

Interprocedural Data Flow Analysis in Soot using Value Contexts

Rohan Padhye

Indian Institute of Technology Bombay
rohanpadhye@iitb.ac.in

Uday P. Khedker

Indian Institute of Technology Bombay
uday@cse.iitb.ac.in

Abstract

An interprocedural analysis is precise if it is flow sensitive and fully context-sensitive even in the presence of recursion. Many methods of interprocedural analysis sacrifice precision for scalability while some are precise but limited to only a certain class of problems.

Soot currently supports interprocedural analysis of Java programs using graph reachability. However, this approach is restricted to IFDS/IDE problems, and is not suitable for general data flow frameworks such as heap reference analysis and points-to analysis which have non-distributive flow functions.

We describe a general-purpose interprocedural analysis framework for Soot using data flow values for context-sensitivity. This framework is not restricted to problems with distributive flow functions, although the lattice must be finite. It combines the key ideas of the tabulation method of the functional approach and the technique of value-based termination of call string construction.

The efficiency and precision of interprocedural analyses is heavily affected by the precision of the underlying call graph. This is especially important for object-oriented languages like Java where virtual method invocations cause an explosion of spurious call edges if the call graph is constructed naively. We have instantiated our framework with a flow and context-sensitive points-to analysis in Soot, which enables the construction of call graphs that are far more precise than those constructed by Soot's SPARK engine.

Categories and Subject Descriptors F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program Analysis

General Terms Algorithms, Languages, Theory

Keywords Interprocedural analysis, context-sensitive analysis, points-to analysis, call graph

1. Introduction

Interprocedural data flow analysis incorporates the effects of procedure calls on the callers and callees. A context-insensitive analysis does not distinguish between distinct calls to a procedure. This causes the propagation of data flow values across interprocedurally invalid paths (i.e. paths in which calls and returns may not match)

resulting in a loss of precision. A context-sensitive analysis restricts the propagation to valid paths and hence is more precise.

Two most general methods of precise flow and context-sensitive analysis are the *Functional* approach and the *Call Strings* approach [11]. The functional approach constructs summary flow functions for procedures by reducing compositions and meets of flow functions of individual statements to a single flow function, which is used directly in call statements. However, constructing summary flow functions may not be possible in general. The tabulation method of the functional approach overcomes this restriction by enumerating the functions as pairs of input-output data flow values for each procedure, but requires a finite lattice.

The call strings method remembers calling contexts in terms of unfinished calls as call strings. However, it requires an exponentially large number of call strings. The technique of value based termination of call string construction [5] uses data flow values to restrict the combinatorial explosion of contexts and improves the efficiency significantly without any loss of precision.

Graph reachability based interprocedural analysis [9, 10] is a special case of the functional approach. Formally, it requires flow functions $2^A \mapsto 2^A$ to distribute over the meet operation so that they can be decomposed into meets of flow functions $A \mapsto A$. Here A can be either a finite set D (for IFDS problems [9]) or a mapping $D \mapsto L$ (for IDE problems [10]) from a finite set D to a lattice of values L . Intuitively, A represents a node in the graph and a function $A \mapsto A$ decides the nature of the edge from the node representing the argument to the node representing the result. Flow function composition then reduces to a transitive closure of the edges resulting in paths in the graph.

The efficiency and precision of interprocedural analyses is heavily affected by the precision of the underlying call graph. This is especially important for object oriented languages like Java where virtual method invocations cause an explosion of spurious call edges if the call graph is constructed naively.

Soot [12] has been a stable and popular choice for hundreds of client analyses for Java programs, though it has traditionally lacked an interprocedural framework. Bodden [4] has recently implemented support for interprocedural analysis using graph reachability. The main limitation of this approach is that it is not suitable for general data flow frameworks with non-distributive flow functions such as heap reference analysis or points-to analysis. For example, consider the Java statement $x = y.n$ to be processed for points-to analysis. If we have points-to edges $y \rightarrow o_1$ and $o_1.n \rightarrow o_2$ before the statement (where o_1 and o_2 are heap objects), then it is not possible to correctly deduce that the edge $x \rightarrow o_2$ should be generated after the statement if we consider each input edge independently. The flow function for this statement is a function of the points-to graph as a whole and cannot be decomposed into independent functions of each edge and then merged to get a correct result.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOAP '13 June 20, 2013, Seattle, Washington, USA.

Copyright © 2013 ACM ISBN 978-1-4503-2201-0/13/06...\$15.00

We have implemented a generic framework for performing flow and context-sensitive interprocedural data flow analysis that does not require flow functions to be distributive. However, the flow functions must be monotonic and the lattice of data flow values must be finite. The framework uses value-based contexts and is an adaptation of the tabulation method of the functional approach and the modified call strings method.

Our implementation is agnostic to any analysis toolkit or intermediate representation as it is parameterized using generic types. Since our core classes are similar to the intra-procedural framework of Soot, it integrates with Soot's Jimple IR seamlessly.

We have instantiated our framework with a flow and context-sensitive points-to analysis in Soot, which enables the construction of call graphs that are far more precise than those constructed by Soot's SPARK engine.

The rest of the paper is organized as follows: Section 2 describes our method. Section 3 outlines the API of our implementation framework. Section 4 presents the results of call graph construction. Finally, Section 5 concludes the paper by describing the current status and future possibilities of our work.

2. Interprocedural Analysis Using Value Contexts

The tabulation method of the functional approach [11] and the modified call strings approach [5] both revolve around the same key idea: if two or more calls to a procedure p have the same the data flow value (say x) at the entry of p , then all of them will have an identical data flow value (say y) at the exit of p . The tabulation method uses this idea to enumerate flow functions in terms of pairs of input-output values (x, y) whereas the modified call strings method uses it to partition call strings based on input values, reducing the number of call strings significantly.

The two methods lead to an important conclusion: Using data flow values as contexts of analysis can avoid re-analysis of procedure bodies. We make this idea explicit by defining a *value context* $X = \langle \text{method}, \text{entryValue} \rangle$, where *entryValue* is the data flow value at the entry to a procedure *method*. Additionally, we define a mapping $\text{exitValue}(X)$ which gives the data flow value at the exit of *method*. As data flow analysis is an iterative process, this mapping may change over time (although it will follow a descending chain in the lattice). The new value is propagated to all callers of *method* if and when this mapping changes. With this arrangement, intraprocedural analysis can be performed for each value context independently, handling flow functions in the usual way; only procedure calls need special treatment.

Although the number of value contexts created per procedure is theoretically proportional to the size of the lattice in the worst-case, we have found that in practice the number of distinct data flow values reaching each procedure is often very small. This is especially true for heap-based analyses that use bounded abstractions, due to the locality of references in recursive paths. This claim is validated in Section 4, in which we present the results of a points-to analysis.

Algorithm

Figure 1 provides the overall algorithm. Line 1 declares three globals: a set of contexts that have been created, a transition table mapping a context and call site of a caller method to a target context at the called method and a work-list of context-parametrized control-flow graph nodes whose flow function has to be processed.

The procedure `INITCONTEXT` (lines 2-11) initializes a new context with a given method and entry value. The exit value is initialized to the $\top$ element. IN/OUT values at all nodes in the method body are also initialized to $\top$, with the exception of the method's entry node, whose IN value is initialized to the context's entry value. All nodes of this context are added to the work-list.

```

1: global contexts, transitions, worklist
2: procedure INITCONTEXT( $X$ )
3:   ADD(contexts,  $X$ )
4:   Set EXITVALUE( $X$ )  $\leftarrow \top$ 
5:   Let  $m \leftarrow \text{METHOD}(X)$ 
6:   for all nodes  $n$  in the body of  $m$  do
7:     ADD(worklist,  $\langle X, n \rangle$ )
8:     Set IN( $X, n$ )  $\leftarrow \top$  and OUT( $X, n$ )  $\leftarrow \top$ 
9:   end for
10:  Set IN( $X, \text{ENTRYNODE}(m)$ )  $\leftarrow \text{ENTRYVALUE}(X)$ 
11: end procedure
12: procedure DOANALYSIS
13:  INITCONTEXT( $\langle \text{main}, BI \rangle$ )
14:  while worklist is not empty do
15:    Let  $\langle X, n \rangle \leftarrow \text{REMOVENEXT}(\text{worklist})$ 
16:    if  $n$  is not the entry node then
17:      Set IN( $X, n$ )  $\leftarrow \top$ 
18:      for all predecessors  $p$  of  $n$  do
19:        Set IN( $X, n$ )  $\leftarrow \text{IN}(X, n) \sqcap \text{OUT}(X, p)$ 
20:      end for
21:    end if
22:    Let  $a \leftarrow \text{IN}(X, n)$ 
23:    if  $n$  contains a method call then
24:      Let  $m \leftarrow \text{TARGETMETHOD}(n)$ 
25:      Let  $x \leftarrow \text{CALLENTRYFLOWFUNCTION}(X, m, n, a)$ 
26:      Let  $X' \leftarrow \langle m, x \rangle$   $\triangleright x$  is the entry value at  $m$ 
27:      Add an edge  $\langle X, n \rangle \rightarrow X'$  to transitions
28:      if  $X' \in \text{contexts}$  then
29:        Let  $y \leftarrow \text{EXITVALUE}(X')$ 
30:        Let  $b_1 \leftarrow \text{CALLEXITFLOWFUNCTION}(X, m, n, y)$ 
31:        Let  $b_2 \leftarrow \text{CALLLOCALFLOWFUNCTION}(X, n, a)$ 
32:        Set OUT( $X, n$ )  $\leftarrow b_1 \sqcap b_2$ 
33:      else
34:        INITCONTEXT( $X'$ )
35:      end if
36:    else
37:      Set OUT( $X, n$ )  $\leftarrow \text{NORMALFLOWFUNCTION}(X, n, a)$ 
38:    end if
39:    if OUT( $X, n$ ) has changed then
40:      for all successors  $s$  of  $n$  do
41:        ADD(worklist,  $\langle X, s \rangle$ )
42:      end for
43:    end if
44:    if  $n$  is the exit node then
45:      Set EXITVALUE( $X$ )  $\leftarrow \text{OUT}(X, n)$ 
46:      for all edges  $\langle X', c \rangle \rightarrow X$  in transitions do
47:        ADD(worklist,  $\langle X', c \rangle$ )
48:      end for
49:    end if
50:  end while
51: end procedure

```

Figure 1. Algorithm for performing inter-procedural analysis using value contexts.

The `DOANALYSIS` procedure (lines 12-51) first creates a value context for the `main` method with some boundary information (BI). Then, data flow analysis is performed using the traditional work-list method, but distinguishing between nodes of different contexts.

A node is removed from the work-list and its IN value is set to the meet of the OUT values of its predecessors (lines 16-21). For nodes without a method call, the OUT value is computed using the normal flow function (line 37). For call nodes, parameter passing is handled by a call-entry flow function that takes as input the IN value at the node, and the result of which is used as the entry value at the

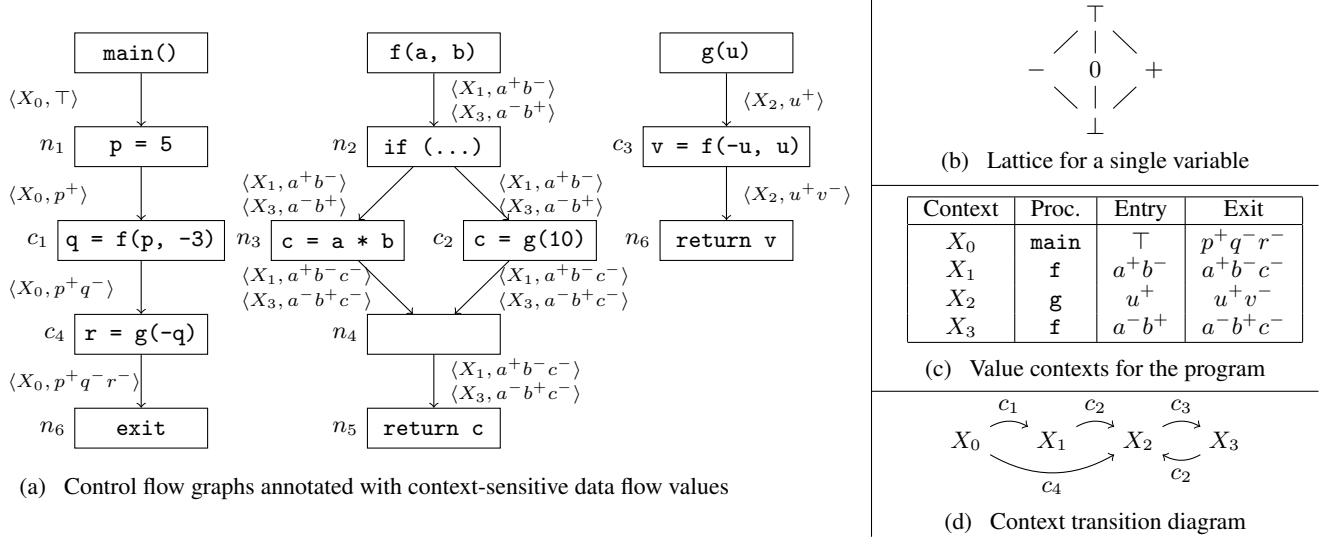


Figure 2. A motivating example of a non-distributive sign-analysis performed on a program with mutually recursive procedures.

callee context (lines 24-26). The transition from caller context and call-site to callee context is also recorded (line 27). If a context with the target method and computed entry value has not been previously created, then it is initialized now (line 34). Otherwise, the exit value of the target context is used as the input to a call-exit flow function, to handle returned values. A separate call-local flow function takes as input the IN value at the call node, and propagates information about local variables. The results of these two functions are merged into the OUT value of the call node (lines 29-32).

Once a node is processed, its successors are added to the work-list if its OUT value has changed in this iteration (lines 39-43). If the node is the exit of its procedure (lines 44-49), then the exit value of its context is set and all its callers are re-added to the work-list.

The termination of the algorithm follows from the monotonicity of flow functions and the finiteness of the lattice (which bounds the descending chain as well as the number of value contexts).

The algorithm can easily be extended to handle multiple entry/exit points per procedure as well as virtual method calls by merging data flow values across these multiple paths. It can also be easily adapted for backward data flow analyses.

Example

Consider the program in Figure 2 (a), for which we wish to perform a simplified *sign analysis*, to determine whether a scalar local variable is negative, positive or zero. The call from `main` to `f` at c_1 will only return when the mutual recursion of `f` and `g` terminates, which happens along the program path $n_2n_3n_4n_5$. Notice that the arguments to `f` at call-site c_3 are always of opposite signs, causing the value of variable c to be negative after every execution of n_3 in this context. Thus, `f` and hence `g` always returns a negative value.

To compute this result using the algorithm described above, we use data flow values that are elements of the lattice in Figure 2 (b), where $\top$ indicates an uninitialized variable and $\perp$ is the conservative assumption. We use superscripts to map variables to a sign or $\perp$, and omit uninitialized variables.

At the start of the program no variables are initialized and hence the analysis starts with the initial value context $X_0 = \langle \text{main}, \top \rangle$. For work-list removal, we will use lexicographical ordering of contexts (newer first) before nodes (reverse post-order).

The flow function of $\langle X_0, n_1 \rangle$ is processed first, which makes p positive (written as p^+). The next node picked from the work-list

is c_1 , whose call-entry flow function passes one positive and one negative argument to parameters a and b of procedure `f` respectively. Thus, a new value context $X_1 = \langle f, a^+b^- \rangle$ is created and the transition $\langle X_0, c_1 \rangle \rightarrow X_1$ is recorded.

Analysis proceeds by processing $\langle X_1, n_2 \rangle$ and then $\langle X_1, c_2 \rangle$, which creates a new value context $X_2 = \langle g, u^+ \rangle$ due to the positive argument. The transition $\langle X_1, c_2 \rangle \rightarrow X_2$ is recorded. When $\langle X_2, c_3 \rangle$ is processed, the arguments to `f` are found to be negative and positive respectively, creating a new value context $X_3 = \langle f, a^-b^+ \rangle$ and a transition $\langle X_2, c_3 \rangle \rightarrow X_3$.

The work-list now picks nodes of context X_3 , and when $\langle X_3, c_2 \rangle$ is processed, the entry value at `g` is u^+ , for which a value context already exists – namely X_2 . The transition $\langle X_3, c_2 \rangle \rightarrow X_2$ is recorded. The exit value of X_2 is at the moment $\top$ because its exit node has not been processed. Hence, the call-exit flow function determines the returned value to be uninitialized and the OUT of $\langle X_3, c_2 \rangle$ gets the value a^-b^+ . The next node to be processed is $\langle X_3, n_3 \rangle$, whose flow function computes the sign of c to be negative as it is the product of a negative and positive value. The IN value at $\langle X_3, n_4 \rangle$ is $(a^-b^+c^- \sqcap a^-b^+) = a^-b^+c^-$. Thus, the sign of the returned variable c is found to be negative. As n_4 is the exit node of procedure `f`, the callers of X_3 are looked up in the transition table and added to the work-list.

The only caller $\langle X_2, c_3 \rangle$ is now re-processed, this time resulting in a hit for an existing target context X_3 . The exit value of X_3 being $a^-b^+c^-$, the returned variable v gets a negative sign, which propagates to the exit node n_6 . The callers of X_2 , namely $\langle X_1, c_2 \rangle$ and $\langle X_3, c_2 \rangle$, are re-added to the work-list.

$\langle X_3, c_2 \rangle$ is processed next, and this time the correct exit value of target context X_2 , which is u^+v^- , is used and the OUT of $\langle X_3, c_2 \rangle$ is set to $a^-b^+c^-$. When its successor $\langle X_3, n_4 \rangle$ is subsequently processed, the OUT value does not change and hence no more nodes of X_3 are added to the work-list. Analysis continues with nodes of X_1 on the work-list, such as $\langle X_1, c_2 \rangle$ and $\langle X_1, n_3 \rangle$. The sign of c is determined to be negative and this propagates to the end of the procedure. When exit node $\langle X_1, n_5 \rangle$ is processed, the caller of X_1 , namely $\langle X_0, c_1 \rangle$, is re-added to the work-list. Now, when this node is processed, q is found to be negative.

Value-based contexts are not only useful in terminating the analysis of recursive procedures, as shown above, but also as a simple *cache table* for distinct call sites. For example, when $\langle X_0, c_4 \rangle$ is

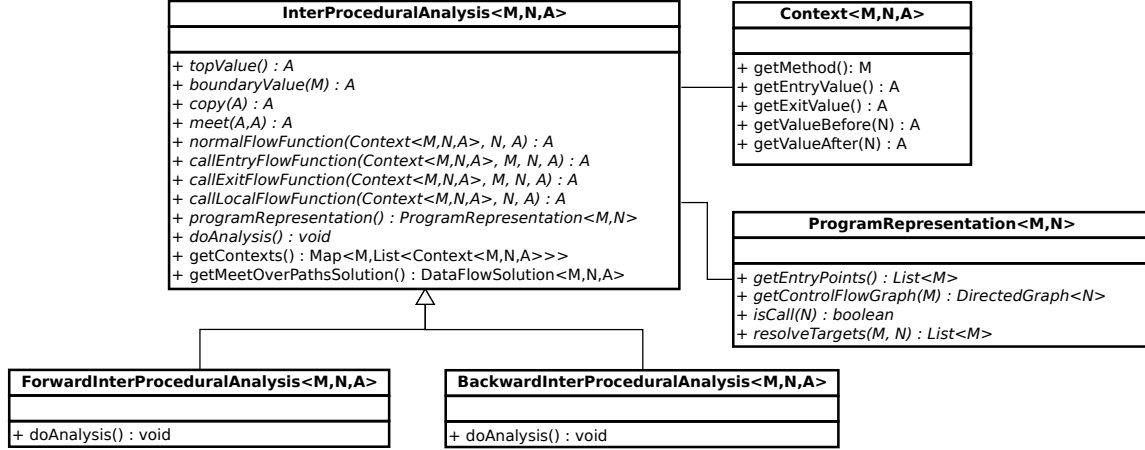


Figure 3. The class diagram of our generic interprocedural analysis framework.

processed, the positive argument results in a hit for X_2 , and thus its exit value is simply re-used to determine that r is negative. Figure 2 (b) lists the value contexts for the program and Figure 2 (c) shows the transitions between contexts at call-sites.

A context-insensitive analysis would have merged signs of a and b across all calls to f and would have resulted in a $\perp$ value for the signs of c , v , q and r . Our context-sensitive method ensures a precise data flow solution even in the presence of recursion.

Notice that the flow function for n_3 is non-distributive since $f_{n_3}(a^+b^-) \sqcap f_{n_3}(a^-b^+) = a^+b^-c^- \sqcap a^-b^+c^- = a^\perp b^\perp c^-$ but $f_{n_3}(a^+b^- \sqcap a^-b^+) = f_{n_3}(a^\perp b^\perp) = a^\perp b^\perp c^\perp$. Hence this problem does not fit in the IFDS/IDE framework, but such flow functions do not pose a problem to our algorithm.

3. Implementation Framework

The implementation framework consists of a handful of core classes as shown in Figure 3. The use of generic types makes the framework agnostic to any particular toolkit or IR. The classes are parameterized by three types: M represents the type of a method, N represents a node in the control flow graph and A is the type of data flow value used by the client analysis. The framework can be naturally instantiated for Soot using the type parameters `SootMethod` and `Unit` for M and N respectively.

Users would extend `ForwardInterProceduralAnalysis` or `BackwardInterProceduralAnalysis`, which are subclasses of an abstract class `InterProceduralAnalysis`. The abstract methods `topValue`, `boundaryValue`, `copy` and `meet` provide a hook for client analyses to express initial lattice values and basic operations on them. The major functionality of the client analysis would be present in the `*FlowFunction` methods, whose roles were explained in Section 2. Additionally, clients are expected to provide a `ProgramRepresentation` object, which specifies program entry points (for which boundary values are to be defined) and resolves virtual calls. Our framework ships with default program representations for Soot’s Jimple IR. The launch point of the analysis is the `doAnalysis` method, which is implemented as per the algorithm from Figure 1 in the directional sub-classes.

The `Context` class encapsulates information about a value context. Every context is associated with a method, an entry value and an exit value, each of which can be retrieved using the corresponding *getter* methods. The `getValueBefore` and `getValueAfter` methods return data flow values for a context just before and after a node respectively. This is the recommended way for accessing the

results of the analysis in a context-sensitive manner. A mapping of methods to a list of all its contexts is available through the `getContexts` method of the `InterProceduralAnalysis` class. Alternatively, `getMeetOverValidPathsSolution` can be used to obtain a solution that is computed by merging data flow results across all contexts of each method. The `DataFlowSolution` class (not shown in the figure) simply provides `getValueBefore` and `getValueAfter` methods to access the resulting solution.

4. The Role of Call Graphs

We initially developed this framework in order to implement heap reference analysis [6] using Soot, because it could not be encoded as an IFDS/IDE problem. However, even with our general framework, performing whole-program analysis turned out to be infeasible due to a large number of interprocedural paths arising from conservative assumptions for targets of virtual calls.

The SPARK engine [7] in Soot uses a flow and context insensitive pointer analysis on the whole program to build the call graph, thus making conservative assumptions for the targets of virtual calls in methods that are commonly used such as those in the Java library. For example, it is not uncommon to find call sites in library methods with 5 or more targets, most of which will not be traversed in a given context. Some call sites can even be found with more than 250 targets! This is common with calls to virtual methods defined in `java.lang.Object`, such as `hashCode()` or `equals()`.

When performing whole-program data flow analysis, the use of an imprecise call graph hampers both efficiency, due to an exponential blow-up of spurious paths, and precision, due to the meet over paths that are actually interprocedurally invalid, thereby diminishing the gains from context-sensitivity.

Soot provides a context-sensitive call graph builder called `PADDLE` [8], but this framework can only perform k -limited call-site or object-sensitive analysis, and that too in a flow-insensitive manner. We were unable to use `PADDLE` with our framework directly because at the moment it not clear to us how the k -suffix contexts of `PADDLE` would map to our value-contexts.

Call Graph Construction using Points-To Analysis

We have implemented a flow and context-sensitive points-to analysis using our interprocedural framework to build a call graph on-the-fly. This analysis is both a demonstration of the use of our framework as well as a proposed solution for better call graphs intended for use by other interprocedural analyses.

Benchmark		Time	Methods (M)		Contexts (X)		X/M		Clean	
			Total	App.	Total	App.	Total	App.	Total	App.
SPEC JVM98	compress	1.15s	367	54	1,550	70	4.22	1.30	50	47
	jess	140.8s	690	328	17,280	9,397	25.04	28.65	34	30
	db	2.19s	420	56	2,456	159	5.85	2.84	62	46
	mpegaudio	4.51s	565	245	2,397	705	4.24	2.88	50	47
	jack	89.33s	721	288	7,534	2,548	10.45	8.85	273	270
DaCapo 2006	antlr	697.4s	1,406	798	30,043	21,599	21.37	27.07	769	727
	chart	242.3s	1,799	598	16,880	4,326	9.38	7.23	458	423

Table 1. Results of points-to analysis using our framework. “App.” refers to data for application classes only.

The data flow value used in our analysis is a points-to graph in which nodes are allocation sites of objects. We maintain two types of edges: $x \rightarrow m$ indicates that the root variable x may point to objects allocated at site m , and $m.f \rightarrow n$ indicates that objects allocated at site m may reference objects allocated at site n along the field f . Flow functions add or remove edges when processing assignment statements involving reference variables. Nodes that become unreachable from root variables are removed. Type consistency is maintained by propagating only valid casts.

The points-to graphs at each statement only maintain objects reachable from variables that are local to the method containing the statement. At call statements, we simulate assignment of arguments to locals of the called method, as well as the assignment of returned values to a local of the caller method. For static fields (and objects reachable from them) we maintain a global flow-insensitive points-to graph. For statements involving static loads/stores we operate on a temporary union of local and global graphs. The call graph is constructed on-the-fly by resolving virtual method targets using type information of receiver objects.

Points-to information cannot be precise for objects returned by native methods, and for objects shared between multiple threads (as our analysis is flow-sensitive). Thus, we introduce the concept of a *summary node*, which represents statically unpredictable points-to information and is denoted by the symbol $\perp$. For soundness, we must conservatively propagate this effect to variables and fields that involve assignments to summary nodes. The rules for summarization along different types of assignment statements are as follows:

Statement	Rule used in the flow function
$x = y$	If $y \rightarrow \perp$, then set $x \rightarrow \perp$
$x.f = y$	If $y \rightarrow \perp$, then $\forall o : x \rightarrow o$, set $o.f \rightarrow \perp$
$x = y.f$	If $y \rightarrow \perp$ or $\exists o : y \rightarrow o$ and $o.f \rightarrow \perp$, then set $x \rightarrow \perp$
$x = p(a_1, a_2, \dots)$	If p is unknown, then set $x \rightarrow \perp$, and $\forall o : a_i \rightarrow o, \forall f \in \text{fields}(o)$ set $o.f \rightarrow \perp$

The last rule is drastically conservative; for soundness we must assume that a call to an unknown procedure may modify the fields of arguments in any manner, and return any object. An important discussion would be on what constitutes an *unknown* procedure. Native methods primarily fall into this category. In addition, if p is a virtual method invoked on a reference variable y and if $y \rightarrow \perp$, then we cannot determine precisely what the target for p will be. Hence, we consider this call site as a *default* site, and do not enter the procedure, assuming worst-case behaviour for its arguments and returned values. A client analysis using the resulting call graph with our framework can choose to do one of two things when encountering a *default* call site: (1) assume worst case behaviour for its arguments (eg. in liveness analysis, assume that all arguments and objects reachable from them are live) and carry on to the next statement, or (2) fall-back onto Soot’s default call graph and follow the targets it gives.

A related approach partitions a call graph into calls from application classes and library classes [2]. Our call graph is partitioned into call sites that we can precisely resolve to one or more valid targets, and those that cannot due to statically unpredictable factors.

Experimental Results

Table 1 lists the results of points-to analysis performed on seven benchmarks. The experiments were carried out on an Intel Core i7-960 with 19.6 GB of RAM running Ubuntu 12.04 (64-bit) and JDK version 1.6.0_27. Our single-threaded analysis used only one core.

The first two columns contain the names of the benchmarks; five of which are the single-threaded programs from the SPEC JVM98 suite [1], while the last two are from the DaCapo suite [3] version 2006-10-MR2. The third column contains the time required to perform our analysis, which ranged from a few seconds to a few minutes. The fourth and fifth columns contain the number of methods analyzed (total and application methods respectively). The next two columns contain the number of value-contexts created, with the average number of contexts per method in the subsequent two columns. It can be seen that the number of distinct data flow values reaching a method is not very large in practice. As our analysis ignores paths with method invocations on null pointers, it was inappropriate for other benchmarks in the DaCapo suite when using stub classes to simulate the suite’s reflective boot process.

The use of *default* sites in our call graph has two consequences: (1) the total number of analyzed methods may be less than the total number of reachable methods and (2) methods reachable from *default* call sites (computed using SPARK’s call graph) cannot be soundly optimized by a client analysis that jumps over these sites. The last column lists the number of *clean* methods which are not reachable from *default* sites and hence can be soundly optimized. In all but two cases, the majority of application methods are clean.

In order to highlight the benefits of using the resulting call graph, just listing the number of edges or call-sites alone is not appropriate, as our call graph is context-sensitive. We have thus computed the number distinct paths in the call graph, starting from the entry point, which are listed in Table 2. As the total number of call graph paths is possibly infinite (due to recursion), we have counted paths of a fixed length length k , for $1 \leq k \leq 10$. For each benchmark, we have counted these paths using call graphs constructed by our Flow and Context-sensitive Pointer Analysis (FCPA) as well as SPARK, and noted the difference as percentage savings ($\Delta\%$) from using our context-sensitive call graph. The option *implicit-entry* was set to *false* for SPARK.

The savings can be clearly observed for $k > 5$. For $k = 10$, SPARK’s call graph contains more than 96% spurious paths for three of the benchmarks, and 62-92% for the remaining. The gap only widens for larger values of k (for which the number of paths was too large to compute in some cases).

Client analyses using our interprocedural framework can be configured to use our context-sensitive call graphs which avoid these spurious paths, hence enabling efficient and precise solutions.

Depth $k =$		1	2	3	4	5	6	7	8	9	10
compress	FCPA	2	5	7	20	55	263	614	2,225	21,138	202,071
	SPARK	2	5	9	22	57	273	1,237	23,426	545,836	12,052,089
	$\Delta\%$	0	0	22.2	9.09	3.51	3.66	50.36	90.50	96.13	98.32
jess	FCPA	2	5	7	30	127	470	4,932	75,112	970,044	15,052,927
	SPARK	2	5	9	32	149	924	24,224	367,690	8,591,000	196,801,775
	$\Delta\%$	0	0	22.2	6.25	14.77	49.13	79.64	79.57	88.71	92.35
db	FCPA	2	5	11	46	258	1,791	21,426	215,465	2,687,625	42,842,761
	SPARK	2	5	13	48	443	4,726	71,907	860,851	13,231,026	245,964,733
	$\Delta\%$	0	0	15.4	4.17	41.76	62.10	70.20	74.97	79.69	82.58
mpegaudio	FCPA	2	14	42	113	804	11,286	129,807	1,772,945	27,959,747	496,420,128
	SPARK	2	16	46	118	834	15,844	250,096	4,453,608	87,096,135	1,811,902,298
	$\Delta\%$	0	12	8.7	4.24	3.60	28.77	48.10	60.19	67.90	72.60
jack	FCPA	2	18	106	1,560	22,652	235,948	2,897,687	45,480,593	835,791,756	17,285,586,592
	SPARK	2	18	106	1,577	27,201	356,867	5,583,858	104,211,833	2,136,873,586	46,356,206,503
	$\Delta\%$	0	0	0	1.08	16.72	33.88	48.11	56.36	60.89	62.71
antlr	FCPA	6	24	202	560	1,651	4,669	18,953	110,228	975,090	11,935,918
	SPARK	6	24	206	569	1,669	9,337	107,012	1,669,247	27,670,645	468,973,725
	$\Delta\%$	0	0	1.9	1.58	1.08	49.99	82.29	93.40	96.48	97.45
chart	FCPA	6	24	217	696	2,109	9,778	45,010	517,682	7,796,424	164,476,462
	SPARK	6	24	219	714	2,199	20,171	306,396	7,676,266	192,839,216	4,996,310,985
	$\Delta\%$	0	0	0.9	2.52	4.09	51.52	85.31	93.26	95.96	96.71

Table 2. Number of k -length call graph paths for various benchmarks using SPARK and FCPA (Flow and Context-sensitive Pointer Analysis).

5. Conclusion and Future Work

We have presented a framework for performing value-based context-sensitive inter-procedural analysis in Soot. This framework does not require distributivity of flow functions and is thus applicable to a large class of analyses including those that cannot be encoded as IFDS/IDE problems. Another advantage of our method is the context-sensitive nature of the resulting data flow solution, which can be useful in dynamic optimizations.

In order to deal with the difficulties in whole-program analysis performed over an imprecise call graph, we constructed call graphs on-the-fly while performing a flow and context-sensitive points-to analysis. This analysis also demonstrated a sample use of our framework and showed that it was practical to use data flow values as contexts because the number of distinct data flow values reaching each method is often very small.

The interprocedural framework has been released and is available at <https://github.com/rohanpadhye/vasco>. However, our points-to analysis implementation is experimental and makes heavy use of HashMaps and HashSets, thus running out of memory for very large programs. We would like to improve this implementation by using bit-vectors or maybe even BDDs for compact representation of points-to sets.

The precision of our call graphs is still limited in the presence of *default* sites, which are prominent due to the liberal use of *summary* nodes in the points-to graphs. We would like to reduce the number of *summary* nodes by simulating some commonly used native methods in the Java library, and also by preparing a summary of initialized static fields of library classes. Our hope is that these extensions would enable our call graph to be fully complete, thereby enabling users to precisely perform whole-program analysis and optimization for all application methods in an efficient manner. We believe that improving the precision of program analysis actually helps to improve its efficiency, rather than hamper it.

Acknowledgments

We would like to thank the anonymous reviewers for suggesting the inclusion of a non-distributive example and the separation of call/return flow functions. These changes improved the paper.

References

- [1] <http://www.spec.org/jvm98>. Accessed: April 3, 2013.
- [2] K. Ali and O. Lhoták. Application-only call graph construction. In *Proceedings of the 26th European conference on Object-Oriented Programming, ECOOP'12*, 2012.
- [3] S. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2006.
- [4] E. Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP '12*, 2012.
- [5] U. P. Khedker and B. Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: resurrecting the classical call strings method. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction, CC'08/ETAPS'08*, 2008.
- [6] U. P. Khedker, A. Sanyal, and A. Karkare. Heap reference analysis using access graphs. *ACM Trans. Program. Lang. Syst.*, 30(1), Nov. 2007.
- [7] O. Lhoták and L. Hendren. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th international conference on Compiler construction, CC'03*, 2003.
- [8] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1), Oct. 2008.
- [9] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '95*, 1995.
- [10] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1-2), Oct. 1996.
- [11] M. Sharir and A. Pnueli. Two approaches to interprocedural dataflow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., 1981.
- [12] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, CASCON '99*, 1999.