Scalable Propagation-Based Call Graph Construction Algorithms

Frank Tip
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
tip@watson.ibm.com

Jens Palsberg
Dept. of Computer Science
Purdue University
West Lafayette, IN 47907
palsberg@cs.purdue.edu

ABSTRACT

Propagation-based call graph construction algorithms have been studied intensively in the 1990s, and differ primarily in the number of sets that are used to approximate run-time values of expressions. In practice, algorithms such as RTA that use a single set for the whole program scale well. The scalability of algorithms such as 0-CFA that use one set per expression remains doubtful.

In this paper, we investigate the design space between RTA and 0-CFA. We have implemented various novel algorithms in the context of *Jax*, an application extractor for Java, and shown that they all scale to a 325,000-line program. A key property of these algorithms is that they do not analyze values on the run-time stack, which makes them efficient and easy to implement. Surprisingly, for detecting unreachable methods, the inexpensive RTA algorithm does almost as well as the seemingly more powerful algorithms. However, for determining call sites with a single target, one of our new algorithms obtains the current best tradeoff between speed and precision.

1. INTRODUCTION

A key task that is required by most approaches to whole-program optimization is the construction of a call graph approximation. Using a call graph, one can remove methods that are not reachable from the main method, replace dynamically dispatched method calls with direct method calls, inline methods calls for which there is a unique target, and perform more sophisticated optimizations such as interprocedural constant propagation, object inlining, and transformations of the class hierarchy. In the context of object-oriented languages with dynamic dispatch, the crucial step in constructing a call graph is to compute a conservative approximation of the set of methods that can be invoked by a given virtual (i.e., dynamically dispatched) method call.

OOPSLA '00, 10/00 Minneapolis, MN, USA
© 2000 ACM ISBN 1-58113-200-x/00/0010...$5.00

Call-graph construction algorithms have been studied intensively in the 1990s. While their original formulations use a variety of formalisms, most of them can be recast as set-based analyses. The common idea is to abstract an object into the name of its class, and to abstract a set of objects into the set of their classes. For any given call site $e.m()$, the goal is then to compute a set of class names S_e that approximates the run-time values of the receiver expression e. Once the sets S_e are determined for all expressions e, the class hierarchy can be examined to identify the methods that can be invoked.

Most call graph construction algorithms differ primarily in the *number* of sets that are used to approximate run-time values of expressions. Examples:

Number of sets used to approximate run-time values of expressions	Algorithm name
No sets	Class Hierarchy Analysis (CHA) [9, 10]
One set for the whole program	Rapid Type Analysis (RTA) [6, 5]
One set per expression	0-CFA (Control-Flow Analysis) [33, 17]
Several sets per expression	k-CFA, $k > 0$ [33, 17]

Intuitively, algorithms that use more sets compute more precise call graphs, but need more time and space to do the construction. In practice, the scalability of the algorithms at either end of the spectrum is fairly clear. The CHA and RTA algorithms at the low end of the range scale well and are widely used. The k-CFA algorithms (for $k > 0$) at the high end seem not to scale well at all [17]. The scalability of 0-CFA remains doubtful, mostly due to the large amounts of space required to represent the many different sets that arise. Recent work by Fähndrich et al. [14, 37] give grounds for optimism, although their recent results are obtained on a machine with 2,048 Megabytes of memory [37]. In the case of Java, another complicating factor for 0-CFA is that sets of class names need to be computed for locations on the run-time stack. Those locations are unnamed, and to facilitate 0-CFA, it seems necessary to first do a program transformation that names all the locations in some fashion, as done in various recent work [41, 38, 21]. Such transformations introduce both time and space overhead.

With the investigation of the scalability of 0-CFA still pending, our research focuses on the following questions:

- Are there interesting design points in the space between RTA and 0-CFA?
- Can we achieve better precision than RTA without analyzing values on the run-time stack?

We have implemented several novel algorithms in the context of *Jax*, an application extractor for Java, and shown that they all scale to a 325,000-line program. A key property of the algorithms is that they do not require simulation of the run-time stack, which makes them easy to implement and which helps efficiency. Our algorithms associate a single distinct set with each class, method, and/or field (but not each expression) in an application. Surprisingly, for detecting unreachable methods, the inexpensive RTA does almost as well as the seemingly more powerful algorithms. However, for determining call sites with a single target, one of our new algorithms obtains the current best tradeoff between speed and precision.

In summary, the results for the most precise of the new algorithms look as follows:

- The constructed call graphs tend to contain only slightly fewer method definitions (i.e., methods that have a body) when compared to RTA: up to 3.0% fewer method definitions, and 1.6% fewer method definitions on average, but in several cases significantly fewer edges (i.e., calling relationships between method definitions): up to 29.0% fewer edges, and 7.2% fewer edges on average.
- An in-depth study of the constructed call graphs revealed that the most precise of our algorithms uniquely resolves up to 26.3% of the virtual call sites that are deemed polymorphic by RTA (12.5% on average).
- Associating a distinct set of types with each *method* in a class has a significantly greater impact on precision than using a distinct set for each *field* in a class.
- The algorithms scale well: the running time is within an order of magnitude of a well-tuned RTA implementation in all cases. The most precise of our algorithms runs up to 8.3 times slower than RTA, and the correlation of this "slowdown factor" with program size appears to be weak (for the largest benchmark, our most expensive algorithm ran 5.0 times slower than RTA).
- The algorithms do not require exorbitant amounts of space. All our measurements were performed on a IBM ThinkPad 600E PC with 288 MB memory. None of our benchmarks required more than 200MB of heap space.

The remainder of this paper is organized as follows. Section 2 presents our new algorithms, and discusses their relation to several previous algorithms such as RTA. Section 3 discusses implementation issues. In Section 4, we compare the results computed by the new algorithms with those obtained with RTA. Section 5 presents related work. Finally, directions for future work are presented in Section 6.

2. THE ALGORITHMS

We will use a set-based framework to present both some existing and some new algorithms. This will enable easy comparison and help put our work in context. Figure 1 shows the relationships between four new algorithms (shown in a shaded area, and code-named CTA, MTA, FTA, and XTA) that will be presented shortly, and four well-known previous algorithms: RA (Reachability Analysis), CHA (Class Hierarchy Analysis), RTA (Rapid Type Analysis), and 0-CFA. The ordering from left to right corresponds to increased accuracy and increased cost. The increased cost stems from increased amounts of information used in resolving virtual method calls. The algorithms to the left of the new algorithms have been shown to scale well, whereas the scalability of 0-CFA remains doubtful. In Section 4, we will present results that demonstrate the scalability of the new algorithms.

2.1 Previous Algorithms

Since our new algorithms can be viewed as a natural "next step" with respect to previous work, we will first discuss some relevant previous algorithms. These algorithms progressively take more information into account when resolving virtual method calls.

2.1.1 Name-Based Resolution (RA)

We will begin by giving a set-constraint formulation of Reachability Analysis (RA), a simple algorithm for constructing call graphs that only takes into account the name of a method. (A slightly more advanced version of this algorithm relies on the equality of method *signatures* instead of method *names*.) Variations of RA have been presented in many places (see, e.g., [34]) and used in the context of tree-shakers for Lisp [16].

RA can be defined in terms of a set variable R (for "reachable methods") that ranges over sets of methods, and the following constraints, derived from the program text:

1. $main \in R$ ($main$ denotes the main method in the program)
2. For each method M, each virtual call site $e.m(\ldots)$ occurring in M, and each method M' with name m: $(M \in R) \Rightarrow (M' \in R)$.

Intuitively, the first constraint reads "the main method is reachable," and the second constraint reads "if a method is reachable, and a virtual method call $e.m(\ldots)$ occurs in its body, then every method with name m is also reachable." It is straightforward to show that there is a least set R that satisfies the constraints, and a solution procedure that computes that set. The reason for computing the least R that satisfies the constraints is that this maximizes the complement of R, i.e., the set of unreachable methods that can be removed safely.

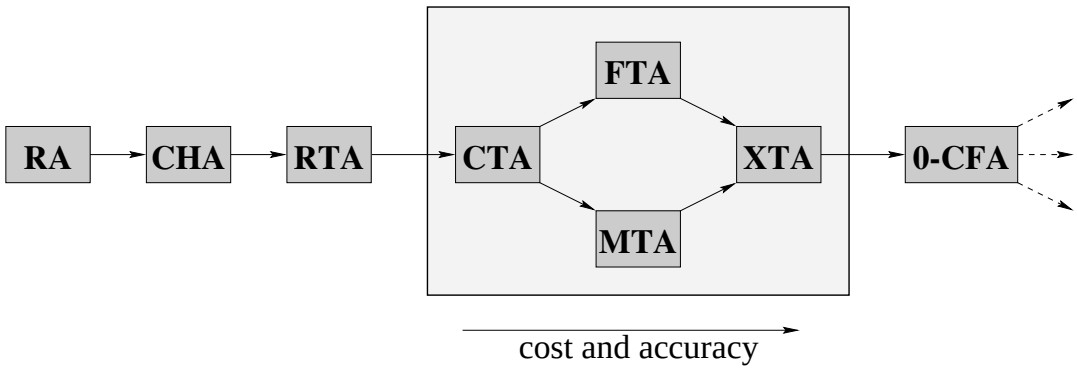

Figure 1: Schematic overview of the algorithms studied in this paper, and their relationship to several previous algorithms.

2.1.2 Class Hierarchy Analysis (CHA)

We can extend the constraint system for the basic reachability analysis to also take class hierarchy information into account. The result is known as *class hierarchy analysis* (CHA) [9, 10]. We will use the notation $StaticType(e)$ to denote the static type of the expression e, $SubTypes(t)$ to denote the set of declared subtypes of type t, and the notation $StaticLookup(C, m)$ to denote the definition (if any) of a method with name m that one finds when starting a static method lookup in the class C. Like RA, CHA uses just one set variable R ranging over sets of methods. The constraints:

1. $main \in R$ ($main$ denotes the main method in the program)
2. For each method M, each virtual call site $e.m(\ldots)$ occurring in M, and each class $C \in SubTypes(StaticType(e))$ where $StaticLookup(C, m) = M'$:
   $(M \in R) \Rightarrow (M' \in R)$.

Intuitively, the second constraint reads: "if a method is reachable, and a virtual method call $e.m(\ldots)$ occurs in the body of that method, then every method with name m that is inherited by a subtype of the static type of e is also reachable."

2.1.3 Rapid Type Analysis (RTA)

We can further extend CHA to take class-instantiation information into account. The result is known as *rapid type analysis* (RTA) [6, 5]. RTA uses both a set variable R ranging over sets of methods, and a set variable S which ranges over sets of class names. The variable S approximates the set of classes for which objects are created during a run of the program. The constraints:

1. $main \in R$ ($main$ denotes the main method in the program)
2. For each method M, each virtual call site $e.m(\ldots)$ occurring in M, and each class $C \in SubTypes(StaticType(e))$ where $StaticLookup(C, m) = M'$:
   $(M \in R) \wedge (C \in S) \Rightarrow (M' \in R)$.
3. For each method M, and for each "new $C()$" occurring in M:
   $(M \in R) \Rightarrow (C \in S)$.

Intuitively, the second constraint refines the corresponding constraint of CHA by insisting that $C \in S$, and the third constraint reads: "S contains the classes that are instantiated in a reachable method."

RTA is easy to implement, scales well, and has been shown to compute call graphs that are significantly more precise than those computed by CHA [6]. We are aware of several whole-program analysis systems that rely on RTA to compute call graphs (e.g., the *Jax* application extractor of [40].) In Section 4, we will use RTA as the baseline against which we compare the new call graph construction algorithms that we are about to present.

2.2 New Algorithms

The new algorithms use multiple set variables that range over sets of classes. We will associate these set variables with program entities such as classes, methods, and fields. The idea is that by giving each program entity a more precise "local" view of the types of objects available, call sites may be resolved more accurately.

2.2.1 Separate sets for methods and fields (XTA)

We will first present an algorithm that uses a distinct set variable S_M for each method M, and a distinct set variable S_x for each field x. We call this analysis XTA. We will use the notation $ParamTypes(M)$ for the set of static types of the arguments of the method M (*excluding* method M's `this` pointer), and the notation $ReturnType(M)$ for the static return type of M. We also extend the function $SubTypes(\cdot)$ to work on a set of types:

$$SubTypes(Y) = \bigcup_{y \in Y} SubTypes(y)$$

The following constraints define XTA:

1. $main \in R$ ($main$ denotes the main method in the program)
2. For each method M, each virtual call site $e.m(\ldots)$ occurring in M, and each class $C \in$

$SubTypes(StaticType(e))$ where $StaticLookup(C, m) = M'$:

$$(M \in R) \wedge (C \in S_M)$$

$$\Rightarrow \begin{cases} M' \in R \ \wedge \\ SubTypes(ParamTypes(M')) \cap S_M \subseteq S_{M'} \ \wedge \\ SubTypes(ReturnType(M')) \cap S_{M'} \subseteq S_M \ \wedge \\ C \in S_{M'} \end{cases}$$

3. For each method M, and for each "new $C()$" occurring in M:
$$(M \in R) \Rightarrow C \in S_M$$

4. For each method M in which a read of a field x occurs:
$$(M \in R) \Rightarrow S_x \subseteq S_M$$

5. For each method M in which a write of a field x occurs:
$$(M \in R) \Rightarrow (SubTypes(StaticType(x)) \cap S_M) \subseteq S_x$$

Intuitively, the second constraint refines the corresponding constraint of RTA by *(i)* insisting that objects of the target class C are available in the local set S_M associated with M, *(ii)* adding two inclusions that capture a flow of data from M to M', and from M' back to M, and *(iii)* stating that an object of type C (the "this" pointer) is available in M'. The third constraint refines the corresponding constraint of RTA by adding the class name C to just the set variable for the method M. The fourth constraint reflects a data flow from a field to a method body, and the fifth constraint reflects a data flow from a method body to a field, taking hierarchy information and creation point information into account.

2.2.2 Algorithms in the space between RTA and XTA

There is a spectrum of analyses between RTA and XTA. We have experimented with the following ones:

- **CTA:** The algorithm CTA uses a distinct set variable S_C for each class C. Intuitively, the set variable S_C unifies the flow information for all methods and fields. The constraints for CTA can be obtained by adding the following constraints to the definition of XTA:
  1. If a class C defines a method M: $S_C = S_M$.
  2. If a class C defines a field x: $S_C = S_x$.
- **MTA:** The algorithm MTA uses a distinct set variable S_C for each class C, and a set variable S_x for every field x. Intuitively, the set variable S_C unifies the flow information for all methods (but not fields.) The constraints for MTA can be obtained by adding the following constraints to the definition of XTA:
  1. If a class C defines a method M: $S_C = S_M$.
- **FTA:** The algorithm FTA uses a distinct set variable S_C for each class C, and a set variable S_M for every method M. Intuitively, the set variable S_C unifies the flow information for all fields (but not methods.) The constraints for MTA can be obtained by adding the following constraints to the definition of XTA:
  1. If a class C defines a field x: $S_C = S_x$.

Many other possibilities exist. For example, one could unify the sets associated with fields in the same class if they have the same type.

2.2.3 Summary

Let us now summarize the algorithms with which we have done experiments. For a given program, define:

$\mathcal{C}$: the number of classes in the program
$\mathcal{M}$: the number of methods in the program
$\mathcal{F}$: the number of fields in the program.

In the following table, the first column gives the number of set variables used to approximate run-time values of expressions.

Number of sets	Algorithm
0	CHA
1	RTA
$\mathcal{C}$	CTA
$\mathcal{C} + \mathcal{F}$	MTA
$\mathcal{C} + \mathcal{M}$	FTA
$\mathcal{M} + \mathcal{F}$	XTA

All of our new algorithms and also 0-CFA can be executed in $O(n^2 \times \mathcal{C})$ time, where n is the number of set variables [28]. We can view 0-CFA as an extension of XTA in the following way. Rather than using just one set variable for each method, 0-CFA uses one set variable for each argument and each expression that evaluates to an object, including references to objects on the run-time stack. The main problem for 0-CFA is that stack locations are unnamed in the Java virtual machine, so it seems necessary to first do a program transformation that names all the locations in some fashion, as done in various recent work [41, 38, 21].

The lattice that was shown previously in Figure 1 illustrates the relationships between the algorithms in terms of cost and accuracy. Section 5 discusses further how these algorithms compare to other algorithms.

3. IMPLEMENTATION ISSUES

We implemented several of the new algorithms of Section 2 in the context of *Jax*, an application extractor for Java [40]. Our implementation relies on "JikesBT" (IBM's Jikes Bytecode Toolkit)[1] for reading in the Java class files that constitute an application, and for creating an internal representation of the classes in which the string-based references of the class file format are represented by pointer references. *Jax* uses RTA for constructing call graphs, and our new algorithms reuse several important data structures that were previously designed for RTA. Since the algorithms of Section 2 are fairly simple, the amount of new code we had to write is only about 4000 lines.

The implementation performs the XTA algorithm in an iterative, propagation-based style. Three work-lists are associated with each program component (i.e., method or field)

[1] Jikes Bytecode Toolkit is a publically available class library for manipulating Java class files. See `www.alphaworks.ibm.com/tech/jikesbt`.

that keep track of "processed" types that have been propagated onwards from the component to other components, "current" types that will be propagated onwards in the current iteration, and "new" types that are propagated to the component in the current iteration and that will be propagated onwards in the next iteration.

The FTA and MTA algorithms are implemented by using a shared set for all the methods and fields in a class, respectively. Note that in the case of MTA (FTA) propagations between different methods (fields) in the same class are not needed. However, once a type is propagated to a method (field) in class C, the other methods (fields) in C still have to be revisited because onward propagations from those methods (fields) may have to take place. Due to time constraints, we have not completed the implementation of the CTA algorithm yet.

We use a combination of array-based and hash-based data structures that allow efficient membership-test operations, element addition, and iteration through all elements. We found that it is important to make *all* of these operations very efficient. Since the propagation of elements is filtered by types of method parameters, method return types, and types of fields, it is very important to efficiently implement subtype-tests. We use an approach described in [42] that relies on associating two integers with each class, corresponding to a pre-order and a post-order traversal of the class hierarchy. Using this numbering scheme, the existence of a subclass-relationship between two classes can be determined in unit time by comparing the associated numbers.

Applying the algorithms to realistic Java applications forced us to address several pragmatic issues:

direct method calls. Direct method calls can be modeled using simple set-inclusions between the sets associated with the callee and the caller.

arrays. Arrays are modeled as classes with one instance field that represents all of its elements. A method m is assumed to read an element from array A if: (i) an object of type A is propagated to m, and (ii) m contains an `aaload` byte code instruction. Similarly, a method m is assumed to write to A-element if: (i) an object of type A is propagated to m, and (ii) m contains an `aastore` instruction.

exception handling. The use of exception handling may cause nontrivial flow of types between methods, since exception objects may skip several stack frames before being caught. Any approach for tracking this flow of types precisely is fraught with complexity, and—in our opinion— unlikely to be very worthwhile, since the number of types involved is likely to be small (only subtypes of `java.lang.Throwable` are involved), and the hierarchy of user-defined exception types is often not very large or complex. Therefore, we use a single, global set of types that represents the run-time type of all expressions in the entire program whose static type is a subtype of `java.lang.Throwable`, and use that set to resolve all method calls on exception objects.

stack examination. While conducting experiments, we observed that better precision along with greater efficiency can be achieved by examining the instruction that follows a method call (or field read). If this instruction is of the form `checkcast` C, an exception is thrown unless the run-time type of the object returned by the method) is a subtype of C. In such cases, we can exclude from the types being propagated to the calling method any type that is not a subtype of C. Similarly, if the instruction that follows a method call is a stack `pop` operation, we can avoid propagating from the callee to the caller altogether. These situations occur frequently in the presence of polymorphic containers such as vectors and hash-tables.

incomplete applications. While we have described how our algorithms construct call graphs for complete applications with a single entry point, any realistic implementation must deal with situations where applications extend classes in the standard libraries, call library methods, and override library methods that are invoked from outside the application. Our basic approach is to associate a single set of objects S_E with the "outside world" (i.e., all code outside the application). This set interacts with the other sets as follows:

- If a method m calls a method m' outside the application, we propagate $(S_m \cap \mathit{ParamTypes}(m'))$ to S_E. For virtual methods, any types passed via the `this` pointer are also propagated to S_E.
- Whenever a method m writes to a field f outside the application, we propagate $(S_m \cap \mathit{Type}(f))$ to S_E. Read-accessing an external field causes similar flow in the opposite direction.
- If a virtual method in the application overrides an external method m, we make the conservative assumption that the external code contains a call to m'. Our approach is to use the set S_E to determine the set of methods in the application that may be invoked by the dynamic dispatch mechanism. For each such method m', we use the parameter types and return types of m' to model the flow of objects between S_E and $S_{m'}$.

We have observed that the above scheme is in certain cases unnecessarily conservative, because objects passed to a library method do not always pollute the global set S_E. Based on these observations, our implementation incorporates two refinements to the above scheme:

- For calls to certain methods in the standard libraries, we know that propagation to the set S_E is unnecessary, because the objects passed to the method will not be the receiver of subsequent method calls. The constructor of class `java.lang.Object` is a prime example in this category. We have identified about 25 heavily-called library methods for which calls can be ignored.
- A separate set of objects S_C can be associated with an external class C in cases where the objects passed to methods in C only interact with the other external classes in limited ways. For example, we can associate a distinct set with class `java.util.Vector`. Care has to

benchmark	# classes	# methods	#fields (reference-typed)	# virtual call sites
Hanoi	44	379	232 (107)	285
Ice Browser	76	761	500 (253)	922
mBird	2,050	17,946	6739 (4284)	3,269
Cindy	468	4,449	3075 (1677)	5,085
CindyApplet	468	4,449	3075 (1677)	2,502
eSuite Sheet	588	5,590	4305 (1412)	4,459
eSuite Chart	733	8,302	5448 (2141)	8,074
javaFig 1.43	161	2,108	1526 (971)	3,482
BLOAT	282	2,677	1255 (541)	6,623
JAX 6.3	309	2,754	1252 (579)	3,836
javac	210	1,512	1107 (406)	3,621
Res. System	2,332	21,495	12487 (6334)	23,640

Table 1: Benchmark characteristics.

be taken, because objects may flow from this class to other external classes (e.g., due to a call to `java.util.Vector.elements()`). There are three other collection classes that we modeled similarly.

reflection and dynamic loading. Nearly all of our benchmarks use dynamic loading and reflection. Since it is impossible for a static analysis to determine which classes may be accessed using these mechanisms, we have to manually supply the analysis with information about where objects are created. Issues related to whole-program analysis in the presence of these mechanisms are discussed at length in [39].

4. RESULTS

For a range of benchmarks, we have measured five characteristics of the results of MTA, FTA, and XTA, with the results of RTA as a baseline:

- the number of types available per method,
- the number of reachable methods,
- the number of edges in the call graphs,
- the number monomorphic and polymorphic call sites, and
- the running times.

Of particular interest is the classification of call sites into monomorphic and polymorphic ones, and we provide a detailed study of how our algorithms improve on RTA. While the number of reachable methods decreases little, we have found significant reductions in the number of edges in the constructed call graphs for several of the benchmarks. More importantly, we found a significant increase in the number of monomorphic call sites when moving from RTA to, especially, XTA.

4.1 Benchmark characteristics

Table 1 lists the benchmark applications that we used to evaluate our algorithm, and provides a number of relevant statistics for each of them. These benchmarks cover a wide spectrum of programming styles and are publically available (except for *Mockingbird* and *Reservation System*).

Hanoi is an interactive applet version of the well-known "Towers of Hanoi" problem, and is shipped with *Jax*. *ICE Browser*[2] is a simple internet browser. Mockingbird (*mBird*) is a proprietary IBM tool for multi-language interoperability. It relies on, but uses only limited parts of, several large class libraries (including Swing, now part of JDK 1.2, and IBM's XML parser). *Cinderella*[3] is an interactive geometry tool used for education and self-study in schools and universities. *CindyApplet* is an applet that allows users to interactively solve geometry exercises that were created with *Cinderella*. It is contained in the same class file archive as *Cinderella*. *Lotus eSuite Sheet* and *Lotus eSuite Chart* are interactive spreadsheet and charting applets, which are examples shipped with Lotus' *eSuite*[4] productivity suite (DevPack 1.5 version). *JavaFig*[5] (version 1.43 (22.02.99)) is a Java version of the `xfig` drawing program. *BLOAT*[6] is a byte-code optimizer developed at Purdue University. Version 6.3 of *Jax* itself was used as a benchmark. *javac*[7] is the SPEC JVM 98 version Sun's `javac` compiler. Our largest benchmark, *Reservation System*, is an interactive front-end for an airline, hotel, and car rental reservation system developed by an IBM customer, and consists of approximately 325,000 lines of Java source code.

Table 1 shows the number of classes, methods, and fields for each of the benchmarks. The table also shows the number of reference-typed fields in the application (i.e., fields whose type is a reference to a class), which is indicated between brackets in the "fields" column. Our system creates one

[2]See `www.icesoft.no`.
[3]See `www.cinderella.de`.
[4]See `www.esuite.lotus.com`.
[5]See `tech-www.informatik.uni-hamburg.de/applets/javafig`.
[6]See `www.cs.purdue.edu/homes/hosking/pjama.html`.
[7]See `www.specbench.org`.

set for each reference-typed field that is accessed from a reached method. Hence, this number is a bound on the number of field-sets that are created. Table 1 also shows the number of virtual call sites in each benchmark. For reasons we will explain shortly, virtual calls to methods outside the application are excluded from this statistic.

4.2 Set sizes

Statistics such as the number of reached methods and the percentage of uniquely resolved method calls are the measures by which call graph construction algorithms are traditionally compared. Since such measures all depend on the number of types available in a method, it is interesting to examine the average set of types available in each method as a more "absolute" measure. Table 4.2 shows, for each of the algorithms we implemented, the total number of types (instantiated classes), the average number of types available in each method body, and the latter as a percentage of the former. In the case of RTA, which uses only one set, the average number of types per method is is the same as the total number of types.

Table 2 tells us several interesting things:

- There are only a very few cases where RTA determines a larger total number of types than the other algorithms.
- Using MTA, an average 57.8% of all types is available in each method, a roughly two-fold reduction over RTA (where all types are available in each method). FTA does much better than MTA and determines that, on average, only 15.6% of all types are available in each method. XTA is better still, with 12.3% of the types being available on average.

While these reductions in average set size are substantial, it seems that there is still room for improvement. For *Reservation System*, we compute that, on average, about 200 types are available in each method when XTA is used. This number seems high, considering that the average size of a method is in the order of 15-20 lines of source code.

4.3 Reached methods

Table 3 shows, for each of the benchmarks, the number of methods in the call graphs computed by RTA, MTA, FTA, and XTA. Also shown are the percentage reductions of MTA, FTA, and XTA relative to RTA.

These statistics do not include `abstract` methods, which do not have a body, do not call other methods, and which cannot be the target of a dynamic dispatch. The rationale for excluding `abstract` methods has to do with the following observation: In cases where a virtual method m is called, but where m cannot be the target of a dynamic dispatch, it is possible to remove the m's body and make m into an `abstract` method without affecting program behavior (in fact, this is one of the optimizations performed by *Jax*). Therefore, counting abstract methods as first-class citizens makes it impossible to distinguish between call graphs that contain the same set of method headers, but different sets of method implementations. One could of course provide detailed statistics that include the number of `abstract` methods as well as the number of non-`abstract` methods, but we do not consider this additional detail to be very worthwhile.

In summary, we find that:

- MTA computes call graphs with 0 to 2.3% fewer method definitions than RTA (0.6% on average),
- FTA computes call graphs with 0 to 2.7% fewer method definitions than RTA (1.4% on average), and
- XTA computes call graphs with 0 to 3.0% fewer method definitions than RTA (1.6% on average).

4.4 Call graph edges

The next measure we study is the number of *edges* in the computed call graphs. In determining this number, we count each direct call site (i.e., a call that does not involve a dynamic dispatch) as one, and each virtual call site as the number of "target" methods that may be invoked by a dynamic dispatch from that site. Multiple calls to the same method m within a method body are counted separately (although our analyses will treat each of these calls similarly).

Since we do not know whether or not classes outside the application have been instantiated, it is not possible to accurately determine the number of targets of calls to virtual methods outside the application. Therefore, in performing these measurements, any calls to methods outside the application are ignored.

The results are shown in Table 4. It is clear that several of the new algorithms do eliminate substantially more edges than RTA does. The results can be summarized as follows:

- MTA computes call graphs with 0 to 4.7% fewer edges than RTA (1.6% on average),
- FTA computes call graphs with 0.1% to 26.7% fewer edges than RTA (6.6% on average), and
- XTA computes call graphs with 0.3% to 29.0% fewer edges than RTA (7.2% on average).

4.5 Uniquely resolved call sites

One of the key goals in the optimization of object-oriented programs is to find "monomorphic" virtual call sites from which only a single method can be invoked. Such call sites can be transformed into direct calls, and subsequently inlined and optimized further.

Table 5 classifies the virtual call sites in each of the benchmark applications as "unreached" (i.e., occurring in an unreached method), "monomorphic" (i.e, having a single target), or "polymorphic" (i.e., having multiple targets). Calls to methods outside the application are ignored again, since we cannot accurately determine the number of targets in such cases. We can conclude the following from Table 5:

- The percentage of virtual call sites that is unreached varies significantly from one benchmark to another.

benchmark	RTA	MTA			FTA			XTA		
	# types	# types	#types/method (avg)		# types	#types/method (avg)		#types	#types/method (avg)	
Hanoi	19	19	7.1	37.3%	19	3.8	20.0%	19	2.8	14.8%
Ice Browser	59	59	23.4	39.7%	59	7.1	12.0%	59	4.5	7.7%
mBird	178	178	90.0	50.6%	178	13.3	7.5%	178	11.6	6.5%
Cindy	238	237	153.2	64.6%	237	36.5	15.4%	237	28.3	11.9%
CindyApplet	105	104	64.6	62.1%	104	17.2	16.5%	104	13.9	13.4%
eSuite Sheet	174	174	96.0	55.2%	174	21.1	12.1%	174	13.8	7.9%
eSuite Chart	303	303	224.1	74.0%	303	48.6	16.0%	303	24.0	7.9%
javaFig 1.43	110	110	73.4	66.7%	110	21.2	19.3%	110	17.1	15.5%
BLOAT	209	209	163.8	78.4%	209	44.7	21.4%	209	42.4	20.3%
JAX 6.3	221	221	70.9	32.1%	221	10.0	4.5%	221	8.3	3.8%
javac	171	171	106.4	62.2%	171	39.9	23.3%	171	35.5	20.8%
Res. System	1,174	1174	833.8	71.0%	1172	228.2	19.5%	1172	200.0	17.1%
AVERAGE				57.8%			15.6%			12.3%

Table 2: Average set size per method for RTA, MTA, FTA, and XTA on each of the benchmarks.

benchmark	RTA	MTA	FTA	XTA	(RTA-MTA)/RTA	(RTA-FTA)/RTA	(RTA-XTA)/RTA
Hanoi	183	179	178	178	2.2%	2.7%	2.7%
Ice Browser	644	644	643	643	0.0%	0.2%	0.2%
mBird	1,862	1,855	1,825	1,824	0.4%	2.0%	2.0%
Cindy	2,437	2,412	2,404	2,403	1.0%	1.4%	1.4%
CindyApplet	1,237	1,209	1,203	1,202	2.3%	2.7%	2.8%
eSuite Sheet	2,414	2,400	2,385	2,367	0.6%	1.2%	1.9%
eSuite Chart	4,428	4,419	4,313	4,296	0.2%	2.6%	3.0%
javaFig 1.43	1,441	1,435	1,413	1,411	0.4%	1.9%	2.1%
BLOAT	2,143	2,142	2,120	2,091	0.0%	1.1%	2.4%
JAX 6.3	1,900	1,894	1,894	1,892	0.3%	0.3%	0.4%
javac	1,366	1,366	1,366	1,366	0.0%	0.0%	0.0%
Res. System	11,232	11,227	11,204	11,201	0.0%	0.2%	0.3%
AVERAGE					0.6%	1.4%	1.6%

Table 3: Number of methods in the call graphs computed by RTA, MTA, FTA, and XTA on each of the benchmarks.

benchmark	RTA	MTA	FTA	XTA	(RTA-MTA)/RTA	(RTA-FTA)/RTA	(RTA-XTA)/RTA
Hanoi	400	386	382	379	3.5%	4.5%	5.3%
Ice Browser	1,594	1,594	1,593	1,588	0.0%	0.1%	0.4%
mBird	8,061	8,036	7,772	7,760	0.3%	3.6%	3.7%
Cindy	15,457	14,729	11,331	10,967	4.7%	26.7%	29.0%
CindyApplet	5,223	4,990	4,399	4,347	4.5%	15.8%	16.8%
eSuite Sheet	7,171	7,149	7,093	7,071	0.3%	1.1%	1.4%
eSuite Chart	14,669	14,648	13,857	13,771	0.1%	5.5%	6.1%
javaFig 1.43	5,128	5,064	4,963	4,961	1.2%	3.2%	3.3%
BLOAT	19,384	18,772	16,704	16,672	3.2%	13.8%	14.0%
JAX 6.3	7,053	7,018	6,904	6,895	0.5%	2.1%	2.2%
javac	13,154	13,154	13,115	13,113	0.0%	0.3%	0.3%
Res. System	46,130	45,944	44,792	44,412	0.4%	2.9%	3.7%
AVERAGE					1.6%	6.6%	7.2%

Table 4: Number of edges in the call graphs computed by RTA, MTA, FTA, and XTA on each of the benchmarks.

benchmark	RTA			MTA			FTA			XTA		
	unreached	mono	poly	unreached	mono	poly	unreached	mono	poly	unreached	mono	poly
Hanoi	34.0%	61.6%	4.4%	34.0%	62.3%	3.7%	34.0%	62.7%	3.3%	34.0%	62.7%	3.3%
Ice Browser	4.0%	91.4%	4.7%	4.0%	91.4%	4.7%	4.0%	91.4%	4.7%	4.0%	91.6%	4.5%
mBird	14.2%	73.4%	12.3%	14.2%	73.5%	12.2%	17.4%	70.8%	11.8%	17.4%	70.9%	11.7%
Cindy	49.3%	45.0%	5.7%	49.5%	45.0%	5.5%	49.4%	45.2%	5.4%	49.4%	45.5%	5.0%
CindyApplet	72.0%	24.6%	3.4%	72.1%	24.6%	3.3%	72.3%	24.4%	3.3%	72.3%	24.5%	3.2%
eSuite Sheet	28.1%	68.4%	3.5%	28.1%	68.4%	3.5%	28.1%	69.1%	2.8%	28.2%	69.1%	2.8%
eSuite Chart	13.3%	76.6%	10.1%	13.3%	76.6%	10.1%	15.7%	75.7%	8.7%	15.7%	76.0%	8.3%
javaFig 1.43	9.1%	87.1%	3.9%	9.1%	87.4%	3.6%	9.7%	87.2%	3.1%	9.7%	87.2%	3.1%
BLOAT	6.6%	82.4%	11.1%	6.6%	82.4%	11.1%	6.7%	82.5%	10.8%	7.0%	82.2%	10.8%
JAX 6.3	18.7%	75.9%	5.4%	18.9%	75.7%	5.4%	18.9%	76.6%	4.5%	18.9%	76.8%	4.3%
javac	3.0%	77.6%	19.4%	3.0%	77.6%	19.4%	3.0%	77.6%	19.4%	3.0%	77.7%	19.3%
Res. System	18.1%	72.0%	9.9%	18.1%	72.2%	9.7%	18.2%	73.1%	8.7%	18.2%	74.0%	7.9%
AVERAGE			7.8%			7.7%			7.2%			7.0%

Table 5: Classification of virtual call sites according to the RTA, MTA, FTA, and XTA algorithms, for each of the benchmarks.

For example, only 3.0% of the virtual call sites in `javac` are unreached, whereas 72.0% of the virtual call sites in `CindyApplet` are unreached.

- Restricting our attention to *reached* virtual call sites, we find that *all* of the algorithms classify a vast majority of the call sites as monomorphic. Specifically, we find that:
  - RTA classifies between 3.4% and 19.4% of all call sites as polymorphic (7.8% on average).
  - MTA classifies between 3.3% and 19.4% of all call sites as polymorphic (7.7% on average).
  - FTA classifies between 2.8% and 19.4% of all call sites as polymorphic (7.2% on average).
  - XTA classifies between 2.8% and 19.3% of all call sites as polymorphic (7.0% on average).

In summary, we can conclude that RTA does a very good job in classifying virtual call sites as monomorphic. For the benchmarks we study in this paper, only 7.8% of all virtual call sites are classified as polymorphic (on average), which leaves little room for improvement. XTA classifies an average of 7.0% of all virtual call sites as polymorphic.

4.6 Detailed comparison

Table 6 shows a detailed comparison of the call graphs constructed by RTA and MTA/FTA/XTA, for each of the benchmarks. Each call site in the RTA call graph is classified as one of the following:

- **mono-to-unreached**: virtual call sites that were resolved to a single target method in the RTA call graph, and that became unreached in the MTA/FTA/XTA call graphs (due to the fact that the method containing the call site in question became unreachable).
- **mono-to-mono**: virtual call sites that were resolved to a single target method in both the RTA and the MTA/FTA/XTA call graphs.
- **poly-to-unreached**: virtual call sites that were resolved to more than 1 target in the RTA call graph, and that became unreached in the MTA/FTA/XTA call graphs.
- **poly-to-mono**: virtual call sites that were resolved to more than 1 target in the RTA call graph, but to a unique target in the MTA/FTA/XTA call graphs.
- **poly-to-poly**: virtual call sites that were resolved to more than 1 target in both the RTA and the MTA/FTA/XTA call graphs.

To determine how much more accurate the MTA/FTA/XTA algorithms are when compared to RTA in relative terms, we can observe the following:

- Any call sites determined to be unreachable by MTA/FTA/XTA have no impact on an application's performance. After all, they are never executed.
- Any call sites determined to be monomorphic by RTA will not be improved by a better algorithm (because they either stay monomorphic, or they become unreached).

Hence, what remains are the **poly-to-mono** and the **poly-to-poly** categories. The *ratio* between these categories reflects the *relative* improvement of MTA/FTA/XTA over RTA.

As an example, consider *Reservation System*, our largest benchmark. This application contains a total of 23,640 virtual call sites. If we subtract (i) all call sites that are determined to be monomorphic by RTA, and (ii) all call sites that are determined to be unreachable by XTA, we are left with 2,824 call sites that are determined to be polymorphic by RTA. XTA determines that 569 of these call sites are, in fact, monomorphic. Hence, XTA is capable of devirtualizing 569/2,824 = 20.1% of the call sites deemed polymorphic by RTA. If we apply this line of reasoning to Table 6, we find that:

- MTA finds a single target for up to 15.8% of the call sites deemed polymorphic by RTA (2.9% on average).

benchmark	mono->unreached		mono->mono		poly->unreached		poly->mono		poly->poly	
Hanoi	0	(MTA)	266	(MTA)	0	(MTA)	3	(MTA)	16	(MTA)
	0	(FTA)	266	(FTA)	0	(FTA)	5	(FTA)	14	(FTA)
	0	(XTA)	266	(XTA)	0	(XTA)	5	(XTA)	14	(XTA)
Ice Browser	0	(MTA)	877	(MTA)	0	(MTA)	0	(MTA)	45	(MTA)
	1	(FTA)	876	(FTA)	0	(FTA)	0	(FTA)	45	(FTA)
	1	(XTA)	876	(XTA)	0	(XTA)	2	(XTA)	43	(XTA)
mBird	0	(MTA)	2,807	(MTA)	0	(MTA)	4	(MTA)	458	(MTA)
	141	(FTA)	2,666	(FTA)	8	(FTA)	12	(FTA)	442	(FTA)
	141	(XTA)	2,666	(XTA)	8	(XTA)	16	(XTA)	438	(XTA)
Cindy	8	(MTA)	4,510	(MTA)	0	(MTA)	9	(MTA)	548	(MTA)
	11	(FTA)	4,507	(FTA)	0	(FTA)	24	(FTA)	543	(FTA)
	14	(XTA)	4,504	(XTA)	0	(XTA)	52	(XTA)	515	(XTA)
CindyApplet	8	(MTA)	2,185	(MTA)	3	(MTA)	12	(MTA)	294	(MTA)
	21	(FTA)	2,172	(FTA)	3	(FTA)	3	(FTA)	303	(FTA)
	25	(XTA)	2,168	(XTA)	3	(XTA)	14	(XTA)	292	(XTA)
eSuite Sheet	1	(MTA)	4,272	(MTA)	2	(MTA)	0	(MTA)	184	(MTA)
	1	(FTA)	4,272	(FTA)	2	(FTA)	45	(FTA)	139	(FTA)
	2	(XTA)	4,271	(XTA)	2	(XTA)	45	(XTA)	139	(XTA)
eSuite Chart	2	(MTA)	7,186	(MTA)	2	(MTA)	1	(MTA)	883	(MTA)
	209	(FTA)	6,979	(FTA)	32	(FTA)	100	(FTA)	754	(FTA)
	217	(XTA)	6,971	(XTA)	32	(XTA)	135	(XTA)	719	(XTA)
javaFig 1.43	1	(MTA)	3,333	(MTA)	0	(MTA)	12	(MTA)	136	(MTA)
	23	(FTA)	3,311	(FTA)	2	(FTA)	26	(FTA)	120	(FTA)
	25	(XTA)	3,309	(XTA)	2	(XTA)	26	(XTA)	120	(XTA)
BLOAT	0	(MTA)	5,838	(MTA)	0	(MTA)	2	(MTA)	783	(MTA)
	26	(FTA)	5,812	(FTA)	0	(FTA)	20	(FTA)	765	(FTA)
	58	(XTA)	5,780	(XTA)	0	(XTA)	20	(XTA)	765	(XTA)
JAX 6.3	11	(MTA)	3,571	(MTA)	0	(MTA)	2	(MTA)	252	(MTA)
	11	(FTA)	3,571	(FTA)	0	(FTA)	46	(FTA)	208	(FTA)
	11	(XTA)	3,571	(XTA)	0	(XTA)	55	(XTA)	199	(XTA)
javac	0	(MTA)	2,898	(MTA)	0	(MTA)	0	(MTA)	723	(MTA)
	0	(FTA)	2,898	(FTA)	0	(FTA)	0	(FTA)	723	(FTA)
	0	(XTA)	2,898	(XTA)	0	(XTA)	2	(XTA)	721	(XTA)
Res. System	4	(MTA)	20,797	(MTA)	2	(MTA)	48	(MTA)	2,789	(MTA)
	22	(FTA)	20,779	(FTA)	14	(FTA)	324	(FTA)	2,501	(FTA)
	23	(XTA)	20,778	(XTA)	15	(XTA)	569	(XTA)	2,255	(XTA)

Table 6: Detailed comparison of the call graphs constructed by RTA and by MTA/FTA/XTA for each of the benchmarks.

benchmark	RTA	MTA	FTA	XTA	MTA/RTA	FTA/RTA	XTA/RTA
Hanoi	0.2	1.1	1.0	1.2	5.5	5.0	6.0
Ice Browser	0.4	1.7	2.4	2.1	4.3	6.0	5.3
mBird	1.1	4.8	5.2	5.8	4.4	4.7	5.3
Cindy	2.0	10.8	8.7	9.3	5.4	4.4	4.7
CindyApplet	0.7	3.0	2.4	2.6	4.3	3.4	3.7
eSuite Sheet	1.5	5.4	6.4	6.9	3.6	4.3	4.6
eSuite Chart	7.0	33.0	24.3	19.9	4.7	3.5	2.8
javaFig 1.43	0.8	3.4	4.5	4.9	4.3	5.6	6.1
BLOAT	1.8	13.1	7.9	8.7	7.3	4.4	4.8
JAX 6.3	1.2	3.4	3.2	3.5	2.8	2.7	2.9
javac	1.2	12.1	8.9	10.0	10.1	7.4	8.3
Res. System	44.5	329.4	309.1	250.2	7.4	6.9	5.6
AVERAGE					5.3	4.9	5.0

Table 7: Running times (in seconds) of the RTA, XTA, and YTA algorithms on each of the benchmarks.

- FTA finds a single target for up to 26.3% of the call sites deemed polymorphic by RTA (9.9% on average), and
- XTA finds a single target for up to 26.3% of the call sites deemed polymorphic by RTA (12.5% on average).

4.7 Running times

Table 7 shows the running time for the RTA, MTA, FTA, and XTA algorithms on each of the benchmarks[8]. In summary, we found that the XTA algorithm is up to 8.3 times slower than RTA. The correlation between the slowdown factor and program size appears to be weak: XTA is only 5.0 times slower than RTA on *Reservation System*. Based on our experiences we believe that, on a large machine, our algorithms should have no problems with million-line programs.

Surprisingly, the MTA and FTA algorithms were in several instances somewhat slower than XTA. Due to time constraints, we have not been able to analyze the source of these slowdowns. A possible explanation is that the increased number of types available in a method results in additional work in resolving the call sites within that method. However, it seems unlikely that this would negate all the benefits from a decreased number of propagations between sets. We would expect an efficient implementation of MTA/FTA (e.g, using techniques by Fähndrich et al [13]) to be significantly more efficient than XTA.

4.8 Assessment

Our experiments have demonstrated that it is feasible to construct propagation-based call graph construction algorithms that use more than a single set of objects to approximate the run-time values of expressions. Regarding the precision of these algorithms, we have observed that:

- The algorithms are slightly more accurate than RTA in terms of the number of reached methods.
- In several cases, the algorithms are significantly more accurate than RTA in terms of the number of edges between the methods in the call graph.
- More importantly, the reduction in the number of edges is to a significant extent derived from call sites that are classified as polymorphic by RTA, but as monomorphic call sites by our new algorithms.

With respect to the number of sets one should use, we conclude that:

- Using a distinct set for each method in the program is useful, because it improves accuracy.
- Using a unified set to represent the fields in a class does not lead to a great loss of accuracy.

[8]Measurements taken on an IBM ThinkPad 600E PC with a 300Mhz processor and 288MB of main memory. We used the Sun JDK 1.1.8 VM with the just-in-time compiler developed at the IBM Tokyo Research Laboratory [22]. None of the benchmarks required more than 200MB of heap space.

In light of the improved results of XTA over FTA in some cases, we consider the XTA algorithm the best choice. If heap space is at a premium, FTA offers reduced space consumption in exchange for a slight loss of precision. MTA seems a poor choice since it computes call graphs that have roughly the same precision as those computed by RTA, but it is more complex to implement. Although we do not have data for CTA, we know it is less accurate than MTA, and the same arguments can be made to argue why it is not a very good choice.

5. RELATED WORK

5.1 Propagation-based algorithms

The idea of doing a propagation-based program analysis with one set variable for each expression is well known. This so-called *monovariant* style of analysis can be done in $O(n^3)$ time where n is the number of expressions. When the goal is to construct a call graph approximation in object-oriented or functional languages, then that style of analysis is known as 0-CFA [33], and when the goal is to do points-to analysis for C programs, then that style of analysis is often referred to as "Andersen's analysis" [3, 31]. 0-CFA has been implemented for a variety of languages, including dynamically-typed object-oriented languages [29, 28, 1], functional languages [33, 19], and statically-typed object-oriented languages, including Java [11, 38, 21]. The experience has been that the effectiveness of the approaches is language-dependent, and perhaps even programming-style dependent.

The idea of *polyvariance* is to associate more than one set variable with each expression, and thereby obtain better precision for each call site. Polyvariant analysis was pioneered by Sharir and Pnueli [32], and Jones and Muchnick [25]. In the 1990s the study of polyvariant analysis has been intensive. Well known are the k-CFA algorithms of Shivers [33], the poly-k-CFA of Jagannathan and Weeks [24], and the cartesian product algorithm of Agesen [1, 2]. A particularly simple polyvariant analysis was presented by Schmidt [30]. Frameworks for defining polyvariant analyses have been presented by Stefanescu and Zhou [36], Jagannathan and Weeks [23], and Nielson and Nielson [26]. Successful applications of polyvariant analysis include the optimizing compiler of Chambers et al [17], and of Hendren et al [12], and the partial evaluator of Consel [8]. As far as we know, these polyvariant approaches have not been tried on programs of 300,000+ lines of code.

5.2 Algorithms not based on propagation

Calder and Grunwald [7] investigated a particularly simple approach to inlining based on the *unique name* measure, that is, inlining in cases where there statically is a unique target for a given call site.

A variation of 0-CFA is the unification-based approach, also known as the equality-based approach, pioneered for call graph construction by [20], and later adapted to points-to analysis for C by Steensgaard [35]. A comparison of Andersen's analysis and Steensgaard's analysis has been presented by Shapiro and Horwitz [31]. The unification-based approach is cheaper and less precise than the 0-CFA-style approach.

A broader comparison was given by Foster, Fähndrich, and Aiken [15]; they compared both polymorphic versus monomorphic and equality-based versus inclusion-based points-to analysis. Their main conclusion is that the monomorphic inclusion-based algorithm is a good choice because 1) it usually beats the polymorphic equality-based algorithm, 2) it is not much worse than the polymorphic inclusion-based algorithm, and 3) it is simple to implement because it avoids the complications of polymorphism.

An experimental comparison of RTA and a unification-based approach to call graph construction was carried out by DeFouw, Grove, and Chambers [11]. Their paper presents a family of algorithms that blend propagation and unification, thereby in effect dynamically determining which set variables to unify based on how propagation proceeds. Members of the family include RTA, 0-CFA, and a number of algorithms with cost and precision in between. Our algorithms, by contrast, uses static criteria to decide which set variables are to be merged, and then performs the usual propagations between them. This is potentially a poorer choice, both for accuracy and analysis time, than the approach in [11]. Our static criterion for merging set variables stems from our desire to keep the algorithm simple, in particular by avoiding analysis of the run-time stack.

Ashley [4] also presented an algorithm that blends unification and propagation, in the setting of Scheme.

6. FUTURE WORK

A comparison of our algorithms, 0-CFA, and the variations presented by Sundaresan et al. [38] and Ishizaki et al. [21] seems to require a framework in which a program transformation names all stack locations. This would be a significant extension to our framework so it may be easier to do a comparison in the settings of [38, 21]. Questions that could be addressed by such a comparison include: 1) does 0-CFA use significantly more time and space than our algorithms for large benchmarks? 2) is the potential extra precision of 0-CFA worth the increased cost? and 3) when used in a compiler for devirtualization of monomorphic calls, does 0-CFA give significantly better speedups than our algorithms? Answers may well shed more light on which algorithm to choose. Perhaps 0-CFA needs to be a fair bit better than the other algorithms before it becomes tempting to implement the program transformation that enables 0-CFA for Java bytecodes.

While adding Hindley-Milner polymorphism seems not to be worthwhile [15], we have conducted some initial experiments with the use of data-polymorphism. The idea is well known: treat each distinct allocation site as a separate class, and keep the fields in these artificial classes distinct [27, 18]. Similarly, distinct sets may be used when methods are invoked on objects of the same type but allocated at different sites. Data-polymorphism has the potential of significantly increasing the cost (more elements have to be propagated, and the number of distinct sets may increase as well). However, accuracy may improve as well because unrelated instantiations of the same type are kept separate, thereby leading to a more precise analysis of their fields.

Acknowledgments

We are grateful to Aldo Eisma, David Grove, Tony Hosking, and especially Craig Chambers for useful discussions and feedback on drafts of this paper. The anonymous OOPSLA referees also provided many helpful suggestions.

Palsberg is supported by a National Science Foundation Faculty Early Career Development Award, CCR–9734265, and by IBM.

7. REFERENCES

[1] Agesen, O. Constraint-based type inference and parametric polymorphism. *Proceedings of the First International Static Analysis Symposium (SAS'94)* (September 1994), 78–100. Springer-Verlag LNCS vol. 864.

[2] Agesen, O. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Stanford University, December 1995. Appeared as Sun Microsystems Laboratories Technical Report SMLI TR-96-52.

[3] Andersen, L. O. Self-applicable C program specialization. In *Proceedings of PEPM'92, Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (June 1992), pp. 54–61. (Technical Report YALEU/DCS/RR-909, Yale University).

[4] Ashley, J. M. A practical and flexible flow analysis for higher-order languages. In *Proceedings of POPL'96, 23nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages* (1996), pp. 184–194.

[5] Bacon, D. F. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, Computer Science Division, University of California, Berkeley, Dec. 1997. Report No. UCB/CSD-98-1017.

[6] Bacon, D. F., and Sweeney, P. F. Fast static analysis of C++ virtual function calls. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, 1996), pp. 324–341. *SIGPLAN Notices* 31(10).

[7] Calder, B., and Grunwald, D. Reducing indirect function call overhead in C++ programs. *Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages* (January 1994), 397–408.

[8] Consel, C. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Proceedings of PEPM'93, Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (1993), pp. 145–154.

[9] Dean, J., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. Tech. Rep. 94-12-01, Department of Computer Science, University of Washington at Seattle, December 1994.

[10] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP'95)* (Aarhus, Denmark, Aug. 1995), W. Olthoff, Ed., Springer-Verlag, pp. 77–101.

[11] DeFouw, G., Grove, D., and Chambers, C. Fast interprocedural class analysis. In *Conference Record of the Twenty-Fifth ACM Symposium on Principles of Programming Languages* (San Diego, CA, January 1998), pp. 222–236.

[12] Emami, M., Ghiya, R., and Hendren, L. J. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (1994), pp. 242–256.

[13] Fähndrich, M., and Aiken, A. Program analysis using mixed term and set constraints. In *Proceedings of SAS'97, International Static Analysis Symposium* (1997), Springer-Verlag (*LNCS*), pp. 114–126.

[14] Fähndrich, M., Foster, J. S., Su, Z., and Aiken, A. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (1998), pp. 85–96.

[15] Foster, J. S., Fähndrich, M., and Aiken, A. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proceedings of SAS 2000, 7th Static Analysis Symposium* (2000), J. Palsberg, Ed., pp. 175–198.

[16] Goldberg, A., and Robson, D. *Smalltalk-80—The Language and its Implementation.* Addison-Wesley, 1983.

[17] Grove, D., DeFouw, G., Dean, J., and Chambers, C. Call graph construction in object-oriented languages. In *Proceedings of the Twelfth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)* (Atlanta, GA, 1997), pp. 108–124. *SIGPLAN Notices* 32(10).

[18] Grove, D., DeFouw, G., Dean, J., and Chambers, C. Call graph construction in object-oriented languages. In *Proceedings of OOPSLA'97, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (1997), pp. 108–124. *SIGPLAN Notices* 32(10).

[19] Heintze, N. Set-based analysis of ML programs. In *Proceedings of ACM Conference on LISP and Functional Programming* (1994), pp. 306–317.

[20] Henglein, F. Dynamic typing. In *Proceedings of ESOP'92, European Symposium on Programming* (1992), Springer-Verlag (*LNCS* 582), pp. 233–253.

[21] Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H., and Nakatani, T. A study of devirtualization techniques for a Java just-in-time compiler. In *Proceedings of the Fifteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)* (Minneapolis, Minnesota), 2000).

[22] Ishizaki, K., Kawahito, M., Yasue, T., Takeuchi, M., Ogasawara, T., Suganuma, T., Onodera, T., Komatsu, H., and Nakatani, T. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM SIGPLAN JavaGrande Conference* (San Francisco, CA, June 1999).

[23] Jagannathan, S., and Weeks, S. A unified treatment of flow analysis in higher-order languages. In *Proceedings of POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages* (1995), pp. 393–407.

[24] Jagannathan, S., and Wright, A. Effective flow analysis for avoiding run-time checks. In *Proceedings of SAS'95, International Static Analysis Symposium* (Glasgow, Scotland, September 1995), Springer-Verlag (*LNCS* 983).

[25] Jones, N., and Muchnick, S. A flexible approach to interprocedural data flow analysis of programs with recursive data structures. In *Ninth Symposium on Principles of Programming Languages* (1982), pp. 66–74.

[26] Nielson, F., and Nielson, H. R. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Proceedings of POPL'97, 24th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages* (1997), pp. 332–345.

[27] Oxhøj, N., Palsberg, J., and Schwartzbach, M. I. Making type inference practical. In *Proceedings of ECOOP'92, Sixth European Conference on Object-Oriented Programming* (Utrecht, The Netherlands, July 1992), Springer-Verlag (*LNCS* 615), pp. 329–349.

[28] Palsberg, J., and Schwartzbach, M. *Object-Oriented Type Systems.* John Wiley & Sons, 1993.

[29] Palsberg, J., and Schwartzbach, M. I. Object-oriented type inference. In *Proceedings of OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications* (Phoenix, Arizona, October 1991), pp. 146–161.

[30] Schmidt, D. Natural-semantics-based abstract interpretation. In *Proceedings of SAS'95, International Static Analysis Symposium* (Glasgow, Scotland, September 1995), Springer-Verlag (*LNCS* 983).

[31] Shapiro, M., and Horwitz, S. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the Twenty-Fourth ACM Symposium on Principles of Programming Languages* (Paris, France, 1997), pp. 1–14.

[32] Sharir, M., and Pnueli, A. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis, Theory and Applications*, S. Muchnick and N. Jones, Eds. 1981.

[33] Shivers, O. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, CMU, May 1991. CMU–CS–91–145.

[34] Srivastava, A. Unreachable procedures in object oriented programming. *ACM Letters on Programming Languages and Systems 1*, 4 (December 1992), 355–364.

[35] Steensgaard, B. Points-to analysis in almost linear time. In *Proceedings of the Twenty-Third ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL, January 1996), pp. 32–41.

[36] Stefanescu, D., and Zhou, Y. An equational framework for flow analysis of higher-order functional programs. In *Proceedings of ACM Conference on LISP and Functional Programming* (1994), pp. 318–327.

[37] Su, Z., Fähndrich, M., and Aiken, A. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of POPL'00, 27nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages* (2000), pp. 81–95.

[38] Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., and Godin, C. Practical virtual method call resolution for Java. In *Proceedings of the Fifteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)* (Minneapolis, Minnesota), 2000).

[39] Sweeney, P. F., and Tip, F. Extracting library-based object-oriented applications. In *Proceedings of the Eighth International Symposium on the Foundations of Software Engineering (FSE-8)* (November 2000). To appear. A previous version of this paper appeared as IBM Research Report RC 21596, November 1999.

[40] Tip, F., Laffra, C., Sweeney, P. F., and Streeter, D. Practical experience with an application extractor for Java. In *Proceedings of the Fourteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)* (Denver, CO), 1999), pp. 292–305. *SIGPLAN Notices* 34(10).

[41] Vallé-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., and Sundaresan, V. Optimizing java bytecode using the soot framework: Is it feasible? In *Proceedings of CC'00, International Conference on Compiler Construction* (2000), Springer-Verlag (*LNCS*).

[42] Vitek, J., Horspool, R. N., and Krall, A. Efficient type inclusion tests. In *Proceedings of OOPSLA'97, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (1997), pp. 142–157. *SIGPLAN Notices* 32(10).