Eighth IEEE International Working Conference on Source Code Analysis and Manipulation

Fast and Precise Points-to Analysis

Jonas Lundberg, Tobias Gutzmann, and Welf Löwe
School of Mathematics and System Engineering
Växjö University, 351 95 Växjö, Sweden
Email: (Jonas.Lundberg|Tobias.Gutzmann|Welf.Lowe)@vxu.se

***Abstract*—Many software engineering applications require points-to analysis. Client applications range from optimizing compilers to program development and testing environments to reverse-engineering tools. In this paper, we present a new context-sensitive approach to points-to analysis where calling contexts are distinguished by the points-to sets analyzed for their target expressions. Compared to other well-known context-sensitive techniques, it is faster—twice as fast as the *call string* approach and by an order of magnitude faster than the *object-sensitive* technique—and requires less memory. At the same time, it provides higher precision than the call string technique and is similar in precision to the object-sensitive technique. These statements are confirmed by experiments.**

I. 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.

The basis for many points-to analysis, and program analysis in general, is the theory of monotone dataflow frameworks [1], [2]. 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.

In a *context-insensitive* program analysis, analysis values of different call sites may get propagated to the same method and get mixed there. The analysis value is the merger of *all* calls targeting that method. A *context-sensitive* analysis addresses the problem caused by this issue by distinguishing between the different calling contexts of a method. It analyzes a method separately for each calling context [3]. 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 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 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 [4]. The call string approach defines a context by the first k callers, i.e., return addresses on the call stack top [5], referred to as the family of k-CFA analyses. The functional approach uses some abstractions of the call site's actual parameters to distinguish different contexts [4], [6]. Both the call string and the functional approaches were evaluated, and put into a common framework by Grove et al. [6].

A rather new functional approach designed for object-oriented languages is referred to as *object-sensitivity* [7], [8]. It distinguishes contexts by analyzing the targeted method for each abstract object in the implicit *this*-parameter separately. Similarly to k-CFA, we can define a family of k-object-sensitive algorithms distinguishing contexts by the top k abstract target objects on the call stack. A *simplified* version of 1-object-sensitivity—simplified in the sense that it merges analysis values of different contexts coming from method parameters when analyzing a method—improves, compared to 1-CFA, the precision of side-effect analysis and, to a lesser degree, call graph construction [7], [8]. 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 [9]. The contributions of this paper 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 the other two, (iii) is more precise than 1-CFA, and (iv) is almost as precise as 1-object-sensitivity.
- We present two general precision metrics suites that cover different granularities and aspects of precision corresponding to two different types of client applications.

In Section II, we outline our flow-sensitive analysis schema. It has been presented before [10], but we include a brief version of the material for understandability and completeness of this paper. In Section III, we present our new context-sensitive points-to analysis. In Section IV, we present our precision metrics, experimental setup, and results. Finally, Section V discusses related work and Section VI concludes this paper.

II. SSA-Based Simulated Execution

First, we present the analysis value representation, which consists of sets of abstract objects and a heap-memory abstrac-

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.26

tion, as well as our SSA-based program representation, named *Points-to SSA*. We then introduce our analysis algorithm, which is an abstract interpretation using a *simulated execution* of the program.

A. Analysis Value, Memory, and Program Representations

Our points-to analysis computes sets of references to abstract objects. An *abstract object* o is an analysis abstraction that represents one or more run-time objects. The mapping from, possibly infinitely many, run-time objects to finitely many abstract objects is called a *name schema*. In this paper, we will use the following name schema: each *syntactic creation point* s corresponds to a unique abstract object o_s. Thus, the set of all allocation sites in a program defines a finite set of abstract objects denoted O, and every abstract object $o_s \in O$ 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 general, a reference expression refers to more than one abstract object. Hence, each *points-to set* $v \subseteq O$ is a set of abstract objects. We use the notation $Pt(a)$ to denote the points-to set referenced by an expression a.

Each abstract object $o \in O$ denotes a unique set of *object fields* $[o, f] \in OF$ where $f \in F$ is a unique identifier of a field. In turn, each object field $[o, f]$ denotes a heap *memory slot* $([o, f], v)$ where v is a points-to set. The points-to set v associated with a given object field $[o, f]$ may change from one program point to another due to field store operations.

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 $([o, f], v)$. In our approach, we are using 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 sets. The memory is therefore equipped with two operations

$$Mem.get(OF) \rightarrow V \quad \text{and} \quad Mem.set(OF, V)$$

with the interpretation of reading the points-to set stored in an object field $[o, f] \in OF$, and merging the points-to set $v \in V$ with the points-to set already stored in an object field $[o, f] \in OF$, respectively. Note that we never override previously stored object field values in memory store operations. Instead, we merge (set union) the new set with the old ones, i.e., we perform *weak updates*. An example is given later on when we present our handling of the store operation.

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* $x \in X = [0, h_m]$ where h_m is the maximum memory size. It corresponds to the case where all object fields contain all abstract objects. Hence, $h_m = |OF| \cdot |O|$.

Our points-to analysis uses an SSA-based program representation [11], [12] where each method is represented by a *method graph*. Nodes correspond to operations; local variables v are resolved to dataflow edges connecting the unique defining operation-nodes to operation-nodes that use v. 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 ϕ-operation nodes.

Figure 1 shows a simple "Linked List" implementation (class `L`) and the corresponding Points-to SSA graphs. The basic idea can be understood just by comparing the source code with the respective graphs. Note 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`. ϕ-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 a semantic abstraction of a method, 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 in 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 *use* the new memory size value. Thus, memory sizes are considered as data and memory size edges have the same semantics—including the use of ϕ-nodes at join points—as def-use edges for other types of data. The introduction of memory size edges in Points-to SSA is important since they also imply a correct order in which the memory accessing operations are analyzed ensuring that an analysis is a flow-sensitive abstraction of the semantics of the program.

Node types may have attributes that refer to node specific information, e.g., each *Alloc* node is decorated with a class identifier C that identifies the class of the object to be created.

Each type of node is associated with a unique *analysis semantics* (or *transfer function*), which is a mapping from input- to output-values that may have a side-effect on the memory. Values are carried through so-called *ports*. In our example, ports labeled *x* specify memory sizes, *a* targets for member accesses, and *v* general points-to sets.

As an example for a Points-to SSA transfer function, Algorithm A 1 shows the analysis semantics for the $Store^f$ node type, which abstracts the actual semantics of a field store statement $a.f = v$. For each abstract object o in the address reference a, it looks up the points-to set previously stored in object field $[o, f]$. If the new set to be stored would change the

```
public class L {
  V value = null;
  L next = null;

  public L (V v) {
    value = v;
  }

  public void append(V v) {
    if (next == null)
      next = new L(v);
    else
      next.append(v);
  }

  public void putAt(int n, V v) {
    int count = 0;
    L l = this;
    while (count < n) {
      l = l.next;
      count++;
    }
    l.value = v;
  }
}
```

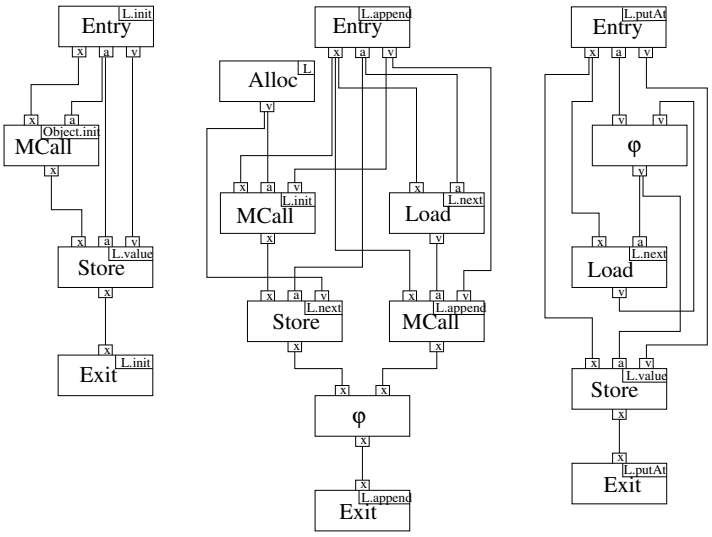


Fig. 1. Source code fragment and corresponding Points-to SSA graphs.

A 1 $Store^f : [x_{in}, a, v] \mapsto x_{out}$

$x_{out} = x_{in}$
for each $o \in Pt(a)$ **do**
  $prev = Mem.get([o, f])$
  if $v \not\sqsubseteq prev$ **then**
    $merge = prev \sqcup v$
    $Mem.set([o, f], merge)$
    $x_{out} = Mem.getSize()$
  end if
end for
return x_{out}

memory (i.e., if $v \not\sqsubseteq prev$), we union v with the previous set and save the result. Also note that we compute a new memory out-port value (a new memory size) if the memory has been changed during this operation.

B. Simulated Execution

Our dataflow analysis technique simulates the actual execution of a program: starting at one or more entry methods, it analyzes the statements of a method in execution order, interrupts this analysis when a call expression occurs to follow the call, continues analyzing the potentially called methods, and resumes with the calling method once the analysis of the called methods is completed. The resulting analysis is flow-sensitive—i.e., it takes into account the order in which statements in a program are executed [3]—in the sense that a memory accessing operation (a call or a field access) $a_1.x$ will never be affected by another memory access $a_2.x$ that is executed after $a_1.x$ in all runs of a program. This makes simulated execution strictly more precise than the frequently used flow-insensitive whole program points-to graph approach [6]–[9], [13], which was verified by experiments [10]. While a strict ordering of the two approaches, from a performance point of view, is impossible, the SSA-based simulated execution appears, on average, faster.

The simulated execution approach can be seen as a recursive interaction between the analysis of an individual method and the analysis semantics associated with monomorphic calls handling the transition from one method to another[1].

III. Context-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 will only consider the case $k = 1$ in this paper, hence, we can base the abstraction on the topmost stack frame, i.e., on target and return addresses and the actual call parameters of a call site. In this section, we present four different context definitions: *Insens*, *CallSite*, *ObjSens*, and *ThisSens*. The first three represent the well known context-*in*sensitive, 1-CFA, and 1-object-sensitive approaches. The last one is our new context-sensitive approach, *1-this-sensitivity*.

A. Context Definitions

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

[1]Polymorphic calls are mapped to selections over possible target methods m_i, which are then processed as a sequence of monomorphic calls targeting m_i.

Insens: $cs_i \mapsto \{(m)\}$
: All calls targeting method m are mapped to the same context. This is the context-insensitive baseline approach.

CallSite: $cs_i \mapsto \{(m, cs_i)\}$
: Calls from the same call site cs_i are mapped to the same context.

ObjSens:$cs_i \mapsto \{(m)\}$ if $m.isStatic$,
$\{(m, o^1), \ldots, (m, o^p)\}$ otherwise.
: Calls targeting the same receiving abstract object $o^i \in Pt(a)$ are mapped to the same record. Static calls are handled context-insensitively.

ThisSens:$cs_i \mapsto \{(m)\}$ if $m.isStatic$,
$\{(m, Pt(a))\}$ otherwise.
: Calls targeting the same points-to set $Pt(a)$ are mapped to the same context. Static calls are handled context-insensitively.

This-sensitivity (*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-sensitivity analyzes a method separately for each *set of abstract objects* reaching `this`. We discuss the differences between these two approaches in more detail in the following section.

B. 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 suited for the analysis of object-oriented programs [7]–[9]. 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 [8], [9].

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:
$m(V\ v)$ {return v;} $\mapsto V$
Call 1:
$Pt(a_1) = \{o_a^1\},\ Pt(v_1) = \{o_v^1\}$
$r_1 = a_1.m(v_1)$
Call 2:
$Pt(a_2) = \{o_a^1, o_a^2\},\ Pt(v_2) = \{o_v^2\}$
$r_2 = a_2.m(v_2)$

We have two 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_a^1, and a_2 also o_a^2. In an object-sensitive analysis, both calls target the context (m, o_a^1), and the return values get mixed in the second call:

$$Pt(r_1) = \{o_v^1\} \text{ and } Pt(r_2) = \{o_v^1, o_v^2\}.$$

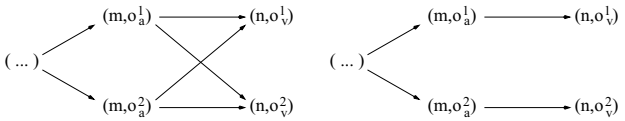
In a this-sensitive analysis, the two calls target different contexts, and no mixing of return values occurs:

$$Pt(r_1) = \{o_v^1\} \text{ and } Pt(r_2) = \{o_v^2\}.$$

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

Example 2
Method m:
$m(\ )\ \{V\ v = this.f;\ v.n();\}$
Call:
$Pt(a) = \{o_a^1, o_a^2\}$
$Pt([o_a^1, f]) = \{o_v^1\}$
$Pt([o_a^2, f]) = \{o_v^2\}$
$a.m()$

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 $(m, \{o_a^1, o_a^2\})$ that reads $\{o_v^1, o_v^2\}$ from memory and then calls the context $(n, \{o_v^1, o_v^2\})$. This gives the object call graph as given on the left below.



This-sensitive call graph. Object-sensitive call graph.

In the object-sensitive approach, the call $a.m()$ is analyzed in two different contexts (m, o_a^1) and (m, o_a^2), which read $\{o_v^1\}$ and $\{o_v^2\}$, respectively, and then call the different contexts (n, o_v^1) and (n, o_v^2), respectively. This leads to the object call graph as given on the right above. That is, the object-sensitive approach gives a more precise result in this example since the results of the memory read operation `v=this.f` never get mixed in this scenario.

Thus, none of the two approaches is strictly more precise than the other. Our experiments show that precision is very similar in practice with both approaches, cf. Section IV. 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`.

Analysis Cost: This-sensitivity might use an exponential number of contexts and requires therefore, in theory, an exponential amount of memory. In practice, however, our experiments show that the number of contexts used by this-sensitivity is (on average) lower than the number used by object-sensitivity. Furthermore, a low number of used contexts

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 observation is one of the major reasons why this-sensitivity is an order of magnitude faster than object-sensitivity, cf. Section IV-E.

Finally, it is straightforward to add an ad hoc mechanism that recognizes situations in which the number of contexts explodes. For instance, when a given method has been associated with more than N contexts, the analysis may be widened, e.g., simply merging two contexts. This is the approach that should be used in any non-experimental implementation of this-sensitivity, but since none of the programs studied in this paper shows exponential behavior for this-sensitivity, we omit a deeper discussion of such possible mechanisms.

C. Implementation Details

Our Java implementation of the analysis presented above reads and analyzes Java bytecode. We use the *Soot* framework, version 2.2.3, as our bytecode reader [14]. 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 have not experienced any major memory problems with the actual points-to analysis so far.

Our analysis implementation is currently incomplete in the sense that it does not correctly handle certain features related to class loading and reflection. To our knowledge, no feasible approach to handling these features is known.

IV. 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.

Our experiments do not target any specific client application that requires reference information, e.g., compiler optimization or software engineering activity. We have chosen to use a set of general (artificial) 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 Sections IV-A and IV-B along with relevant metrics for each type. Although measuring the effects for specific clients is also a necessary part when evaluating a new approach, we consider this to be the next step after a more general evaluation.

A. 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 [9], [15], *cast safety analysis* to avoid unnecessary run-time type checking [9], [16], *metrics-based analyses* to compute coupling and cohesion metrics involving members and classes [17], [18], *source code browsers* that need to resolve various source code references, and *software testing* where class dependencies determine the test order [19]–[21].

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 k [7], [8], [22], *escape analysis* that identifies method (or thread) local objects to improve garbage collection (and to remove synchronization operations) [23]–[25], *Memory leak debugging* to identify references that prevent garbage collection [13], static *design pattern detection* to identify the interaction among possible participating objects [26], reverse engineering of UML *interaction diagrams* [27], and architectural recovery by *class clustering* to avoid erroneous groupings of classes/instances [28], [29].

B. Used Metrics and Benchmarks

In order to compare our new approach this-sensitivity with the other two approaches, we have used a benchmark containing 13 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 225 to 796 classes. All programs are presented in Table I.

The programs in the upper half of the table are taken from well-known test suites [30]–[32], and we have picked those programs that were (i) larger than 200 classes, and (ii) freely available on the Internet. They are a bit "older" and are analyzed using version 1.3.1 of the Java standard library. In the lower half, we have our own set of "more recent" test programs. They are all publicly available and are analyzed using version 1.4.2 of the Java standard library. The program `obfusc0.73` is a source code obfuscator that comes with the Java source code transformation framework Recoder v0.73 [33]. All experimental data presented in this

	General				*SourceCode*				*ObjectIdentity*			
Program	Class	Method	Object	Time [s]	Node	Edge	PCall	Cast	ONode	OEdge	Heap	Enter
antlr	225	910	2,615	1.3	808	2,432	131	49	4,441	57,906	19,004	26,138
javac	307	2,131	4,007	24.8	1,714	6,603	668	477	18,046	329,189	64,797	224,971
javadoc	416	1,845	4,363	11.2	1,192	3,992	176	116	19,321	274,139	54,851	207,740
jython	322	2,093	5,022	41.1	1,788	5,448	287	140	42,099	1,565,265	193,889	422,035
ps	396	1,605	3,172	3.4	1,153	10,220	291	573	8,258	504,348	69,285	116,335
sablecc-j	649	2,990	4,524	14.9	2,045	17,205	434	342	15,095	162,877	56,128	218,336
soot-c	796	3,070	14,733	22.9	2,859	12,990	964	710	26,005	765,797	77,928	609,428
emma2.0	749	3,401	7,004	62.7	1,691	4,410	118	108	37,652	444,087	83,153	452,542
javacc3.2	274	1,579	7,706	4.2	1,068	3,350	39	404	8,989	223,630	3,578	71,749
pmd3.2	508	2,642	4,162	3.9	1,693	3,807	52	94	12,120	118,903	19,595	56,086
jess4.5	308	1,139	3,113	3.4	706	2,108	42	73	4,754	156,291	6,972	28,355
obfusc0.73	688	3,714	3,847	14.4	3,490	10,435	422	420	15,937	128,500	15,268	104,850
xsltc1.2	663	3,038	7,301	175.1	1,959	9,385	501	531	53,786	2,243,036	208,038	649,942

TABLE I
BENCHMARK INFORMATION AND CONTEXT-INSENSITIVE RESULTS

paper is the median value of three runs on the same computer (Dell Inspiron 5150, 1GB, Pentium 4, 3.2GHz under Windows 2000) using Sun's JVM 1.4.2.

Table I 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* shows the number of used classes (*Class*), the number of reachable methods (*Method*), the number of abstract objects (*Object*), and the analysis time (*Time*). All time measurements show the *analysis time*. That is, Points-to SSA graph construction and analysis setup are not included[2].

The sections *SourceCode* and *ObjectIdentity* show the context-insensitive results for the precision metrics related to the previously mentioned types of client applications. They will be explained below.

In order to avoid taking into account results due to the same set of Java library and JVM start-up classes again and 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 `xsltc1.2` 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.

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 any inter-procedural analysis. The other two are directly related to *method inlining* and *cast safety analysis*.

The *ObjectIdentity* Metrics Suite

The set of metrics presented here is most relevant for client applications of type *ObjectIdentity*.

- *ONode, OEdge*: The Application Object Member Graph (AOMG) is a graph consisting of two node types: object methods $[o, m]$ and object fields $[o, f]$, and three edge types: object call $[o^i, m_p] \rightarrow [o^j, m_q]$, object field store $[o^i, m] \rightarrow [o^j, f]$, object field load $[o^i, f] \rightarrow [o^j, m]$. *ONode* and *OEdge* is the number of nodes and edges in an AOMG where at least one of the participants is an application object member.
- *Heap*: The number of abstract objects referenced by the application object fields. That is, we have summed up the sizes for all points-to sets stored in all application object fields.
- *Enter*: The number of abstract objects entering an application method. That is, we have counted the number of different abstract objects that enter an application method (i.e., out-port values for entry, field load, and call nodes) and summed these up.

The AOMGs are easy to derive since we know the set of abstract objects referenced by the implicit variable *this* in each method (or context), and we know the targets of all member accesses $a.x$. A small number of *OEdge* indicates small *this* value sets as well as a precise resolution of member accesses (relevant in, e.g., reverse engineering of UML *interaction diagrams* [27]).

The *Heap* metric can be seen as the size of the abstract heap associated with the application objects. It is a metric that puts

[2]The longest time we measured was for `xsltc1.2`, which took 172 seconds, including file reading. Graph construction, however, is not at all optimized for speed

all focus on the precision of the memory store operation and is of direct relevance for a number of memory management optimizations (e.g., *side-effect* and *escape* analyses).

Enter focuses on the flow of abstract objects between different parts of a program. A low value indicates a precise analysis that narrows down the flow of abstract objects from one part of the program to another (e.g., *object tracing*).

C. 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*, *ThisSens*, and *ObjSens* are presented in Table II. All results in this, and the following tables, are given as a multiple of the context-insensitive results presented in Table I. For example, for the `sablecc-j` benchmark, the number of unresolved polymorphic calls in analysis *CallSite* is $434 \times 0.81 = 352$, where 434 is the number for the metric *PCall* given in Table I. Tables II-IV use both a simple mean (*average*) and median value (*median*) to report the overall results. The median values are included to reduce the effects of outliers (e.g., `sablecc-j` in Table II), which are over-emphasized in *average*.

First, there is no significant difference between the three approaches for this set of metrics. Second, they only provide slightly better results than the context-insensitive analysis, with one major exception: In the benchmark `sablecc-j`, the number of call graph edges is reduced by 73-74% when using a context-sensitive analysis. Our results are in agreement with those of Lhoták and Hendren, who also explained the outlier `sablecc-j`: They traced this increase in precision to the own *map*-implementation of `sablecc-j`, where different maps store different types of objects, but all maps use the same kind of generic map entry object. Thus, in context-insensitive analysis, the contents of all maps get mixed, and consequently, the analysis cannot compute that methods like `toString()` and `equals()` are called for only some but not all of the maps [9]. Thus, client applications of type *SourceCode* would probably not notice any significant change.

D. The ObjectIdentity Metrics Suite — Results

In this section, we present the results when measuring the precision using the *ObjectIdentity* Metrics Suite. The results are presented in Table III.

First, all three approaches are much more precise than context-insensitive analysis. These results indicate that client applications of type *ObjectIdentity* are likely to benefit from using a context-sensitive analysis. The results for the metric *Heap* show a significant precision improvement (47-59%) 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 59% 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. [7], [8] that this type of client applications can benefit from context-sensitive analysis. Our experiments confirm theses results.

Second, note that the two functional approaches, on average and median, are clearly more precise than *CallSite*. This- and object-sensitivity have in common that the context in which a method is analyzed depends on the value of the implicit `this`-variable. That is, the analysis values of two calls targeting different `this`-sets are never mixed. This is important in object-oriented languages where a large part of all field accesses and calls are targeting `this`. The focus on `this` makes this- and object-sensitivity more precise in the analysis of OO programs than CFA (that was designed for the analysis of functional and imperative programs). This is also in agreement with previous results, where object-sensitivity and the call string approach were compared to each other [8], [9]. However, our use of metrics suites 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 metric *OEdge* where *ObjSens* is significantly 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. The difference in this particular kind of situation was studied in Example 2, Section III-B.

E. Time and Memory Measurements

Table IV shows the number of used contexts 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 metric. 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 independent. Both the time and context measurements are given as a multiple of the context-insensitive results presented in Table I.

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 avarage, only 8% 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 (`antlr`, `ps`, `sablecc-j`, `emma2.0`, and `javacc3.2`) 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 II-B.

	CallSite				ThisSens				ObjSens			
Program	Node	Edge	PCall	Cast	Node	Edge	PCall	Cast	Node	Edge	PCall	Cast
antlr	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98	1.00	1.00	1.00	0.98
javac	1.00	1.00	0.99	0.98	1.00	1.00	0.98	0.98	1.00	1.00	0.98	0.97
javadoc	1.00	1.00	0.99	0.92	1.00	1.00	0.99	0.93	1.00	1.00	1.00	0.92
jython	1.00	1.00	1.00	0.99	1.00	1.00	1.00	0.99	1.00	1.00	1.00	0.99
ps	1.00	1.00	1.00	0.98	1.00	1.00	1.00	1.00	0.99	0.99	1.00	1.00
sablecc-j	1.00	0.27	0.81	0.99	0.99	0.26	0.71	0.98	0.99	0.27	0.71	0.98
soot-c	1.00	0.99	0.97	0.99	1.00	0.99	0.94	1.00	1.00	0.99	0.94	1.00
emma2.0	0.98	0.98	0.97	0.96	0.98	0.98	0.97	0.96	0.98	0.98	0.97	0.96
javacc3.2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
pmd3.2	1.00	1.00	0.96	0.96	1.00	1.00	0.96	0.96	1.00	1.00	0.96	0.95
jess4.5	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
obfusc0.73	1.00	1.00	0.99	0.99	1.00	1.00	0.99	1.00	1.00	1.00	0.99	1.00
xsltc1.2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00	1.00
average	**1.00**	**0.94**	**0.98**	**0.98**	**1.00**	**0.94**	**0.97**	**0.98**	**1.00**	**0.94**	**0.97**	**0.98**
median	**1.00**	**1.00**	**0.99**	**0.99**	**1.00**	**1.00**	**0.99**	**0.99**	**1.00**	**1.00**	**1.00**	**0.99**

TABLE II
RESULTS RELEVANT FOR *SourceCode* CLIENTS

	CallSite				ThisSens				ObjSens			
Program	ONode	OEdge	Heap	Enter	ONode	OEdge	Heap	Enter	ONode	OEdge	Heap	Enter
antlr	0.71	0.26	0.27	0.77	0.70	0.22	0.09	0.70	0.70	0.16	0.09	0.70
javac	0.90	0.73	0.57	0.85	0.71	0.58	0.38	0.39	0.71	0.40	0.35	0.40
javadoc	0.91	0.59	0.56	0.83	0.78	0.42	0.44	0.69	0.80	0.41	0.43	0.68
jython	0.83	0.70	0.23	0.55	0.80	0.68	0.22	0.53	0.79	0.18	0.22	0.53
ps	0.97	0.93	0.10	0.97	0.96	0.87	0.10	0.94	0.96	0.63	0.10	0.94
sablecc-j	0.62	0.25	0.05	0.12	0.63	0.18	0.05	0.37	0.63	0.14	0.05	0.37
soot-c	0.81	0.34	0.93	0.60	0.74	0.26	0.91	0.72	0.74	0.24	0.91	0.72
emma2.0	0.88	0.27	0.39	0.67	0.65	0.15	0.17	0.44	0.65	0.15	0.17	0.44
javacc3.2	0.74	0.09	0.97	0.43	0.73	0.09	0.97	0.42	0.73	0.09	0.97	0.42
pmd3.2	0.90	0.82	0.80	0.81	0.88	0.81	0.80	0.79	0.88	0.80	0.80	0.79
jess4.5	0.90	0.55	0.66	0.80	0.90	0.54	0.36	0.80	0.90	0.36	0.36	0.80
obfusc0.73	0.85	0.63	0.65	0.64	0.84	0.61	0.59	0.63	0.84	0.50	0.58	0.63
xsltc1.2	0.87	0.46	0.66	0.82	0.64	0.30	0.27	0.28	0.64	0.17	0.26	0.28
average	**0.84**	**0.59**	**0.53**	**0.68**	**0.77**	**0.44**	**0.41**	**0.59**	**0.77**	**0.32**	**0.41**	**0.59**
median	**0.87**	**0.55**	**0.57**	**0.77**	**0.74**	**0.42**	**0.36**	**0.63**	**0.74**	**0.24**	**0.35**	**0.63**

TABLE III
RESULTS RELEVANT FOR *ObjectIdentity* CLIENTS

ThisSens also outperforms the other two approaches when it comes to the number of used contexts. It requires, on average, 17.8% fewer contexts than *CallSite* and 44.5% fewer contexts than *ObjSens*. This is a bit surprising since, remembering from Section III-B, the 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 used call-sites, which in turn 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 the two other approaches 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 $a.m(\ldots)$, where *N = sizeOf(Pt(a))*, may require that method m (and all its callees transitively) must be processed N times, one for each abstract object in $Pt(a)$. That is, a single call targeting a large points-to set where $N \gg 1$ may generate a cascade of new methods 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. Scenarios in Java programs where $N \gg 1$ most often involve objects of classes `String` and `StringBuffer`, which can make up more than half of all abstract objects used in an analysis. This explanation is supported by an observed 94% reduction in the analysis time for *ObjSens* on `javacc3.2` when using a class-based name schema for all objects of classes `String` and `StringBuffer`.

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. [8], 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 graph (the method

	CallSite		ThisSens		ObjSens	
Program	Context	Time	Context	Time	Context	Time
antlr	4.87	1.69	3.36	0.97	3.91	4.65
javac	4.59	4.83	4.19	1.56	7.11	5.71
javadoc	4.42	3.04	4.61	1.33	10.04	12.65
jython	4.44	0.98	4.74	1.07	15.58	23.35
ps	9.36	6.23	3.06	0.94	5.37	21.22
sablecc-j	3.22	0.52	3.05	0.47	3.56	1.51
soot-c	6.34	3.64	3.45	1.31	5.20	10.88
emma2.0	3.56	0.96	3.91	0.63	11.63	9.15
javacc3.2	8.21	1.61	8.26	0.91	9.26	24.81
pmd3.2	3.11	1.71	2.88	1.24	5.25	10.53
jess4.5	3.54	2.38	3.16	1.26	5.35	6.85
obfusc0.73	3.57	3.24	2.73	1.23	3.36	3.67
xsltc1.2	6.44	2.90	6.61	1.14	11.67	9.97
average	**5.05**	**2.60**	**4.15**	**1.08**	**7.48**	**11.1**
median	**4.44**	**2.38**	**3.45**	**1.14**	**5.37**	**9.97**

TABLE IV
USED CONTEXTS AND ANALYSIS TIME

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 haven't experienced any sign of exponential behavior in our experiments, we still recommend any non-experimental implementation of this-sensitivity to use a guarded approach (cf. discussion in Section III-B) to ensure a polynomial behavior for any input program.

V. RELATED WORK

A context-insensitive version of the SSA-based simulated execution approach used in this paper was presented before [10]. Our program representation Points-to SSA is closely related to Memory SSA [34], [35]. Memory SSA is an extension to the traditional approach to SSA [11], [12].

The number of papers explicitly dealing with context-sensitive points-to analysis of object-oriented programs is rapidly growing [7]–[9], [13], [36], [37]. 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 [38] and Ryder [3].

Many authors use a call string approach and approximative method summaries to reduce the cost of having multiple contexts [36], [37]. Sometimes, ordered binary decision diagrams (OBDD) are used to efficiently exploit commonalities among similar contexts [9], [13], [34], which allows handling of a very large number of contexts at reasonable memory cost.

Milanova et al. [7], [8] present a technique named object-sensitivity. Object-sensitivity uses the (abstract) receiver object to distinguish different contexts. The object-sensitive and call string approaches were compared in different works [8], [9]. These also show that 1-object-sensitivity scales to programs containing hundreds of classes.

Whaley et al. [13] present a k-call-string based analysis with no fixed upper limit (k) that only takes acyclic call paths into account. Experiments using this approach report reasonable analysis costs but no improved precision compared to object-sensitivity [9], [13].

VI. CONCLUSIONS

In this paper, we present a new context-sensitive approach to points-to analysis where the target context associated with a call site $a.m(\ldots)$ is determined by the pair $(m, Pt(a))$, where $Pt(a)$ 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. [7], [8].

We have experimentally evaluated this-sensitivity by comparing it with two well-known context-sensitive approaches (1-object-sensitivity and 1-CFA). Our measurements show that this-sensitivity is much faster, and requires less memory, than the two other. In fact, it is, on average, only 8% slower than our context-insensitive analysis.

We have used two different 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*, *escape analysis*, and reverse engineering of UML *interaction*

diagrams.

The experiments using the *SourceCode* metrics suite show no significant difference between the three context-sensitive approaches. Furthermore, they only provide slightly better results than the context-insensitive analysis. Our conclusion is that client applications of type *SourceCode* can probably safely avoid the trouble of adding any kind of context-sensitivity to their 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 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 [8], [9]. It seems likely that client applications which require precise information about individual objects and their interaction would benefit from using 1-object-sensitivity or this-sensitivity rather than 1-CFA.

Finally, 1-object-sensitivity and this-sensitivity provide almost identical results in three out of four 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 an order of magnitude slower than this-sensitivity.

REFERENCES

[1] T. Marlowe and B. Ryder, "Properties of data flow frameworks: A unified model," *Acta Informatica*, vol. 28, pp. 121–163, 1990.

[2] F. Nielsen, H. R. Nielsen, and C. Hankin, *Principles of Program Analysis, 2nd Edition*. Springer, 2005.

[3] B. G. Ryder, "Dimensions of precision in reference analysis of object-oriented programing languages," in *International Conference on Compiler Construction (CC'03)*, 2003, pp. 126–137.

[4] M. Sharir and A. Pnueli, *Two Approaches to Interprocedural Data Flow Analysis*, ser. In S. Muchnick and N. Jones, editors, Program Flow Analysis: Theory and Applications. Prentice Hall, 1981.

[5] O. Shivers, "Control-flow analysis of higher-order languages," PhD thesis, Carnegie-Mellon University, CMU-CS-91-145, Tech. Rep., 1991.

[6] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," in *Procedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*, 1997, pp. 108–124.

[7] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to and side-effect analyses for Java," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02)*, July 2002.

[8] ——, "Parameterized object sensitivity for points-to analysis for Java," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 1, pp. 1–41, 2005.

[9] O. Lhoták and L. Hendren, "Context-sensitive points-to analysis: is it worth it?" in *International Conference on Compiler Construction (CC'06)*, ser. LNCS, A. Mycroft and A. Zeller, Eds., vol. 3923. Vienna: Springer, March 2006, pp. 47–64.

[10] J. Lundberg and W. Löwe, "A scalable flow-sensitive points-to analysis," in *Compiler Construction – Advances and Applications, Festschrift on the occasion of the retirement of Prof. Dr. Dr. h.c. Gerhard Goos, Lecture Notes in Computer Science (LNCS)*, 2007.

[11] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[12] S. S. Muchnick, *Advanced Compiler Design Implementation*. San Francisco, California: Morgan Kaufmann Publishers, 1997.

[13] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proceedings of the Conference on Programmimg Language Design and Implementation (PLDI'04)*, June 2004.

[14] "Soot: a Java Optimization Framework," www.sable.mcgill.ca/soot.

[15] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani, "A study of devirtualization techniques for a Java Just-In-Time compiler," in *Procedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, 2000, pp. 294–310.

[16] R. O'Callahan, "The generalized aliasing as a basis for software tools," Ph.D. dissertation, Carnegie Mellon University, December 2000.

[17] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, june 1994.

[18] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*, ser. Prentice Hall Object-Oriented Series. Upper Saddle River, New Jersey: Prentice Hall, 1999.

[19] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[20] K. Tai and J. Daniels, "Interclass test order for object-oriented software," *Journal of Object-Oriented Programming*, vol. 12, no. 2, pp. 18–25, 1999.

[21] A. Milanova, A. Rountev, and B. G. Ryder, "Constructing precise object relation diagrams," in *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, October 2002.

[22] L. R. Clausen, "A Java byte code optimizer using side-effect analysis," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1031–1045, 1999.

[23] B. Blanchet, "Escape analysis for object-oriented languages. Application to Java," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, 1999, pp. 20–34.

[24] J. Bogda and U. Hölzle, "Removing unnecessary synchronization in Java," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, 1999, pp. 35–46.

[25] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff, "Escape analysis for Java," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, 1999, pp. 1–19.

[26] J. Seemann and J. W. von Gudenberg, "Pattern-based design recovery of Java software," in *Proceedings of the International Symposium on Foundations of Software Sngineering (FSE'98)*, 1998, pp. 10–16.

[27] P. Tonella and A. Potrich, "Reverse engineering of the interaction diagrams from c++ code," in *ICSM '03: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2003, p. 159.

[28] S.Mancoridis, B. Mitchell, Y. Chen, and E. Ganser, "Bunch: A clustering tool for the recovery and maintenance of software systems," in *Proceeding of International Conference of Software Maintenance (ICSM'99)*, August 1999.

[29] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, and F. Vokolos, "Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences," in *Proceedings of the International Conference on Software Maintenance (ICSM'05)*, September 2005.

[30] "Ashes Suite Collection," http://www.sable.mcgill.ca/ashes.

[31] "SPEC JVM98 Benchmarks," http://www.spec.org/osg/jvm98.

[32] "DaCapo benchmark suite," http://www.dacapobench.org, 2006.

[33] "The Recoder Homepage," http://recoder.sourceforge.net.

[34] M. Trapp, "Optimerung objektorientierter programme," Ph.D. dissertation, Universität Karlsruhe, December 1999.

[35] M. Trapp, G. Lindenmaier, and B. Boesler, "Documentation of the intermediate representation Firm," Fakultät für Informatik, Universität Karlsruhe, Germany, Tech. Rep., August 2002.

[36] R. Chatterjee, B. Ryder, and W. Landi, "Relevant context inference," in *Symposium on Principles of Programming Languages (POPL'99)*, 1999, pp. 133–146.

[37] E. Ruf, "Effective synchronization removal for Java," in *Proceedings of the Conference on Programmimg Language Design and Implementation (PLDI'00)*, 2000, pp. 208–218.

[38] M. Hind, "Pointer analysis: Haven't we solved this problem yet," in *Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, 2001, pp. 54–61.