Fast Online Pointer Analysis

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

Pointer analysis benefits many useful clients, such as compiler optimizations and bug finding tools. Unfortunately, common programming language features such as dynamic loading, reflection, and foreign language interfaces, make pointer analysis difficult. This article describes how to deal with these features by performing pointer analysis online during program execution. For example, dynamic loading may load code that is not available for analysis before the program starts. Only an online analysis can analyze such code, and thus support clients that optimize or find bugs in it. This article identifies all problems in performing Andersen's pointer analysis for the full Java language, presents solutions to these problems, and uses a full implementation of the solutions in a Java virtual machine 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, and add up to 64.5 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

ACM Reference Format:

Hirzel, M., von Dincklage, D., Diwan, A., and Hind, M. 2007. Fast online pointer analysis. ACM Trans. Program. Lang. Syst. 29, 2, Article 11 (April 2007), 55 pages. DOI = 10.1145/1216374.1216379 http://doi.acm.org/10.1145/1216374.1216379.

A preliminary version of parts of this article appeared in the *European Conference on Object-Oriented Programming 2004.*

This work was supported by NSF ITR Grant CCR-0085792, an NSF Career Award CCR-0133457, and 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.

Authors' addresses: M. Hirzel, M. Hind, IBM T. J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532; email: {hirzel,hindm}@us.ibm.com; D. von Dincklage, A. Diwan, University of Colorado, Boulder, CO 80309; email: {danielvd,diwan}@colorado.edu.

Permission to make digital or hard copies part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2007 ACM 0164-0925/2007/04-ART11 $5.00. DOI 10.1145/1216374.1216379 http://doi.acm.org/10.1145/1216374.1216379

1. INTRODUCTION

Pointer analysis benefits many useful clients, such as optimizations (e.g., in the compiler or garbage collector) and programmer productivity tools (e.g., for bug finding or refactoring). Pointer analysis is useful whenever a client depends on possible values of pointers at runtime.

An *online* analysis occurs during program execution. In contrast, an *offline* analysis completes before the program begins executing. An online analysis incrementally analyzes new code when it is dynamically loaded into the running program. Some of that code may not be available at all for offline analysis before the program begins executing, and thus can only be analyzed by online analysis.

Besides being the natural way to deal with dynamically loaded code, an online analysis also enables novel techniques for dealing with reflection and foreign language interfaces. All three features are commonly used in Java, both in the standard libraries and in many user programs. We developed the first online version of Andersen's pointer analysis [1994]. Our online analysis deals with all Java language features in the general setting of an executing Java virtual machine. Thus, the benefits of points-to information become available to optimizations in just-in-time compilers and other components of the language runtime system. Online analysis provides information about the entire program, including parts that are not available offline, for which an offline analysis cannot provide any information.

The rest of this section describes challenges that dynamically loaded code, reflection, and foreign language interfaces (Sections 1.1 to 1.3, respectively) present for program analysis, the possible space of solutions (Section 1.4), and our specific solution to these challenges (Section 1.5). Section 1.6 outlines the rest of this article.

1.1 Dynamic Loading

An offline analysis does not know where code will be loaded from at runtime, which code will be loaded, and whether a given piece of code will be loaded. For these reasons, some of the code can be unavailable to offline analysis, hampering its ability to yield useful information.

1.1.1 *Offline Analysis Does Not Know Where Code Will Be Loaded From.* Applications can "load" code by *generating* it during execution, rather than reading it from a file. A user-defined class loader in Java can create code in an array of bytes at runtime, then hand it to the underlying runtime system to execute with ClassLoader. *defineClass* (**byte**[] b). Other languages have similar features. Many production-strength applications use runtime code generation to create custom-tailored code snippets for performance-critical tasks, such as serialization or data structure traversal. In fact, using jar and grep on the bytecode files of the Java 1.5 libraries reveals that they use *defineClass* for XML processing.

Applications can also load code over the *network*. In Java, a user-defined class loader can download a stream of bytes from another machine, then call *defineClass* for it. This feature is popular for distributed extensible applications.

Another situation where parts of the program are not available to offline analysis is *library development*. The developer of a library cannot analyze all code offline because some of that code will be written by users of the library. This restricts the scope of clients of the analysis, such as optimizations and productivity tools.

1.1.2 *Offline Analysis Does Not Know Which Code Will Be Loaded.* Dynamic loading poses problems for an offline analysis, even when all code is available before runtime. In Java, an application may load code with 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 any particular run loads only one calendar class. Even worse, the relevant classes may be available only in the execution environment, and not in the development environment.

1.1.3 *Offline Analysis Does Not Know Whether a Given Piece of Code Will Be Loaded.* Dynamic loading poses problems for an offline analysis, even when all code is available before runtime and there is no explicit class loading. In Java, the runtime system 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 indeterminable: This depends on the input, and even when the input is known, it is undecidable. In general, an offline whole-program analysis has to analyze more code than gets executed, making it less efficient (analyzing more code wastes time and space) and less precise (the code may exhibit behavior never encountered at runtime). In situations like these, online analysis can be both more efficient and more precise.

1.1.4 *Issues with Unavailable Code.* Section 1.1.1 describes how some code can be unavailable for offline analysis. Offline analysis only analyzes the available code, and therefore does not supply its clients with information about the unavailable code. For example, optimizations cannot use offline analysis to improve the performance of code that is not available offline. Similarly, bug finding tools based on offline analysis will miss bugs in unavailable code.

When some code is unavailable offline, even the code that *is* available is harder to analyze because the offline analysis must make assumptions about what happens at the boundary to unavailable code. This typically means sacrificing a desirable analysis property: (i) Offline analysis can make pessimistic assumptions at the boundary, sacrificing some degree of precision. (ii) Alternatively, it can make optimistic assumptions at the boundary, sacrificing soundness. (iii) Offline analysis can require the user to write a specification for what the unavailable code does, sacrificing ease-of-use and maintainability. (iv) Finally, offline analysis can reject programs involving unavailable code, sacrificing the ability to analyze programs that use certain language features, either directly or through the standard libraries.

Online analysis avoids these issues because all code that is executed is available to online analysis. Thus, it can support online clients that operate on code which is not available offline. Also, it does not need to sacrifice precision, soundness, usability, or coverage at the boundary to unavailable code because there is no boundary.

1.2 Reflection

Reflection allows expressing certain design patterns and frameworks more conveniently that are hard or impossible to express within the confines of more traditional language features. For example, Java libraries can use reflection to implement a class-independent *clone* method. When a program analysis tries to predict the effect of reflection, the result is often quite imprecise because reflection can manipulate the arbitrary program entities named by strings that are constructed at runtime. Hence, many offline analyses elect to be unsound with respect to reflection rather than to sacrifice precision, or elect to require manual specifications of the possible effect of reflection in a particular program or library. Online analysis can avoid these issues by waiting for reflection to actually happen, and then updating its results if the program uses reflection in a way it has not before. This prevents the loss of precision, soundness, or usability that offline analysis may experience.

1.3 Foreign Language Interfaces

Foreign language interfaces allow code written in different languages to interoperate. For example, the JNI (Java native interface) allows bidirectional data and control flow between Java and native code, such as compiled C code. This allows interaction with the operating system and with legacy libraries, and gives developers the choice of the right language for the right task. For example, file I/O in Java is implemented using JNI. To analyze the whole application, we would need an analysis for each of the languages involved. But even this may not be enough when the code in the foreign language is unavailable; for example, JNI loads native code dynamically.

In the absence of an analysis for the foreign language, we can analyze just the part of the application written in one language. This means that the analysis must make assumptions at the boundary to the foreign code: It can be pessimistic (sacrificing some precision), optimistic (sacrificing soundness), require user specifications, or reject certain programs.

Online analysis gives an alternative: It can instrument the places wherein the analyzed language makes calls to, or receives calls from, the foreign language. The instrumentation checks whether the foreign code affects the analyzed code in a way it has not before, and updates analysis results accordingly. This prevents the loss of precision, soundness, or usability that offline analysis may experience, without requiring a separate analysis for the foreign code. In addition, online analysis does not require all foreign code to be available offline.

1.4 Offline vs. Online: Picking the Right Analysis

Clients of program analyses have some or all of the following requirements, but may prioritize them differently:

—*Code coverage:* The analysis should characterize all parts of the program.

—*Feature coverage:* The analysis should accept code using all language features.

—*Input coverage:* The analysis results should predict program behavior with respect to all possible program inputs.

—*Analysis time:* The analysis should take little time before the program runs.

—*Runtime:* The analysis should take little time while the program runs.

—*Precision:* The analysis should not overapproximate runtime behavior.

—*Soundness:* The analysis should not underapproximate runtime behavior.

—*Ease-of-use:* The analysis should require no work on the part of the human user.

—*Ease-of-engineering:* Developing and maintaining the analysis code should not take too much time.

Offline analysis cannot achieve both full code coverage and full feature coverage if one of the features of the analyzed language is dynamic loading. Online analysis cannot achieve full input coverage or zero runtime overhead because the input determines the code it encounters, and the analysis itself takes time while the program is running. Offline analyses often sacrifice one or more of soundness, ease-of-use, code coverage, and feature coverage, particularly when dealing with reflection or foreign language interfaces. This is not strictly necessary, and the fact that it happens anyway indicates that most analysis designers rate other requirements, such as better precision, as more important for their clients.

The choice of offline versus online analysis depends on how well the requirements of the client match the strengths and weaknesses of the analysis. In general, online analysis is better for optimizations (they should be fully automatic and deal with all programs, but can be specialized to the current input by using an invalidation framework), and offline analysis is better for productivity tools (they should be general with respect to all possible inputs, but need not necessarily deal with all programs or all language features).

1.5 Contributions

This article describes an online version of Andersen's pointer analysis [1994]. The online analysis elegantly deals with dynamically loaded code, as well as with reflection and a foreign language interface. This article

—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 [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 article 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 event that is relevant to pointer analysis, the average analysis pause takes under 0.1 seconds, and all pauses together take 64.5 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. Our implementation of Andersen's analysis is flow- and context-insensitive. Although this article should also be helpful for performing flow- or context-sensitive analyses online, these analyses pose additional challenges that need to be addressed. For example, correctly dealing with exceptions [Choi et al. 1999] or multithreading [Grunwald and Srinivasan 1993] is more difficult in a flow-sensitive than in a flow-insensitive analysis.

—*Other languages:* This article shows how to deal with dynamic class loading, reflection, and the JNI in Java. Dynamic class loading is a form of dynamic loading of code, and we expect our solutions to be useful for other forms of dynamical loading, 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 JNI is a form of foreign language interface, and we expect our solutions to be useful for foreign language interfaces of other languages.

1.6 Overview of the Article

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

2. BACKGROUND

Section 2.1 describes ways in which pointer analyses abstract data flow and points-to relationships in programs. Section 2.2 gives a concrete example for how pointer analysis manipulates these abstractions. Section 2.3 defines online and incremental analyses.

2.1 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, and subsequent sections give details for *how* it works. While this section enumerates several alternative abstractions, the lists are not intended to be exhaustive.

2.1.1 *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 cannot hide in a nonpointer 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 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).

(b) 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 article assumes context-insensitive analysis, using Option (a).

2.1.2 *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, nor 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 the following.

(a) For each type, represent all instances of that type by one h-node (type-based heap abstraction).

(b) For each allocation site, represent all heap objects allocated at that allocation site by one h-node (allocation-site-based heap abstraction).

Most of this work assumes allocation-site-based heap abstraction, using Option (b).

2.1.3 *Fields.* Fields are instance variables or array elements. A field is like a variable, but is stored inside of a heap object. Just as the analysis is concerned only with variables of reference type, it is concerned only with fields of reference type, and ignores nonpointer fields. Multiple instances of a field can coexist at runtime because multiple instances of the object in which it resides 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) 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).
(b) 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 work assumes field-sensitive analysis, thus using Option (b).

2.1.4 *PointsTo Sets.* PointsTo sets are sets of h-nodes that 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 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 article assumes flow-insensitive analysis, thus using Option (a). When the analysis finds that $h \in \text{pointsTo}(v)$, it predicts that any of the variables represented by v may point to any of the heap objects represented by h. Likewise, when the analysis finds that $h \in \text{pointsTo}(h'.f)$, it predicts that the field f of any of the heap objects represented by h' may point to any of the heap objects represented by h.

2.2 Example

Consider a program with just three statements:

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

In the terminology of Section 2.1, 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

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

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

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

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 may point to all targets that y points to (due to Statement 3), but possibly more.

An analysis component called a “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:

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

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

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

Again, the analysis represents statements with flowTo sets

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

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

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

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

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

2.3 Kinds of Interprocedural Analysis

An *incremental* interprocedural analysis updates 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 much less expensive than computing them from scratch. Also, the computational complexity of updating results must be no worse than that 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 article differs in that it uses incrementality to deal with dynamically loaded code.

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

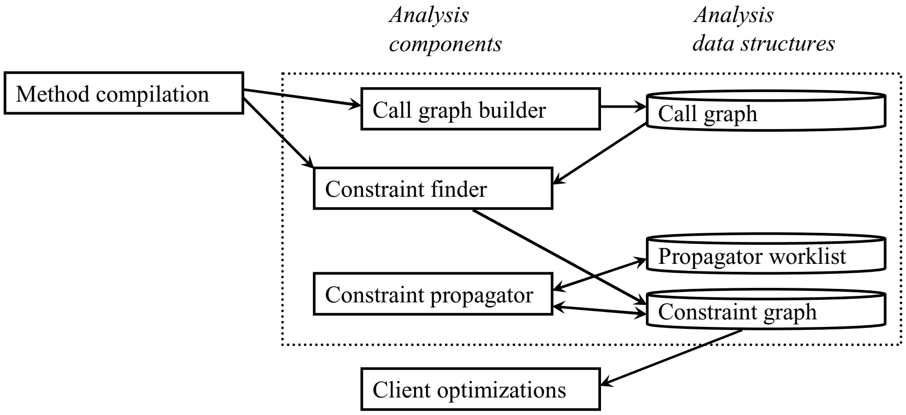


Fig. 1. Offline architecture.

A *modular* interprocedural analysis (e.g., Chatterjee et al. [1999], Liang and Harold [1999], Cheng and Hwu [2000]) performs most of its work in an intramodule step, and less work in an intermodule step. By "most" and "less," we mean that the intramodule step must be much more expensive than the intermodule step. Also, the computational complexity of the intramodule step must be no better than that 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 in order 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 in order to be used online.

3. OFFLINE ANALYSIS

This section describes the parts of our version of Andersen's pointer analysis that could be used for Java-like languages without dynamic features. We present them separately here to make the article easier to read, and to allow Section 4 to focus on techniques for performing online analyses. Our implementation combines the techniques from Sections 3 to 6 into a sound analysis for the full Java language.

3.1 Offline Architecture

Figure 1 shows the architecture for performing Andersen's pointer analysis offline. In an offline setting, bytecode for the entire program is available before running it, and "method compilation" constructs an intermediate representation (IR) from bytecode. The call graph builder uses the IR as input, and creates a call graph. The constraint finder uses the IR as input for creating

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]

intraprocedural constraints, and 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.

3.2 Analysis Data Structures

The call graph models possible method calls (Section 3.2.1). The constraint graph models the effect of program code on pointsTo sets (Section 3.2.2). The propagator worklist keeps track of possibly violated constraints (Section 3.2.3). Our analysis represents pointsTo sets in the constraint graph with Heintze's shared bit sets (Section 3.2.4).

3.2.1 *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 and interface methods, a given call site may have edges to multiple potential callee methods.

3.2.2 *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 2.1.1–2.1.3. 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 2.1.4.

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 2.2, 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) is equivalent to $v \in$ 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

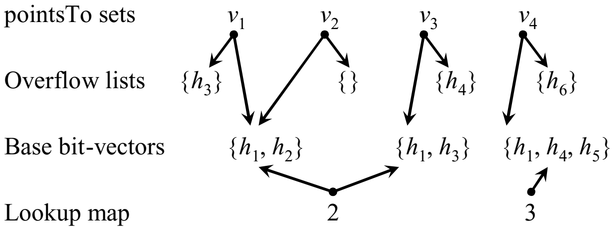


Fig. 2. Example for Heintze's pointsTo set representation.

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 models the pointers themselves with pointsTo sets. Clients are interested in the pointsTo sets, but not in the flow sets, which are only used as an internal representation for the analysis to compute and update the pointsTo sets.

3.2.3 *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 may not hold for the current pointsTo set solution. If this is the case, the propagator has some more work to do in order 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'. Other implementations of Andersen's analysis, such as BANE (Berkely ANalysis Engine) [Fähndrich et al. 1998], also put $h.f$-nodes on the worklist.

3.2.4 *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 [Heintze 1999]. 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.

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 $\{h_1, h_2\}$, as shown in Figure 2. Since pointsTo(v_1) also includes h_3, its overflow list contains the single element h_3; by similar reasoning, pointsTo(v_2)'s overflow list is empty.

Figure 3 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; we picked the value 24). 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; we configured

```
 1: if $h \notin base$ and $h \notin overflow$
 2:     if $|overflow| < overflowLimit$
 3:         $overflow.add(h)$
 4:     else
 5:         $desiredSet \leftarrow base \cup overflow \cup \{h\}$
 6:         for $overflowSize \leftarrow 0$ to $newOverflowThreshold$
 7:             for each $candidate \in lookupMap[|desiredSet| - overflowSize]$
 8:                 if $candidate \subseteq desiredSet$
 9:                     $base \leftarrow candidate$
10:                     $overflow \leftarrow desiredSet - candidate$
11:                     return
12:         // we get here only if there was no suitable candidate
13:         $base \leftarrow desiredSet$
14:         $overflow \leftarrow \{\}$
15:         $lookupMap[|base|].add(base)$
```

Fig. 3. Adding a new element h to a pointsTo set in Heintze's representation.

newOverflowThreshold = 12). 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).

3.3 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 [Ghiya 1992; Burke et al. 1994; Emami et al. 1994]. 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]. Constructing a call graph on-the-fly would require an architectural change: In Figure 1, the input to the call graph builder would come both from method compilation and from the constraint graph (which contains the pointsTo sets). Instead of improving the precision using on-the-fly call graph construction based on the pointsTo sets, a better alternative may be to obtain the most precise possible call graph using low-overhead exhaustive profiling [Qian and Hendren 2004].

3.4 Offline Constraint Finder

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

3.4.1 *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

Table II. Finding Constraints for Assignments

Statement		Actions	Represent constraints
$v' = v$	(move $v \to v'$)	flowTo(v). $add(v')$	pointsTo(v) $\subseteq$ pointsTo(v')

$v' = v.f$	(load $v.f \to v'$)	flowTo($v.f$). $add(v')$	**for each** $h \in$ pointsTo(v) :
			pointsTo($h.f$) $\subseteq$ pointsTo(v')

$v'.f = v$	(store $v \to v'.f$)	flowTo(v). $add(v'.f)$,
		flowFrom($v'.f$). $add(v)$
			for each $h \in$ pointsTo(v') :
			pointsTo(v) $\subseteq$ pointsTo($h.f$)

a: $v =$ **new**...	(alloc $h_a \to v$)	pointsTo(v). $add(h_a)$	$\{h_a\} \subseteq$ pointsTo(v)

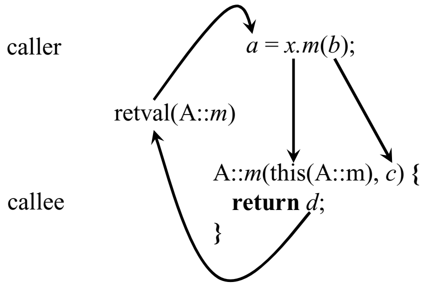


Fig. 4. Finding constraints for parameters and return values.

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 this 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 possible targets pointsTo(v') of v'.

3.4.2 *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. Figure 4 shows these flow-to relations as arrows. For example, the arrow $d \to$ retval(A :: m) denotes the analysis action flowTo(v_d).add($v_{\text{retval(A::}m)}$).

3.4.3 *Exceptions.*

Exception handling leads to a 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. We could easily imagine making this more

```
 1: while worklist not empty
 2:     while worklist not empty
 3:         remove node $v$ from worklist
 4:         for each $v' \in$ flowTo($v$)                          // move $v \rightarrow v'$
 5:             pointsTo($v'$).$add$(pointsTo($v$))
 6:             if pointsTo($v'$) changed, add $v'$ to worklist
 7:         for each $v'.f \in$ flowTo($v$)                        // store $v \rightarrow v'.f$
 8:             for each $h \in$ pointsTo($v'$)
 9:                 pointsTo($h.f$).$add$(pointsTo($v$))
10:         for each field $f$ of $v$
11:             for each $v' \in$ flowFrom($v.f$)                  // store $v' \rightarrow v.f$
12:                 for each $h \in$ pointsTo($v$)
13:                     pointsTo($h.f$).$add$(pointsTo($v'$))
14:             for each $v' \in$ flowTo($v.f$)                    // load $v.f \rightarrow v'$
15:                 for each $h \in$ 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 $v' \in$ flowTo($v.f$)                        // load $v.f \rightarrow v'$
20:             for each $h \in$ pointsTo($v$)
21:                 pointsTo($v'$).$add$(pointsTo($h.f$))
22:                 if pointsTo($v'$) changed, add $v'$ to worklist
```

Fig. 5. Lhoták-Hendren worklist propagator.

precise, for example, by assuming that throws can only reach catch clauses in the current method or its (transitive) callers.

3.5 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.

3.5.1 *Lhoták-Hendren Worklist Propagator.* Figure 5 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 this 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. Other propagators, such as the one in BANE [Fähndrich et al. 1998], do not need the second part.

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 $v_1 \rightarrow v_2.f$ in Lines 7–9 can change pointsTo($h_2.f$) for some $h_2 \in$ pointsTo(v_2). When this occurs, simply putting v_2 on the worklist is insufficient because the stored value can be retrieved via an unrelated variable v_3. For example, assume that:

$$\begin{array}{ll} \text{flowTo}(v_1) = \{v_2.f\}, & \text{flowTo}(v_3.f) = \{v_4\}, \\ \text{pointsTo}(v_1) = \{h_1\}, & \\ \text{pointsTo}(v_2) = \{h_2\}, & \text{pointsTo}(v_3) = \{h_2\}, \\ \text{pointsTo}(v_4) = \{\}, & \text{pointsTo}(h_2.f) = \{\} \end{array}$$

Assume v_1 is on the worklist. Processing the store $v_1 \rightarrow v_2.f$ propagates h_1 from pointsTo(v_1) to pointsTo($h_2.f$). Because flowTo($v_3.f$) $= \{v_4\}$, there is a constraint pointsTo($h_2.f$) $\subseteq$ pointsTo(v_4), but it is violated, since pointsTo($h_2.f$) $= \{h_1\} \nsubseteq \{\} =$ pointsTo(v_4). Processing the load $v_3.f \rightarrow v_4$ would repair this constraint by propagating h_1 from pointsTo($h_2.f$) to 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.

3.5.2 *Type Filtering.* The propagator admits only those h-nodes into a pointsTo set that are compatible with its type. The operation

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

is really implemented as:

```
for each $h \in$ pointsTo($a$)
  if typeOf($h$) is a subtype of typeOf($b$)
    pointsTo($b$).$add$($\{h\}$)
```

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

3.5.3 *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 creates only $h.f$-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 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 (as does BANE [Fähndrich et al. 1998]).

4. ONLINE ANALYSIS

This section describes how to change the offline analysis from Section 3 to obtain an online analysis. It is organized around the online architecture in Section 4.1.

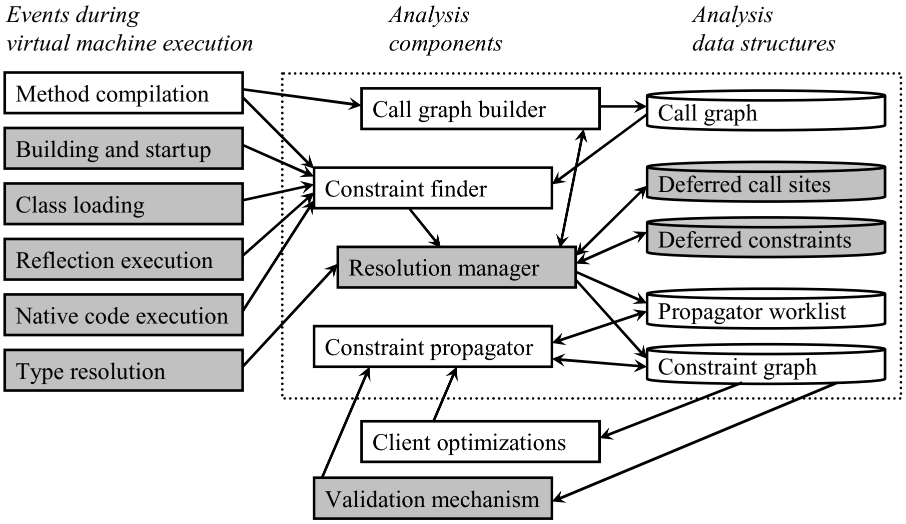


Fig. 6. Online architecture. Components absent from the offline architecture (Fig. 1) are shaded.

4.1 Online Architecture

Figure 6 shows the architecture for performing Andersen's pointer analysis online. The online architecture subsumes the offline architecture from Figure 1, and adds additional functionality (shown as shaded) for dealing with dynamically loaded code and other dynamic program behavior that cannot be analyzed prior to execution. In the online case, method compilation is performed by a JIT, a just-in-time compiler that compiles bytecode to machine code when the method gets executed for the first time.

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). The validation mechanism behaves like a client. 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.

4.2 Online Call Graph Builder

As described in Section 3.3, 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 gradually incorporate the call sites and potential callee methods that the JIT compiler encounters during runtime, and it has to work in the presence of unresolved types.

4.2.1 *Incorporating Call Sites and Callees as They Are Encountered.* For each call edge, either the call site 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 (Figure 6, 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 (Figure 6, arrow from call graph builder to call graph). Then, it stores the new callee method node in a map that is keyed by its signature, for future retrieval when new call sites are encountered.

—Analogously, when the JIT compiler compiles a new call site (Figure 6, arrow from method compilation to call graph bilder), 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 (Figure 6, arrow from call graph builder to call graph). Then, it stores the new call site node in a map that is keyed by its signature, for future retrieval when new callees are encountered.

4.2.2 *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: Eager loading would require hiding any visible 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, thus 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 (Figure 6, arrow from call graph builder to resolution manager).

4.3 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 does not yet need for obtaining sound results. The arrows from the call graph builder and constraint finder to the resolution manager in Figure 6 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 Figure 6 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 Figure 6), 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 slowdown 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.

4.4 Online Constraint Finder

Compared to the offline constraint finder (Section 3.4), the online constraint finder has to satisfy three additional requirements: It has to capture more input-generating events, find interprocedural constraints whenever more call edges are encountered, and work in the presence of unresolved types.

4.4.1 *Capturing More Input-Generating Events.* In the offline architecture (Figure 1), the only input to the constraint finder is the compiler's intermediate representation of methods. In the online architecture (Figure 6) several events generate 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 6.3.1).

—*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 startup. 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 this for Jikes RVM in Section 6.3.5.

—*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, Sect. 4.5]. This attribute initializes fields, yet is not represented as code, but rather as metadata in the class file.

—*Reflection execution*: Section 4.6 describes how to analyze reflection online.

—*Native code execution*: Section 4.7 describes how to analyze native code online.

4.4.2 *Capturing Call Edges as They Are Encountered.* Whenever the online call graph builder (Section 4.2) adds a new call edge to the call graph (Figure 6, arrow from call graph builder to call graph), the constraint finder is notified (Figure 6, arrow from call graph to constraint finder), so it can model the data flow for parameter passing and return values. This works as follows:

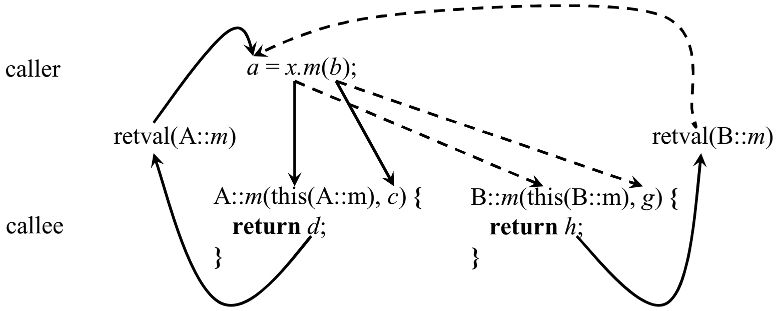


Fig. 7. Call constraints when adding a new callee.

- —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 callsite 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 $c \to m$, 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.

For the example, in Figure 4, 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 $x \to$ this(B::m), $b \to g$, and retval(B::m) $\to a$, shown as dashed arrows in Figure 7.

4.4.3 *Dealing with Unresolved Types.* As discussed in Section 4.2.2, Java bytecode can refer to unresolved types. This prohibits type filtering in the propagator (Section 3.5.2), 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 (Figure 1), the online constraint finder sends its outputs to the resolution manager first (Figure 6), which only reveals them to the propagator when they become resolved. This approach is sound because the types will be resolved when the instructions are executed for the first time before the types are resolved, the constraints involving them are not needed. The analysis deals with unresolved

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

types 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 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.

—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 yet resolved itself, the information will be added in the future when that node gets resolved.

4.5 Online Constraint Propagator

Method compilation and other constraint-generating events happen throughout program execution. Whereas 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 of Figure 6 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 3.2.3) of changed parts in the constraint graph to avoid incurring the full propagation cost every time.

In architectural terms, the online propagator's worklist is populated by the constraint finder (Figure 6), whereas the offline propagator's worklist was only used internally by the propagator (Figure 1). Exposing the worklist was the only "external" change to the propagator to support online analysis. In addition,

```
 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. 8. Reflection example.

we found that "internal" changes that make the constraint propagator more incremental are invaluable for achieving good overall analysis performance. We will describe these incrementalizations in Section 5.1.

4.6 Reflection

Java allows code to access methods, fields, and types by name based on strings computed at runtime. For example, in Figure 8, 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 cannot determine what actions the 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 cannot 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 string [Christensen et al. 2003] or reflection analysis [Livshits et al. 2005] to predict which entities reflection manipulates. However, both are offline analyses, and while they 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 significant imprecision into the analysis for the sake of a few operations that are 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, this 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 repropagation.

4.7 Native Code

The Java native interface (JNI) allows Java code to interact with dynamically loaded native code. A *downcall* transfers control from Java to native code, and an *upcall* transfers control from native code to Java. Downcalls are calls to native methods. Upcalls include Java method calls, Java field and array manipulation, allocation of Java objects, and calls to JVM services such as the class loader. Upcalls go through the JNI API, which is equivalent to Java reflection (in fact, Jikes RVM implements this by calling Java methods that use reflection).

In general, the analysis can not analyze the dynamically loaded native code, since it is compiled architecture-specific machine code. But in order to soundly analyze the Java code, it needs to model possible data transfer from native code to Java. Specifically, it needs to predict the set of pointers that native code returns from downcalls or passes as parameters to upcalls.

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 downcalls, while being precise for parameters passed in upcalls. If a downcall returns a pointer to a heap-allocated object, the constraint finder assumes that the object could have originated from any allocation site. This is imprecise, but easy to implement. Type filtering (Section 3.5.2) limits the imprecision by reducing the set of h-nodes returned by a JNI method based on its declared return type. For upcalls, our analysis simply uses the same instrumentation as for reflection.

5. 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 5.1) and to manually fine-tune the precision of the analysis along various dimensions to improve performance (Section 5.2).

5.1 Incrementalizing the Propagator

As discussed in Section 2.3, incrementality is necessary, but not sufficient, for online program analysis. Section 4.5 described some steps towards making the propagator incremental. However, incrementality is not an absolute property; rather, there is a continuum of increasingly incremental algorithms.

Sections 5.1.1 to 5.1.4 describe several such algorithms, each more incremental than the previous. None of these algorithms affect precision: They all compute the same fixed point on the constraint set.

5.1.1 *Lhoták-Hendren Propagator and Propagating from New Constraints Only.* This section briefly reviews the starting point of our propagator incrementalization. Section 3.5 describes the propagator that Lhoták and Hendren presented in their initial paper about the SPARK offline pointer analysis framework for Java [2003]. Figure 5 gives the pseudocode, 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 3.2.3 describes the worklist data structure: This is a list of v-nodes that may appear on the left side of unresolved constraints. Section 4.5 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 Figure 5 efficient.

5.1.2 *IsCharged Bit for $h.f$-nodes.* We found that after some iterations of the outer loop (Line 1) of the Lhoták-Hendren propagator in Figure 5, 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 this 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.

Figure 9 shows the algorithm that we presented as Figure 3 in our previous paper about this analysis [2004], which maintains isCharged bits on $h.f$-nodes to speed-up the iterative part of Figure 5 (Lines 18–22). We say an $h.f$-node is charged if processing a load might propagate new pointsTo set elements from $h.f$. 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 Figure 5. In Figure 9, 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.

5.1.3 *Caching Charged $h.f$-node Sets.* Lines 20–22 of Figure 9 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_2) for some other variable v_2, it is wasteful to compute the subset twice. Instead, the propagator in Figure 10 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 Figure 9. Instead, our

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

Fig. 9. IsCharged bit for $h.f$-nodes. Changes compared to Fig. 5 are annotated with comments.

```
1-19: identical with Lines 1-19 in Fig. 9
20: for each $v.f$
21:     for each $v' \in$ flowTo($v.f$)                                     // load $v.f \rightarrow v'$
22:         if (pointsTo($v$), $f$) $\in$ chargedHFCache
23:             chargedHF $\leftarrow$ chargedHFCache[pointsTo($v$), $f$]
24:         else
25:             chargedHF $\leftarrow$ {$h.f$ for $h \in$ pointsTo($v$), if isCharged($h.f$)}
26:             chargedHFCache[pointsTo($v$), $f$] $\leftarrow$ chargedHF
27:         for each $h.f \in$ chargedHF
28:             pointsTo($v'$).add(pointsTo($h.f$))
29:             if pointsTo($v'$) changed, add $v'$ to worklist
30: for each $h.f$, isCharged($h.f$) $\leftarrow$ false
```

Fig. 10. Caching charged $h.f$-node sets.

implementation exploits the fact that the pointsTo sets are already shared (see Heintze's representation, Section 3.2.4). 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 whether $h.f$ is charged.

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

5.1.4 *Caching the Charged h-node Set.* The iterative part of the algorithm in Figure 10 still loops over all load edges $v.f \rightarrow v'$ in Lines 20 and 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 Figure 11 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 Figure 11 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 of the 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 Figure 10 (compare to Lines 31–32 in Figure 11). 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).

5.2 Varying Analysis Precision

Section 2.1 states that our analysis is context-insensitive, abstracts the heap with allocation sites, and is field-sensitive. These choices yield a reasonable overall precision/performance tradeoff. But in fact, manually revising the choices for precision in a few places improves performance. Based on conversations with different pointer analysis implementers, we believe that these kinds of precision adjustments are common practice. We record them here to demystify what can seem like black magic to the uninitiated, and to encourage the community to come up with techniques for automatically, rather than manually, revising choices for precision online. Section 8.3 will evaluate the effects of these optimizations.

5.2.1 *Selectively Use Context Sensitivity.* Our context-insensitive analysis (Section 2.1.1) gives imprecise results, particularly for methods that are called from many places and modify or return their argument objects. In such situations, a 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 2.1.1) improves precision,

```
while worklist not empty, or isCharged($h.f$) for any $h.f$-node
    while worklist not empty
        remove node $v$ from worklist
        for each $v' \in$ flowTo($v$)                              // move $v \rightarrow v'$
            pointsTo($v'$).add(pointsTo($v$))
            if pointsTo($v'$) changed, add $v'$ to worklist
        for each $v'.f \in$ flowTo($v$)                            // store $v \rightarrow v'.f$
            for each $h \in$ pointsTo($v'$)
                pointsTo($h.f$).add(pointsTo($v$))
                if pointsTo($h.f$) changed
                    isCharged($h.f$) $\leftarrow$ true
                    chargedH.add($h$)                              // new in Fig. 11
        for each field $f$ of $v$
            for each $v' \in$ flowFrom($v.f$)                      // store $v' \rightarrow v.f$
                for each $h \in$ pointsTo($v$)
                    pointsTo($h.f$).add(pointsTo($v'$))
                    if pointsTo($h.f$) changed
                        isCharged($h.f$) $\leftarrow$ true
                        chargedH.add($h$)                          // new in Fig. 11
            if pointsTo($v$) $\cap$ chargedH $\neq$ {}             // new in Fig. 11
                for each $v' \in$ flowTo($v.f$)                    // load $v.f \rightarrow v'$
                    for each $h \in$ (pointsTo($v$) $\cap$ chargedH)   // compare to Fig. 9 Line 17
                        pointsTo($v'$).add(pointsTo($h.f$))
                        if pointsTo($v'$) changed, add $v'$ to worklist
    for each $v.f$
        if pointsTo($v$) $\cap$ chargedH $\neq$ {}                 // new in Fig. 11
            for each $v' \in$ flowTo($v.f$)                        // load $v.f \rightarrow v'$
                if (pointsTo($v$), $h$) $\in$ chargedHFCache
                    chargedHF $\leftarrow$ chargedHFCache[pointsTo($v$), $f$]
                else
                    ch $\leftarrow$ pointsTo($v$) $\cap$ chargedH          // new in Fig. 11
                    chargedHF $\leftarrow$ {$h.f$ for $h \in$ ch, if isCharged($h.f$)} //compare to
                                                                   Fig. 10 Line 25
                    chargedHFCache[pointsTo($v$), f] $\leftarrow$ chargedHF
                for each $h.f \in$ chargedHF
                    pointsTo($v'$).add(pointsTo($h.f$))
                    if pointsTo($v'$) changed, add $v'$ to worklist
    for each $h.f$, isCharged($h.f$) $\leftarrow$ false
    chargedH $\leftarrow$ {}
```

Fig. 11. Caching the charged h-node set.

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.

5.2.2 *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 2.1.2, allocation-site-based heap abstraction is more precise than type-based heap abstraction). 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 that 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.

5.2.3 *Selectively Relax Field Sensitivity.* If there are many stores into a field accessed via different h-nodes, but they all store similar pointsTo sets, then field sensitivity 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 $h.f_{\mathrm{StringBuffer}.\mathit{value}}$ point to the same few instances.

We have extended our analysis to selectively fall back to being field-based. In our runs we apply field sensitivity everywhere except for fields affected as described earlier by context sensitivity, namely, StringBuffer.*value*, String.*value*, and VM_Assembler.*machineCodes*.

6. IMPLEMENTATION

Section 6.1 describes how we convinced ourselves that our analysis implementation is sound. Section 6.2 mentions some performance issues that we found and fixed. Finally, Section 6.3 shows how our implementation tackles the challenges of a concrete VM. Readers who are less interested in detailed implementation issues may skip ahead to Section 7.

6.1 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.

6.1.1 *Validation Mechanism.* We validate the pointer analysis results at GC (garbage collection) time (Figure 6, 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. The validation mechanism does not check whether the pointsTo sets of local variables are correct.

Since it checks the correctness of pointsTo sets during GC, the validation mechanisms triggers constraint propagation just before GC starts (Figure 6, 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.

6.1.2 *Validation Anecdotes.* Our validation methodology helped us find many bugs, some of which were quite subtle. We describe 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, Sect. 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, Sect. 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 this 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 this attribute (Section 4.4.1).

6.2 Fixing Performance Issues

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 issues: situations where the analysis is still sound, but takes excessive time and space. Mentioning them here may help readers avoid pitfalls in implementing similar analyses.

6.2.1 *Use a Bit Mask for Type Filtering.* Section 3.5.2 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 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 3.2.4). Therefore, the operation

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

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

```
$\text{pointsTo}(b) \leftarrow \text{pointsTo}(b) \cup (\text{pointsTo}(a).\mathit{base} \cap \text{typeMask}(\text{typeOf}(b)))$
 for each $h \in \text{pointsTo}(a).\mathit{overflow}$
    if $h \in \text{typeMask}(\text{typeOf}(b))$
       $\text{pointsTo}(b) \leftarrow \text{pointsTo}(b) \cup \{h\}$
```

6.2.2 *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 cannot 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.

6.2.3 *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, both local variables and stack slots can have different, conflicting types at different program points. This is not an artifact of our JIT compiler's IR, but a general property of JVM-independent bytecode. 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 so naively was too slow. We fixed it by introducing helper v-nodes of more precise types in some cases where it helps performance, thereby using the normal type filtering when propagating to the helper v-nodes and avoiding the need to filter at field access time.

6.3 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.

6.3.1 *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 wherein 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.

6.3.2 *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. Since the old constraints are still around, the analysis does not benefit from any possible improvements in precision that recompilation could cause.

6.3.3 *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. The alternative would be to add a special case for clients that need to know what points to type descriptors.

6.3.4 *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.

6.3.5 *Building and Startup.* 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 preallocated Java objects and precompiled 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 of the *code* in the boot image as usual, with the constraint finder from Sections 3.4 and 4.4. 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 it is known statically that certain parts of the application use no dynamic language features, either based on user assertions or on compiler analysis.

7. CLIENTS

This section investigates example clients of our analysis, and how they can deal with its dynamic nature. In general, each client triggers constraint propagation when it requires sound analysis results (Figure 6, arrow from client optimizations to constraint propagator), and then consumes the resulting pointsTo sets (Figure 6, arrow from constraint graph to client optimizations). So far, we have only used our online pointer analysis for connectivity-based garbage collection (Section 7.3).

7.1 Method Inlining

Method inlining 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 Figure 6, 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 the inlining decisions (preexistence-based inlining [Detlefs and Agesen 1999] and guards [Hölzle and Ungar 1994; Arnold and Ryder 2002], respectively) that would give our analysis more slack in repropagating after it finds new constraints. Qian and Hendren [2005] study the usefulness of pointer analysis for inlining in a VM, and survey invalidation techniques.

As an aside, inlining is also an example for how online pointer analysis can subtly interact with online optimization. Updating pointer information may invalidate the inlining results, and inlining may force updating of pointer information. We do not anticipate this to be a problem in practice, since the analysis results in the version of the method with inlining will in general be more precise than the original ones. Thus, adding them should not reduce overall precision.

7.2 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 this in turn could help optimizations in a JVM achieve higher speedups if pointer analysis information was available in the JVM. Invalidation would proceed similarly to method inlining.

7.3 Connectivity-Based Garbage Collection

CBGC (connectivity-based garbage collection) is a new garbage collection technique that requires pointer analysis [Hirzel et al. 2003]. It 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, it 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. On the other hand, CBGC does need full conservative points-to information when performing a garbage collection; thus, it requests a full propagation before collecting. Between collections, CBGC does not need conservative points-to information.

7.4 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 previous concrete examples 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.

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 work 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. The suite includes all SPECjvm98 benchmarks and several other Java programs. Column "Command line arguments" shows the workload; for *pseudojbb*, the workload consists of one warehouse and 70,000 transactions.

Table V characterizes the benchmarks. Row *average* is the arithmetic mean of all benchmarks except for *null*. Columns "Compiled methods" and "Loaded classes" of Table V 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 andclasses, which includes the

Table IV. Benchmark Descriptions

Program	Command line arguments	Description
null		Empty main method, does nothing
compress	`-m1 -M1 -s100`	LZW compression
db	`-m1 -M1 -s100`	Memory-resident database
hsql	`-clients 1 -tpc 50000`	Relational database engine
ipsixql	`3 2`	Queries against persistent XML document
jack	`-m1 -M1 -s100`	Parser generator
javac	`-m1 -M1 -s100`	Java source to bytecode compiler
javalex	`qb1.lex`	Scanner generator
jess	`-m1 -M1 -s100`	Java Expert Shell System
mpegaudio	`-m1 -M1 -s100`	Decompresses audio files
mtrt	`-m1 -M1 -s100`	Multithreaded raytracer
pseudojbb		Java business benchmark for server-side Java
richards		Simulates task dispatcher in OS kernel
soot	-W –app -t -d Hello –jimple	Java bytecode analyzer and optimizer
xalan	`1 1`	XSLT tree transformation language processor

Table V. Benchmark Characteristics

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

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 has 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).

If we had implemented our analysis in a JVM that is not written in Java, it would need to analyze much less code, and likely perform better. Another way to achieve this would be by isolating the JVM from the application as far as the analysis is concerned. This is difficult to achieve in Jikes RVM, and we did not pursue it. On the other hand, our implementation prompted us to solve some other challenges (Section 6.3) that are gaining relevance as the benefits of writing system code in a high-level managed language gain wider recognition (see, e.g., Hunt et al. [2005]). Among other things, our analysis is actually a hybrid of offline and online analysis, since part of it is performed when building the JVM (Section 6.3.5).

Columns "Alloc." and "Time" of Table V characterize the workload, using Jikes RVM without our pointer analysis. Column "Alloc." is the amount 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.

8.1.2 *Method Compilation Over Time.* Figure 12 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 milliseconds. 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 startup. The graphs for other benchmarks have a similar shape, and therefore we omit them.

Figure 12 shows that after an initial warm-up period of around 0.3 seconds, *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.

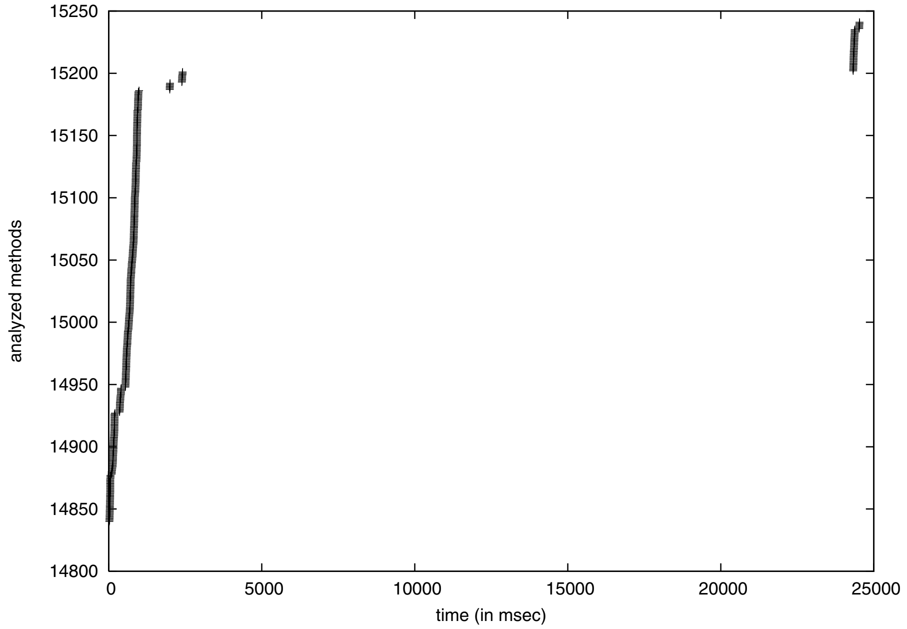


Fig. 12. Method compilation over time, for *mpegaudio*. The first shown data point is the *main*() method.

Since context-insensitive Andersen-style analysis has cubic time complexity, optimizations that increase code size can dramatically increase analysis overhead. In our experience, overly aggressive inlining can increase constraint propagation time by up to a factor of five 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 6.3.5). We force Jikes RVM to be more cautious about inlining inside boot-image methods by disabling inlining at build time, with the effect that at system start, only those VM methods that carry an explicit inlining pragma are inlined. However, the adaptive system still recompiles some hot boot-image methods if an inlining opportunity is discovered later [Arnold et al. 2000].

8.2 Performance With All Optimizations

This section evaluates the performance of our online analysis when all optimizations from Section 5 are enabled, and all solutions to performance issues from Section 6.2 have been applied. Section 8.2.1 introduces terminology, Section 8.2.2 evaluates 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 2.1), $O(n^2)$ time for constraint finding including call graph construction, and $O(n^3)$ time for constraint propagation (Figure 1), where n is the code size (indicated by the number of methods and classes in Table V).

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, this 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 (Figure 6). In general, $i = O(n^3)$, but thanks to our worklist-driven architecture, this 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 (Section 6.3.5), and then at runtime whenever a client of the pointer analysis needs points-to information and there are unresolved constraints.

This article investigates three kinds of propagation: Eager, At GC, and At End. *Eager* propagation occurs whenever an event from the left column of Figure 6 generates new constraints. *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]. *At End* propagation occurs just once at the end of program execution. With respect to cumulative propagation cost, Eager represents the worst case and At End represents the best. With respect to the cost of an individual propagation, Eager represents the best case and At End represents the worst. In both respects, At GC is an example real case.

8.2.2 *Memory Usage.* Figure 13 gives the total allocation of the application and runtime system ("No analysis") compared to the total allocation of constraint finding and by the propagator. Total allocation is the cumulative size of all objects allocated during the run; many objects become garbage and get collected, so the total allocation is at least as high as, and usually much higher than, the amount of data in the heap at any given point in time. Total allocation gives an indication of GC pressure. Without pointer analysis, the average benchmark allocates 359.4MB of memory. This includes allocation by the application itself, as well as allocation by the runtime system such as the JIT compilers, but excludes the boot image. Since the boot image needs to contain constraints for the code and data, our analysis inflates the boot-image size from 31.5MB to 69.4MB.

On average, constraint finding without propagation allocates 67.7MB more than the benchmark without the analysis, bringing the total to 427.1MB. On average, At End propagation allocates another 3.5MB (total 430.5MB), whereas At GC propagation allocates 11.3MB (total 438.3MB). With At GC propagation or At End propagation, most of the space overhead of the pointer analysis comes from constraint finding, rather than from propagation. Constraint finding allocates flowTo sets, deferred unresolved constraints, and the call graph. We have

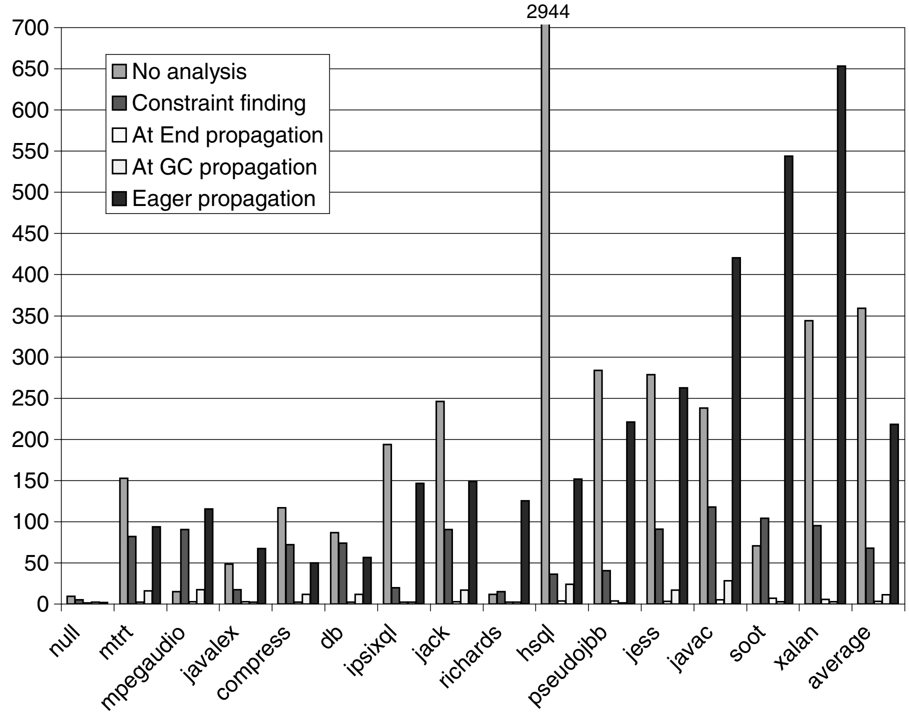


Fig. 13. Memory usage in MB.

not yet optimized their representation. Propagation allocates pointsTo sets. The low allocation for At End and At GC propagation indicates that Heintze's shared representation for pointsTo sets [Heintze 1999] is reasonably compact for none-too-eager clients.

On average, Eager propagation allocates 218.3MB. Together with the 67.7MB allocation for constraint finding, this adds 286.0MB to the 359.4MB average allocation without pointer analysis. Some data structures, such as deferred constraints or shared bit-sets, become garbage and get recycled throughout the run. The total allocation for eager propagation depends on the number of compiled methods and loaded classes (see Table V). For example, the benchmark with the most compiled methods is *xalan*, and Eager propagation allocates 653.2MB of memory for *xalan*. Clients that are content with lower propagation frequency techniques, such as CBGC, are rewarded with lower space overhead, but space overhead remains a challenge.

8.2.3 *Time for Constraint Finding.* Table VI gives the time overhead of finding constraints from methods (column "Analyzing methods") and from resolution events (column "Resolving classes and arrays") on top of runtime without constraint finding or propagation. The cost of finding constraints for methods dominates. Also, as the benchmark runtime increases, the percentage of time spent in constraint finding decreases. For example, the time spent in constraint finding is a negligible percentage of the runtime for our longest-running

Table VI. Time for Constraint Finding

Program	Analyzing methods		Resolving classes and arrays	
null	*0.3s*	*(195%)*	*0.1s*	*(24%)*
mtrt	1.3s	(10%)	0.4s	(1%)
mpegaudio	1.5s	(6%)	0.5s	(1%)
javalex	1.1s	(5%)	0.3s	(1%)
compress	1.0s	(3%)	0.3s	(0%)
db	1.0s	(4%)	0.2s	(0%)
ipsixql	1.8s	(28%)	0.8s	(7%)
jack	2.8s	(33%)	0.9s	(8%)
richards	1.3s	(54%)	0.8s	(17%)
hsql	1.6s	(1%)	0.5s	(0%)
pseudojbb	2.2s	(8%)	0.4s	(1%)
jess	3.0s	(23%)	1.1s	(4%)
javac	2.8s	(19%)	1.1s	(3%)
soot	8.5s	(312%)	1.7s	(37%)
xalan	5.8s	(19%)	2.2s	(5%)
average	*2.6s*	*(38%)*	*0.8s*	*(6%)*

The percentages are normalized to time without analysis.

Table VII. Time for Constraint Propagation (in seconds)

	Eager	At GC	At End
Program	*k* ×(Avg±Std) = Total	*k* × (Avg ± Std) = Total	1 × 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*

benchmark, *hsql*, but large for our shortest-running benchmarks (e.g., *null*). Soot is a worst case for this experiment: It has a large code base, but runs only three seconds.

8.2.4 *Time for Constraint Propagation.* Table VII shows the cost of constraint propagation. For example, with Eager propagation on javac, the propagator runs 1,275 times, taking 0.1 seconds on average, with a standard deviation of 0.3 seconds. All of the Eager propagation pauses add up to 182.3 seconds. The At End propagation data gives an approximate sense for how long propagation would take if we could wait for exactly the right moment to

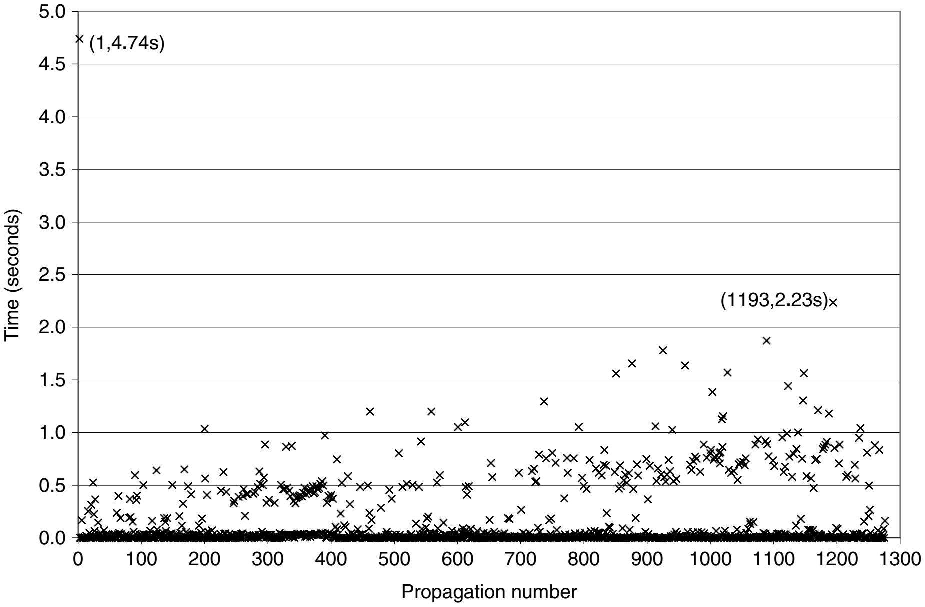


Fig. 14. Time for constraint propagation for *javac* with eager propagation.

propagate just once, after all methods are compiled, but before the application does the bulk of its computation. For example, At End propagation on javac would take 16.4 seconds. However, finding exactly the right moment to propagate is usually not possible.

Table VII shows that when propagating more frequently, individual propagations become faster. When all the optimizations are enabled, an eager propagation takes 0.1 seconds on average. 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 nontrivial aggregate cost.

Figure 14 presents the spread of propagation times for *javac*. A point (x, y) in this graph says that propagation x took y seconds. Out of 1,275 propagations in *javac*, 996 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 runtime propagation is the first because it finds a fixed point for all constraints that arise during VM startup, before Jikes RVM loads the application itself. The omitted graphs for other benchmarks have a similar shape. Figure 14 considers eager propagation times; as discussed in Table VII, at gc propagations are slower individually, but there are fewer of them.

8.2.5 *Runtime Constraint Counts.* This section explores how many constraints arise from each kind of runtime event. A constraint is an individual flow-to edge between two v-nodes or one v-node and one $v.f$-node. The runtime events are the inputs to the constraint finder in the left column of Figure 6,

Table VIII. Runtime Constraint Counts

Program	JIT compiler	Reflection	Clone	Native
Change?	Yes + No	Yes + No	Yes + No	Yes + No
null	*2,736 + 1,388*	*18 + 22*	*50 + 930*	*3 + 0*
mtrt	13,890 + 4,470	20 + 70	33 + 2,250	3 + 0
mpegaudio	9,055 + 3,836	20 + 93	50 + 3,362	3 + 0
javalex	8,523 + 2,066	18 + 26	60 + 10,800	3 + 0
compress	6,265 + 3,697	20 + 80	50 + 1,994	3 + 0
db	8,222 + 3,742	20 + 54	50 + 2,214	3 + 0
ipsixql	6,460 + 2,026	18 + 22	50 + 3,754	3 + 0
jack	16,404 + 7,141	19 + 138	50 + 3,862	3 + 0
richards	5,096 + 1,731	18 + 24	50 + 3,850	3 + 0
hsql	12,700 + 3,763	22 + 96	50 + 4,838	3 + 0
pseudojbb	39,529 + 3,624	43 + 507,067	33 + 3,945	3 + 0
jess	62,926 + 4,657	63 + 198	46 + 4,284	3 + 0
javac	44,832 + 10,343	20 + 638	142 + 1,123,018	3 + 0
soot	73,764 + 13,853	21 + 41	50 + 17,338	4 + 0
xalan	27,073 + 6,249	51 + 275	88 + 404,260	3 + 0
average	*23,910 + 5,086*	*27 + 36,344*	*57 + 113,555*	*3 + 0*

except for building and startup. The constraint finder may try to add a constraint that is already there. For example, when reflection calls the same method twice from the same place, the second call does not give rise to any new constraints. We are interested both in how many constraints are newly added and in how often the constraint finder tries to add an existing constraint, but does not because this would not change the constraint graph.

Table VIII shows the number of constraints from the different kinds of runtime events. For each kind of event, the table distinguishes how many constraints changed the constraint graph (Change = Yes or No). Column "JIT compiler" shows constraints from method compilation, class loading, type resolution, call graph building, and constraint propagation. Column "Reflection" counts all reflection constraints except for those caused by clone or native upcalls. Column "Clone" shows constraints from copying fields or array slots with reflection while cloning an object. Column "Native" shows constraints from the instrumentation on upcalls from native code.

Most constraints added to the constraint graph at runtime (Change = Yes) come from the JIT compiler (23,910 on average); all other runtime events together add 87 constraints, on average. Clone and reflection often attempt to add constraints that are already there (Change = No). The optimizing compiler of Jikes RVM calls clone on every compiler phase object for every compilation, causing a few thousand clone constraints for all benchmarks, but*javalex*,*javac*, *soot*, and *xalan* have even more significant numbers of clones in application code. An analysis might be able to deduce constraints for clone at compile time, reducing runtime overhead. The benchmark *pseudojbb* often uses reflection to allocate objects, generating 507,110 reflection constraints, most of which are redundant. Upcalls from native code cause only three constraints on average, indicating that in our benchmarks, native code seldom passes pointer arguments to Java methods or assigns pointers to Java fields or array slots.

Table IX. Constraint Propagation Time for Different Propagators

Program	Num. of Props	Prop. from new constraints (Section 4.5)		IsCharged for $h.f$ (Section 5.1.2)		Caching charged $h.f$ (Section 5.1.3)		Caching charged h (Section 5.1.4)	
		Avg ± Std	Total	Avg ± Std	Total	Avg ± Std	Total	Avg ± Std	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*

The time is in seconds. These numbers are slightly different from the "Eager" column in Table VII due to instrumentation artifacts.

Clone, reflection, and native upcalls are rare enough that the overhead from instrumenting them for the pointer analysis is not a big issue. On the other hand, they cause enough constraints to warrant an automatic approach, rather than requiring the user to specify manually what effect they can have on points-to relations. This is especially true for a few of the benchmarks that use these features extensively.

8.3 Effect of Optimizations

The results presented so far enabled all of the techniques in Sections 5 and 6.2. This section explores how they affect propagation time individually and in combination.

8.3.1 *Performance When Incrementalizing.* Table IX shows the effect of individual optimizations from Section 5.1. For these experiments, all of the optimizations from Section 5.2 and all solutions to performance issues from Section 6.2 are active. Comparing the average propagation cost of the first incremental propagator from Section 4.5 with the average propagation cost of the final algorithm from Section 5.1.4 shows that the incrementalizations are effective: The average propagation time drops from 1.2 seconds to 0.1 seconds. On average, the total propagation time drops from 750.7 seconds to 64.5 seconds, a factor of twelve speedup.

8.3.2 *Performance When Varying Sensitivity.* This section evaluates the three selective precision changes from Section 5.2, along with the technique of using a bit set for type filtering from Section 6.2.1. The other solutions to

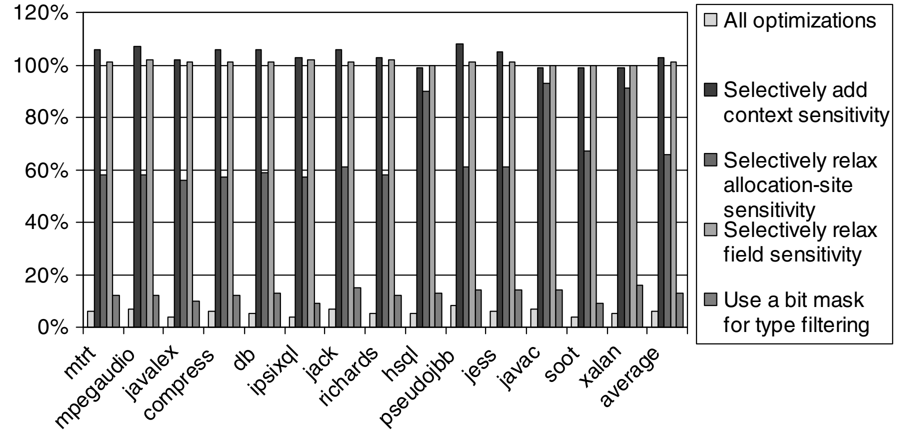


Fig. 15. Individual optimizations: cost in time compared to the unoptimized algorithm.

performance issues had a smaller impact, and exploring these is less interesting because we would want to include them in an implementation, anyway. Therefore, for these experiments, all optimizations from Sections 5.2, 6.2.2, and 6.2.3 are always active.

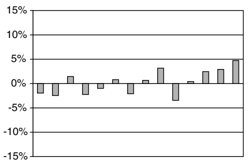
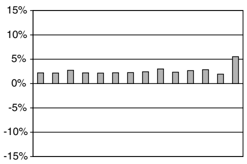
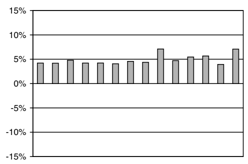
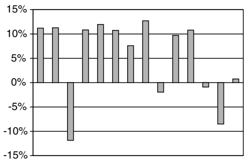
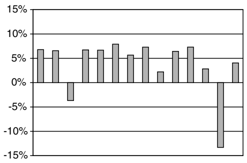
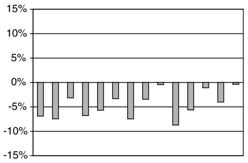
Section 8.3.2 shows the impact of individual optimizations and explores interactions of multiple optimizations, and Section 8.3.3 evaluates how the optimizations affect precision.

—*Individual optimizations*: Figure 15 gives the total propagation time when using the techniques from Sections 5.2 and 6.2.1 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 sixteen 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.

—*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 X 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. All numbers are normalized to the time without any of the optimizations. The value T_O is the total propagation time when using optimization O alone, as given in Figure 15. 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 individually.

Table X. Interactions of Pairs of Optimizations

	Selectively add context sensitivity	Selectively relax allocation-site sensitivity	Selectively relax field sensitivity
Use a bit mask for type filtering			
Selectively relax field sensitivity			
Selectively relax allocation-site sensitivity			

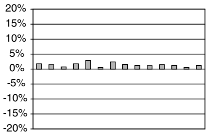
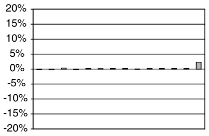
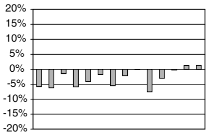
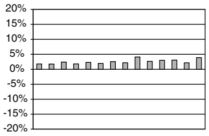
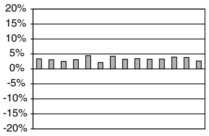
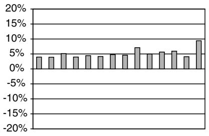
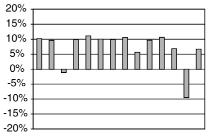
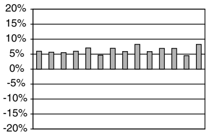
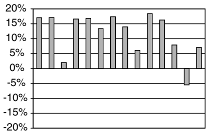
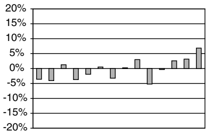
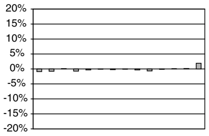
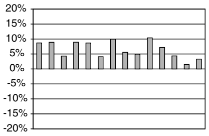
Here, min$(T_{O1}, T_{O2}) - T_{O1,O2}$, how much better is it to perform both optimizations than to perform only the better single optimization? The bars, left-to-right, are for *mtrt*, *mpegaudio*, *javalex*, *compress*, *db*, *ipsixql*, *jack*, *richards*, *hsql*, *pseudojbb*, *jess*, *javac*, *soot*,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

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.

—*Triples of optimizations*: Table XI 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. All numbers are normalized to the time without any of the optimizations. The interpretation of this table is similar to that of Table X: 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 XI, row "Context and allocation-site", column "Selectively relax field sensitivity" shows the strongest positive interaction. The row itself corresponds to the worst pairwise interaction from Table X. This motivates the need for the optimization "Selectively relax field sensitivity": In our

Table XI. Interactions of Triples of Optimizations

	Selectively add context sensitivity	Selectively relax allocation-site sensitivity	Selectively relax field sensitivity	Use a bit mask for type filtering
Bit mask and field				
Bit mask and allocation-site				
Field and allocation-site				
Context and field				
Context and allocation-site				
Bit mask and context				

Here, $\min(T_{O1,O2}, T_{O3}) - T_{O1,O2,O3}$, 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*.

experiments, it was 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 5 to improve performance, the three optimizations in Section 5.2 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

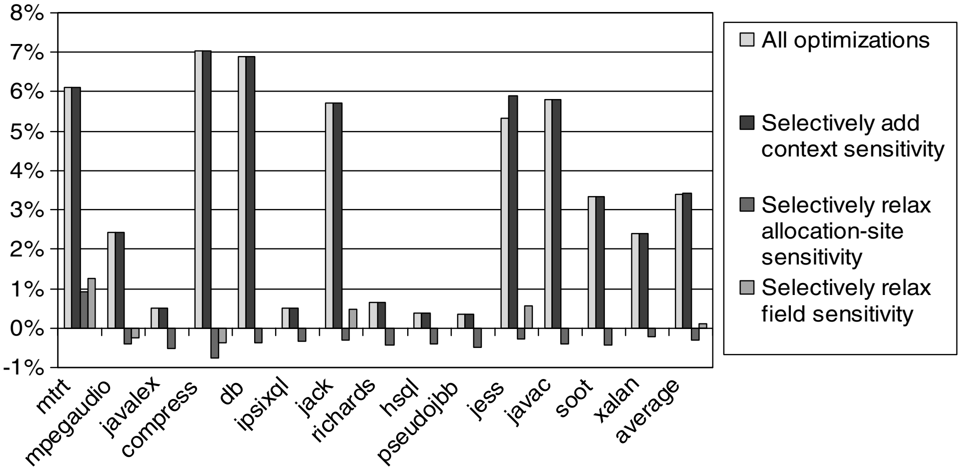


Fig. 16. Precision. How the optimizations change the number of CBGC partitions.

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 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 one, it also has fewer, larger strongly connected components. Thus, a less precise analysis will have fewer partitions than a more precise analysis. Figure 16 gives the change in the number of partitions relative to using none of these optimizations.

Figure 16 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 3, Section 9.2 discusses work related to the online analysis from Section 4, Section 9.3 discusses work related to the analysis optimizations from Section 5, and Section 9.4 discusses work related to our validation technique from Section 6.1.

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.

9.1.1 *Where Does Andersen Fit In?.* The body of literature on pointer analyses is vast [Hind 2001]. At one extreme, as exemplified both by Steensgaard

[1996] and analyses with type-based heap abstraction [Harris 1999; Tip and Palsberg 2000; Diwan et al. 2001], the analyses are fast but imprecise. At the other extreme, as 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 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 analyses with type-based heap abstraction are too imprecise for CBGC. On the other hand, we felt that much more precise shape 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] analyses. 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 aforementioned papers present solutions for static Java features that make pointer analyses difficult, such as object fields, virtual method invocations, etc.

Rountev et al. [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 et al. [2001] compare both Steensgaard's and Andersen's analyses for static Java, and evaluate tradeoffs 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 believe that they are part of the 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 et al. [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 deoptimize (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.* Kotzmann and Mössenböck published a sound online escape analysis that deals with dynamic class loading, reflection, and JNI [2005]. Their paper appeared one year after we published our online pointer analysis. Next, we discuss some earlier analyses that deal with dynamic class loading. None of these analyses deal with reflection or JNI, nor validate 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 does not 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 to be used by 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 [2004] solve the problems posed by dynamic class loading similarly to the way we solve them; for example, their approach to unresolved references is analogous to our approach in Section 4.3.

9.3 Varying Analysis Precision

There has also been significant prior work in varying analysis precision to improve performance. Ryder [2003] describes precision choices for modeling program entities in a reference analysis (pointer analysis, or weaker variants with type-based heap abstraction). Plevyak and Chien [1994] and Guyer and Lin [2003] describe mechanisms for automatically varying flow and context sensitivity during the analysis. Sridharan and Bodık describe a demand-driven analysis that automatically varies context and field sensitivity [2006]. 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 the limit studies that other researchers have used to evaluate and debug various compiler analyses [Larus and Chandra 1993; Mock et al. 2001; Diwan et al. 2001; Liang et al. 2002]. Kotzmann and Mössenböck use their escape analysis for stack allocation [2005]. Validation runs maintain a heap-allocated copy of each stack-allocated object. Each field store updates both copies. Each field load retrieves both values and asserts their equality. This methodology convincingly demonstrates that the analysis is sound in practice.

10. CONCLUSIONS

Language features, such as dynamic generation and loading of code, reflection, and foreign language interfaces, create the following challenges for program analyses: (i) Dynamically generated and loaded code is, in general, not available for analysis until the application actually runs; moreover, two runs, even with the same inputs, may use different code if, for example, the two runs are in different environments (e.g., different classpaths). Thus a program analysis has to gracefully handle unavailable code. (ii) With reflection we do not know what field, method, or type a reflective action refers to until the program is actually run; moreover, two executions of a reflective action may refer to different entities. (iii) With foreign language interfaces we have to analyze code written in multiple languages in order to understand what a program is doing; moreover, the foreign language code is often low-level in nature (e.g., using unsafe casts) and thus may not even be amenable to static analysis.

This article presents and evaluates an online version of Andersen's pointer analysis in a Java virtual machine that handles all the aforementioned challenges. Because the analysis is online, time spent on the analysis disrupts application execution; thus, this work presents and evaluates various optimizations to reduce disruptions. These optimizations improve analysis performance by two orders of magnitude. If the analysis runs every time it gets new information (e.g., a new class is loaded), most analysis runs take under 0.1 seconds, and the total analysis time is 64.5 seconds on our benchmarks, on average.

Finally, since writing a pointer analysis that handles all the complexities of a modern language is hard, this article describes and employs a novel technique to find bugs in pointer analyses.

ACKNOWLEDGMENTS

We thank Steve Fink, Johannes Henkel, Balaji Iyengar, Ondřej Lhoták, Christoph Reichenbach, and Barbara Ryder for feedback on earlier drafts of this article. We thank the anonymous reviewers of ECOOP and TOPLAS for their comments.

REFERENCES

Agrawal, G., Li, J., and Su, Q. 2002. Evaluating a demand driven technique for call graph construction. In *Proceedings of the 11th International Conference on Compiler Construction (CC)*. Lecture Notes on Computer Science, vol. 2304. Springer Verlag, 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 Syst. J. 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. 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 Not. 35*, 10 (Oct.), 47–65. In *Proceedings of the 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 *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 2374. Springer Verlag.

Bacon, D. F. and Sweeney, P. F. 1996. Fast static analysis of C++ virtual function calls. *ACM SIGPLAN Not. 31*, 10 (Oct.), 324–341. In *Proceedings of the 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 Not. 38*, 5 (May), 103–114. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

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

Burke, M., Carini, P., Choi, J.-D., and Hind, M. 1994. Flow-Insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Extended version published as Res. Rep. RC 19546, IBM T. J. Watson Research Center, September.

Burke, M. and Torczon, L. 1993. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Trans. Program. Lang. Syst. 15*, 3 (Jul.), 367–399.

Chatterjee, R., Ryder, B. G., and Landi, W. A. 1999. Relevant context inference. In *Proceedings of the 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 Not. 35*, 5 (May), 57–69. In *Proceedings of the 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 *Proceedings of the 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 Not. 34*, 10 (Oct.), 1–19. In *Proceedings of the 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 *Proceedings of the 10th International Static Analysis Symposium (SAS)*. Lecture Notes in Computer Science, vol. 2694. Springer Verlag. 1–18.

Cierniak, M., Lueh, G.-Y., and Stichnoth, J. M. 2000. Practicing JUDO: Java under dynamic optimizations. *ACM SIGPLAN Not. 35*, 5 (May), 13–26. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

Cooper, K. D., Kennedy, K., and Torczon, L. 1986. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM SIGPLAN Not. 21*, 7 (Jul.), 58–67. In *Proceedings of the Symposium on Compiler Construction (SCC)*.

Das, M. 2000. Unification-Based pointer analysis with directional assignments. *ACM SIGPLAN Not. 35*, 5 (May), 35–46. In *Proceedings of the 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 *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 952. Springer Verlag. 77–101.

Detlefs, D. and Agesen, O. 1999. Inlining of virtual methods. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 1628. Springer Verlag. 258–278.

Diwan, A., McKinley, K. S., and Moss, J. E. B. 2001. Using types to analyze and optimize object-oriented programs. *ACM Trans. Program. Lang. Syst. 23*, 1 (Jan.), 30–72.

Duesterwald, E., Gupta, R., and Soffa, M. L. 1997. A practical framework for demand-driven interprocedural data flow analysis. *ACM Trans. Program. Lang. Syst. 19*, 6 (Nov.), 992–1030.

Emami, M., Ghiya, R., and Hendren, L. J. 1994. Context-Sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Not. 29*, 6 (Jun.), 242–256. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

Fähndrich, M., Foster, J. S., Su, Z., and Aiken, A. 1998. Partial online cycle elimination in inclusion constraint graphs. *ACM SIGPLAN Not. 33*, 5 (May), 85–96. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

Fernández, M. F. 1995. Simple and effective link-time optimization of Modula-3 programs. *ACM SIGPLAN Not. 30*, 6 (Jun.), 103–115. In *Proceedings of the 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 *Proceedings of the 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. August.

Ghiya, R. 1992. Interprocedural aliasing in the presence of function pointers. ACAPS Tech. Memo 62, McGill University. December.

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

Grunwald, D. and Srinivasan, H. 1993. Data flow equations for explicitly parallel programs. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*.

Guyer, S. and Lin, C. 2003. Client-Driven pointer analysis. In *Proceedings of the 10th International Static Analysis Symposium (SAS)*. Lecture Notes in Computer Science, vol. 2694. Springer Verlag. 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 *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Lecture Notes in Computer Science, vol. 768. Springer Verlag. 522–545.

Harris, T. 1999. Early storage reclamation in a tracing garbage collector. *ACM SIGPLAN Not. 34*, 4 (Apr.), 46–53. In *Proceedings of the International Symposium on Memory Management (ISMM)*.

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

Heintze, N. and Tardieu, O. 2001a. Demand-Driven pointer analysis. *ACM SIGPLAN Not. 36*, 5 (May), 24–34. In *Proceedings of the 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 Not. 36*, 5 (May), 254–263. In *Proceedings of the 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 *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 113–123.

Hirzel, M., Diwan, A., and Hertz, M. 2003. Connectivity-Based garbage collection. *ACM SIGPLAN Not. 38*, 11 (Nov.), 359–373. In *Proceedings of the 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 *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, vol. 3086. Springer Verlag. 96–122.

Hirzel, M., Henkel, J., Diwan, A., and Hind, M. 2003. Understanding the connectivity of heap objects. *ACM SIGPLAN Not. 38*, 2s (Feb.), 143–156. In *Proceedings of the International Symposium on Memory Management (ISMM)*.

Hölzle, U., Chambers, C., and Ungar, D. 1992. Debugging optimized code with dynamic deoptimization. *ACM SIGPLAN Not. 27*, 7 (Jul.), 32–43. In *Proceedings of the 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 Not. 29*, 6 (Jun.), 326–336. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

Hunt, G., Larus, J., Abadi, M., Aiken, M., Barham, P., Fähndrich, M., Hawblitzel, C., Hodson, O., Levi, S., Muprhy, N., Steensgaard, B., Tarditi, D., Wobber, T., and Zill, B. 2005. An overview of the Singularity project. Tech. Rep. MSR-TR-2005-135, Microsoft Research.

King, A. C. 2003. Removing synchronization (extended version). Tech. rep. 11-03, Computing Laboratory, University of Kent.

Kotzmann, T. and Mössenböck, H. 2005. Escape analysis in the context of dynamic compilation and deoptimization. In *Virtual Execution Environments (VEE)*.

LARUS, J. R. AND CHANDRA, S. 1993. Using tracing and dynamic slicing to tune compilers. Tech. Rep. 1174. August.

LATTNER, C. AND ADVE, V. 2003. Data structure analysis: An efficient context-sensitive heap analysis. Tech. Rep. UIUCDCS-R-2003-2340, Computer Science Department, University of Illinois at Urbana-Champaign. April.

LE, A., LHOTÁK, O., AND HENDREN, L. 2005. Using inter-procedural side-effect information in JIT optimizations. In *Proceedings of the 14th International Conference on Compiler Construction (CC)*. Lecture Notes in Computer Science, vol. 3443. Springer Verlag. 287–304.

LHOTÁK, O. AND HENDREN, L. 2003. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction (CC)*. Lecture Notes in Computer Science, vol. 2622. Springer Verlag. 153–169.

LIANG, D. AND HARROLD, M. J. 1999. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th European Software Engineering Conference, and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, O. Nierstraz and M. Lemoine, eds. Lecture Notes in Computer Science, vol. 1687. Springer Verlag. 199–215.

LIANG, D., PENNINGS, M., AND HARROLD, M. J. 2001. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. 73–79.

LIANG, D., PENNINGS, M., AND HARROLD, M. J. 2002. Evaluating the precision of static reference analysis using profiling. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 22–32.

LINDHOLM, T. AND YELLIN, F. 1999. *The Java Virtual Machine Specification*, 2nd ed. Addison-Wesley.

LIVSHITS, B., WHALEY, J., AND LAM, M. S. 2005. Reflection analysis for Java. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*.

MOCK, M., DAS, M., CHAMBERS, C., AND EGGERS, S. J. 2001. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. 66–72.

PALECZNY, M., VICK, C., AND CLICK, C. 2001. The Java Hotspot server compiler. In *Proceedings of the 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 Not. 36*, 11 (Nov.), 195–210. In *Proceedings of the 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 Not. 29*, 10 (Oct.), 324–324. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

QIAN, F. AND HENDREN, L. 2004. Towards dynamic interprocedural analysis in JVMs. In *Proceedings of the 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 *Proceedings of the 14th International Conference on Compiler Construction (CC)*. Lecture Notes in Computer Science, vol. 3443. Springer Verlag. 255–270.

ROUNTEV, A. AND CHANDRA, S. 2000. Off-Line variable substitution for scaling points-to analysis. *ACM SIGPLAN Not. 35*, 5 (May), 47–56. In *Proceedings of the 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 Not. 36*, 11 (Nov.), 43–55. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

RUF, E. 2000. Effective synchronization removal for Java. *ACM SIGPLAN Not. 35*, 5 (May), 208–218. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

RYDER, B. G. 2003. Dimensions of precision in reference analysis for object-oriented programming languages. In *Proceedings of the 12th International Conference on Compiler Construction (CC)*, G. Hedin, ed. Lecture Notes in Computer Science, vol. 2622. Springer Verlag. 126–137.

SAGIV, M., REPS, T., AND WILHELM, R. 1999. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th Symposium on Principles of Programming Languages (POPL)*. 105–118.

SHAPIRO, M. AND HORWITZ, S. 1997. The effects of the precision of pointer analysis. In *Proceedings of the 4th International Symposium on Static Analysis (SAS)*. Lecture Notes in Computer Science, vol. 1302. Springer Verlag. 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 Not. 35*, 5 (May), 196–207. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

SRIDHARAN, M. AND BODÍK, R. 2006. Refinement-Based context-sensitive points-to analysis for Java. In *Proceedings of the Programming Language Design and Implementation (PLDI)*.

STEENSGAARD, B. 1996. Points-To analysis in almost linear time. In *Proceedings of the 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 Not. 36*, 11 (Nov.), 180–195. In *Proceedings of the 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 Not. 35*, 10 (Oct.), 264–280. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

TIP, F. AND PALSBERG, J. 2000. Scalable propagation-based call graph construction algorithms. *ACM SIGPLAN Not. 35*, 10 (Oct.), 281–293. In *Proceedings of the 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 *Proceedings of the 9th International Conference on Compiler Construction (CC)*. Lecture Notes in Computer Science, vol. 1781. Springer Verlag. 18–34.

VIVIEN, F. AND RINARD, M. 2001. Incrementalized pointer and escape analysis. *ACM SIGPLAN Not. 36*, 5 (May), 35–46. In *Proceedings of the 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 *Proceedings of the 9th International Symposium on Static Analysis (SAS)*. Lecture Notes in Computer Science, vol. 2477. Springer Verlag. 180–195.

Received June 2005; revised February 2006; accepted August 2006