments] | Compiled methods | Loaded classes | Alloc. [MB] | Time [s] |
|---|---|---|---|---|---|
| *null* | | *15,298* | *1,263* | *9.5* | *0* |
| mtrt | `-m1 -M1 -s100` | 15,858 (+560) | 1,404 (+141) | 152.8 | 13 |
| mpegaudio | `-m1 -M1 -s100` | 15,899 (+601) | 1,429 (+166) | 15.0 | 26 |
| javalex | `qb1.lex` | 16,058 (+760) | 1,289 (+26) | 48.5 | 23 |
| compress | `-m1 -M1 -s100` | 16,059 (+761) | 1,290 (+27) | 116.8 | 30 |
| db | `-m1 -M1 -s100` | 16,082 (+784) | 1,284 (+21) | 86.5 | 25 |
| ipsixql | `3 2` | 16,275 (+977) | 1,348 (+85) | 193.9 | 6 |
| jack | `-m1 -M1 -s100` | 16,293 (+995) | 1,324 (+61) | 245.9 | 8 |
| richards | | 16,293 (+995) | 1,336 (+73) | 12.0 | 2 |
| hsql | `-clients 1 -tpc 50000` | 16,323 (+1,025) | 1,316 (+53) | 2,944.3 | 153 |
| pseudojbb | | 16,453 (+1,155) | 1,318 (+55) | 283.7 | 27 |
| jess | `-m1 -M1 -s100` | 16,489 (+1,191) | 1,420 (+157) | 278.4 | 13 |
| javac | `-m1 -M1 -s100` | 16,795 (+1,497) | 1,418 (+155) | 238.2 | 15 |
| soot | `-W --app -t -d Hello --jimple` | 17,023 (+1,725) | 1,486 (+223) | 70.9 | 3 |
| xalan | `1 1` | 17,342 (+2,044) | 1,504 (+241) | 344.2 | 31 |
| *average* | | *16,374 (+1,076)* | *1,369 (+106)* | *359.4* | *27* |

Columns "Compiled methods" and "Loaded classes" of Table IV characterize the code size. The numbers in parentheses show how much code each benchmark adds beyond *null*. Jikes RVM compiles a method if it belongs to the boot image, or when the program executes it for the first time. The loaded classes also include classes in the boot image. Our pointer analysis has to deal with all these methods and classes, which includes the benchmark's own code, library code used by the benchmark, and code belonging to Jikes RVM, including code of the pointer analysis itself. Benchmark *null* provides a baseline: it represents approximately the amount that Jikes RVM adds to the size of the application. This is an approximation 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).

Analysis in Jikes RVM has to deal with many more methods and classes 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, such as relying on garbage collection for memory management. Furthermore, the absence

of artificial boundaries between the analysis, other parts of the runtime system, and the application exposes more opportunities for optimizations. For example, the analysis can speed up garbage collection (by providing information on where pointers may point to), and garbage collection in turn can speed up the analysis (by efficiently reclaiming garbage produced by the analysis). 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.

Columns "Alloc." and "Time" of Table IV characterize the workload, using Jikes RVM without our pointer analysis. Column "Alloc." is the number of MB allocated during the run, excluding the boot image, but including allocation on behalf of Jikes RVM runtime system components such as the JIT compilers. Column "Time" is the runtime in seconds, using generous heap sizes (not shown). Our choice of workloads turns *hsql* into an extreme point with a comparatively long running time (2 minutes 33 seconds), and turns *soot* into an extreme point with a short running time (3 seconds) despite a large code base.

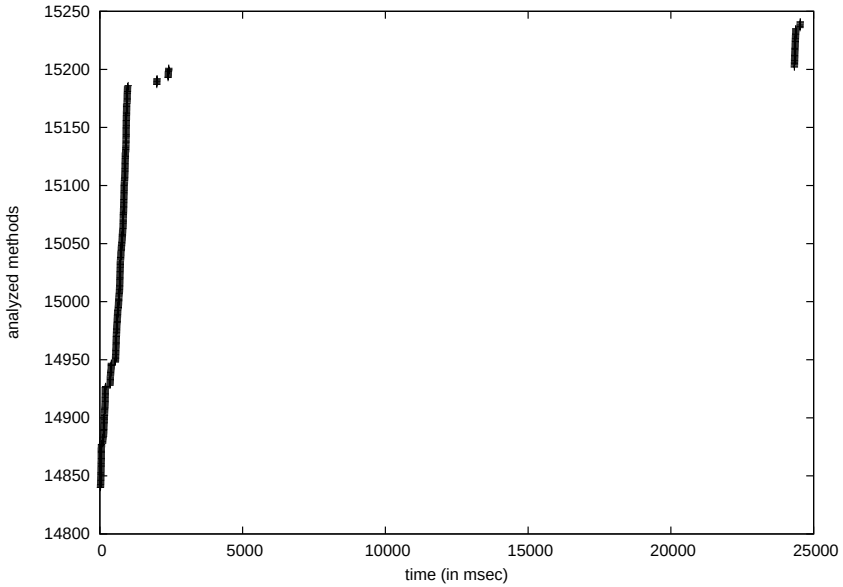



Fig. 13. Method compilation over time, for *mpegaudio*. The first shown data point is the *main*() method.

8.1.2 *Method compilation over time.* Fig. 13 shows how the number of analyzed methods increases over a run of *mpegaudio*. The experiment used Jikes RVM without our pointer analysis. The x-axis shows time in milli-seconds. The y-axis shows compiled methods, for which an online analysis would have to find constraints. The first data point is the *main*() method; all methods analyzed before that are either part of the boot image, or compiled during virtual machine start-up. The graphs for other benchmarks have a similar shape, and therefore we omit them.

Fig. 13 shows that after an initial warm-up period of around 0.3s, *mpegaudio* executes only methods that it has encountered before. For an online analysis, this means that behavior stabilizes quickly, at which point the analysis time overhead drops to zero. At the end of the run, there is a phase shift that requires new code. For an online analysis, this means that even after behavior appears to have stabilized, the analysis must be ready and able to incorporate new facts, and possibly even invalidate results used by clients. In general, the analysis needs to keep some data structures around for this, so its space overhead stays positive even when the time overhead asymptotically approaches zero.

8.1.3 *Infrastructure.* We implemented our pointer analysis in Jikes RVM 2.2.1. To evaluate our analysis in the context of a client of the analysis, we also implemented CBGC (connectivity-based garbage collection) in our version of Jikes RVM. CBGC [Hirzel et al. 2003] is a novel garbage collector that depends on pointer analysis for its effectiveness and efficiency. We use the partitioner component of CBGC to evaluate the precision of the analysis.

Since Andersen's analysis has cubic time complexity, optimizations that increase code size can dramatically increase analysis overhead. In our experience, too aggressive inlining can increase constraint propagation time by up to a factor of 5 for our benchmarks. We used Jikes RVM with its default adaptive optimization system, which performs inlining (and optimizations) only inside the hot application methods, but is more aggressive about methods in the boot image (see Section 7.4.5). We force Jikes RVM to be more cautious about inlining inside boot image methods by disabling inlining at build time. However, the adaptive system still recompiles some hot boot image methods if an inlining opportunity is discovered [Arnold et al. 2000].

## 8.2 Performance with all optimizations

This section evaluates the performance of our online analysis when all optimizations from Section 6 are enabled, and all performance bug fixes from Section 7.3 have been applied. Section 8.2.1 introduces terminology, Section 8.2.2 evaluates the memory usage, Section 8.2.3 evaluates the time for constraint finding, and Section 8.2.4 evaluates the time for constraint propagation.

8.2.1 *Terminology for propagator eagerness.* Performing Andersen's analysis offline requires $O(n^2)$ memory for representing pointsTo sets and other abstractions (Section 3.1), $O(n^2)$ time for constraint finding including call graph construction, and $O(n^3)$ time for constraint propagation (Fig. 2), where $n$ is the code size (indicated by the number of methods and classes in Table IV). Certain Java features prohibit offline pointer analysis (Section 2).

Performing Andersen's analysis online also requires $O(n^2)$ memory and $O(n^2)$ time for constraint finding including call graph construction. In the worst case, it requires $O(e \cdot i)$ time for constraint propagation, where $e$ (eagerness) is how often a client requests up-to-date analysis results, and $i$ (incrementality) is how long each repropagation takes (Fig. 7). In general, $i = O(n^3)$, but thanks to our worklist-driven architecture, it is much faster than propagating from scratch. Another change compared to offline analysis is that $n$ itself, the code size, differs. As discussed in Section 8.1.1, on the one hand $n$ is smaller in an online context, because

not all methods in the code base end up being compiled in a particular execution; and on the other hand, the code size $n$ is larger in a homogeneous-language system like Jikes RVM, because it includes the code for the runtime system itself.

Constraint propagation happens once offline after building the boot image 7.4.5, and then at runtime whenever a client of the pointer analysis needs points-to information and there are unresolved constraints. This paper investigates three kinds of propagation: eager, lazy, and at gc. *Eager* propagation occurs whenever an event from the left column of Fig. 7 generated new constraints. *Lazy* propagation occurs just once at the end of the program execution. *At gc* propagation occurs at the start of every garbage collection, and is an example for supporting a concrete client: connectivity-based garbage collection (CBGC) requires up-to-date points-to information at gc time [Hirzel et al. 2003].

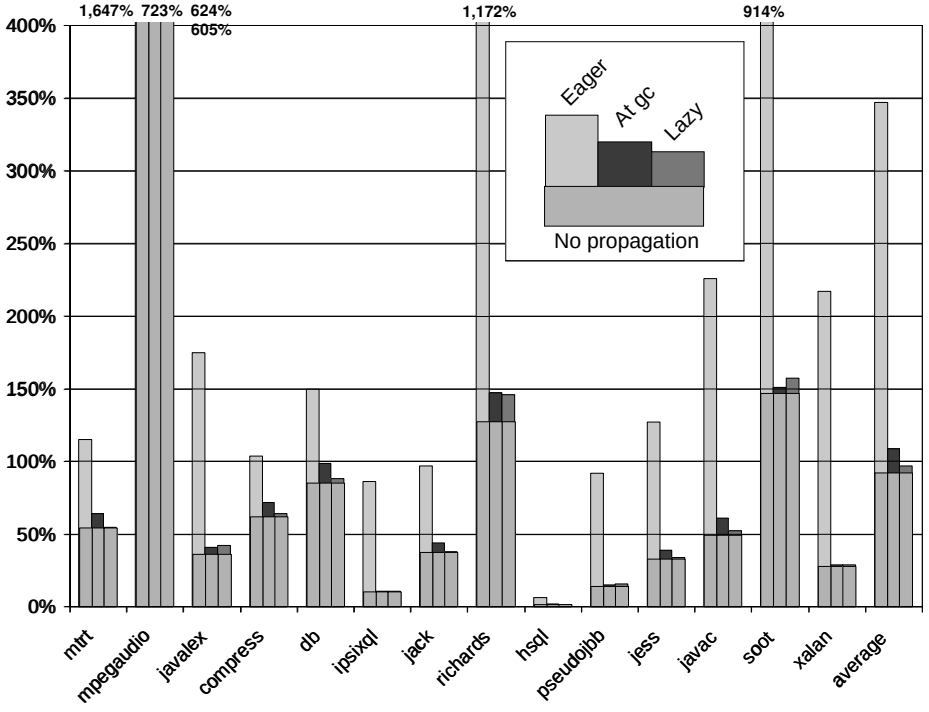



Fig. 14. Memory usage. Total allocation of the analysis, normalized to allocation without analysis.

8.2.2 *Memory usage.* Fig. 14 gives the amount allocated by our pointer analysis on top of what the application and run-time system allocate. These numbers are normalized to the total allocation without the analysis. For example, Column "Alloc." in Table IV shows that the average benchmark allocates a total of 359.4MB, and Fig. 8 shows that with propagation at gc, our analysis allocates another 109%, or 391.7MB, on top of that for the average benchmark.

The bottom part of each set of bars in Fig. 14 gives the space overhead of constraint finding. The difference between this number and the height of the bars gives the space overhead of constraint propagation. With lazy propagation or propagation at gc, most of the space overhead of the pointer analysis comes from the constraints. In other words, the shared bit-set representation for pointsTo sets from CLA [Heintze 1999] is reasonably compact for none-too-eager clients. We have not yet optimized the representation of other analysis data structures, such as flowTo sets, deferred unresolved constraints, and the call graph. Those data structures are allocated by the constraint finder, not the constraint propagator.

As the frequency of propagation increases, the space overhead also rises. Some data structures, such as deferred constraints or shared bit sets, become garbage and get recycled throughout the run. The seemingly high numbers for *richards* and *mpegaudio* are caused by normalizing to their low total allocation of 12MB and 15MB, respectively. For *soot*, normalized total allocation with eager propagation is high because *soot*'s base total allocation is relatively low (71MB), yet it has a large number of methods (17,023), and furthermore, it wraps many allocations in factory methods, which hurts analysis precision. Clients content with lower propagation frequency, such as CBGC, are rewarded with lower space overhead, but space overhead remains a challenge.

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 31.5MB to 69.4MB.

Table V. Time for constraint finding, normalized to time without analysis.

| Program | Analyzing methods | Resolving classes and arrays |
|---|---|---|
| *null* | *195%* | *24%* |
| mtrt | 10% | 1% |
| mpegaudio | 6% | 1% |
| javalex | 5% | 1% |
| compress | 3% | 0% |
| db | 4% | 0% |
| ipsixql | 28% | 7% |
| jack | 33% | 8% |
| richards | 54% | 17% |
| hsql | 1% | 0% |
| pseudojbb | 8% | 1% |
| jess | 23% | 4% |
| javac | 19% | 3% |
| soot | 312% | 37% |
| xalan | 19% | 5% |
| *average* | *38%* | *6%* |

8.2.3 *Time for constraint finding.* Table V gives the time overhead of generating constraints from methods (Column "Analyzing methods") and from resolution events (Column "Resolving classes and arrays"), on top of running time without either constraint generation or constraint propagation. Table V shows that generating constraints for methods is the dominant part of constraint generation. Also, as the benchmark runtime increases, the percentage of time spent in constraint

generation decreases. For example, the time spent in constraint finding is a negligible percentage of the runtime for our longest running benchmark, *hsql*, but is large for our shortest running benchmarks (e.g., *null*). Soot is a worst case for this experiment: it has a large code base but a short runtime (only 3 seconds).

Table VI. Time for constraint propagation, in seconds.

| Program | Eager $k \times$ (Avg±Std) = Total | At GC $k \times$ (Avg± Std) = Total | Lazy $1 \times$ Total |
|---|---|---|---|
| *null* | *2 × ( 2.3±3.2) = 4.6* | *1 × ( 4.7± 0.0) = 4.7* | *1 × 4.8* |
| mtrt | 315 × ( 0.1± 0.3) = 16.1 | 4 × ( 2.3± 2.0) = 9.0 | 1 × 6.8 |
| mpegaudio | 383 × ( 0.0± 0.3) = 17.2 | 4 × ( 2.3± 2.1) = 9.2 | 1 × 7.6 |
| javalex | 208 × ( 0.1± 0.4) = 26.6 | 2 × ( 4.6± 0.2) = 9.2 | 1 × 6.2 |
| compress | 162 × ( 0.1± 0.4) = 13.8 | 4 × ( 2.2± 2.0) = 8.7 | 1 × 6.6 |
| db | 181 × ( 0.1± 0.4) = 17.2 | 4 × ( 2.2± 2.0) = 9.0 | 1 × 6.1 |
| ipsixql | 485 × ( 0.1± 0.3) = 37.3 | 2 × ( 4.2± 0.9) = 8.4 | 1 × 6.2 |
| jack | 482 × ( 0.1± 0.3) = 39.5 | 4 × ( 3.1± 2.8) = 12.5 | 1 × 9.6 |
| richards | 438 × ( 0.0± 0.2) = 8.8 | 2 × ( 2.8± 2.8) = 5.5 | 1 × 5.6 |
| hsql | 482 × ( 0.1± 0.3) = 43.9 | 4 × ( 3.5± 4.2) = 14.1 | 1 × 12.0 |
| pseudojbb | 726 × ( 0.1± 0.2) = 44.3 | 2 × ( 6.2± 1.8) = 12.5 | 1 × 9.3 |
| jess | 833 × ( 0.1± 0.3) = 100.0 | 4 × ( 3.1± 2.8) = 12.3 | 1 × 8.7 |
| javac | 1,275 × ( 0.1± 0.3) = 182.3 | 6 × ( 3.7± 5.4) = 22.0 | 1 × 16.4 |
| soot | 1,588 × ( 0.1± 0.3) = 206.4 | 2 × (11.9±10.1) = 23.7 | 1 × 20.8 |
| xalan | 2,018 × ( 0.1± 0.2) = 149.3 | 2 × ( 8.3± 5.0) = 16.6 | 1 × 13.7 |
| *average* | *684 × ( 0.1±0.3) = 64.5* | *3 × ( 4.3± 3.2) = 12.3* | *1 × 9.7* |

8.2.4 *Time for constraint propagation.* Table VI shows the cost of constraint propagation. For example, with eager propagation on javac, the propagator runs 1,275 times, taking 0.1s on average with a standard deviation of 0.3s. All of the eager propagation pauses add up to 182.3s. The lazy propagation data gives an approximate sense for how long propagation would take if one were to wait for exactly the right moment to propagate just once, after all methods are compiled, but before the application does the bulk of its computation. For example, lazy propagation on javac would take 16.4s. Recall, however, that finding exactly the right moment to propagate is usually not possible (Section 2.3).

We see that for our algorithm with all the optimizations, an eager propagation takes on average 0.1 seconds. Table VI shows that if we propagate more frequently, individual propagations become faster. Thus, our incremental pointer analysis algorithm is effective in avoiding work on parts of the program that have not changed since the last propagation. That said, the total propagation times show that frequent propagations incur a non-trivial aggregate cost. Overall, individual propagations are fast and probably adequate for many clients of pointer analysis.

Fig. 15 presents the spread of propagation times for *javac*. A point (x,y) in this graph says that propagation "x" took "y" milliseconds. Out of 1,275 propagations in *javac*, 996 propagations take one-tenth of a second or less. This figure omits the offline propagation that happens at VM build time, and only shows propagations at application runtime. The most expensive run-time propagation is the first one (at x = 0 and y = 4.7 seconds, on the y-axis), because it finds a fixed-point for all

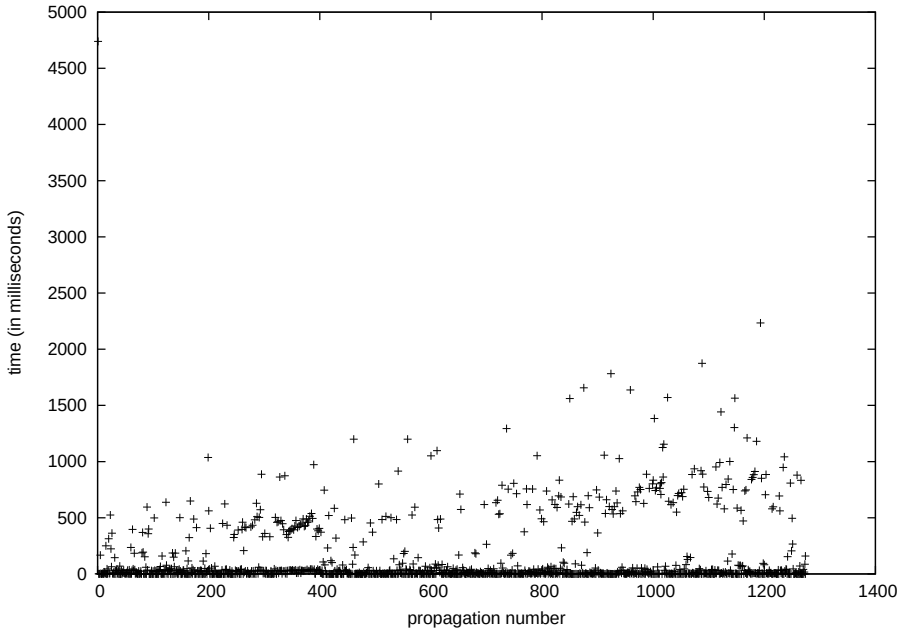



Fig. 15. Time for constraint propagation, for *javac* with eager propagation.

constraints that arise during VM startup, before Jikes RVM loads the application itself. The omitted graphs for other benchmarks have a similar shape. Fig. 15 considers eager propagation times; as discussed in Table VI, at gc propagations are slower individually, but there are fewer of them.

## 8.3 Effect of optimizations

The results presented so far enabled all of the techniques in Sections 6 and 7.3. This section explores how they affect propagation time individually and in combinations.

8.3.1 *Performance when incrementalizing.* Table VII shows the effect of individual optimizations from Section 6.1. For these experiments, all of the optimizations from Section 6.2 and all of the performance bug fixes from Section 7.3 are active. Comparing the average propagation cost of the first incremental propagator from Section 5.5 with the average propagation cost of the final algorithm from Section 6.1.4 shows that the incrementalizations are effective: the average propagation time drops from 1.2s to 0.1s. On average, the total propagation time drops from 750.7s to 64.5s, a factor of 12 speedup.

8.3.2 *Performance when varying sensitivity.* This section evaluates the three selective precision changes from Section 6.2, along with the performance bug fix of using a bit set for type filtering from Section 7.3.1. The other performance bug fixes had a smaller impact, and exploring bug fixes is less interesting, because one would want to include them in an implementation anyway. Therefore, for these

Table VII. Constraint propagation time for different propagators, in seconds. These numbers are slightly different from the "Eager" column in Table VI due to instrumentation artifacts.

| Program | Num. of Props | Prop. from new constraints (Section 5.5) Avg±Std | Prop. from new constraints (Section 5.5) Total | IsCharged for $h.f$ (Section 6.1.2) Avg±Std | IsCharged for $h.f$ (Section 6.1.2) Total | Caching charged $h.f$ (Section 6.1.3) Avg±Std | Caching charged $h.f$ (Section 6.1.3) Total | Caching charged $h$ (Section 6.1.4) Avg±Std | Caching charged $h$ (Section 6.1.4) Total |
|---|---|---|---|---|---|---|---|---|---|
| *null* | *2* | *4.9±5.7* | *9.9* | *4.8±6.8* | *9.6* | *4.0±5.7* | *8.0* | *2.3±3.2* | *4.6* |
| mtrt | 314 | 1.3± 0.9 | 401.9 | 0.4± 0.9 | 122.6 | 0.3± 0.7 | 83.3 | 0.1± 0.3 | 16.1 |
| mpegaudio | 382 | 1.2± 1.0 | 464.4 | 0.4± 0.8 | 149.3 | 0.3± 0.6 | 96.4 | 0.0± 0.3 | 17.2 |
| javalex | 209 | 1.8± 1.2 | 366.1 | 0.8± 1.2 | 160.6 | 0.6± 1.0 | 122.7 | 0.1± 0.4 | 26.6 |
| compress | 161 | 1.4± 1.1 | 231.8 | 0.5± 1.1 | 85.8 | 0.7± 1.6 | 105.5 | 0.1± 0.4 | 13.8 |
| db | 180 | 1.6± 1.6 | 286.1 | 0.6± 1.2 | 106.8 | 0.4± 0.9 | 69.5 | 0.1± 0.4 | 17.2 |
| ipsixql | 485 | 1.4± 1.1 | 699.0 | 0.6± 1.4 | 309.3 | 0.4± 0.9 | 211.7 | 0.1± 0.3 | 37.3 |
| jack | 481 | 1.7± 1.3 | 813.0 | 0.7± 1.5 | 341.6 | 0.4± 0.8 | 197.4 | 0.1± 0.3 | 39.5 |
| richards | 438 | 1.0± 0.4 | 441.4 | 0.2± 0.5 | 74.8 | 0.1± 0.4 | 54.7 | 0.0± 0.2 | 8.8 |
| hsql | 484 | 1.6± 1.4 | 778.4 | 0.7± 1.4 | 320.5 | 0.4± 0.8 | 199.9 | 0.1± 0.3 | 43.9 |
| pseudojbb | 726 | 1.5± 1.3 | 1,109.9 | 0.5± 1.3 | 336.9 | 0.3± 0.8 | 233.4 | 0.1± 0.2 | 44.3 |
| jess | 832 | 1.9± 1.6 | 1,591.4 | 1.0± 1.6 | 807.7 | 0.6± 1.0 | 523.0 | 0.1± 0.3 | 100.0 |
| javac | 1,274 | 2.3± 2.1 | 2,902.5 | 1.1± 1.8 | 1,366.3 | 0.7± 1.2 | 908.1 | 0.1± 0.3 | 182.3 |
| soot | 1,587 | 1.8± 1.4 | 2,805.4 | 0.8± 1.4 | 1,242.0 | 0.5± 1.0 | 856.7 | 0.1± 0.3 | 206.4 |
| xalan | 2,017 | 1.7± 1.2 | 3,394.4 | 0.6± 1.0 | 1,199.0 | 0.4± 0.8 | 899.9 | 0.1± 0.2 | 149.3 |
| *average* | *684* | *1.2±1.0* | *740.7* | *0.6±1.0* | *301.0* | *0.4±0.9* | *207.7* | *0.1±0.3* | *64.5* |

experiments, all optimizations from Sections 6.2, 7.3.2, and 7.3.3 are always active.

Section 8.3.2.1 shows the impact of individual optimizations, Section 8.3.2.2 explores interactions of multiple optimizations, and Section 8.3.3 evaluates how the optimizations affect precision.

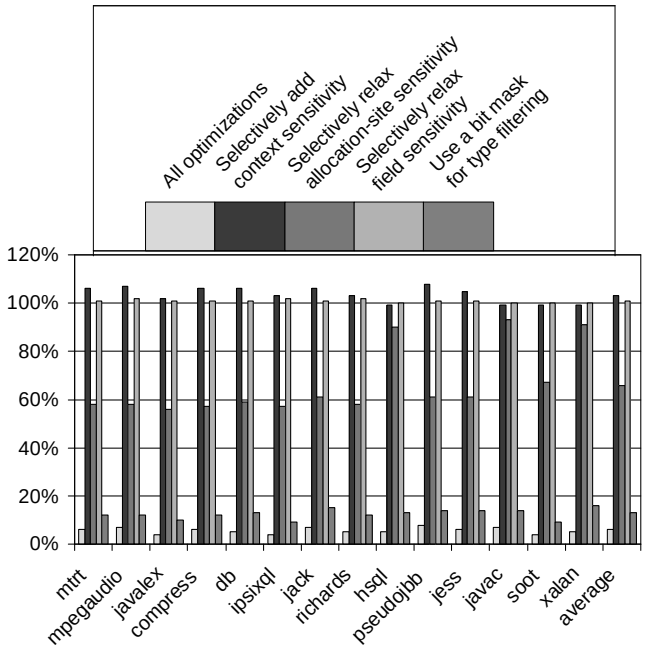



Fig. 16. Individual optimizations: cost in time compared to the unoptimized algorithm.

8.3.2.1 Individual optimizations. Fig. 16 gives the total propagation time when using the techniques from Sections 6.2 and 7.3.1 individually as a percentage of total propagation time without any of these techniques. The "All optimizations" bars show that the techniques collectively have a dramatic effect on performance; on average, they reduce propagation time to 6% of the propagation when all of them are inactive, a factor of 16 speedup. Taken individually, using a bit mask for type filtering is the most effective and relaxing allocation site sensitivity is also beneficial. The other two optimizations, taken alone, slightly degrade performance.

8.3.2.2 Pairs of optimizations. The optimizations have obvious interactions. For example, the optimization "selectively relax field sensitivity" is designed to alleviate the negative effects of context sensitivity; taken alone there is little opportunity for applying this optimization.

Table VIII. Interactions of pairs of optimizations: $\min(T_{O1}, T_{O2}) - T_{O1,O2}$, how much better is it to perform both optimizations than to perform only the better single optimization. The bars, left to right, are for *mtrt*, *mpegaudio*, *javalex*, *compress*, *db*, *ipsixql*, *jack*, *richards*, *hsql*, *pseudojbb*, *jess*, *javac*, *soot*, and *xalan*.

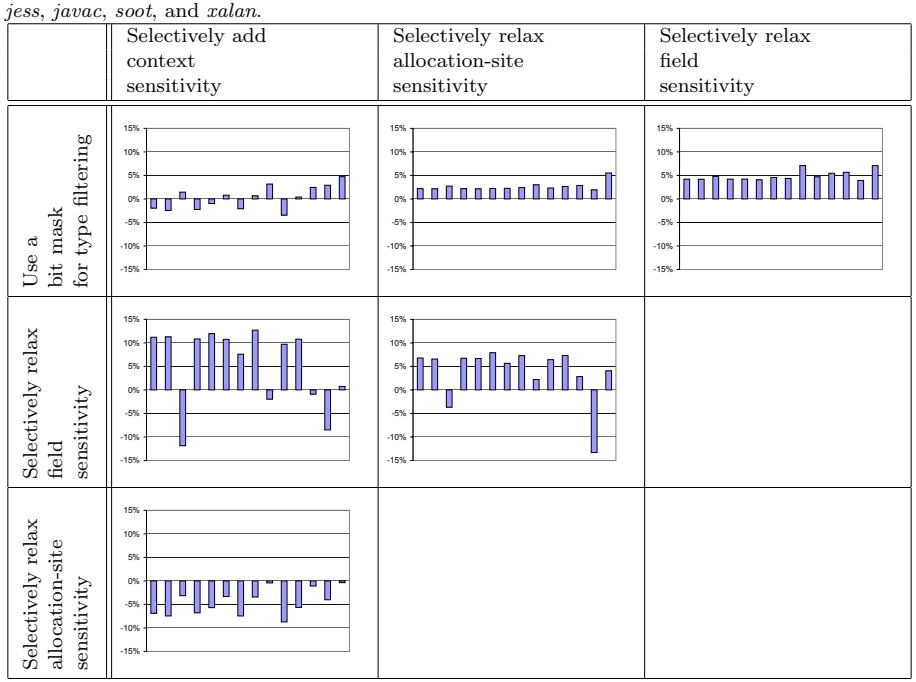



Table VIII explores how pairs of optimizations interact. Each bar shows $\min(T_{O1}, T_{O2}) - T_{O1,O2}$ for the pair $(O1, O2)$ of optimizations given by the table cell, and for the benchmark given by the order of bars specified in the caption. The value $T_O$ is the total propagation time when using optimization $O$ alone, as given in Fig. 16. The value $T_{O1,O2}$ is the total propagation time when applying both $O1$ and $O2$ in the same run. When $\min(T_{O1}, T_{O2}) > T_{O1,O2}$, the bar is positive,

meaning that the combined optimizations sped up propagation more than each one individually.

Using a bit mask for type filtering interacts well with selectively relaxing allocation-site or field sensitivity. This is not surprising: all three optimizations reduce analysis data structures. Selectively adding context sensitivity, on the other hand, increases analysis data structures. When using just a pair of optimizations, selective context sensitivity sometimes interacts positively with bit masks or selective field sensitivity, but always interacts negatively with selective allocation site sensitivity.

8.3.2.3 Triples of optimizations. Table IX explores how triples of optimizations interact. Each bar shows $\min(T_{O1,O2}, T_{O3}) - T_{O1,O2,O3}$, where the row gives the pair of optimizations $(O1, O2)$, the column gives the third optimization $O3$, and the caption tells which bar corresponds to which benchmark. This table is similar to Table VIII: when $\min(T_{O1,O2}, T_{O3}) > T_{O1,O2,O3}$, the bar is positive, meaning that the triple $(O1, O2, O3)$ led to better propagation time than either $(O1, O2)$ or $O3$. Each row has empty cells for the two columns that are already involved in the pair of optimizations.

Table IX, Row "Context and allocation-site", Column "Selectively relax field sensitivity" shows the strongest positive interaction. The row itself corresponds to the worst pair-wise interaction from Table VIII. This motivates why we introduced the optimization "Selectively relax field sensitivity": in our experiments, it it necessary for reaping the full benefit from the optimization "Selectively add context sensitivity". Of course, we relax field sensitivity for exactly those fields for which context sensitivity behaves badly otherwise.

8.3.3 *Precision.* While we designed all the optimizations in Section 6 to improve performance, the three optimizations in Section 6.2 also affect precision. We evaluate their effect on the precision of our pointer analysis with respect to a client of the pointer analysis: the partitioner in CBGC (connectivity-based garbage collection [Hirzel et al. 2003]). Since these optimizations change precision in incomparable ways (they use a different heap model), it is not appropriate to compare their relative effects using pointsTo sets or alias sets [Diwan et al. 2001].

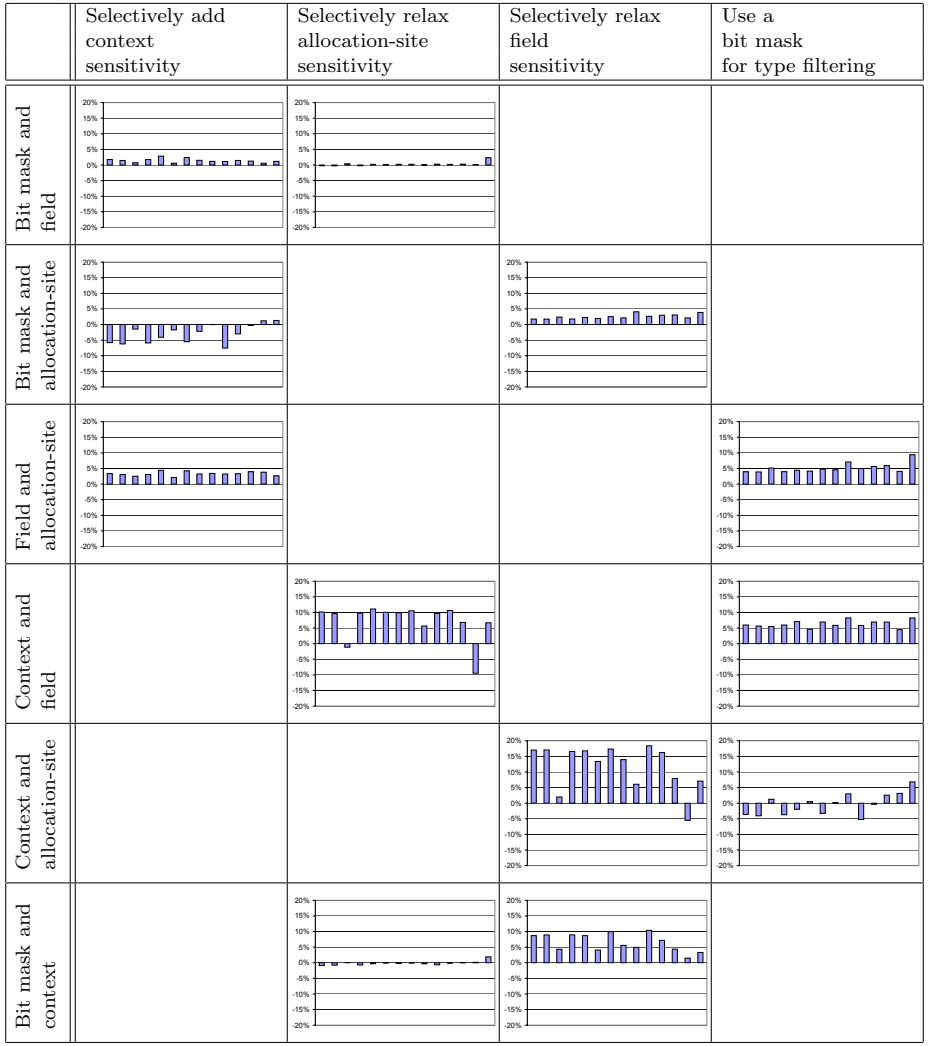
The CBGC partitioner works by placing each allocation site in its own partition and then collapsing strongly-connected components in the partition graph. Since a less precise pointer analysis has (potentially) more points-to edges than a more precise analysis, it also has fewer, larger strongly-connected components than the more precise analysis. Thus, a less precise analysis will have fewer partitions than a more precise analysis. Fig. 17 gives the change in the number of partitions relative to using none of these optimizations.

Fig. 17 shows that context sensitivity slightly improves precision, while the other optimizations have a negligible impact on precision. In other words, collectively these optimizations improve both precision and performance of our pointer-analysis client.

## 9. RELATED WORK

Section 9.1 discusses work related to the offline analysis from Section 4, Section 9.2 discusses work related to the online analysis from Section 5, Section 9.3 discusses work related to the analysis optimizations from Section 6, and Section 9.4 discusses work related to our validation technique from Section 7.1.

Table IX. Interactions of triples of optimizations: $\min(T_{O1,O2}, T_{O3}) - T_{O1,O2,O3}$, how much better is it to add a third optimization than just performing a pair or the third one alone. The bars, left to right, are for *mtrt*, *mpegaudio*, *javalex*, *compress*, *db*, *ipsixql*, *jack*, *richards*, *hsql*, *pseudojbb*, *jess*, *javac*, *soot*, and *xalan*.




### 9.1 Offline analysis

Section 9.1.1 puts our analysis on the map, Section 9.1.2 discusses related offline analyses, and Section 9.1.3 describes related work on how to represent analysis data structures.

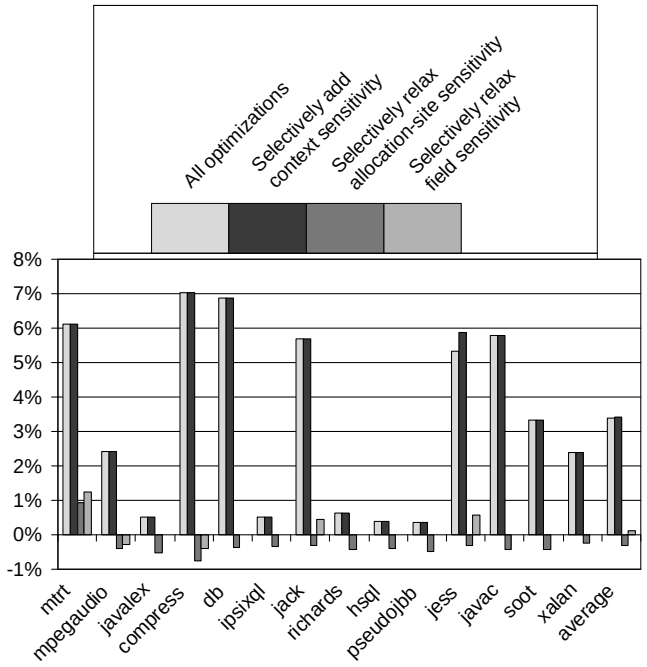



Fig. 17. Precision. How the optimizations change the number of CBGC partitions.

9.1.1 *Where does Andersen fit in.* The body of literature on pointer analyses is vast [Hind 2001]. At one extreme, exemplified by Steensgaard [1996] and type-based analyses [Harris 1999; Tip and Palsberg 2000; Diwan et al. 2001], the analyses are fast, but imprecise. At the other extreme, exemplified by shape analyses [Hendren 1990; Sagiv et al. 1999], 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 CBGC (connectivity-based garbage collection, [Hirzel et al. 2003]). On the one hand, our experiments found that type-based analyses are too imprecise for CBGC. On the other hand, we felt that much more precise shape analysis or context-sensitive analysis would probably be too expensive in an online context. This left us with a choice between Steensgaard's [1996] and Andersen's [1994] analysis. Andersen's analysis is less efficient, but more precise [Shapiro and Horwitz 1997; Hind and Pioli 2000]. 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.

9.1.2 *Andersen for "static Java".* A number of papers describe how to use Andersen's analysis for a subset of Java without features such as dynamic class loading, reflection, or native code [Liang et al. 2001; Rountev et al. 2001; Whaley and Lam 2002; Lhoták and Hendren 2003]. We will refer to this subset language as "static Java". The above papers present solutions for static Java features that make pointer analyses difficult, such as object fields, virtual method invocations, etc.

Rountev, Milanova, and Ryder [2001] formalize Andersen's analysis for static Java using set constraints, which enables them to solve it with BANE (Berkeley ANalysis Engine) [Fähndrich et al. 1998]. Liang, Pennings, and Harrold [2001] compare both Steensgaard's and Andersen's analysis for static Java, and evaluate trade-offs for handling fields and the call graph. Whaley and Lam [2002] improve the efficiency of Andersen's analysis by using implementation techniques from CLA [Heintze and Tardieu 2001b], and improve the precision by adding flow-sensitivity for local variables. Lhoták and Hendren [2003] present SPARK (Soot Pointer Analysis Research Kit), an implementation of Andersen's analysis in Soot [Vallée-Rai et al. 2000], which provides precision and efficiency tradeoffs for various components.

9.1.3 *Representation.* There are many alternatives for storing the flow and pointsTo sets. For example, we represent the data flow between $v$-nodes and $h.f$-nodes implicitly, whereas BANE represents it explicitly [Foster et al. 1997; Rountev et al. 2001]. Thus, our analysis saves space compared to BANE, but may have to perform more work at propagation time. As another example, CLA [Heintze and Tardieu 2001b] stores reverse pointsTo sets at $h$-nodes, instead of storing forward pointsTo sets at $v$-nodes and $h.f$-nodes. The forward pointsTo 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 flowFrom sets.

Finally, there are many choices for how to implement the sets. The SPARK paper evaluates various data structures for representing pointsTo sets [Lhoták and Hendren 2003], 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 [Heintze 1999] to be even more efficient than the hybrid sets used by SPARK. Another shared set representation is BDDs, which have recently become popular for representing sets in pointer analysis [Berndl et al. 2003]. In our experience, Heintze's shared bit vectors are highly efficient; in fact, we speculate that they are a main reason for the good performance of CLA [Heintze and Tardieu 2001b].

## 9.2 Online interprocedural analysis

Section 9.2.1 describes extant analysis, an offline analysis that can yield some of the benefits of online analysis. Section 9.2.2 discusses how clients can deal with the fact that results from online analysis change over time. Section 9.2.3 describes online analyses that deal with Java's dynamic class loading feature.

9.2.1 *Extant analysis.* Sreedhar, Burke, and Choi [2000] 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.

9.2.2 *Invalidation.* Pechtchanski and Sarkar [2001] 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.

9.2.3 *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 validates its results. Furthermore, all are less precise than Andersen's analysis.

Bogda and Singh [2001] and King [2003] adapt Ruf's escape analysis [2000] 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 [Choi et al. 1999]. 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 [Bacon and Sweeney 1996].

Qian and Hendren [2004] adapt Tip and Palsberg's XTA [2000] 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 to base XTA. 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 5.3.

### 9.3 Varying analysis precision

There has also been significant prior work in varying analysis precision to improve performance. Ryder [Ryder 2003] describes precision choices for modeling program entities in a reference analysis (pointer analysis, or weaker type-based variants). Plevyak and Chien [Plevyak and Chien 1994] and Guyer and Lin [Guyer and Lin 2003] describe mechanisms for automatically varying flow and context sensitivity during the analysis. Although those mechanisms happen in an offline, whole-world analysis, and thus, do not immediately apply to online analysis, we believe they can inspire approaches to automate online precision choices.

### 9.4 Validation

Our validation methodology compares pointsTo 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 [Larus and Chandra 1993; Diwan et al. 2001; Liang et al. 2002].

## 10. 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 concrete 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 runtime. Because any inefficiencies in online analysis directly degrade application performance, this paper discusses and evaluates various implementation issues that affect analysis performance. This paper presents several optimizations that improve the performance of online pointer analysis by almost two orders of magnitude compared to our previous paper. On average over our benchmark suite, if the analysis recomputes points-to results upon each program change, most analysis pauses take under 0.1 seconds.

## REFERENCES


Agrawal, G., Li, J., and Su, Q. 2002. Evaluating a demand driven technique for call graph construction. In *11th International Conference on Compiler Construction (CC)*. LNCS, vol. 2304. 29–45.

Alpern, B., Attanasio, C. R., Barton, J. J., Burke, M. G., Cheng, P., Choi, J.-D., Cocchi, A., Fink, S. J., Grove, D., Hind, M., Hummel, S. F., Lieber, D., Litvinov, V., Mergen, M. F., Ngo, T., Russell, J. R., Sarkar, V., Serrano, M. J., Shepherd, J. C., Smith, S. E., Sreedhar, V. C., Srinivasan, H., and Whaley, J. 2000. The Jalapeño virtual machine. *IBM Systems Journal 39,* 1 (Feb.), 211–238.

Andersen, L. O. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU, University of Copenhagen. DIKU report 94/19, available at `ftp://ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z`.

Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. F. 2000. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices 35,* 10 (Oct.), 47–65. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Arnold, M. and Ryder, B. G. 2002. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In *16th European Conference on Object-Oriented Programming (ECOOP)*. LNCS, vol. 2374.

Bacon, D. F. and Sweeney, P. F. 1996. Fast static analysis of C++ virtual function calls. *ACM SIGPLAN Notices 31,* 10 (Oct.), 324–341. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Berndl, M., Lhoták, O., Qian, F., Hendren, L., and Umanee, N. 2003. Points-to analysis using BDDs. *ACM SIGPLAN Notices 38,* 5 (May), 103–114. In *Conference on Programming Language Design and Implementation (PLDI)*.

Bogda, J. and Singh, A. 2001. Can a shape analysis work at run-time? In *Java Virtual Machine Research and Technology Symposium (JVM)*. 13–26.

Burke, M. and Torczon, L. 1993. Interprocedural optimization: Eliminating unnecessary recompilation. *Transactions on Programming Languages and Systems (TOPLAS) 15,* 3 (July), 367–399.

Chatterjee, R., Ryder, B. G., and Landi, W. A. 1999. Relevant context inference. In *26th Symposium on Principles of Programming Languages (POPL)*. 133–146.

Cheng, B.-C. and Hwu, W.-m. W. 2000. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. *ACM SIGPLAN Notices 35,* 5 (May), 57–69. In *Conference on Programming Language Design and Implementation (PLDI)*.

Choi, J.-D., Grove, D., Hind, M., and Sarkar, V. 1999. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. 21–31.

Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V. C., and Midkiff, S. 1999. Escape analysis for Java. *ACM SIGPLAN Notices 34,* 10 (Oct.), 1–19. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Christensen, A. S., Møller, A., and Schwartzbach, M. I. 2003. Precise analysis of string expressions. In *10th International Static Analysis Symposium (SAS)*. LNCS, vol. 2694. 1–18.

Cierniak, M., Lueh, G.-Y., and Stichnoth, J. M. 2000. Practicing JUDO: Java under dynamic optimizations. *ACM SIGPLAN Notices 35,* 5 (May), 13–26. In *Conference on Programming Language Design and Implementation (PLDI)*.

Cooper, K. D., Kennedy, K., and Torczon, L. 1986. Interprocedural optimization: eliminating unnecessary recompilation. *ACM SIGPLAN Notices 21,* 7 (July), 58–67. In *Symposium on Compiler Construction (SCC)*.

Das, M. 2000. Unification-based pointer analysis with directional assignments. *ACM SIGPLAN Notices 35,* 5 (May), 35–46. In *Conference on Programming Language Design and Implementation (PLDI)*.

Dean, J., Grove, D., and Chambers, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *9th European Conference on Object-Oriented Programming (ECOOP)*. LNCS, vol. 952. 77–101.

Detlefs, D. and Agesen, O. 1999. Inlining of virtual methods. In *13th European Conference on Object-Oriented Programming (ECOOP)*. LNCS, vol. 1628. 258–278.

Diwan, A., McKinley, K. S., and Moss, J. E. B. 2001. Using types to analyze and optimize object-oriented programs. *Transactions on Programming Languages and Systems (TOPLAS) 23,* 1 (Jan.), 30–72.

Duesterwald, E., Gupta, R., and Soffa, M. L. 1997. A practical framework for demand-driven interprocedural data flow analysis. *Transactions on Programming Languages and Systems (TOPLAS) 19,* 6 (Nov.), 992–1030.

Fähndrich, M., Foster, J. S., Su, Z., and Aiken, A. 1998. Partial online cycle elimination in inclusion constraint graphs. *ACM SIGPLAN Notices 33,* 5 (May), 85–96. In *Conference on Programming Language Design and Implementation (PLDI)*.

Fernández, M. F. 1995. Simple and effective link-time optimization of Modula-3 programs. *ACM SIGPLAN Notices 30,* 6 (June), 103–115. In *Conference on Programming Language Design and Implementation (PLDI)*.

Fink, S. J. and Qian, F. 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization (CGO)*. 241–252.

Foster, J. S., Fähndrich, M., and Aiken, A. 1997. Flow-insensitive points-to analysis with term and set constraints. Tech. Rep. UCB/CSD-97-964, University of California at Berkeley. Aug.

Grove, D. 1998. Effective interprocedural optimization of object-oriented languages. Ph.D. thesis, University of Washington.

Guyer, S. and Lin, C. 2003. Client-driven pointer analysis. In *10th International Static Analysis Symposium (SAS)*. LNCS, vol. 2694. 214–236.

Hall, M. W., Mellor-Crummey, J. M., Carle, A., and Rodriguez, R. G. 1993. Fiat: A framework for interprocedural analysis and transformations. In *6th Workshop on Languages and Compilers for Parallel Computing (LCPC)*. LNCS, vol. 768. 522–545.

Harris, T. 1999. Early storage reclamation in a tracing garbage collector. *ACM SIGPLAN Notices 34,* 4 (Apr.), 46–53. In *International Symposium on Memory Management (ISMM)*.

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

Heintze, N. and Tardieu, O. 2001a. Demand-driven pointer analysis. *ACM SIGPLAN Notices 36,* 5 (May), 24–34. In *Conference on Programming Language Design and Implementation (PLDI)*.

Heintze, N. and Tardieu, O. 2001b. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. *ACM SIGPLAN Notices 36,* 5 (May), 254–263. In *Conference on Programming Language Design and Implementation (PLDI)*.

Hendren, L. 1990. Parallelizing programs with recursive data structures. Ph.D. thesis, Cornell University.

Hind, M. 2001. Pointer analysis: haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. 54–61. Invited talk.

Hind, M. and Pioli, A. 2000. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis (ISSTA)*. 113–123.

Hirzel, M., Diwan, A., and Hertz, M. 2003. Connectivity-based garbage collection. *ACM SIGPLAN Notices 38,* 11 (Nov.), 359–373. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Hirzel, M., Diwan, A., and Hind, M. 2004. Pointer analysis in the pressence of dynamic class loading. In *18th European Conference on Object-Oriented Programming (ECOOP)*. LNCS, vol. 3086. 96–122.

Hirzel, M., Henkel, J., Diwan, A., and Hind, M. 2003. Understanding the connectivity of heap objects. *ACM SIGPLAN Notices 38,* 2s (Feb.), 143–156. In *International Symposium on Memory Management (ISMM)*.

Hölzle, U., Chambers, C., and Ungar, D. 1992. Debugging optimized code with dynamic deoptimization. *ACM SIGPLAN Notices 27,* 7 (July), 32–43. In *Conference on Programming Language Design and Implementation (PLDI)*.

Hölzle, U. and Ungar, D. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. *ACM SIGPLAN Notices 29,* 6 (June), 326–336. In *Conference on Programming Language Design and Implementation (PLDI)*.

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

Larus, J. R. and Chandra, S. 1993. Using tracing and dynamic slicing to tune compilers. Tech. Rep. 1174. Aug.

Lattner, C. and Adve, V. 2003. 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.

Le, A., Lhoták, O., and Hendren, L. 2005. Using inter-procedural side-effect information in JIT optimizations. In *14th International Conference on Compiler Construction (CC)*. LNCS, vol. 3443. 287–304.

Lhoták, O. and Hendren, L. 2003. Scaling Java points-to analysis using SPARK. In *12th International Conference on Compiler Construction (CC)*. LNCS, vol. 2622. 153–169.

Liang, D. and Harrold, M. J. 1999. Efficient points-to analysis for whole-program analysis. In *Lecture Notes in Computer Science, 1687*, O. Nierstrasz and M. Lemoine, Eds. Springer-Verlag, 199–215. In *7th European Software Engineering Conference, and International Symposium on Foundations of Software Engineering (ESEC/FSE)*.

Liang, D., Pennings, M., and Harrold, M. J. 2001. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. 73–79. Supplement to ACM SIGPLAN Notices.

Liang, D., Pennings, M., and Harrold, M. J. 2002. Evaluating the precision of static reference analysis using profiling. In *International Symposium on Software Testing and Analysis (ISSTA)*. 22–32.

Lindholm, T. and Yellin, F. 1999. *The Java virtual machine specification*, second ed. Addison-Wesley.

Paleczny, M., Vick, C., and Click, C. 2001. The Java Hotspot server compiler. In *Java Virtual Machine Research and Technology Symposium (JVM)*. 1–12.

Pechtchanski, I. and Sarkar, V. 2001. Dynamic optimistic interprocedural analysis: a framework and an application. *ACM SIGPLAN Notices 36,* 11 (Nov.), 195–210. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Plevyak, J. and Chien, A. A. 1994. Precise concrete type inference for object-oriented languages. *ACM SIGPLAN Notices 29,* 10 (Oct.), 324–324. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

Qian, F. and Hendren, L. 2004. Towards dynamic interprocedural analysis in JVMs. In *3rd Virtual Machine Research and Technology Symposium (VM)*. 139–150.

Qian, F. and Hendren, L. 2005. A study of type analysis for speculative method inlining in a JIT environment. In *14th International Conference on Compiler Construction (CC)*. LNCS, vol. 3443. 255–270.

Rountev, A. and Chandra, S. 2000. Off-line variable substitution for scaling points-to analysis. *ACM SIGPLAN Notices 35,* 5 (May), 47–56. In *Conference on Programming Language Design and Implementation (PLDI)*.

Rountev, A., Milanova, A., and Ryder, B. G. 2001. Points-to analysis for Java using annotated constraints. *ACM SIGPLAN Notices 36,* 11 (Nov.), 43–55. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Ruf, E. 2000. Effective synchronization removal for Java. *ACM SIGPLAN Notices 35,* 5 (May), 208–218. In *Conference on Programming Language Design and Implementation (PLDI)*.

Ryder, B. G. 2003. Dimensions of precision in reference analysis for object-oriented programming languages. In *12th International Conference on Compiler Construction (CC)*, G. Hedin, Ed. LNCS, vol. 2622. Springer, Warsaw, Poland, 126–137.

Sagiv, M., Reps, T., and Wilhelm, R. 1999. Parametric shape analysis via 3-valued logic. In *26th Symposium on Principles of Programming Languages (POPL)*. 105–118.

Serrano, M., Bordawekar, R., Midkiff, S., and Gupta, M. 2000. Quicksilver: a quasi-static compiler for Java. *ACM SIGPLAN Notices 35,* 10 (Oct.), 66–82. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Shapiro, M. and Horwitz, S. 1997. The effects of the precision of pointer analysis. In *4th International Symposium on Static Analysis (SAS)*. LNCS, vol. 1302. 16–34.

Sreedhar, V. C., Burke, M., and Choi, J.-D. 2000. A framework for interprocedural optimization in the presence of dynamic class loading. *ACM SIGPLAN Notices 35,* 5 (May), 196–207. In *Conference on Programming Language Design and Implementation (PLDI)*.

Steensgaard, B. 1996. Points-to analysis in almost linear time. In *23rd Symposium on Principles of Programming Languages (POPL)*. 32–41.

Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H., and Nakatani, T. 2001. A dynamic optimization framework for a Java just-in-time compiler. *ACM SIGPLAN Notices 36,* 11 (Nov.), 180–195. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., and Godin, C. 2000. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices 35,* 10 (Oct.), 264–280. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

The Apache Tomcat Project. Apache Tomcat. `http://jakarta.apache.org/tomcat`.

The Eclipse Project. Eclipse. `http://www.eclipse.org`.

Tip, F. and Palsberg, J. 2000. Scalable propagation-based call graph construction algorithms. *ACM SIGPLAN Notices 35,* 10 (Oct.), 281–293. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Vallée-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., and Sundaresan, V. 2000. Optimizing Java bytecode using the Soot framework: Is it feasible? In *9th International Conference on Compiler Construction (CC)*. LNCS, vol. 1781. 18–34.

Vivien, F. and Rinard, M. 2001. Incrementalized pointer and escape analysis. *ACM SIGPLAN Notices 36,* 5 (May), 35–46. In *Conference on Programming Language Design and Implementation (PLDI)*.

Whaley, J. and Lam, M. 2002. An efficient inclusion-based points-to analysis for strictly-typed languages. In *9th International Symposium on Static Analysis (SAS)*. LNCS, vol. 2477. 180–195.