Journal of Universal Computer Science, vol. 18, no. 20 (2012), 2851-2878
submitted: 10/8/11, accepted: 6/5/12, appeared: 1/12/12 © J.UCS

Points-to Analysis: A Fine-Grained Evaluation[1]

Jonas Lundberg
(Linnaeus University, Växjö, Sweden
Jonas.Lundberg@lnu.se)

Welf Löwe
(Linnaeus University, Växjö, Sweden
Welf.Lowe@lnu.se)

Abstract: Points-to analysis is a static program analysis that extracts reference information from programs, e.g., possible targets of a call and possible objects referenced by a field. Previous works evaluating different approaches to context-sensitive Points-to analyses use coarse-grained precision metrics focusing on references between source code entities like methods and classes. Two typical examples of such metrics are the number of nodes and edges in a call-graph. These works indicate that context-sensitive analysis with a call-depth $k = 1$ only provides slightly better precision than context-insensitive analysis. Moreover, these works could not find a substantial precision improvement when using the more expensive analyses with call-depth $k > 1$.

The hypothesis in the present paper is that substantial differences between the context-sensitive approaches show if (and only if) the precision is measured by more fine-grained metrics focusing on individual objects (rather than methods and classes) and references between them. These metrics are justified by the many applications requiring such detailed object reference information.

In order to experimentally validate our hypothesis we make a systematic comparison of ten different variants of context-sensitive Points-to analysis using different call-depths $k \geq 1$ for separating the contexts. For the comparison we use a metric suite containing four different metrics that all focus on individual objects and references between them.

The main results show that the differences between different context-sensitive analysis techniques are substantial, also the differences between the context-insensitive and the context-sensitive analyses with call-depth $k = 1$ are substantial. The major surprise was that increasing the call-depth $k > 1$ *did not* lead to any substantial precision improvements. This is a negative result since it indicates that, in practice, we cannot get a more precise Points-to analysis by increasing the call-depth. Further investigations show that substantial precision improvements can be detected for $k > 1$, but they occur at such a low detail level that they are unlikely to be of any practical use.

Key Words: Static program analysis, Points-to analysis, Context sensitivity
Category: D.2.3, D.3.4, F.3.2

1 Introduction

Points-to analysis is a static program analysis that computes precise object reference information by tracking the flow of objects from one part of a program

[1] The article is a contribution to the Forum for Negative Results.
(See http://www.jucs.org/jucs_3_9/why_we_need_an.)

to another. This reference information is an essential input to many types of client applications in optimizing compilers and software engineering.

Optimizing compilers use reference information to perform optimizations such as: *virtual call resolution* to avoid unnecessary dynamic dispatch and facilitate method inlining, *side-effect analysis* to compute the set of objects that may be modified during the execution of a statement, and *escape analysis* to identify method- or thread-local objects to improve garbage collection and to remove synchronization operations.

Activities related to software engineering include: *metric analyses* computing coupling and cohesion between objects, architectural recovery by *class clustering* proposes groupings of classes, either based on coupling and cohesion or directly on reference information. Source code browsers need *forward and backward slices* of a program point for easy navigation, which, in turn, requires reference information. In *software testing*, class dependencies determine the test order. Finally, static *design pattern detection* needs to identify the interaction among participating classes and object instances in order to exclude false positives.

1.1 Introduction to Points-to Analysis

Object-oriented programs consist of classes, and each class contains a set of methods and fields. Methods create and propagate runtime objects, fields capture them. An *abstract object* o is an analysis abstraction that represents one or more run-time objects and in this article each *syntactic creation point* s corresponds to a unique abstract object $o_s \in O$ representing *all* run-time objects created at s in *any* execution of an analyzed program. During analysis, each variable and object field n in the analyzed program is associated with a points-to set $Pt(n) \subseteq O$ which, when the analysis is completed, will be interpreted as: $Pt(n)$ is the set of abstract objects that may be referenced by n.

In Points-to analysis, each method in the program is represented by a graph with two different node types: abstract objects $o \in O$, and reference variables $v \in RV$. Edges represent assignments of reference values and can be considered as channels for *object propagation*. An allocation edge $o \to l$ means that the abstract object o should be propagated to $Pt(l)$. An assignment edge $r \to l$ means that all abstract objects in $Pt(r)$ should be propagated to $Pt(l)$.

Fields introduce a third type of nodes, object fields nodes $[o, f] \in OF$, that via field read/write assignment edges connect the different method graphs. Finally, for each call $l = a.m(v)$, additional edges are added that correspond to assignments of address a, argument v (and return values ret) to the implicit variable *this*, formal parameter p (and receiving variable l). The result is a whole program representation where inter-procedural edges due to calls and field accesses connect the different method (sub)graphs.

Once the program graph is constructed, the analysis computes the object flow from one part of the program to another by merging the values from predecessor nodes repeatedly until a fixed point is reached.

In *context-insensitive* analysis, arguments of different calls to the same target method (subgraph) m get propagated and mixed there. The result of m's analysis is then the merger of *all* calls targeting m and the effects of m itself. Furthermore, the analysis result, i.e., over-approximated return values and heap updates, affect other callees of m.

A *context-sensitive* analysis aims at reducing this over-approximation by partitioning all calls targeting m into a finite number of *call contexts* (cloned method subgraphs). Context-sensitivity gives, in general, a more precise analysis since the arguments of calls targeting the same method do not get mixed when the calls are analyzed in different contexts. This partitioning can be made even more fine grained by taking more than one level (call-depth k) of call history into account. Thus, in theory, a context-sensitive analysis is more precise than a context-insensitive one, and increasing the call-depth $k > 1$ makes it even more precise. The drawbacks of context-sensitivity are the increased memory costs that come with maintaining a number of contexts for each method, and the increased analysis time required to reach a fixed point. Increasing the call-depth comes with an exponential cost in both analysis time and memory.

1.2 Paper Motivation

The question if and when a more costly context-sensitive Points-to analysis using a call-depth $k \geq 1$ actually pays off in practice is not clearly answered.

On the one hand we have the results of [Lhoták and Hendren 2006, 2008] indicating that a more expensive context-sensitive approach to Points-to analysis does not provide any substantial precision improvements[2]. Their results did not show any major difference in precision between different context-sensitive approaches or when using a call-depth $k > 1$, and only slightly better results than the context-insensitive analysis. However, when evaluating the precision they used a rather coarse-grained metric suite focusing on references between source code entities like methods and classes, i.e., in relations that hold for all instances of a class. Typical examples of such metrics are the number of nodes and edges in a call-graph, and the number of non-resolved polymorphic calls. We will from now on refer to this type of precision metric suite as *SourceLevel*.

On the other hand, [Lundberg et al., 2009] reported substantial differences between different context-sensitive approaches when using a more fine-grained metric suite containing four different precision metrics that all focus on different aspects of individual objects and their references. We will from now

[2] Consequently, they entitled one of their papers: "Context-Sensitive Points-to Analysis: Is it worth it?"

on refer to this type of metric suite as *ObjectLevel*. However, the results of [Lundberg et al., 2009] were limited to a call-depth $k = 1$ and, thus, did not indicate if further precision improvements can be expected for the case $k > 1$.

In summary: Previous experiments comparing different approaches to Points-to analysis indicate that: (i) client applications of type *SourceLevel* can probably safely avoid the trouble of adding any kind of context-sensitivity to their analysis, (ii) client applications of type *ObjectLevel* are likely to benefit from using a $k = 1$ context-sensitive analysis. Within this group of analyses ($k = 1$), there is also a trade-off between precison and cost when comparing different context-sensitive approaches. (iii) The question if we can improve the precision for clients of type *ObjectLevel* by increasing the call-depth, or by some other mean partitioning the level 1 call contexts, is still an open issue.

1.3 Hypothesis, Contributions, and Paper Outline

Our hypothesis H is: We have substantial differences between call-depth $k = 1$ and $k > 1$, or when we by some other mean partition the $k = 1$ call contexts into smaller contexts, if (and only if) precision is measured by metrics of type *ObjectLevel*.

The contributions of this paper are the following: (i) We present and evaluate three different context-sensitive approaches using a call-depth $k \geq 1$. (ii) We introduce two new context-sensitive variants that both can be seen as examples of more precise analyses variants achieved by partitioning the $k = 1$ call contexts into smaller contexts. (iii) We make a thorough experimental evaluation of 10 different context-sensitive variants in order to validate or invalidate our hypothesis H. We perform all experiments using a precision metric suite of type *ObjectLevel*, thus emphasizing the difference in precision between different context-sensitive approaches. In addition to precision results, we also present analysis costs, i.e., analysis time and memory.

Our experiments failed to validate hypothesis H since we could not detect any substantial precision improvments when increasing the call-depth beyond $k = 1$. This is a negative result since it indicates that, in practice, we can not get a more precise Points-to analysis by increasing the call-depth beyond $k = 1$, and this paper is therefore a contribution to the *Forum for Negative Results*.

Paper outline: In Section 2, we present a framework for context-sensitive Points-to analyses to provide a uniform presentation of all the analysis variants that we later on evaluate in our exeriments. Section 3 gives a brief outline of our Points-to analysis implementation. Section 4 presents our experiments, including experimental setup, results, and discussion. In Section 5, entitled *Explaining Negative Results*, we describe why we failed to validate hypothesis H. We also present a new modified hypothesis H' and new experiments supporting it. Section 6 presents related work and Section 7 concludes the paper.

2 A Framework for Context-Sensitive Analyses

This section presents a framework for context-sensitive Points-to analyses. We use this framework not only to give a precise definition of different context-sensitive analysis variants, it also limits the scope of our experimental evaluation in Section 4. Our framework is not complete in the sense that it covers every known Points-to analysis. Alternative Points-to analysis approaches outside the scope of our framework will be briefly discussed at the end of Section 2.1. Finally, in Sections 2.2 and 2.3, we discuss and exemplify the major differences between the different framework instances that are used in our experiments in Section 4.

2.1 Seven Families of Context-Sensitive Points-to Analysis

In our framework for context-sensitive Points-to analyses is each instance of the framework defined by a pair (k, AS) where k is the *call context depth* and AS is an *activation schema*. An activation schema $AS(m, s, a, v_1, \ldots, v_l) \mapsto \{r_1, \ldots, r_n\}$ is an abstraction of a call from a call-site $s:\ res = a.m(v_1, \ldots, v_l)$ that is used to associate each call with one or more abstract *activation records* $r_i, i \in [1, \ldots, n]$. Notice, an activation record r_i is an abstraction of an abstract activation frame $(m, s, a, v_1, \ldots, v_l)$ which in turn is an analysis abstraction of the information contained in a run-time stack frame. For example, given an activation scheme $AS(m, s, a, v_1, \ldots, v_l) \mapsto \{s\}$, s is an abstraction of $(m, s, a, v_1, \ldots, v_l)$ indicating that we in this case partition all calls targeting m based on the call-sites (return addresses). Each partition defines a context; a called method m is analyzed separately for each such context. Hence, an activation schema defines activation records and the activation records partition calls targeting method m into a number of contexts.

The partitioning of calls can be made even more fine grained by using a finite sequence $[r_1, \ldots, r_k]$ of activation records corresponding to the k top-most activation frames on the (analysis abstraction of the) run-time activation stack. That is, for each instance (k, AS) in our framework, we can define the target contexts ctx_m of a call to be a pair $(m, [r_1, \ldots, r_k])$ where $r_j, j \in [1, \ldots, k]$, is the result of applying the activation schema AS on the j-th enclosing call. In the special case of $k = 0$ this degenerates to a context-insensitive analysis where all calls targeting method m are analyzed in the same context $(m, [\,])$.

Let m' be a method containing a call-site $s:\ res = a.m(v_1, \ldots, v_l)$, $ctx_{m'} = (m', [r'_1, \ldots, r'_k])$ be a context in which m' is analyzed; let $AS(m, s, a, v_1, \ldots, v_l) \mapsto \{r_1, \ldots, r_n\}$ be the activation records associated with the call site s. Then, the set of target contexts CTX_m under which m will be analyzed next is defined as:

$$CTX_m = \{(m, [r_i, r'_1, \ldots, r'_{k-1}]) \mid i \in [1, \ldots, n]\}.$$

That is, for each new call, the $k-1$ top-most activation records on the activation stack and data from the current call, are used to define new contexts with a call-depth k.

Each framework instance (k, AS) defines a unique version of context-sensitive Points-to analysis, and each unique activation schema AS defines a *family* k-AS of context-sensitive analyses. We distinguish the following basic activation schemas AS for a given call $s: \ res = a.m(v_1, \ldots, v_l)$ where $Pt(a) = \{o_1, \ldots, o_m\}$:

- *CFA*: $AS \mapsto \{s\}$: Calls from the same call-site s are mapped to the same record, i.e., the stack frames are abstracted by their return address.
- *ThisSens*: $AS \mapsto \{Pt(a)\}$: Calls targeting the same *points-to set* $Pt(a)$ are mapped to the same record, i.e., the stack frames are abstracted by the analysis result for the implicit parameter *this*. Static calls are handled context-insensitively.
- *ObjSens*: $AS \mapsto \{o_1, \ldots, o_m\}$: Calls targeting the same receiving abstract object $o_i \in Pt(a)$ are mapped to the same record, i.e., the stack frames are abstracted by the *elements of* the analysis result for the implicit parameter *this*. Static calls are handled context-insensitively.
- *ThisArg*: $AS \mapsto \{[Pt(a), Pt(v_1), \ldots, Pt(v_l)]\}$: Calls targeting the same points-to value $Pt(a)$, and having the same arguments $Pt(v_i)$, are mapped to the same record. For static calls, $Pt(a)$ is undefined and ignored.
- *ObjArg*: $AS \mapsto \{[o_1, Pt(v_1), \ldots, Pt(v_l)], \ldots, [o_m, Pt(v_1), \ldots, Pt(v_l)]\}$: Calls targeting the same receiving abstract object $o_i \in Pt(a)$, and having the same arguments $Pt(v_i)$, are mapped to the same record. For static calls, $o_1, \ldots, o_l$ are undefined and ignored.
- *ThisCFA*: $AS \mapsto \{[s, Pt(a)]\}$: Calls from the same call site s targeting the same points-to value $Pt(a)$ are mapped to the same record. Static calls are handled using *CFA*.
- *ObjCFA*: $AS \mapsto \{[s, o_1], \ldots, [s, o_m]\}$: Calls from the same call site s targeting the same receiving abstract object $o_i \in Pt(a)$ are mapped to the same record. Static calls are handled using *CFA*.

CFA defines the family of well-known k-CFA analyses distinguishing context by the top k sequence of call-sites [Shivers, 1991] and *ObjSens* the family of well-known k-object-sensitive analyses [Milanova et al., 2005]. *ThisSens* (k-this-sensitivity) is rather new [Lundberg et al., 2009]. *CFA*, *ObjSens*, and *ThisSens* will be presented in more detail in Section 2.2. k-*ThisArgs*, k-*ObjArgs*, k-*ThisCFA*, and k-*ObjCFA* are to our knowledge new and will be discussed in more detail in Section 2.3.

In general, the different variants of Points-to analysis are partially ordered regarding their theoretical precision: (i) All context-sensitive analysis variants are more precise than context-insensitive Points-to analysis. (ii) In each family k-*AS*, the analysis i-*AS* is more precise than j-*AS*, iff $i > j$. (iii) For each k, k-*ThisArgs* (k-*ObjArgs*) is more precise than k-*ThisSens* (k-*ObjSens*). (iv) For each k, k-*ThisCFA* (k-*ObjCFA*) is more precise than both k-*ThisSens* (k-*ObjSens*) and k-*CFA*.

As an extension to the common framework by [Grove et al., 1997], we present this precision ordering in a semi-lattice notation in Figure 1 where edges indicate an "is more precise in theory" relationship. The gray coloring used in the picture is just a simple way of separating the different families k-*AS* from each other. A general ordering of precision cannot be given for all variants. We therefore assess their precision in experiments presented in Section 4.

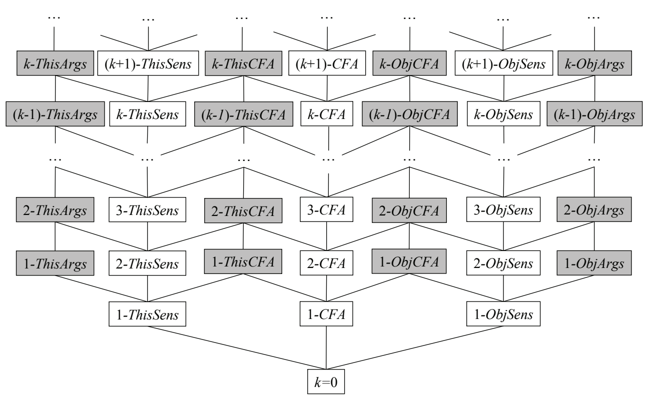

Figure 1: Partial ordering of precisions of different analysis families.

Finally, our framework only takes two dimensions (k, AS) into account and is not complete in the sense that it covers every known Points-to analysis. These dimensions, (k, AS), are orthogonal to other dimensions frequently exploited in context-sensitive analysis and, for brevity of presentation, we do not discuss them in detail. They include: 1) Runtime object abstractions. We used a fixed name schema where each *syntactic object allocation site s* corresponds to a unique abstract object o_s. Alternative approaches are very much possible: [Grove et al., 1997, Tip and Palsberg, 2000] are more coarse-grained, [Milanova et al., 2005, Grove et al., 1997, Smaragdakis et al., 2011] and [Lhoták and Hendren 2006, 2008] are (in certain experiments) more fine-grained. 2) We used an arbitrary but fixed call-depth k whereas other made k adaptable but bounded [Whaley and Lam, 2004, Zhu and Calman, 2004]. 3) We treat all parts of the program uniformly. Demand-driven approaches where certain parts are analyzed with higher precision are also possible [Sridharan and Bodik, 2006].

2.2 Context-Sensitivity by Example

In what follows we will use a simple example to show the effects of using different types of context-sensitivity in a Points-to analysis. This section also serves to give a short introduction to three different analysis techniques (*CFA*, *ObjSens*, *ThisSens*) that was defined in our framework, and that will be used in our experiments in Section 4.

Example
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:
$Pt(a_2) = \{o_a^1, o_a^2\},\ Pt(v_2) = \{o_v^2\}$
$r_2 = a_2.m(v_2)$
Call 3:
$Pt(a_3) = \{o_a^3\},\ Pt(v_3) = \{o_v^3\}$
$r_3 = a_3.m(v_3)$

We have three different calls targeting the same method m, which just returns the provided argument. Each call has a target expression a_i with value $Pt(a_i)$ and an argument v_i with value $Pt(v_i)$.

In a context-insensitive analysis, the arguments of all three calls get mixed, and the resulting return values are a merger of all calls:

$$Pt(r_1) = Pt(r_2) = Pt(r_3) = \{o_v^1, o_v^2, o_v^3\}.$$

The two traditional approaches to define a context are referred to as the *call string* approach and the *functional* approach [Sharir and Pnueli, 1981]. The call string approach (k-*CFA*) partions all calls targeting m based on their call-sites. In the example above, when using 1-*CFA*, we have three different call-sites making a call to method m, each defining a separate call context. Thus, no mixing of the three calls occur and each call is handled separately:

$$Pt(r_1) = \{o_v^1\}, \quad Pt(r_2) = \{o_v^2\}, \quad Pt(r_3) = \{o_v^3\}.$$

Functional approaches use some abstraction of the call site's actual parameters to distinguish different contexts [Sharir and Pnueli, 1981, Grove et al., 1997]. [Milanova et al., 2002, Milanova et al., 2005] presented a functional approach designed for object-oriented languages referred to as *object-sensitivity* (k-*ObjSens*). It distinguishes contexts for a call site $a.m(\ldots)$ by analyzing the target method m separately for each abstract object in the target expressions a.

Returning to the example, we notice that Call 1 and Call 2 with target expressions a_1 and a_2 respectively, have points-to sets $Pt(a_1)$ and $Pt(a_2)$ that both contain the abstract object o_a^1. Thus, in an object-sensitive analysis (1-*ObjSens*), both Call 1 and Call 2 target the context $(m, [o_a^1])$, and the return values of these two calls get mixed:

$$Pt(r_1) = \{o_v^1, o_v^2\}, \quad Pt(r_2) = \{o_v^1, o_v^2\}, \quad Pt(r_3) = \{o_v^3\}.$$

Call 3 is targeting a separate context $(m, [o_a^3])$ and no mixing occurs.

[Lundberg et al., 2009] presented a modified version of object-sensitivity where the target context associated with a call site $a.m(\ldots)$ is determined by a pair $(m, Pt(a))$ with $Pt(a)$ the points-to set of the target expression a. It is called *this-sensitivity* (k-*ThisSens*) since it distinguish contexts of a method by the analysis values of its implicit variable *this*.

In a this-sensitive analysis (1-*ThisSens*), in our example, all three calls target different contexts: $(m, [\{o_a^1\}])$, $(m, [\{o_a^1, o_a^2\}])$, and $(m, [\{o_a^3\}])$. Thus, no mixing occurs and we get the same result as in the call string (1-CFA) approach:

$$Pt(r_1) = \{o_v^1\}, \quad Pt(r_2) = \{o_v^2\}, \quad Pt(r_3) = \{o_v^3\}.$$

In this very simple example we found that 1-CFA and 1-this-sensitivity have similar precision, both are slightly more precise than 1-object-sensitivity, and all three context-sensitive approaches are more precise than the context-insensitive analysis. However, in general, none of the context-sensitive approaches is strictly more precise than the others and we need experiments to assess their precision.

2.3 Argument-Sensitivity and Combined Approaches

Both *ThisArgs* and *ObjArgs* use the call argument set $Pt(v_i)$ to identify a context for a call $a.m(v_1, \ldots, v_n)$. We consider k-*ThisArgs* and k-*ObjArgs* as a *functional refinement* to k-*ThisSens* and k-*ObjSens*, respectively, where we, inspired by [Ryder, 2003], have added a new "dimension of analysis precision", denoted *argument-sensitivity*. However, similar ideas have been used in the Cartesian Product Algorithm (CPA) of [Agesen, 1995] and in the Simple Class Set algorithm (SCS) [Grove et al., 1997]. They both use the *types* of the arguments to identify a context in their call graph construction algorithms.

Our combined approaches *k-ThisCFA* (*k-ObjCFA*) are a combination of *k-CFA* and *k-ThisSens* (*k-ObjSens*) that distinguish contexts by taking both call-site and implicit *this*-parameter into account.

Adding argument-sensitivity to *k-ThisSens* (*k-ObjSens*), or combining *k-CFA* with *k-ThisSens* (*k-ObjSens*) can be seen as an improvement of *k-ThisSens* (*k-ObjSens*) where we have further partitioned each context into a number of "smaller" contexts by separating them due to their arguments or call-sites as well[3]. The fact that each context in both these approaches can be mapped (in a many-to-one mapping) to a unique context in each of the constituent approaches, makes both approaches strictly more precise than each of its constituents (*k-ThisSens*, *k-ObjSens*, or *k-CFA*) .

3 Our Points-to Analysis Implementation

In this section, we outline our Points-to analysis implementation. Sections 3.1 and 3.2 briefly summarize previous work [Lundberg et al 2006, 2009] and are included to make the presentation complete. This whole section can be skipped at a first reading.

3.1 Program Representation

Each method m is represented by a *method graph* g_m. Our Points-to analysis uses a program representation, Points-to SSA, in Static Single Assignment (SSA) form [Cytron et al., 1991, Muchnick, 1997] where nodes correspond to operations and local variables v are resolved to dataflow edges connecting the uniquely 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 the using operations. Join-points in the control flow where several definitions may apply are modeled with special ϕ-operation nodes.

Another feature in Points-to SSA is the use of memory edges to explicitly model dependencies between different memory operations. An operation that may change the memory defines a new memory value and operations that may access this updated memory use the new memory value. Thus, memory is considered as data and memory edges have the same semantics, including the use of ϕ-operations at join-points, as def-use edges for other types of data. The introduction of memory 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 an abstraction of the semantics of the program.

[3] And vice versa, as an improvement of *k-CFA* where we have partitioned each context into a number of "smaller" contexts by separating them due to their implicit *this*-parameter as well.

3.2 Analysis Algorithms

Our dataflow analysis algorithm, called *simulated execution*, is an abstract interpretation of the program that mimics an actual execution: starting at one or more entry methods, it analyzes the operations of a method in execution order, interrupts this analysis when a call operation occurs to follow the call, continues analyzing the potentially called methods, and resumes with the calling method later when the analysis of the called method is completed. 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[4], which handle the transition from one method to another (presented in the next section).

Flow-sensitivity is a concept that is frequently used, but there is no consensus as to its precise definition [Marlowe et al., 1995]. Informally, an analysis is flow-sensitive if it takes control-flow information into account [Hind, 2001]. Many people also require the use of so-called *strong* (or *killing*) updates as a criteria for flow-sensitivity [Ryder, 2003].

Our SSA-based analysis has local (intra-procedural) flow-sensitivity in the strictest sense since our use of an SSA representation incorporates the def-use information needed to identify all places where strong updates of local variables are possible. That dataflow analysis on an SSA-based representation implies local flow-sensitivity has been demonstrated by [Hasti and Horwitz, 1998].

Furthermore, our simulated execution based analysis has a global (inter-procedural) flow-sensitivity in a more general sense since a memory accessing operation (call or field access) $a_1.x$ will never be affected by another memory access $a_2.y$ 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 used by [Grove et al., 1997, Milanova et al., 2005, Lhoták and Hendren, 2008, Whaley and Lam, 2004]. This statement was verified in experiments by [Lundberg and Löwe, 2007].

3.3 Transfer Functions of Calls

The transition from the processing of one method to another is handled in operations of type $MCall^{m,s}$, i.e., monomorphic calls. Algorithm A1 gives the semantics of a $MCall^{m,s}$ operation which handles a call from a call site $s : r = a.m(v_1, \ldots, v_l)$.

The algorithm makes use of three concepts that were introduced in our framework presentation in Section 2.1: abstract *activation records* r, *contexts* ctx, and an *activation stack* that keeps track of currently used activation records. The

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

method $AS : [m, s, a, v_1, \ldots, v_l] \mapsto \{r_1, \ldots, r_n\}$ returns a set of activation records for each call. It is the implementation of this method that determines which activation schema (i.e., family k-AS of context-sensitive analyses) to use.

A 1 $MCall^{m,s} : [a, v_1, \ldots, v_l] \mapsto ret$

```
Record[ ]  records = AS(m, s, a, v_1, ..., v_l)
ret = ∅
for each r ∈ records do
   activation_stack.push(r)
   Context ctx = getContextFor(m, activation_stack.top(k))
   this = ctx.getThis()
   ret = ret ∪ simulate_call_execution(ctx, m, this, v_1, ..., v_l)
   activation_stack.pop()
end for
return ret
```

Algorithm A1 processes one activation record r at a time and starts by pushing the record on the activation stack. The first k top elements of the activation stack are then used to define the context of the upcoming call. We assume that each context *ctx* is aware of the corresponding points-to value for the implicit variable *this*. In short, it is a singleton abstract object set $\{o_i\}, o_i \in Pt(a)$ for the object-sensitive analyses, and the whole set $Pt(a)$ for context-insensitive, CFA, and this-sensitive analyses. This information is embodied in the assignment $this = ctx.getThis()$ that we use to simplify the notations.

3.4 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 [Soot]. We then use the *Shimple* format provided by *Soot* as the starting point to construct the SSA-based graphs for the individual methods.

We use stubs to simulate the behavior of the most frequently used native methods. We handle exceptions, threads, and methods in the Java class library dealing with array manipulation (e.g., `java.lang.System.arraycopy`) in a special but conservative way. Our analysis implementation is currently incomplete in the following sense: (1) It does not handle all features related to dynamic class loading and reflection correctly. To our knowledge, no feasible approach to handle these features is known. (2) Native methods returning `String` and `StringBuffer` objects are treated like allocation sites of `String` and `StringBuffer` objects, respectively.(3) Native methods for which we have no specific stub are not handled correctly. They all return ∅ and are considered to be side-effect free.

4 Experiments

Out of all possible variants *k-AS* of different context-sensitive analyses covered by our framework, we decided to evaluate the following eleven instances:

- A context-insensitive variant named *Insens* where $k = 0$.
- Three versions ($k = 1$, $k = 2$, $k = 3$) of *k-ThisSens*.
- Two versions ($k = 1$, $k = 2$) of *k-ObjSens*.
- Three versions ($k = 1$, $k = 2$, $k = 3$) of *k-CFA*.
- A single version ($k = 1$) of *k-ThisArgs*.
- A single version ($k = 1$) of *k-ThisCFA*.

Insens is our baseline analysis against which all other analyses will be compared. The eight versions of *k-ThisSens*, *k-ObjSens*, and *k-CFA* should validate our hypothesis that we, when using a more fine-grained precision metric suite, should be able to find differences between different context-sensitive analysis families, and to find differences between call-depth $k = 1$ and $k > 1$.

Finally, *1-ThisArgs* and *1-ThisCFA* are included to evaluate the effect of adding argument-sensitivity to an existing analysis (*1-ThisSens*) and to evaluate the effect of combining two well-known approaches (*1-ThisSens* and *1-CFA*) to get a new analysis that is strictly more precise (see Section 2.3).

4.1 The *ObjectLevel* Metric Suite and Other Used Metrics

The reference information that can be extracted from a program using static Points-to analysis is most often used as input to different *client applications*. Our experiments do not target any specific client application that requires reference information. We have chosen to use a set of general precision metrics relevant for a number of different client applications. However, our metric suite, denoted *ObjectLevel*, has a focus on *individual objects* and references to individual objects. We have two reasons for this. First, previous studies (e.g., [Milanova et al., 2005, Lhoták and Hendren, 2008]) evaluating different context-sensitive analyses have often used metrics based on *source code entities* and their relations (e.g., call graphs and resolved polymorphic calls). It turns out that metrics based on this type of information are rather insensitive to the kind of context-sensitive Points-to analysis that is used, and they are therefore not expected to provide any relevant differences between different variants of context-sensitive analyses. This standpoint was presented and motivated by experiments [Lundberg et al., 2009].

Secondly, there are a number of client applications that, in order to be meaningful, require precise information about individual objects and their interactions. Examples are *side-effect analysis* that computes the set of object fields that may be modified during the execution of a statement [Milanova et al., 2002, Milanova et al., 2005, Clausen, 1999], *escape analysis* identifying method-local (or thread-local) objects to improve garbage collection (and to remove synchronization operations) [Blanchet, 1999, Choi et al., 1999], *Memory leak debugging* that identify references preventing garbage collection [Whaley and Lam, 2004], static *design pattern detection* to identify the interaction among possible participating objects [Seemann and von Gudenberg, 1998], reverse engineering of UML *interaction diagrams* [Tonella and Potrich, 2003], and architectural recovery by *class clustering* to avoid erroneous groupings of classes [Mancoridis et al., 1999, Salah et al., 2005].

In order to avoid taking into account the effect of the same set of Java library and Java Virtual Machine (JVM) start-up classes in all experiments, we decided to use the following method when applying our metric suite on the results of 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 `javac` are all those classes having a name starting with `com.sun.tools`. 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 *ObjectLevel* metric suite consists of the following four metrics:

- *Node, Edge*: The Application Object Call Graph (AOCG) is a graph consisting of object methods $[o, m]$ (nodes) and object method calls $[o^i, m_p] \rightarrow [o^j, m_q]$ (edges). *Node* and *Edge* are the number of nodes and edges in an AOCG where at least one of the participants is an application object method.

-*Heap*: The number of abstract objects referenced by the application object fields. That is, we have summed up the sizes of 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., call arguments and return values from field reads and calls) and summed these up.

A small number of *Edge* indicates small *this* value sets and precise resolutions of member accesses (relevant, e.g., in reverse engineering of UML *interaction diagrams*). The *Heap* metric can be seen as the size of the abstract heap associated with the application objects. It is a metric that puts 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 analysis*. *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*.

	General				*ObjectLevel*			
Program	Class	Method	Object	Time[s]	Node	Edge	Heap	Enter
antlr	420	2,222	4,716	5.49	7,274	328,623	167,255	51,118
chart	1,002	6,265	13,893	72.02	15,312	130,862	32,884	384,413
eclipse	889	5,299	10,419	24.03	28,955	444,889	18,127	218,290
fop	436	2,094	4,375	3.25	5,397	82,023	42,980	21,441
javac1.3	490	3,424	5,343	44.17	28,301	529,892	109,332	416,211
javadoc	606	3,423	6,231	47.50	32,952	514,587	202,014	401,888
jython	677	4,778	8,688	159.25	62,315	3,913,594	374,934	752,680
ps	571	2,867	5,400	5.45	12,359	601,162	337,790	134,445
sablecc-j	995	5,941	8,697	28.67	26,321	475,326	149,467	452,581
soot-c	933	4,044	6,270	23.28	20,188	1,044,727	85,332	387,403
emma	856	4,904	10,070	208.81	55,797	1,371,488	569,312	783,392
javacc	311	2,136	8,507	6.80	9,994	531,744	24,938	81,812
jess	364	1,825	3,976	6.59	6,953	403,893	74,582	47,556
pmd	556	3,320	5,301	5.59	16,108	261,646	43,883	95,278

Table 1: Benchmark information and context-insensitive results.

4.2 Experimental Setup

We have used a benchmark containing 14 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 part of the table are taken from well-known test suites[5]; 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 3.2GHz under Linux x86-64, kernel 2.6.22.1).

Table 1 also contains the results of 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*). *Class* and *Method* are computed by our context-insensitive analysis *Insens*; then *Object* is the number of object allocation sites found in the used classes. Column (*Time*) lists the analysis time, i.e., the time needed to perform a context-insensitive points-to analysis. The times required for Points-to SSA graph construction and

[5] Ashes Suite: `http://www.sable.mcgill.ca/ashes`, SPEC JVM98: `http://www.spec.org/osg/jvm98`, DaCapo suite: `http://www.dacapobench.org`.

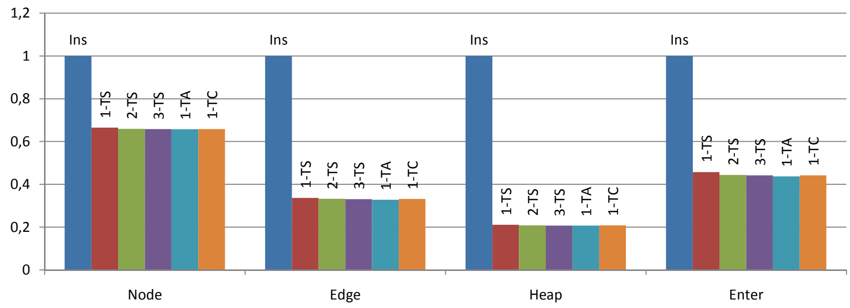

Figure 2: Precision results for *This*-related analyses.

analysis setup are not included[6]. The section *ObjectLevel* in Table 1 shows the context-insensitive results for the precision metrics that we introduced earlier.

4.3 *This*-Related Analyses – Precision Results

In this section, we present the first set of results when measuring the precision using our *ObjectLevel* metric suite. Figure 2 shows the results related to the *this*-family $k = 1, 2, 3$ (`1-TS`, `2-TS`,`3-TS`) of analyses, as well as the two variants *1-ThisArgs* and *1-ThisCFA* (`1-TA`,`1-TC`).

Each bar in the chart shows the average precision compared to the context-insensitive results for a given metrics. More precisely, for a given metrics m_v^p computed for a program p using analysis v, we compute the average value:

$$m_v = \frac{1}{|P|} \sum_{p \in P} \frac{m_v^p}{m_{ins}^p}$$

where *ins* refers to the context-insensitive results and P is the set of programs in our benchmark. For example, the result 0.65 for `1-TS` in metrics *Node* indicates that the Application Object Call Graph (on average) contains 35% fewer nodes when computed using *1-ThisSens* compared to our context-insensitive analysis *Insens*.

First, all three *k-ThisSens* variants (`1-TS`, `2-TS`, `3-TS`) are much more precise than the context-insensitive analysis `Ins`. These results indicate that client applications focusing on individual objects are likely to benefit from using a context-sensitive analysis. For example, the results for the metrics *Heap* show a substantial precision improvement (about 79% on average) when using a context-sensitive analysis, a result that indicates a considerably improved precision in

[6] The longest graph construction time was measured for `Chart` (48.2 seconds) mainly (to 92%) spent on file reading and in third party components (Soot).

the handling of memory store operations. That memory related client application can benefit from this type of context-sensitive analysis was also shown by [Milanova et al., 2002, Milanova et al., 2005] and [Lundberg et al., 2009], both using concrete memory related client applications (side-effect analysis and escape analysis, respectively). Our experiments confirm those results.

Second, and more important to this paper, there is no substantial difference between `1-TS` and `2-TS`, `3-TS`. The two, from a theoretical view-point, more precise approaches (`2-TS`, `3-TS`) only provide slightly better results than `1-TS`. When comparing `1-TS` with `3-TS` we only find an average precision gain for `3-TS` of 1-3% for all four metrics[7]. Thus, increasing the call-depth k for the $This$-family does not have a substantial effect on the precision when evaluated using our *ObjectLevel* metric suite.

The bars in Figure 2 annotated with `1-TA` and `1-TC` present the results of our new approaches *1-ThisArgs* and *1-ThisCFA*. When comparing these two with *1-ThisSens*, we find an increased precision on certain individual benchmarks. For example, *Edge* in `sablecc-j` and *Enter* in `chart` and `emma` improve by about 15% when using the argument-sensitive approach `1-TA`, and *Enter* in `chart` and `emma` improves by about 13% when using the combined approach `1-TC`. However, on average, the overall effect when using these two improvements of *1-ThisSens* is rather insignificant, between 1-4% for all four metrics.

4.4 ObjSens and CFA – Precision Results

Figure 3 shows the results related to the *object*-family of analyses for $k = 1, 2$ (`1-OS`,`2-OS`), and the *CFA*-family of analyses for $k = 1, 2, 3$ (`1-CFA`,`2-CFA`,`3-CFA`). During these experiments we run into out-of-memory problems when analyzing certain benchmark programs (`javadoc, ps, soot-c` in `3-CFA`, `chart`, `jython`, `ps`, `emma` in `2-OS`) although allowing the JVM to use as much as 5.5GB of memory. In order to make a fair comparision between different k values within the two families k-CFA and k-ThisSens we have simply removed `javadoc, ps, soot-c` when computing the averages for `1-CFA,2-CFA`, and `jython`, `ps`, `emma` when computing the averages for `1-OS`, in Figure 3. However, this approach introduces an error when comparing two analyses belonging to different families. For example, the bar charts for the *Heap* metric indicates that `3-CFA` is slightly more precise than `2-OS`. This is actually not the case if we should have restricted the *Heap* metric computation to only those programs that could be analyzed by both `2-OS` and `3-CFA` without any memory problems.

Apart from memory problems, the results for the *object*-family (`1-OS`,`2-OS`) follow the same pattern as the *this*-family results: the more precise approach

[7] Throughout this paper we use *percentages* rather than *percentage points* when we compare different experimental results.

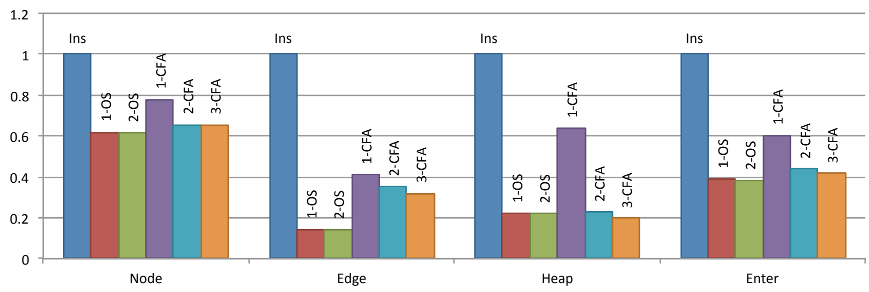

Figure 3: Precision results for k-*ObjSens* and k-*CFA*.

2-OS only provides slightly better results than 1-OS, on average between 1-2% for all four metrics when computed on a comparable set of benchmark programs.

A comparison of Figures 2 and 3 shows that this-sensitivity and object-sensitivity provide almost identical results in three out four metrics. The major difference is in the metrics *Edge* where object-sensitivity is much more precise. This deviation was explained in [Lundberg et al., 2009] by a difference in the handling of memory load operations via the implicit variable `this`. However, the *Edge* improvement for object-sensitivity comes at the price of being more than an order of magnitude slower than this-sensitivity, see Section 4.5, or not even completing analysis due to out-of-memory problems, see discussion above.

The CFA-family stands out from the others as the only one where we can detect a substantial precision increase when increasing the call-depth k. It is most pronounced in the *Heap* metrics where we can detect a 64% reduction in the number of objects referenced by the fields. Thus, increasing the call-depth k when using the CFA-family of analyses is likely to have a substantial effect in memory related client applications. The precision increase for 2-CFA compared to 1-CFA is also noticeable by the other three metrics (*Node* 16%, *Edge* 14%, *Enter* 27%). The difference between 2-CFA and 3-CFA is less pronounced (*Node* 1%, *Edge* 9%, *Heap* 14%, *Enter* 5%).

Finally, the increased precision detected when $k > 1$ still does not make CFA-family more precise than the other two families. For example, 2-TS and 3-TS have a similar precision as 2-CFA and 3-CFA, but the former come with a much lower analysis cost, see Section 4.5.

4.5 Performance Results

Figure 4 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

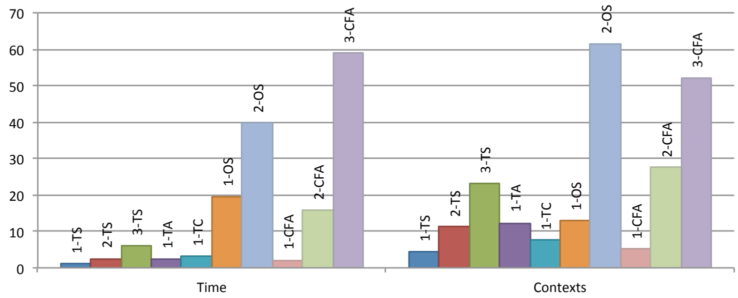

Figure 4: Analysis costs in time and contexts (memory) given as a multiple of the context-insensitive results presented in Table 1.

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 $\Theta(N)$, which is true for our implementation and most other implementations that we know of. Both metrics (*Time* and *Contexts*) are once again given as a multiple of the context-insensitive results presented in Table 1. For example, the `1-OS` results 19.5 (*Time*) and 12.8 (*Contexts*) indicate that `1-OS` requires 19.5 times longer analysis time, and 12.8 times the number of contexts, than the context-insensitive analysis.

The first thing to notice is that the this-family is much faster in general, and requires fewer contexts, than the other two. This is in agreement with results for $k = 1$ presented by [Lundberg et al., 2009]. They also gave an explanation: 1) A much lower number of contexts reduces the processing needed to reach a fixed point, and 2) Object-sensitivity may associate a monomorphic call-site $a.m(..)$ with more than one context. Thus, a call where $|Pt(a)| \gg 1$ may require that method m and all its callees transitively are processed $|Pt(a)|$ times. This problem does not occur in this-sensitivity and CFA, where each call always targets a single context. The time measurements also show that *1-ThisSens* is, on average (median), only 11% (17%) slower than our context-insensitive baseline analysis *Insens*, a number that is likely to be accepted by client applications that can make use of the improved precision.

Secondly, more precise approaches where $k > 1$ come with substantial analysis costs. This is most pronounced for `2-OS` (and `3-CFA`) that on average requires 61 (52) times the contexts, and 40 (59) times longer analysis time, than our context-insensitive analysis. These numbers should probably have been even higher, as we excluded those benchmark programs where `2-OS` and `3-CFA` run out of memory. The cost penalties for *2-ThisSens* (`2-TS`) and our two new approaches *1-ThisArgs* (`2-TA`) and *1-ThisCFA* (`1-TC`) are not as pronounced. They require about twice as much time, and about 2-3 times the number of contexts of *1-ThisSens*. However, the gain in precision for these three approaches is rather insignificant.

5 Discussion: Explaining Negative Results

Our experiments using the *ObjectLevel* metric suite indicate that theoretically more precise analyses with call-depth $k > 1$, and our combined approaches, in practice hardly generate any more precise results at all. These results are contradicting hypothesis H presented in Section 1.3.

Neither disproved nor supported by our experiments in Section 4, 2-this-sensitivity is for example more precise than 1-this-sensitivity. The problem is that this precision improvement is not detected by the *ObjectLevel* metrics suite we have used in these experiments. In order to show improved precision experimentally, we need even more fine-grained metrics to be able to detect the differences. Let us assume that each analysis variant v^p produces results on a certain detail level p, and each metric m^q measures the precision on a certain detail level q. A new revised hypothesis H' can then be expressed as:

H': We can detect a substantial difference in precision when comparing two analysis variants v^p and v^q (assuming v^q more precise than v^p) *iff* we use a metric at detail level q.

Previous experiments provide support for this new hypothesis: Counting nodes and edges in an ordinary call-graph are metrics on *source code level*. The context-insensitive analysis *Insens* works on that level. Hence, theoretically more precise analyses like *1-ThisSens* and *1-ObjSens* where the analysis of a method m is further partitioned into a number of contexts $c_m^1, c_m^2, ...$ will in practice not give any substantially better result. This conclusion was supported in experiments by [Lhoták and Hendren 2006, 2008] and [Lundberg et al., 2009].

Moreover, metrics *Node* and *Edge* in an object call-graph are metrics on *object level*. The *1-ObjSens* analysis works on object-method level. Hence, by using object call-graph based metrics we can detect a substantial precision difference between *1-ObjSens* and the more coarse-grained context-insensitive analysis *Insens* whereas a more precise analysis like *2-ObjSens* where the analysis of an object-method $[o^i, m]$ is further partitioned into a number of contexts $c_m^{i1}, c_m^{i2}, ...$ will not give any substantially better result. These conclusions were supported by experiments presented in [Lhoták and Hendren, 2008] and [Lundberg et al., 2009], and in Section 4.

Finally, a very fine-grained precision metric is likely to show a substantial precision improvement when using a more precise analysis variant. However, such a metric only makes sense if it corresponds to a meaningful client application where the higher precision can make a difference. In the following section we will introduce a very fine-grained precision metric that can be used to detect precision differences between analyses variants using a call-depth $k > 1$. This metric serves to support our revised hypothesis H' and to explain our negative results.

5.1 A Context-graph based Precision Metric

To measure an analysis' capacity to correctly resolve calls at an appropriate detail level for a given analysis $k - AS$, we introduce context call-graphs. A *context call-graph*, or briefly *context-graph*, is a directed graph with contexts c_m as nodes and detected calls from one context c_m to another c_n as edges $c_m \to c_n$. A context-graph is implicitly generated whenever a context-sensitive analysis is executed: A method m is always analyzed in a specific context c_m and method calls make the analysis jump from one context c_m to another c_n. Each such jump is represented by an edge $c_m \to c_n$ in the context-graph.

The advantage of using the context-graph as a basis for computing precision metrics is that its detail level (contexts) is always appropriate for any given analysis $k - AS$. The disadvantage is that it is difficult to compare two different analyses using the context-graph since each analysis uses a different context definition. In order to compare different analyses using a context-graph based metric, we need some kind of normalization.

The *completeness* c for a directed graph with n nodes and e edges is often defined as $c = e/n^2$. It is a normalized metric relating the number of actual edges e with the number of possible edges n^2. A reduced completeness of context-graphs when comparing two *comparable*[8] analyses, for example *1-ThisSens* with *2-ThisSens*, indicates that the more precise analysis (*2-ThisSens*) has detected that certain theoretically possible context calls can never occur. Hence, we expect the context-graph completeness $c^p_{v_1}$ for a given program p and a given analysis v_1 to be smaller than $c^p_{v_2}$, the completeness for a less precise (but comparable) analysis v_2.

Figure 5 shows the average completeness for each analysis. We have normalized each completeness c^p_v with the corresponding context-insensitive completeness c^p_{ins}, in order to compute a meaningful average c^p_v/c^p_{ins} over all benchmark programs p.

In Figure 5 we have grouped the results into four sets of comparable analyses. For example, in the first group is *1-ThisSens* compared with the strictly more precise analyses *1-ThisArgs* and *1-ThisCall*. The important thing to notice here is that we now clearly have a substantial precision improvement when using a more precise analysis. The remaining groups (*k-ThisSens*, *k-ObjSens*, and *k-CFA*)

[8] *Comparable* according to the partial ordering in the precision lattice in Section 2.1.

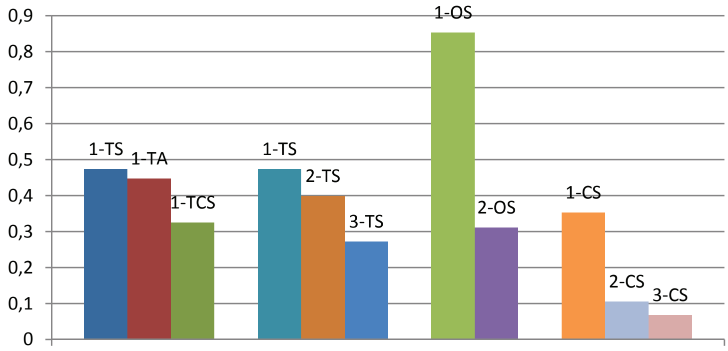

Figure 5: Precision results based on the context-graph completeness.

show that in general, increasing the call-depth k results in a lower completeness indicating a higher analysis precision.

The positive result of introducing the completeness metric is that we now can measure a substantial precision improvement when applying a more precise analysis. The results clearly shows that a more precise analysis can detect that certain call situations, although theoretically possible, will never occur. Furthermore, these results support our revised hypothesis that we can find a substantial difference in precision when comparing two approaches v^p and v^q (assuming v^q more precise than v^p) if (and only if) we use a metric at detail level q. The downside is that we were forced to introduce a rather artificial metric involving context calls to be able to detect these precision improvements.

6 Related Work

We start this section by outlining the results presented by [Lundberg et al., 2009] and [Lhoták and Hendren, 2006, Lhoták and Hendren, 2008]. They both present thorough experimental evaluations of different context-sensitive approaches to Points-to analysis, and are by far the most closely related works. Later on, in Section 6.2, we give a more general presentation of different contributions related to context-sensitive Points-to analysis.

6.1 Evaluating Context-Sensitive Points-to Analysis

This paper is an extention of [Lundberg et al., 2009] where this-sensitivity was first presented and evaluated in a set of experiments comparing 1-this-sensitivity with 1-object-sensitivity and 1-CFA using the *ObjectLevel* metric suite as presented in Section 4.1. Hence, the results were almost identical to the results presented for *1-ThisSens*, *1-ObjSens* and *1-CFA* in Section 4.

However, [Lundberg et al., 2009] also used a metric suite of type *SourceLevel*. The four metrics used in this suite were the number of nodes and edges in a call-graph, the number of casts that may fail, and the number of non-resolved polymorphic call sites. The experiments using this suite showed no substantial difference between the three context-sensitive approaches. (Median results were within 0.5% on all four metrics.) Furthermore, the context-sensitive approaches showed only slightly better results than the context-insensitive analysis. (Median improvements were 0.5-1%.)

[Lhoták and Hendren, 2006, Lhoták and Hendren, 2008] present an extensive experimental evaluation focusing on k-object-sensitivity and k-CFA using a precision metric suite very similar to the *SourceLevel* suite described above. For the case $k = 1$ they also found similar results: Small variations between different context-sensitive approaches and only slightly better results than the context-insensitive analysis. The interesting part here is that they, in addition to call-depth $k = 1$, also used $k = 2$ and $k = 3$. However, due to their usage of a rather coarse-grained metric suite they could not detect any substantial precision improvements for the cases $k > 1$.

6.2 Context-Sensitive Points-to Analysis in General

[Lundberg and Löwe, 2007] presented a context-insensitive version of the SSA-based simulated execution approach used in this article. Our program representation Points-to SSA is closely related to Memory SSA [Trapp, 1999] which is an extension to the traditional SSA [Muchnick, 1997, Cytron et al., 1991].

The number of papers explicitly dealing with context-sensitive Points-to analysis of object-oriented programs is continuously growing. The authors experiment with different context definitions and techniques to reduce the memory cost associated with having multiple contexts for a given method.

Many authors use a call string approach and approximative method summaries to reduce the cost of having multiple contexts [Chatterjee et al., 1999, Ruf, 2000]. Sometimes, ordered binary decision diagrams (OBDD) are used to efficiently exploit commonalities of similar contexts [Lhoták and Hendren, 2008, Whaley and Lam, 2004, Trapp, 1999], which allows handling of a very large number of contexts at a reasonable memory cost. A recently published alternative to OBDDs was presented in [Xiao and Zhang, 2011]. They use a new context encoding scheme called Geometric Encoding and report similar memory costs as when using OBDDs but without the high analysis time penalty.

[Milanova et al., 2002, Milanova et al., 2005] presented the object-sensitive technique as discussed earlier. They also presented a version using context-sensitive abstract objects that later was implemented and evaluated by experiments in [Lhoták and Hendren, 2008] and [Smaragdakis et al., 2011].

[Whaley and Lam, 2004] and [Zhu and Calman, 2004] 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 [Lhoták and Hendren, 2008].

Paddle [Lhoták and Hendren, 2008] and DOOP [Bravenboer et al, 2009] are two frameworks for Points-to analysis of Java programs. They both support many types of context-sensitive analyses, altough none of them contribute with new ones. Paddle uses OBDDs to reduce the memory costs, but it is painfully slow. DOOP allows the user to specify the analysis algorithms declaratively, using Datalog, avoids OBDDs, and reports much faster analysis times than Paddle.

7 Summary and Conclusions

[Lhoták and Hendren, 2008] present an extensive experimental evaluation focusing on k-object-sensitivity and k-CFA including cases where $k > 1$. Their results did not show a substantial difference in precision between different context-sensitive approaches and only slightly better results than the context-insensitive analysis. Their results could have indicated that a more expensive context-sensitive approach to Points-to analysis does not pay off. However, in their evaluation they used a rather coarse-grained precision metric suite focusing on source code entities, e.g., methods and statement and reference relations between them.

On the other hand, [Lundberg et al., 2009] reported substantial differences between different context-sensitive approaches when using more fine-grained precision metrics focusing on individual objects and references between them.

In this paper we present and investigate the following *hypothesis*: We have substantial differences between different context-sensitive approaches if (and only if) the precision is measured by more fine-grained metrics focusing on individual objects and references between them. That is, we expect to find differences between CFA, object-, and this-sensitivity. We also expect to find differences between call-depth $k = 1$ and $k > 1$.

In order to validate or invalidate our hypothesis, we have made a thorough experimental evaluation of ten different context-sensitive variants: k-this-sensitivity where $k = 1, 2, 3$, k-*CFA* where $k = 1, 2, 3$, and k-object-sensitivity where $k = 1, 2$. We also evaluated two new context-sensitive analysis variants: 1) a modified version of 1-this-sensitivity denoted *1-ThisArgs* that uses *all* arguments of a method to distinguish a context, and 2) a combination of 1-this-sensitivity and 1-CFA, denoted *1-ThisCFA*, that distinguishes contexts by taking both the call site and the implicit *this*-parameter into account.

To emphasize the difference in precision we have in all experiments used a metric suite named *ObjectLevel* with four precision metrics that all focus on different aspects of individual objects and their interactions. These metrics are justified by the many applications requiring such precise points-to information.

Experimental results: 1-object-sensitivity and 1-this-sensitivity provide almost identical results in three out of four precision metrics. The major difference lies in the metric indicating a precise resolution of object member accesses. Here, 1-object-sensitivity is clearly more precise. However, this comes at the price of being more than an order of magnitude slower than 1-this-sensitivity. Both 1-object-sensitivity and 1-this-sensitivity are clearly more precise than *1-CFA*.

Increasing the call-depth k does *not* give any substantial precision increase for k-this-sensitivity and k-object-sensitivity. When comparing 1-this-sensitivity (1-object-sensitivity) with 3-this-sensitivity (2-object-sensitivity), we find only precision gains in the range of 1-3% (0.5-2%) for all four metrics[9].

The CFA-family stands out as the only variant where we can detect a substantial precision increase when increasing the call-depth k. For example, when comparing *1-CFA* with *2-CFA*, we find precision gains for *2-CFA* in the range of 14-64% for all four metrics. However, the increased precision detected when $k > 1$ still does not make the CFA-family more precise than the other two families. For example, 3-this-sensitivity has a similar precision as *3-CFA*, but comes with a substantially lower analysis cost (time and memory).

Our new variants *1-ThisArgs* and *1-ThisCFA* have a positive effect on some individual benchmarks. For certain programs and metrics we find precision improvements with about 15% and 8%, respectively. However, on average, the overall precision improvements when using *1-ThisArgs* (*1-ThisCFA*) instead of 1-this-sensitivity is rather low, between 1-4% (0.5-1%) for all four metrics.

Conclusion: A substantial precision gain when using the theoretically more precise analyses *1-ThisArgs* or *1-ThisCFA* cannot be detected. Neither can it be achieved by using an increased call-depth $k > 1$. These are negative results since they indicate that in practice we cannot substantially improve the precision of a Points-to analysis by increasing the call-depth, or by some other means partition the call contexts into smaller ones.

Explaining Negative Results: Increasing the call-depth k leads to a more precise analysis but this improvement can not be detected by our *ObjectLevel* metric suite. We need to use even more fine-grained metrics to detect any differences.

Assume that each analysis variant v^p produces results on a certain detail level p, and that each metric m^q measures the precision on a certain detail level q. By introducing detail levels we can formulate a *revised hypothesis*: There is a

[9] These percentages are computed using averages over all programs in our benchmark.

substantial difference in precision when comparing two analysis variants v^p and v^q (assuming v^q more precise than v^p) *iff* we use a metric at detail level q.

In order to validate this revised hypothesis we repeated our experiments using a very detailed, but artificial, precision metric based on context call-graphs. The advantage of a context call-graph based metric is that its detail level (contexts) is always appropriate for any given context-sensitive analysis. The new experiments show a substantial precision improvement when increasing the call-depth k. These results confirm our revised hypothesis and clearly show that a more precise analysis can detect that certain context calls that are possible according to a less precise analysis can never occur. The downside is that the results, e.g., that a certain call between two contexts in *3-CFA* can never occur, now are at a detail level that is not likely to be useful in any realistic client application.

References

[Agesen, 1995] Agesen, O. (1995). The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'95)*, pages 2–26.

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

[Bravenboer et al, 2009] Bravenboer, M. and Smaragdakis, Y. (2009). Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'09)*, pages 243–262.

[Chatterjee et al., 1999] Chatterjee, R., Ryder, B., and Landi, W. (1999). Relevant context inference. In *Symposium on Principles of Programming Languages (POPL'99)*, pages 133–146.

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

[Clausen, 1999] Clausen, L. R. (1999). A Java byte code optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045.

[Cytron et al., 1991] Cytron, R., Ferrante, J., Rosen, B., Wegman, M., and Zadeck, K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490.

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

[Hasti and Horwitz, 1998] Hasti, R. and Horwitz, S. (1998). Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 97–105.

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

[Lhoták and Hendren, 2006] Lhoták, O. and Hendren, L. (2006). Context-sensitive points-to analysis: is it worth it? In Mycroft, A. and Zeller, A., editors, *International Conference on Compiler Construction (CC'06)*, volume 3923 of *LNCS*, pages 47–64.

[Lhoták and Hendren, 2008] Lhoták, O. and Hendren, L. (2008). Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):1–53.

[Lundberg et al., 2009] Lundberg, J., Gutzmann, T., Edvinsson, M., and Löwe, W. (2009). Fast and precise points-to analysis. *Journal of Information and Software Technology*, 51(10):1428 – 1439.

[Lundberg and Löwe, 2007] Lundberg, J. and Löwe, W. (2007). A scalable flow-sensitive points-to analysis. Technical report, Matematiska och systemtekniska institutionen, Växjö Universitet.

[Mancoridis et al., 1999] Mancoridis, S., Mitchell, B., Chen, Y., and Ganser, E. (1999). Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 50–59.

[Marlowe et al., 1995] Marlowe, T. J., Ryder, B. G., and Burke, M. G. (1995). Defining flow sensitivity for data flow problems. *Laboratory of Computer Science Research Technical Report*, Number LCSR-TR-249.

[Milanova et al., 2002] Milanova, A., Rountev, A., and Ryder, B. G. (2002). Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proc. of the Int. Symposium on Software Testing and Analysis (ISSTA'02)*, pages 1–11.

[Milanova et al., 2005] Milanova, A., Rountev, A., and Ryder, B. G. (2005). Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41.

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

[Ruf, 2000] Ruf, E. (2000). Effective synchronization removal for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'00)*, pages 208–218.

[Ryder, 2003] Ryder, B. G. (2003). Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction (CC'03)*, volume 2622 of *LNCS*, pages 126–137. Springer.

[Salah et al., 2005] Salah, M., Denton, T., Mancoridis, S., Shokoufandeh, A., and Vokolos, F. (2005). Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In *Proceedings of the International Conference on Software Maintenance (ICSM'05)*, pages 155–164.

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

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

[Shivers, 1991] Shivers, O. (1991). Control-flow analysis of higher-order languages. Technical report, PhD thesis, Carnegie-Mellon University, CMU-CS-91-145.

[Smaragdakis et al., 2011] Smaragdakis, Y., Bravenboer, M., and Lhoták, O. (2011). Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'11)*, pages 17–30.

[Soot] Soot: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot`.

[Sridharan and Bodik, 2006] Sridharan, M. and Bodik, R. (2006). Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'06)*, pages 387–400.

[Tip and Palsberg, 2000] Tip, F. and Palsberg, J. (2000). Scalable propagation-based call graph construction algorithms. *ACM SIGPLAN Notices*, 35(10):281–293.

[Tonella and Potrich, 2003] Tonella, P. and Potrich, A. (2003). Reverse engineering of the interaction diagrams from C++ code. In *Proceedings of the International Conference on Software Maintenance (ICSM'03)*, page 159.

[Trapp, 1999] Trapp, M. (1999). *Optimierung Objektorientierter Programme.* PhD thesis, Universität Karlsruhe.

[Whaley and Lam, 2004] Whaley, J. and Lam, M. S. (2004). Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI'04)*, pages 131–144.

[Xiao and Zhang, 2011] Xiao, X. and Zhang, C. (2011). Geometric Encoding: Forging the high performance context sensitive points-to analysis for Java. In *Proc. of the Int. Symposium on Software Testing and Analysis (ISSTA'11)*, pages 188–198.

[Zhu and Calman, 2004] Zhu, J. and Calman, S. (2004). Symbolic pointer analysis revisited. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'04)*, pages 145–157.