Fast and precise points-to analysis

Jonas Lundberg, Tobias Gutzmann *, Marcus Edvinsson, Welf Löwe

Software Technology Group, School of Mathematics and System Engineering, Växjö University, 351 95 Växjö, Sweden

A R T I C L E   I N F O

Article history:
Available online 6 May 2009

Keywords:
Points-to analysis
Dataflow analysis
Escape analysis

A B S T R A C T

Many software engineering applications require points-to analysis. These client applications range from optimizing compilers to integrated program development environments (IDEs) and from testing environments to reverse-engineering tools. Moreover, software engineering applications used in an edit-compile cycle need points-to analysis to be fast and precise.

In this article, we present a new context- and flow-sensitive approach to points-to analysis where calling contexts are distinguished by the points-to sets analyzed for their call target expressions. Compared to other well-known context-sensitive techniques it is faster in practice, on average, twice as fast as the call string approach and by an order of magnitude faster than the object-sensitive technique. In fact, it shows to be only marginally slower than a context-insensitive baseline analysis. At the same time, it provides higher precision than the call string technique and is similar in precision to the object-sensitive technique. We confirm these statements with experiments using a number of abstract precision metrics and a concrete client application: escape analysis.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Points-to analysis is a static program analysis that extracts reference information from a given input program, e.g., possible targets of a call and possible objects referenced by a field. This reference information is an essential input to many types of client applications in optimizing compilers and software engineering tools.

Examples of such client applications are: metrics analyses computing coupling and cohesion between objects [1,2] and architectural recovery by class clustering proposing groupings of classes, either based on coupling and cohesion or directly on reference information [3,4]. Source code browsers compute forward and backward slices [5] of a program point which, in turn, requires reference information. In software testing, class dependencies determine the test order [6–8]. Reverse engineering of UML interaction diagrams requires very precise reference information in order to be useful [9]. Finally, static design pattern detection needs to identify the interaction among participating classes and object instances in order to exclude false positives [10].

In real life, where the program to be analyzed may contain hundreds or even thousands of classes, there is a tradeoff between the analysis precision and the analysis time and memory costs. This situation is problematic for user-interactive client analyses, where in many cases response time is essential and a delay of several minutes unacceptable. Especially for these client applications, scalably fast and precise points-to analysis is very much desirable.

The basis for many points-to analysis approaches, and program analysis in general, is the theory of monotone dataflow frameworks [11,12]. A program is represented by a program graph; its nodes correspond to program points, its edges to control and data dependencies between them. The analysis iteratively computes values for each node by merging values from predecessor nodes and by applying transfer functions representing the abstract program behavior at these nodes.

A context-insensitive program analysis, analysis values of different call sites are propagated to the same method and get mixed there. The analysis value is then the merger of all calls targeting that method. A context-sensitive analysis addresses this source of imprecision by distinguishing between different calling contexts of a method. It analyzes a method separately for each calling context [13]. Context-sensitivity will therefore, in general, give a more precise analysis. The drawbacks are the increased memory cost that comes with maintaining a large number of contexts and their analysis values, and the increased analysis time required to reach a fixed point.

Context-sensitive approaches use a finite abstraction of the top sequence of the call stack possibly occurring at each call site in order to separate different call contexts. The two traditional approaches to define a context are referred to as the call string approach and the functional approach [14]. The call string approach defines a context by the first $k$ callers, i.e., return addresses on the
call stack top [15], referred to as the family of k-CFA (Control Flow Analysis). The functional approach uses some abstractions of the call site’s actual parameters to distinguish different contexts [14, 16]. Both the call string and the functional approaches were evaluated and put into a common framework by Grove et al. [16].

A functional approach designed for object-oriented languages is referred to as object-sensitivity [17, 18]. It distinguishes contexts by separately analyzing the targeted method for each abstract object in the implicit this-parameter. Similarly to k-CFA, a family of k-object-sensitive algorithms distinguishing contexts by the top k abstract target objects on the call stack can be defined. In the two papers [17, 18], a simplified version of 1-object-sensitivity was evaluated. Here, only method parameters and return values are treated context-sensitively. The authors report, compared to 1-CFA, increased precision of side-effect analysis and, to a lesser degree, call graph construction. Both approaches show similar costs in time and memory. These results generalize to variants where k > 1, which, however, are very costly in terms of memory and provide only a small increase in precision [19].

The contributions of this article are the following:

- We present a new functional approach to points-to analysis denoted this-sensitivity.
- We experimentally evaluate this-sensitivity by comparing it with two well-known context-sensitive approaches (1-CFA and a complete version of 1-object-sensitivity – complete in the sense that it does not merge analysis values of different contexts). Our measurements show that this sensitivity (i) is twice as fast as 1-CFA and an order of magnitude faster than 1-object-sensitivity, (ii) requires less memory than 1-object-sensitivity, (iii) is more precise than 1-CFA, and (iv) is almost as precise as 1-object-sensitivity.
- We evaluate the precision by using one concrete client application (escape analysis) and two abstract precision metrics suited for different granularities and aspects of precision corresponding to two different types of client applications.

The remainder of this article is as follows: In Section 2, we outline Points-to SSA, our SSA-based program representation where each method is represented by a sparse method graph. Points-to SSA has been presented elsewhere [20], we include a brief version of this material for the understandability and completeness of this article. In Section 3, we present our context-sensitive simulated execution analysis technique where the analysis of a method (in a specific context) is interrupted when a call occurs, and later resumed when the analysis of the called method is completed. Then, our new context-sensitive points-to-analysis is presented in Section 4. In Section 5, we present our precision metrics, experimental setup, and results. Finally, in Section 6, we discuss related work and conclude this article in Section 7.

2. Analysis values and program representation

In this section, we introduce the representation of analysis values, which are sets of abstract objects and a heap-memory abstraction, and our program representation called Points-to SSA. They are then used in the actual analysis algorithm, which is described in Section 3.

2.1. Analysis values

In points-to analysis, we need to represent references to abstract objects and an abstraction of the heap-memory.

An abstract object o is an analysis abstraction that represents one or more run-time objects. The mapping from run-time to abstract objects is called a name schema. In this article, we use the following name schema: each syntactic creation point s corresponds to a unique abstract object o. Thus, the set of all allocation sites in a program defines a finite set of abstract objects denoted O, and every abstract object o ∈ O can be seen as an analysis abstraction representing all run-time objects created at the corresponding allocation site s in any execution of the analyzed program.

In the analysis, reference variables will in general hold references to more than one abstract object. Hence, we assume that each points-to value ν in the analysis of a program is an element in the points-to value lattice Lν = {V, ⊃, ∩, T, ⊥} where V = 2O is the power set of O, T ⊂ O = ∅, and ⊃, ∩, and ⊥ are the set operations (union) and (intersection). The height of the points-to value lattice is h1 = |O|. We use the notation pt(a) to refer to the points-to value that is referenced by the expression a.

Each abstract object o ∈ O has a unique set of object fields [o, f] ∈ OF where f ∈ F is a unique identifier of a field (capturing references). Each object field [o, f] is in turn associated with a memory slot ([o, f], ν) where ν is a points-to value. A memory slot represents the abstract object references stored in object field [o, f].

The abstraction of the heap-memory associated with an analyzed program, referred to as abstract memory, Mem, is defined as the set of all memory slots ([o, f], ν). In our approach, we use a single global memory configuration. Our reason for introducing an abstract memory is not only to mimic the run-time behavior; it is a necessary construct to handle field store and load operations and the transport of abstract objects from one method to another that follows as a result of these operations. We think of the abstract memory as a mapping from object fields to points-to values. The memory is therefore equipped with two operations

\[ \text{Mem.get}(OF) \rightarrow V \quad \text{and} \quad \text{Mem.addTo}(OF, V) \]

with the interpretation of reading the points-to value stored in an object field [o, f] ∈ OF, and merging the points-to value ν ∈ V with the points-to value already stored in an object field [o, f] ∈ OF, respectively. Note that we never override previously stored object field values in memory store operations, i.e., we never execute strong updates. Instead, we merge the new value with the old one using the points-to value lattice’s join operation, i.e., we perform weak updates.

The abstract memory is updated as a side effect of the analysis. In order to quickly determine the fixed point, we use memory sizes indicating whether or not the memory has changed. In what follows, we refer to the size of the abstract memory as a memory size x ∈ X = [0, hm] where hm is the maximum memory size. It corresponds to the case where all object fields contain all abstract objects, hence, h_m = |OF| = |O|.

In order to apply the theory of monotone dataflow frameworks to memory size values as well, we introduce a lattice L_X referred to as the memory size lattice. The memory size lattice L_X is a single ascending chain of integers, i.e., \( L_X = \{X, \sqcup, \sqcap, T, \bot\} \) where \( X = \{0, 1, 2, \ldots, h_m\} \), \( T = h_m \), \( \bot = 0 \), \( X_1 \cup X_2 = \max(X_1, X_2) \), and \( X_1 \cap X_2 = \min(X_1, X_2) \). The height of \( L_X \) is \( h_m \).

2.2. Points-to SSA

Points-to SSA is our graph-based program representation. In Points-to SSA, local variables are resolved to dataflow edges connecting operations (nodes) that define variables to operations (nodes) that use these variables. As a result, every def-use relation via local variables is explicitly represented as an edge between the defining and using operations. Join-points in the control flow where several definitions may apply are modeled with special op-nodes using possible definitions valid in the different branches and introducing new definitions.
Fig. 1 shows a simple “Linked List” implementation (class L) and the corresponding Points-to SSA graphs. Each method is represented by a graph and each node in the graph represents an operation in the method. We have for example Entry and Exit nodes representing method entry/exit points, and Store and Load nodes representing field write/read operations. The ports at the top of a node represent operation input values (e.g., memory size \( x \), target address values \( a \), and the values \( v \) to store in the Store nodes) and the ports at the bottom represent operation results (e.g., a new memory size \( x \) in the Store nodes). Edges connecting node ports represent the flow of values from defining nodes (operation results) to using nodes (operation input values). More details regarding these notations will be presented later on.

Notice that the constructor L.init starts by calling its super constructor Object.init and that object creation, in L.append, is done in two steps: we first allocate an object of class L and then call the constructor L.init. \( \phi \)-nodes are used in L.append to merge the memory size values from the two selective branches, and in L.putAt as the loop head of the iteration.

A Points-to SSA method graph can be seen as an abstraction of a method’s semantics, an SSA graph representation specially designed for points-to analysis. It is an abstraction since we have removed all operations not directly related to reference computations, e.g., operations related to primitive types. Moreover, we abstracted from the semantics of the remaining operations by giving them an abstract analysis semantics.

Another feature of Points-to SSA is the use of memory edges to explicitly model (direct, indirect, and anti-) dependencies between different memory operations. An operation that may change the memory defines a new memory size value, and operations that may access this updated memory have an abstract analysis semantics.

The reference-related semantics of different language constructs (e.g., calls and field accesses) are described by a set of operation node types. Each node type \( n \) has a number of in-ports \( \text{in}(n) = [\text{in}1(n), \ldots, \text{in}k(n)] \) and a number of out-ports \( \text{out}(n) = [\text{out}1(n), \ldots, \text{out}k(n)] \). The in-ports represent input values to the operation in question, whereas the out-ports represent the results produced by the operation. All ports have a fixed type \( V \) or \( X \) and a current analysis value of that type \( v \in V \) or \( x \in X \).

An edge \( e = \text{out}(\text{src}) \rightarrow \text{in}(\text{tg}) \) connects an out-port of a node \( \text{src} \) with an in-port of a node \( \text{tg} \). An edge may only connect an-ports and in-ports of the same type. An out-port \( \text{out}(n) \) may be connected to one or more outgoing edges. An in-port \( \text{in}(n) \) is always connected to a single incoming edge. The last property reflects our underlying SSA approach – each value has one, and only one, definition.

Certain node types have attributes that refer to node specific, static information. For example, each Alloc \( C \) node is decorated with a class identifier \( C \) that identifies the class of the object to be created.

Finally, each type of node is associated with a unique analysis semantics (or transfer function) which can be seen as a mapping from in-ports to out-ports that may have a side-effect on the memory. As an example, Algorithm 1 shows the analysis semantics for the Store \( L \) node, which abstracts the actual semantics of a field write statement \( a.f = v \).

Algorithm 1. Store \( L \): \( X_{\text{in}}, a, v \rightarrow X_{\text{out}} \)

For each abstract object \( o \) in the address reference \( a \), we look up the points-to value previously stored in object field \( o.f \). If the new value to be stored changes the memory (i.e., if \( v \not\subseteq \text{prev} \)), we merge \( v \) with the previous value and save the result. Notice also that we compute a new memory out-port value (a new memory size) if the memory has been changed during this operation.
3. Context-sensitive simulated execution

Our dataflow analysis technique, called simulated execution, is an abstract interpretation of the program based on the abstract analysis and program representation discussed in the previous section. It simulates the actual execution of a program where the analysis of a method is interrupted when a call occurs, and later resumed when the analysis of the called method was completed.

The simulated execution approach can be seen as a recursive interaction between the analysis of an individual Points-to SSA method graph and the transfer function associated with monomorphic calls, which handle the transition from one method to another. Polymorphic calls are handled as selections over possible target methods \(m\), which are then processed as a sequence of monomorphic calls targeting \(m\). The context-sensitive approach described here is a modification of the context-insensitive approach described in previous work [20]. The most noticeable difference is that we associate each monomorphic call targeting a method \(m\) with a number of contexts, and process \(m\) separately for each such context.

In Section 3.1, we describe the processing of individual method graphs. In Section 3.2, we describe the analysis of calls in a context. In Section 3.3, we present our context-sensitive handling of monomorphic calls. Our approach to intra- and inter-procedural flow-sensitivity is discussed in Section 3.4.

3.1. Method graph processing

For each method graph, we have a pre-computed node order that is determined by the data and memory dependencies between the nodes. We compute a topological sorting for forward edges. To order the nodes in loops, we use a so-called interval analysis [21,22] where we identify inner and outer loops and their loop heads (always \(o\)-nodes).

The method processing starts in the method entry node, follows the node ordering, and iterates over loops until a fixed point is reached. Inner loops are stabilized before their outer loops. Consequences of this approach are: (1) All nodes in a method graph \(g_m\) are analyzed at least once every time method \(m\) is analyzed. (2) All nodes, except the loop head \(o\)-nodes, have all their predecessor nodes updated before they are analyzed themselves. (3) The order in which the nodes are analyzed respects all control and data dependencies and is therefore an abstraction of the control-flow of an actual execution. The final point is a crucial step to assure flow-sensitivity in the SSA-based simulated execution technique.

The above properties of analyzing single method graphs is taken into consideration by processMethod as given in Algorithm 2. It should only be considered as a rough outline of the approach actually implemented. The idea is simple: We start by initializing the method entry node with the method input to be used in this particular method activation. We then analyze the method nodes repeatedly until we reach the method exit node. Therefore, we compute a node’s transfer function given by the node type, update the successor in-ports, and determine the next node to analyze to get its values stable.

Algorithm 2. processMethod : \((m, [x_{in}, a, v_1, \ldots, v_n]) \rightarrow [x_{out}, r]\)

\[
\begin{align*}
& n = m.\text{entryNode} \\
& in(n) = [x_{in}, a, v_1, \ldots, v_n] \\
& \text{do} \\
& \quad n.\text{computeTransferFunction()} \\
& \quad n.\text{updateSuccess()} \\
& \quad n = n.\text{next()} \\
& \text{while} n\neq m.\text{exitNode} \\
& \text{return in}(n)
\end{align*}
\]

The transition from one method to another is embedded in the statement \(n.\text{computeTransferFunction()}\) if \(n\) is of a monomorphic call type \(\text{MCall}(\cdot, \cdot)\). Note that the processing of a call in turn may lead to the analysis of the call target method \(m\) as defined in processMethod.

3.2. Call processing

Our approach to analyzing individual calls (see Algorithm 3) describes the handling of a call to method \(m\) in a context \(ctx^m\). For the understanding of our call processing, it is safe to assume that all calls to \(m\) are associated with only one context \(ctx^m\), i.e., that we perform a context-insensitive analysis. This is generalized to more contexts in Section 3.3.

The processing of (recursive) method calls must guarantee that the analysis terminates and that the analysis values reach a global fixed point.

The crucial step to ensure termination is that each context \(ctx^m\) is associated with two attributes \(\text{pre}_v\text{args}\) and \(\text{pre}_v\text{return}\) where we store previous input and return values of the calls to \(m\) in that context \(ctx^m\). The former of these attributes is used to decide whether we have seen a more general call targeting \(m\) in the same context \(ctx^m\) before, i.e., if \([x_{in}, a, v_1, \ldots, v_n] \subseteq \text{pre}_v\text{args}\), in which case we interrupt the call processing and reuse the previous result from \(\text{pre}_v\text{return}\).

Algorithm 3. processCall\((ctx^m, [x_{in}, a, v_1, \ldots, v_n]) \rightarrow [x_{out}, r]\)

\[
\begin{align*}
& \text{– if } ctx^m \text{ was already analyzed with larger parameters before} \\
& \text{if } [x_{in}, a, v_1, \ldots, v_n] \subseteq ctx^m.\text{pre}_v\text{args} \text{ then} \\
& \quad \text{return } ctx^m.\text{pre}_v\text{return} \\
& \text{end if} \\
& \text{if } ctx^m.\text{pre}_v\text{args} = ctx^m.\text{pre}_v\text{args} \cup [x_{in}, a, v_1, \ldots, v_n] \text{ then} \\
& \quad \text{return } ctx^m.\text{pre}_v\text{return} \\
& \text{end if} \\
& \text{if } ctx^m.\text{is}\_\text{active} \text{ then} \\
& \quad ctx^m.\text{is}\_\text{recursive} = \text{true} \\
& \quad \text{return } ctx^m.\text{pre}_v\text{return} \\
& \text{end if} \\
& \text{if } ctx^m.\text{is}\_\text{active} = \text{true} \\
& \quad [x_{out}, r] = \text{processMethod}(m, ctx^m.\text{pre}_v\text{args}) \\
& \text{– if } ctx^m \text{ was not recursively called within processMethod} \\
& \text{if } ctx^m.\text{is}\_\text{recursive} \text{ then} \\
& \quad ctx^m.\text{pre}_v\text{return} = [x_{out}, r] \\
& \quad ctx^m.\text{is}\_\text{active} = \text{false} \\
& \text{return } [x_{out}, r] \\
& \text{end if} \\
& \text{– while } ctx^m\text{'s recursive call results haven't reached fixed point} \\
& \text{while } ctx^m.\text{pre}_v\text{return} \subseteq [x_{out}, r] \text{ do} \\
& \quad ctx^m.\text{pre}_v\text{return} = [x_{out}, r] \\
& \quad [x_{out}, r] = \text{processMethod}(m, ctx^m.\text{pre}_v\text{args}) \\
& \text{end while} \\
& \text{ctx^m.\text{is}\_\text{recursive} = \text{false}} \\
& \text{ctx^m.\text{is}\_\text{active} = \text{false}} \\
& \text{return } [x_{out}, r]
\end{align*}
\]

The alternative, a call targeting \(m\) in \(ctx^m\) with new arguments, leads to a new method activation where we process the target method \(m\) by invoking processMethod using the merged input \(\text{pre}_v\text{args} \cup [x_{in}, a, v_1, \ldots, v_n]\). We also update the two attributes \(\text{pre}_v\text{args}\) and \(\text{pre}_v\text{return}\) in preparation for the next call targeting \(m\) in \(ctx^m\).

Termination of our analysis is ensured since we incrementally merge our arguments \(\text{pre}_v\text{args} \cup [x_{in}, a, v_1, \ldots, v_n]\) before we start processing a method \(m\). Thus, the sequence of arguments \(args\) used for a given context \(ctx^m\) forms an ascending chain satisfying \(args_0 \subset args_1 \subset \ldots \subset args_n\).

\]
Each such chain must have finite length since our value lattices have finite heights (both $L_x$ and $L_y$ are finite). Thus, each method can only be processed a finite number of times, and analysis termination is guaranteed. This argument also holds for calls involving recursion; terminations is guaranteed for these programs as well.

In order to guarantee that the fixed point is reached, especially in loops induced by recursive method calls, we need a few more attributes associated with each context: is-active is used to check if we are processing a call in a context that is currently being analyzed, i.e., if $m$ is called recursively in $ctx_m$. In this case, we directly return prev_return for the recursive call and undefined $[0, \bot]$ if we have no previous results, respectively. Also we set is_recursive $= true$ which indicates, upon return from processMethod, that we have seen a recursive call during processMethod. In this case, we need to stabilize the results by iteratively reinvoking processMethod until the fixed point is reached.

### 3.3. Flow sensitivity

The way we associate a call $a.m(\ldots)$ with a number of contexts under which the method $m$ shall be analyzed depends on the call stack abstraction used. Each such abstraction defines a family of different context-sensitive analyses that can be parameterized by a call stack depth $k$. We only consider the case $k = 1$ in this article, hence, we can base the abstraction on the topmost stack frame, i.e., on the target and the return addresses and the actual call parameters of a call site. The case $k > 1$ is a matter of future work. However, previous experiments with 2-CFA and 2-object-sensitivity show a much increased memory cost but only a small increase in performance point of view, is impossible [20]. However, the SSA-based simulated execution approach was, on average, 22% faster.

### 4. Context sensitivity

A context definition is a rule that associates a call with a set of contexts under which the method should be analyzed. Actually, ObjSens is the only context definition (in this selection) that may associate a call with more than one context. Each context is in turn defined by a tuple; the tuple elements, its number and content, depend on what context definition we are using. In this article, we will use the following context definitions for a given call from a call site $cs_l : a.m(v_1, \ldots, v_n)$ where $Pt(a) = \{o^1, \ldots, o^r\}$.

#### Insens: $cs_l \mapsto \{m\}$

All calls targeting method $m$ are mapped to the same context. This is the context-insensitive baseline approach.

#### CallSite: $cs_l \mapsto \{(m, cs_l)\}$

Calls from the same call site $cs_l$ are mapped to the same context.

#### ObjSens: $cs_l \mapsto \{(m)\}$ if $m\text{.isStatic}$,

$\{(m, o^1), \ldots, (m, o^r)\}$ otherwise.

Calls targeting the same receiving abstract object $o \in Pt(a)$ are mapped to the same context. Static calls are handled context-insensitively.
This sensitivities (ThisSens) is to our knowledge new. In contrast to object-sensitivity, which analyzes a method separately for each abstract object reaching the implicit this-variable, this-sensitivities analyzes a method separately for each set of abstract objects reaching this. For example, given a (non-static) call \( a.m(v_1) \) with \( Pt(a) = \{ o^1, o^2 \} \), ThisSens would map it to the single context \( (m, \{ o^1, o^2 \}) \), whereas ObjSens would map it to the two contexts \( (m, \{ o^1 \}) \) and \( (m, \{ o^2 \}) \). We discuss the differences between these two approaches in more detail in the following.

4.2. Object- and this-sensitivity

The object-sensitive approach has been thoroughly studied during the last years, and there seems to be an agreement that this technique is particularly well suited for the analysis of object-oriented programs [17–19]. The difference between our new this-sensitivity and object-sensitivity is that we are using a different context definition for a given call site. Therefore, we compare the two approaches in more detail. 1-CFA is left out of the discussion since it has been compared to object-sensitivity before [18,19].

4.2.1. Analysis precision

It is not obvious which of the two techniques, object- or this-sensitivity, is more precise. In what follows, we will present two scenarios where one technique provides higher precision than the other and vice versa. This proves that neither of the two approaches is strictly more precise than the other.

The first example shows a situation where this-sensitivity provides higher precision:

**Example 1.** Method \( m \):

\[ m(V \ v) \{ \text{return } v; \} \rightarrow V \]

Call 1:

\[ Pt(a_1) = \{ o^n_v \}, \ Pt(v_1) = \{ o^n_v \} \]

\[ r_1 = a_1.m(v_1) \]

Call 2:

\[ Pt(a_2) = \{ o^n_2, o^n_2 \}, \ Pt(v_2) = \{ o^n_2 \} \]

\[ r_2 = a_2.m(v_2) \]

We have two consecutive calls targeting the same method \( m \), which just returns the provided argument. The two calls target expressions \( a_1 \) and \( a_2 \), whose respective points-to sets both contain the abstract object \( o^n_v \) and \( o^n_2 \). In an object-sensitive analysis, both calls target the context \( (m, \{ o^n_v \}) \), and the return values get mixed in the second call:

\[ Pt(r_1) = \{ o^n_v \} \text{ and } Pt(r_2) = \{ o^n_2, o^n_2 \}. \]

Here, the points-to set \( r_1 \) from call 1 gets mixed into the points-to set of \( r_2 \), as the call \( a_2.m() \) is analyzed under both the contexts \( (m, \{ o^n_v \}) \) and \( (m, \{ o^n_2 \}) \).

In a this-sensitive analysis, the two calls target different contexts:

\[ Pt(r_1) = \{ o^n_v \} \text{ and } Pt(r_2) = \{ o^n_2 \}. \]

Here, \( a_2.m() \) is analyzed under the single context \( (m, \{ o^n_v, o^n_2 \}) \), so that no mixing of return values occurs.

The second example shows a situation where object-sensitivity provides higher precision:

**Example 2.** Method \( m \):

\[ m() \{ V \ v = this.f; \ v.n(); \} \]

Call:

\[ Pt(a) = \{ o^n_v, o^n_2 \} \]

\[ Pt(o^n_1.f) = \{ o^n_1 \} \]

\[ Pt(o^n_2.f) = \{ o^n_2 \} \]

\[ a.m() \]

Here, the method \( m \) reads from memory and calls another method \( n \) on the read-result. In the this-sensitive approach, the call \( a.m() \) targets the context \( (m, \{ o^n_v, o^n_2 \}) \) that reads \( \{ o^n_1, o^n_2 \} \) from memory and then calls the context \( (n, \{ o^n_1 \}) \). This gives the upper of the following two call graphs.

\[ (m, o^n_1_v, a) \rightarrow (n, o^n_1_v) \]

This-sensitive call graph.

\[ (m, o^n_2_v, a) \rightarrow (n, o^n_2_v) \]

Object-sensitive call graph.

In the object-sensitive approach, the call \( a.m() \) is analyzed under two different contexts \( (m, o^n_v) \) and \( (m, o^n_2) \), which read \( \{ o^n_v \} \) and \( \{ o^n_2 \} \), respectively, and then call the contexts \( (n, o^n_1) \) and \( (n, o^n_2) \), respectively. This leads to the lower of the two object call graphs given above. That is, the object-sensitive approach gives a more precise result in this example since the results of the memory read operation \( v = \text{this.f} \) never get mixed in this scenario.

Thus, neither of the two approaches is strictly more precise than the other. Our experiments show that these approaches have similar precision in practice, cf. Section 5. However, these two examples nicely illustrate the fundamental differences between the two approaches. This-sensitivity, with the potential to use many more contexts for a given method, is better in separating two different calls targeting the same method. Object-sensitivity, where each abstract object is treated separately, has a more precise handling of operations (calls and field accesses) targeting the implicit variable this.

4.2.2. Analysis cost

Object-sensitivity may use a number of contexts polynomial in the number of abstract objects, whereas this-sensitivity might use an exponential number of contexts and requires therefore, in theory, an exponential amount of memory and time. In practice, however, our experiments show that the number of contexts used by this-sensitivity is (on average) less than half of the number used by object-sensitivity. Furthermore, a low number of contexts used does not only reduce the memory requirements, but also speeds up the analysis by reducing the amount of processing required to reach the fixed point. This non-obvious experimental observation
5. Experiments

In this section, we evaluate our new approach, this-sensitivity, by comparing it with 1-CFA and 1-object-sensitivity. In fact, we compare our implementations thereof, which we refer to as ThisSens, CallSite, and ObjSens, respectively. The differences between the theory and our implementation are discussed in Section 5.1.

We have defined a set of general precision metrics relevant for a large number of different client applications. More precisely, we tried to identify two types of client applications that require different granularity of reference information as their input. These two types are presented in Section 5.2 along with relevant metrics for each type. Section 5.3 presents escape analysis, a concrete client analysis that requires points-to information. Our experimental setup is presented in Section 5.4, and the results of our experiments are discussed in Sections 5.5–5.8.

5.1. Implementation details

Our Java implementation of the presented analysis reads and analyzes Java bytecode. We use the Soot framework, version 2.3.0, as our bytecode reader [27]. We then use the Shimple format provided by Soot as the starting point to construct the SSA-based graphs for the individual methods.

In the context-sensitive analysis, all contexts are equipped with a node-to-values map where all current analysis values are saved. Using this approach, we avoid cloning the method graphs. Moreover, our points-to set implementation is similar to the hybrid-set implementation that comes with Soot, and we make sure that we never save multiple copies of identical sets.

We use stubs to simulate the behavior of native methods. We have a conservative handling of exceptions, threads, and methods in the Java class library dealing with array manipulation (e.g., java.lang.System.arraycopy). Our analysis implementation is currently incomplete in the following sense: (1) It does not handle features related to dynamic class loading and reflection correctly. To our knowledge, no feasible approach to handling these features is known. (2) Native methods returning String and StringBuffer objects are treated like allocation sites. Methods returning void are considered to be side-effect free, and all other (non-void) methods return ⊥.

5.2. Two types of client applications

The reference information that can be extracted from a program using static points-to analysis is in most cases used as input to different client applications. These client applications can be further divided into different domains such as compiler optimizations, software development, and reverse engineering. In this section, we will take an orthogonal approach and try to focus on what granularity of reference information the client applications need rather than their domain.

The first type of client application that we have identified, denoted SourceCode clients, is primarily interested in source code entities and reference relations between them, i.e., in relations between source code entities like classes, methods, fields, and statements, that hold for any execution of the program and for all instances of a class. Examples are all client applications that require a call graph as input, i.e., most types of inter-procedural program analysis. Other examples of SourceCode clients are: virtual call resolution to avoid dynamic dispatch and facilitate method inlining [19,28], cast safety analysis to avoid unnecessary run-time type checking [19,29], metrics-based analyses to compute coupling and cohesion metrics involving members and classes [1,2], source code browsers in IDEs [30,31] that need to resolve various source code references, and software testing where class dependencies determine the test order [6–8].

Another type of client application, denoted ObjectIdentity clients, is primarily interested in individual objects and references to individual objects. Examples of ObjectIdentity clients are: side-effect analysis that computes the set of object fields that may be modified during the execution of a statement [17,18,32], escape analysis that identifies method-local (or thread-local) objects to improve garbage collection (and to remove synchronization operations) [33–35], Memory leak debugging to identify references that prevent garbage collection [26], static design pattern detection to identify the interaction among possible participating objects [10], reverse engineering of UML interaction diagrams [9], and architectural recovery by class clustering to avoid erroneous groupings of classes/instances [3,4].

5.2.1. Application entities

In order to avoid taking into account results due to the same set of Java library and Java Virtual Machine (JVM) start-up classes over and over again, we decided to use the following method when applying our metrics suites on the results of the points-to analysis: we selected a subset of all classes in each benchmark program and denoted them application classes. A simple name filter on the fully qualified class names did this job. For example, the application classes of xsltcl1.2 are all those classes having a name starting with org.apache. Members defined in these classes are denoted application members and abstract objects corresponding to allocations of these classes are denoted application objects. We did not consider any class from the Java standard library as an application class in any of the benchmark programs.

5.2.2. The SourceCode metrics suite

The set of precision metrics presented here is most relevant for client applications of type SourceCode. These metrics are frequently used when evaluating different approaches to points-to analysis.

- **Node, Edge:** The number of nodes (methods) and edges (calls) in a call graph where at least one of the participants (caller or callee) is an application method.
- **PCall:** The number of potentially polymorphic call sites located in an application class.
- **Cast:** The number of casts (located in an application class) potentially failing at run-time.

The call graph related metrics, Node and Edge, are relevant for inter-procedural analyses. The other two are related to method inlining and cast safety analysis.

5.2.3. The ObjectIdentity metrics suite

The set of metrics presented here is most relevant for client applications of type ObjectIdentity.
5.3. Escape analysis

Escape analysis is a static program analysis that finds abstract objects escaping from the methods of a program. We say that an object escapes from a method if it is created during the execution of the method and is still accessible after the execution of the method. An object escapes if it is returned by the method and assigned to a field of another escaping object, or assigned to an object allocated outside the method. Static variables are considered fields of a virtual object allocated outside any method.

On top of our points-to analysis, we have implemented a standard escape analysis [35,36]. Our implementation uses points-to analysis as a black box preparing for an easy switching from one context-sensitive analysis technique to another.

Escape analysis has, among others, successfully been applied to the stack-allocated heap-objects [36–38]: a non-escaping object, which is – outside any loop – heap-allocated in a method and can as well be allocated on the stack-frame of the method avoiding the garbage collection of objects. Our metrics \( E(m) \) counts the number of abstract objects allocated in and escaping from a method by unifying the sets of objects escaping any context of the method. Lower values are better since more objects could be stack-allocated.

The above idea has also been generalized before: an object allocated outside a loop in a method and escaping from a method can as well be allocated in a transitive caller method \( n \), if \( i \) it does not escape from \( n \), \( ii \) \( n \) dominates \( m \), and \( iii \) each execution of \( n \) corresponds to at most one execution of \( m \). Therefore, our second metric \( \text{Esc}(m) \) counts the number of abstract objects allocated transitively in a method, and escaping from a method by unifying the sets of objects escaping any context of the method. Lower values are again better since more objects could be potentially stack-allocated.

5.4. Experimental setup

We have used a benchmark containing 16 different programs. Since we analyze Java bytecode, we characterize the size of a program in terms of “number of classes and methods” rather than “lines of code” – our benchmark programs range from 311 to 1002 classes. All programs are presented in Table 1.

The programs in the upper half of the table are taken from well-known test suites [39–41], and we have picked all those programs that were (i) larger than 300 classes, and (ii) freely available on the Internet. In the lower half, we have our own set of “more recent” test programs, which are also freely available. All programs are analyzed using version 1.4.2 of the Java standard library, and all experimental data presented in this article is the median value of three runs on the same computer (Dell PowerEdge 1850, 6GB RAM, Dual Intel Xeon 3.2 GHz under Linux x86-64, kernel 2.6.22.)

Table 1 contains data taken from our context-insensitive analysis Insens. This set of data will be our baseline result which we compare the context-sensitive results with. The first section General is provided to give a rough overview of the different programs, and shows the number of used classes (Class), the number of reachable methods (Method), and the number of abstract objects (Object). It further lists the analysis time (Time), i.e., the time needed to perform the points-to analysis. The times
required for Points-to SSA graph construction and analysis setup are not included. The sections SourceCode, ObjectIdentity, and Escape show the context-insensitive results for the precision metrics that we introduced earlier.

5.5. *The SourceCode metrics suite — results*  

In this section, we present the first set of results when measuring the precision using the *SourceCode* metrics suite. The results related to our context-sensitive approaches *CallSite*, *ObjSens* and *ThisSens* are presented in Table 2. All results in this and the following tables are given as a multiple of the context-insensitive results presented in Table 1. For example, for the *sablecc-j* benchmark, the number of unresolved polymorphic calls in analysis *CallSite* is 506 × 0.834 = 422, where 506 is the number for the metrics *PCall*.

---

**Table 2**  
Results relevant for *SourceCode* clients.

<table>
<thead>
<tr>
<th>Program</th>
<th>CallSite</th>
<th>ObjSens</th>
<th>ThisSens</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Node</td>
<td>Edge</td>
<td>Heap</td>
</tr>
<tr>
<td>Antr</td>
<td>0.836</td>
<td>0.302</td>
<td>0.891</td>
</tr>
<tr>
<td>Bloat</td>
<td>0.776</td>
<td>0.357</td>
<td>0.680</td>
</tr>
<tr>
<td>Chart</td>
<td>0.811</td>
<td>0.692</td>
<td>0.524</td>
</tr>
<tr>
<td>Eclipse</td>
<td>0.409</td>
<td>0.123</td>
<td>0.165</td>
</tr>
<tr>
<td>Fop</td>
<td>0.779</td>
<td>0.379</td>
<td>0.904</td>
</tr>
<tr>
<td>JavaV</td>
<td>0.883</td>
<td>0.619</td>
<td>0.535</td>
</tr>
<tr>
<td>Javadoc</td>
<td>0.855</td>
<td>0.482</td>
<td>0.668</td>
</tr>
<tr>
<td>Jython</td>
<td>0.713</td>
<td>0.527</td>
<td>0.243</td>
</tr>
<tr>
<td>Ps</td>
<td>0.928</td>
<td>0.628</td>
<td>0.727</td>
</tr>
<tr>
<td>Sablecc-j</td>
<td>0.727</td>
<td>0.145</td>
<td>0.108</td>
</tr>
<tr>
<td>Soot-c</td>
<td>0.753</td>
<td>0.234</td>
<td>0.901</td>
</tr>
<tr>
<td>Emma2.0</td>
<td>0.743</td>
<td>0.331</td>
<td>0.839</td>
</tr>
<tr>
<td>JavaCc3.2</td>
<td>0.739</td>
<td>0.079</td>
<td>0.484</td>
</tr>
<tr>
<td>Jess4.5</td>
<td>0.887</td>
<td>0.399</td>
<td>0.872</td>
</tr>
<tr>
<td>Pmd3.2</td>
<td>0.829</td>
<td>0.501</td>
<td>0.846</td>
</tr>
<tr>
<td>Xsltc1.2</td>
<td>0.901</td>
<td>0.327</td>
<td>0.599</td>
</tr>
<tr>
<td>Average</td>
<td>0.786</td>
<td>0.395</td>
<td>0.624</td>
</tr>
<tr>
<td>Median</td>
<td>0.795</td>
<td>0.368</td>
<td>0.674</td>
</tr>
</tbody>
</table>

---

1 The longest graph construction time we measured was for *Chart* (48.2 s). A large part of this time (92%) was spent on file reading and in third party components (Soot).
In this section, we present the results when measuring the precision using the ObjectIdentity metrics suite. The results are presented in Table 3.

First, all three approaches are much more precise than context-insensitive analysis. This results indicate that client applications of type ObjectIdentity are likely to benefit from using a context-sensitive analysis. The results for the metrics Heap show a significant precision improvement (38–78%) when using a context-sensitive analysis, a result that indicates a considerably improved precision in the handling of memory store operations. This comes as no surprise for the two functional approaches (both come with a 78% improvement), since, due to encapsulation, almost every memory store operation is done using the implicit variable this. It was also shown by Milanova et al. [17,18] that this type of memory related client application can benefit from context-sensitive analysis. Our experiments confirm these results.

Second, the two functional approaches, on average and median, are clearly more precise than CallSite. This- and object-sensitivity have in common that the contexts in which a method is analyzed depends on the value of the implicit this-variable. That is, the analysis values of two calls targeting disjoint this-sets are never mixed. This is important in object-oriented programs where a large part of all field accesses and calls targets this. The focus on this makes this-and object-sensitivity more precise in the analysis of object-oriented programs than CFA, which was designed for the analysis of functional and imperative programs. This is also in agreement with previous results comparing object-sensitivity and the call string approach [18,19]. However, our use of the metrics suite that focus on individual objects makes the difference between the two types of context definitions more obvious. It seems likely that client applications that require precise information about individual objects and their interaction would benefit from using one of the two functional approaches.

Finally, the two functional approaches provide almost identical results in three out of four metrics. The major difference is in the metrics OEdge, where ObjSens is considerably more precise. The higher precision is probably due to a more precise handling of memory load operations via the implicit variable this, a situation that is very common in most programs. This situation was studied in Example 2, Section 4.2.

5.7. Escape Analysis — results

The picture that the abstract points-to metrics show is confirmed by the concrete client application escape analysis, as shown in Table 4. The results measured by Esc and Esc improve clearly (up to 6% and 17%, respectively, on average) when the precision of the underlying points-to analysis improves.

This confirms our assumption that client analyses which are interested in individual objects and their relations improve when based on more precise context-sensitive points-to results.

In one case (emma2.0), the overall number Esc of objects defined in and escaping from a method is reduced to 75% and for Esc, the number of objects defined in a method or its transitive descendants and escaping, is reduced to 22%.

It should come at no surprise that the reduction in the direct escapes Esc is somewhat smaller than the reduction in the transitive escapes Esc*: an object that is identified as not escaping a method in which it is allocated directly, cannot escape from a caller of m either. Hence, small improvements in Esc can imply larger differences in Esc.

Again, the two functional approaches are more precise than CallSite, which is an effect that our abstract points-to metrics already indicate and explain. The two functional approaches ObjSens and ThisSens provide again almost identical results.

5.8. Time and memory measurements

Table 5 shows the number of used contexts (Ctx) and the analysis time required for each approach. We have used the number of used contexts, rather than a direct memory measurement, as our memory cost metrics. This is based on the assumption that the memory cost of maintaining N contexts for a given method m is O(N), which is true for our implementation and most other implementations that we know of. Furthermore, it is simple and implementation and machine independent. Ctx is the average number of used contexts per method, and time is again given as a multiple of the context-insensitive results presented in Table 1.

The first thing to notice is that ThisSens is about twice as fast as CallSite, and more than an order of magnitude faster than ObjSens, on average and median. The time measurements also show that ThisSens is, on average (median), only 29% (20%) slower than our context-insensitive analysis Insens, a number that is likely to be accepted by client applications that can make use of the improved precision. The fact that ThisSens is even faster than Insens on a
number of benchmarks (e.g., mablecc-j and emma2.0) can be considered as a positive side-effect of the improved memory store precision (as manifested by the Heap results). Improved memory store precision implies fewer memory changes, which consecutively implies that the fixed point iteration stabilizes faster, cf. Section 3.

ObjSens is also much more costly than the other two approaches when it comes to the number of used contexts (Ctx). It requires, on average, 2.2 (2.4) times as many contexts as ThisSens (CallSite).

This is a bit surprising since, remembering from Section 4.2, that this-sensitive analysis potentially requires an exponential number of contexts in theory. In practice, on the other hand, it turns out that, on average, the number of used this-sets is smaller than the number of receiving abstract objects. Furthermore, a low number of used contexts does not only reduce the memory requirements, but even speeds up the analysis by reducing the processing required to reach a fixed point. This non-obvious observation is the key to understand why this-sensitivity outperforms object-sensitivity despite of an exponential worst-case scenario. Another reason is that object-sensitivity may associate a call site with more than one context. Thus, a call a.m(...) where \( N = |Pr(a)| \), may require that method \( m \) (and all its callees transitively) be processed \( N \) times, one for each abstract object in \( Pr(a) \). That is, a single call targeting a large points-to set where \( N > 1 \) may generate a cascade of new contexts to process. This problem does not occur in this-sensitivity where each call always targets a single context, which also simplifies the analysis implementation considerably.

Finally, our results where object-sensitivity is more than an order of magnitude slower than the context-insensitive analysis may at first glance seem contradictory to the results presented by Milanova et al. [18], where object-sensitivity was only slightly slower. The explanation is simple: they presented object-sensitivity in theory, but implemented a simplified version where only a fraction of the program (the method parameter nodes) was treated in a context-sensitive manner; all other node types were treated insensitively. This simplification will of course speed up the analysis at the cost of some precision loss. Our implementation follows the theory and treats all method contexts separately.

Although we have not experienced any sign of exponential behavior in our experiments\(^2\), we still recommend any non-experimental implementation of this-sensitivity to use a guarded approach (cf. discussion in Section 4.2) to guarantee a polynomial behavior for any input program.

6. Related work

A context-insensitive version of the SSA-based simulated execution approach used in this article was presented before [20]. Our program representation Points-to SSA is closely related to Memory SSA [42,43]. Memory SSA is an extension to the traditional approach to SSA [21,44].

Points-to SSA is a graph-based representation specially designed for points-to analysis where we have removed all operations not directly related to reference computations. A similar approach (a sparse graph designed for points-to analysis) was used by Binkley and Lyle [45] to compute program slices in C programs. It is, however, not SSA-based.

The number of papers explicitly dealing with context-sensitive points-to analysis of object-oriented programs is growing [17–19,26,46,47]. The active research within this area demonstrates its expected potential to improve the analysis precision. The papers experiment with different context definitions and techniques to reduce the memory cost associated with having multiple contexts for a given method. It should also be noted that many approaches targeting object-oriented programs have an “imperative counterpart”, which often pre-dates the object-oriented work. People interested in more general reviews of the area should take a look at the papers of Hind [24] and Ryder [13].

Many authors use a call string approach and approximative method summaries to reduce the cost of having multiple contexts [46,47]. Sometimes, ordered binary decision diagrams (OBDD) are used to efficiently exploit commonalities among similar contexts [19,26,42], which allows handling of a very large number of contexts at reasonable memory cost.

Milanova et al. [17,18] present the object-sensitive technique as discussed earlier. The object-sensitive and call string approaches were compared in various works [18,19]. These works also show that 1-object-sensitivity scales to programs containing hundreds of classes.

Whaley et al. [26] present a k-call-string-based analysis with no fixed upper limit (k) that only takes acyclic call paths into account. Calls within strongly connected components are treated context-insensitively. They report reasonable analysis costs due to their use of OBDDs to handle all contexts. However, their analysis technique was later on compared to 1-object-sensitivity, which was found to be “clearly better” both in terms of precision and scalability [19]. 1-object-sensitivity, in turn, is similar in precision to 1-this-sensitivity but much more costly, as shown in this article.

7. Summary, conclusion, and future work

In this article, we present a new context-sensitive approach to points-to analysis where the target context associated with a call site a.m(...) is determined by the pair \((m, Pr(a))\), where \(Pr(a)\) is the points-to set of the target expression \(a\). Hence, we distinguish analysis contexts of a method by its implicit variable \(this\). We have therefore named it this-sensitivity. It is a modified version of object-sensitivity presented by Milanova et al. [17,18].

We have experimentally evaluated 1-this-sensitivity by comparing it with two well-known context-sensitive approaches (1-object-sensitivity and 1-CFA). Our measurements show that 1-this-sensitivity is much faster than the other two. It is in fact, on average (median), only 35% (21%) slower than our context-insensitive analysis.

We have used two different abstract metrics suites to evaluate the precision of our new approach. Each metrics suite is targeted to a specific group of client applications.

The first metrics suite, denoted SourceCode, is most relevant for client applications that are primarily interested in source code entities and reference relations between them, i.e., in relations that hold for all instances of a class. An example is call graph construction. The second metrics suite, denoted ObjectIdentity, is most relevant for client applications that are primarily interested in individual objects and references to individual objects. Examples of ObjectIdentity clients are side-effect analysis and reverse engineering of UML interaction diagrams. Finally, we have implemented a client application – escape analysis – representing a concrete ObjectIdentity client.

The experiments using the SourceCode metrics suite show no significant difference between the three context-sensitive approaches. Furthermore, they only provide insignificantly better results than the context-insensitive analysis.

The experiments using the ObjectIdentity metrics suite show that all three context-sensitive approaches are much more precise than the context-insensitive analysis. Furthermore, 1-object-sensitivity and 1-this-sensitivity are clearly more precise than 1-CFA. This is in agreement with previous results, where 1-object-sensitivity and 1-CFA were compared with each other [18,19].
The results of the concrete client application escape analysis also improve significantly when a more precise context-sensitive analysis is used. Again, 1-object-sensitivity and 1-this-sensitivity provide almost identical results and are slightly more precise than 1-CFA.

Finally, 1-object-sensitivity and 1-this-sensitivity provide almost identical results on escape analysis and in three out of four \texttt{ObjectIdentity} metrics. The major difference lies in the metrics indicating a precise resolution of object member accesses. Here, 1-object-sensitivity is significantly more precise. However, this comes at the price of being more than an order of magnitude slower than 1-this-sensitivity.

In conclusion, we recommend 1-this-sensitivity if the client application requires quite precise points-to-information and the analysis is on an edit-compile cycle. We recommend 1-object-sensitivity only if the client application requires very precise points-to-information and the analysis could be executed as a batch job. Finally, client applications of type \texttt{SourceCode} can probably safely avoid the trouble of adding any kind of context-sensitivity to their analysis.

We are currently working on three different approaches to extend 1-this-sensitivity to the next “level of precision” by (1) implementing support for k-this-sensitivity, (2) implementing support for context-sensitive abstract objects, and (3) including all arguments, not only this, into our functional context definitions. We refer to the latter as argument-sensitivity and consider it to be an orthogonal next “level of precision”.

References