Towards Path-Sensitive Points-to Analysis

Tobias Gutzmann, Jonas Lundberg and Welf Löwe
Software Technology Group
Växjö University
Växjö, Sweden
Email: {Tobias.Gutzmann,Jonas.Lundberg,Welf.Lowe}@vxu.se

***Abstract*— Points-to analysis is a static program analysis aiming at analyzing the reference structure of dynamically allocated objects at compile-time. It constitutes the basis for many analyses and optimizations in software engineering and compiler construction. Sparse program representations, such as Whole Program Points-to Graph (WPP2G) and Points-to SSA (P2SSA), represent only dataflow that is directly relevant for points-to analysis. They have proved to be practical in terms of analysis precision and efficiency. However, intra-procedural control flow information is removed from these representations, which sacrifices analysis precision to improve analysis performance.**

We show an approach for keeping control flow related information even in sparse program representations by representing control flow effects as operations on the data transferred, i.e., as dataflow information. These operations affect distinct paths of the program differently, thus yielding a certain degree of path-sensitivity. Our approach works with both WPP2G and P2SSA representations.

We apply the approach to P2SSA-based and flow-sensitive points-to analysis and evaluate a context-insensitive and a context-sensitive variant. We assess our approach using abstract precision metrics. Moreover, we investigate the precision improvements and performance penalties when used as an input to three source-code-level analyses: dead code, cast safety, and null pointer analysis.

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. It computes for each variable and field v the set of possibly referenced objects, the *points-to set* $pt(v)$.

Points-to analysis is the basis of many other analyses in compiler construction and software engineering. Many of these "client analyses" of points-to analysis are used in program optimization and, hence, have no effects on the source code. However, quite a few of them have direct relevance for program designers since they

- compute precise views on the source code that help designers in program comprehension and debugging, e.g., call graph construction and program slicing,
- prove correctness of certain properties statically, e.g., cast safety and that objects accessed cannot be null,
- lead to source code transformations, e.g., dead code elimination.

Points-to analysis is the approximation of a problem that is, in general, undecidable. Since a program may create countable many runtime objects and may have unknown input, exact points-to information cannot be computed in general. Approximations vary in precision and scalability. Since points-to analysis is not a value in itself, it is important to carefully select the variant fitting best a certain client analysis. It should provide sufficient precision at acceptable performance *in the context of the client analysis*. It has been observed before that very precise points-to analysis may have little effect on a given client analysis, e.g., in the context of call graph construction [1].

Points-to analyses differ, among others, in

- flow-sensitivity: Does the analysis consider the order in which statements are executed?
- context-sensitivity: Does the analysis distinguish different call contexts of methods?
- name schema: How are the countable many runtime objects represented in the analysis, i.e., mapped to abstract objects?
- program representation: How does the analysis abstract from the semantics of programming constructs of a source program?

Flow- and context sensitive variants are more precise and less efficient than their insensitive counterparts, and so are name schemata that distinguish many objects compared to those that only distinguish few. For a more elaborated discussion and classification of points-to analysis variants cf. [2]. In short, the choice of flow-sensitivity, context-sensitivity, and name schema provides a variant of points-to analysis with a certain precision, where higher precision usually means more time and memory are required to perform the analysis.

Points-to analyses that have proved to be efficient in practice usually operate on *sparse* program representations. They abstract from all programming constructs that do not directly affect points-to information. They do not include, e.g., operations related to basic types or control flow statements. Again, compared with full program representations, e.g., intermediate compiler languages, sparse representations trade precision for performance.

A. Contributions

Sparse program representations used for points-to analysis remove intra-procedural control flow information. We present an approach that retains a certain degree of control flow information. Our approach to *path-sensitivity* enriches a program representation with additional dataflow operations. It is orthogonal to other variants of points-to analysis [2] and

fundamentally differs from well-known approaches to path-sensitivity, cf. Section V-A. Hence, it can be seen as a new dimension for varying points-to analysis. In theory, path-sensitivity leads to a non-decreasing precision. In practice, precision of points-to analysis increases and has positive effect on the precision of the source code related client analyses assessed. In particular, we observe an average precision-improvement by

- 31% (25%) in the context of our context-insensitive (context-sensitive) cast-safety analysis,
- 5% (3,5%) in the context of our context-insensitive (context-sensitive) dead code analysis,
- 3,3% (3,1%) in the context of our context-insensitive (context-sensitive) null-pointer analysis.

Although performance results are preliminary, we did not observe prohibitively large slow-downs of the analysis. In the worst case, we observed a 32% (21%) slower context-insensitive (context-sensitive) analysis. In the best case, we even observed a speed-up by 12% (21%).

B. Paper Outline

The remainder of the paper is organized as follows. Section II defines and discusses basic notions and concepts of points-to analysis and sparse representations. Section III introduces our approach to path-sensitive points-to analysis. This section contains the main theoretical contribution of the paper. Section IV assesses our approach in the context of source code related client analyses. It contains the main practical contribution of the present paper. Section V puts our contributions into the context of prior work. Finally, Section VI concludes the paper and shows directions of future work.

II. Program Analysis

In this section, we discuss principles of program analysis, including a short summary of common program representations.

A. Basics

A dataflow analysis needs to *approximate* its analysis results. For points-to analysis, such an approximation is either *optimistic* or *pessimistic*, yielding either *stronger* or *weaker* conclusions than the exact statement, respectively. For a points-to analysis, this means either an under- or overestimation of the obtained points-to sets. An analysis that only uses pessimistic approximations is also called *conservative*. A common approach to support conservative analysis is the use of monotone dataflow frameworks [3], [4]. A common way to achieve monotonous dataflow is to disallow *strong updates* for *transfer functions* – that are the functions that give single operations its analysis semantics – i.e., the previously computed value of a variable is not overwritten but values are added to a variable's set of possible values when an assignment occurs.

B. Name Schema

A program analysis needs to abstract from the values which expressions may take during a real application run in some way. Such an abstraction is called a *name schema*. When a name $n \in N$ is a classification of one or more runtime objects $o(n)$, then the following must hold:

$$\forall n_1, n_2 \in N : n_1 \neq n_2 \Rightarrow o(n_1) \cap o(n_2) = \emptyset$$

Thus, an *abstract* object may denote an arbitrary number of *runtime* objects, but each *runtime* object must be represented by exactly one *abstract* object.

C. Flow Sensitivity

An analysis can be either *flow sensitive* or *flow insensitive*. A flow sensitive analysis takes the order of operations into consideration. This can be the case either on inter-procedural or intra-procedural dataflow. Flow-sensitive analyses are more precise than flow-insensitive analyses, however, they are also more costly.

D. Context Sensitivity

An operation op on a syntactical location s in a program might be reached through multiple predecessor statements, therefore an analysis may distinguish distinct execution paths in order to improve precision. This can be done either on an *intra-procedural* level or on an *inter-procedural* level.

E. Program Representations

Program representations can capture either the full semantics of a program, or they can focus on parts of a program that are sufficient for a given task. The former include basic block graphs used, e.g., by compilers [5].

An example of the latter are sparse *Whole Program Points-to Graphs (WPP2Gs)*, which are used by many scalable Points-to analyses, e.g., [1], [6]–[13]. WPP2Gs contain three different node types: abstract objects, reference variables, and object fields. Edges represent assignments of abstract objects, variables, and fields to (other) variables and fields.

For the rest of this section, we will focus on *Points-to SSA*, the program representation which we use in the evaluation section of this paper. We start with a short summary of technologies that Points-to SSA is based on.

Static Single Assignment form – in short, SSA – is an intermediate representation technique first developed by [14]. Every variable is assigned a value exactly once. For each definition in original form, a new *version* of that variable is created during SSA construction. To decide what version of a variable is valid after meets in the control flow, ϕ-nodes are introduced: ϕ-nodes are artificial operations that take the possible versions of a variable as arguments and decide, depending on control flow, which of these operands is the currently valid definition. SSA form provides many benefits for program analysis, for instance, use-def relations become explicit.

Memory SSA [15]–[17] is a graph-based extension to the traditional SSA. In Memory SSA, the traditional ordering

of operations within a basic block structure is replaced by a directed graph structure. Local variables are resolved to dataflow edges connecting operations (nodes), which has the effect that def-use relations become explicit. Dependencies on accessing the memory are modeled by memory edges, putting memory on the same level as data, including the use of ϕ-nodes at control flow confluence. These memory edges dictate a correct order in which memory accesses must be executed for a given program.

SSA and Memory SSA both capture the full semantics of a program. We will now describe *Points-to SSA* [18], our graph based program representation which we conduct our experiments on in this paper.

Points-to SSA method graphs are an abstraction of Memory SSA method graphs which are specially designed for points-to analysis. We have removed all operations not directly related to reference computations, e.g., operations related to primitive types.

A feature of Points-to SSA is how memory operations are handled. 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 actual memory state – e.g., the analysis values of object fields – is maintained in a separate data structure.

A *Points-to SSA method graph* $G = \{N, E, Entry, Exit\}$ is now defined as a directed, ordered multi-graph where N is a set of Points-to SSA nodes, E is a set of Points-to SSA edges, $Entry$ is a graph entry node satisfying $|pred(Entry)| = 0$, and $Exit$ is a graph exit node satisfying $|succ(Exit)| = 0$.

The reference related semantics of different language constructs (e.g., calls and field accesses) are described by a set of *operation nodes*. Each node n has a number of *in-ports* $in(n) = [in_1(n), \ldots, in_k(n)]$, and a number of *out-ports* $out(n) = [out_1(n), \ldots, out_l(n)]$. The in-ports represent input values for the operation in question whereas the out-ports represent the results produced by the operation. All ports have a fixed *type* – i.e., memory size or points-to set – and a current *analysis value* of that type.

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

Certain node types have attributes that refer to node specific information. For example, each $Read$ node is decorated with a field identifier that identifies the field to be read from memory. Finally, each type of node is associated with a unique *analysis semantics* (or *transfer function*), which can be seen as a mapping from in-ports to out-ports that may have a side-effect on the memory. A detailed description of the underlying analysis algorithms can be found in [18].

These information should give enough insight to follow the examples given in this paper by comparing given source code with the corresponding graphs, e.g., Figure 5.

III. Approach

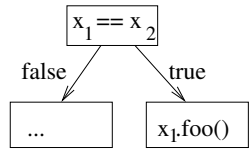

Fig. 1. Simple Control Flow Graph

Our approach, *path-sensitivity*, uses the fact that control flow statements may make branching decisions based upon input variables. While, at runtime, each such input variable has exactly one value, a static analysis will, in general, compute a set of possible runtime values. Thus, it is undecidable which branch is taken. However, statements about the values of the variables involved in deciding the branch can be made, cf. the control flow graph in Figure 1: On the branch that is executed when the equality comparison yields *true*, we can restrict the targets of the call x_1.foo() to $pt(x_1) \cap pt(x_2)$. Should the compared values not be retrieved from local variables or constants, some special treatment can be applied to flow sensitive analysis. We will discuss this in Section III-C and, for now, restrict input to branch decisions made on basis of local variables and constants.

We will now present a general approach that can be applied to many intermediate representations. Afterwards, we will discuss improved implementation techniques for flow-insensitive and flow-sensitive points-to analyses.

We limit our approach to Java, as so does our points-to analysis and different languages might have slightly different semantics, e.g., for instanceof operations, but it should be possible to apply the approach to other programming languages with few adaptations.

A. General Approach

Our approach works for many program representations that are not in SSA form. SSA is usually constructed from another program representation, and the changes can be taken along to SSA based representations, so that this is not a functional constraint.

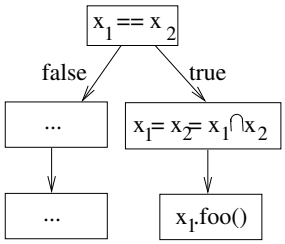

Fig. 2. Insertion of Basic Blocks

For each branch statement for which we can add filter operations – which will be discussed below –, two basic blocks are added. These blocks have the basic block containing the branch statement in question as their only control flow predecessor, and the original control flow successors as their (only) control flow successors. Each of these basic blocks contains redefinitions of the variables which are involved in the branching condition. These redefinitions filter the variables to values as discussed below. Figure 2 shows how the control flow graph from Figure 1 will look after this transformation. Note that the basic blocks need to be inserted only if the successor basic block of the branch statement has multiple control flow predecessors, i.e., the original edge is a critical edge, as for some of the predecessors of that basic block the filter operation may not be valid. Otherwise, the filter operations can be inserted as first statements of the existing basic blocks for simplicity. No action needs to be taken when a branch joins another branch as the redefinitions of the variables lose their scope.

In Table I, we list five possible filter operations that can be applied to our approach and which we will briefly discuss. Each row in the table lists one pattern which may be encountered in a program representation as input to a branching statement, and for which we introduce filter operations in the program, either on the *true*- or *false*-branch, respectively.

Note that for inequality and negation, the two branches are simply to be exchanged. We therefore do not discuss these cases any further.

We have already given reference equality – the first filter operation listed – and its influence on the *true*-branch as an introductory example. For the *false*-branch, however, we cannot narrow the possible values of the involved variables, as an *abstract* object may denote several concrete *runtime* objects, cf. Section II-B; an exception is a comparison with the *null*-constant, our second filter operation. Further abstract objects known to denote a single runtime object could be identified, e.g., objects known to be singletons. These could also be treated like *null*-checks, but a discussion on how to identify such objects is beyond the scope of this paper.

Another variation of this kind of filter is comparing the runtime type of an object with a given type, which is the third filter operation shown. Logically, this filtering has been widely used: [15], for example, transform polymorphic calls in their Memory SSA based program representation into a number of monomorphic calls and insert type-based filters. The code shown in Figure 3 and an outline of the dispatch logic of the polymorphic method invocation of $foo()$ illustrate this process on source-code level.

The fourth proposed filter operation targets a method invocation to *Object.equals()*. Here, $x_1.class^* == Object.class$ means that the polymorphic call resolution for an object a targets the implementation of $Object.equals()$; i.e., the class of a does not overwrite that method. This approach can be extended to other methods, e.g., strings, where both objects need at least to be strings in order to be equal.

Our fifth and last proposed filter is the *instanceof* operation.

```
class A {
  void foo() { ... }
  void bar(A a) { A.foo(); }
}
class B extends A {
  void foo() { }
}
```

Fig. 3. Polymorphic Call

```
static dispatch_foo(A a) {
  if (a.getClass() == B.class)
    B.foo(a);
  else if (a.getClass() == A.class)
    A.foo(a);
  // ...
}
```

Fig. 4. Dispatch Logic

The most common use of the *instanceof* operator is to check an object for a given type and then to cast the object to that type. With the filter operation preceding a cast, we gain two other benefits besides the effect that the cast itself can be regarded as never failing: First, a cast in Java will not fail if the value of the casted expression is *null*; an *instanceof*-operation will, however, return *false* in that very case, so that a benefit in filtering is achieved. Second, on the *false*-branch, values *cannot* be object instances of that type, thus we can include a complementary filter there.

B. Implementation for Flow-Insensitive Analyses

The approach described above is correct for every analysis. However, flow-insensitive analyses will not benefit since variables which share the same declaration also share the same value set. Therefore, a slightly different implementation approach needs to be taken. Instead of assigning a variable x its filtered value as described above, we declare a copy x', assign it the filtered value of x, and replace occurrences of x with x' where valid. To decide where x can be replaced by x', we use the *dominator relation*. A basic block b_1 dominates a basic block b_2 if every path from the start node of a method graph to b_2 must go through b_1.

With x the variable in question and B the newly created basic block for x, every defining and using reference to this variable x is replaced by x' in all operations contained in any basic block dominated by B. If such an x' is assigned a value at any point, these values need to be merged with x again. Since this is required only in flow-insensitive analyses, it is sufficient to introduce an assignment $x = x'$ anywhere in the control flow graph. A generally valid approach not specifically targeted at points-to analysis is to introduce a merge operation $x = x \cup x'$ as the last statement of each basic block that is dominated by B but which has at least one control flow successor that is not dominated by B any more, i.e., dominated blocks that are predecessors of dominance frontiers of B.

With this algorithm, we perform a belated def-use analysis for the filter operations. While it at first appears that we

TABLE I
FILTER OPERATIONS

Condition	true branch	false branch
$x_1 == x_2$	$pt(x_1) = pt(x_2) = pt(x_1) \cap pt(x_2)$	<none>
$x == null$	$pt(x) = pt(x) \cap \{null\}$	$pt(x) = pt(x) \backslash \{null\}$
$x.getClass() == T.class$	$pt(x) = \{a \in pt(x) \mid a.getClass() == T.class\}$	$pt(x) = \{a \in pt(x) \mid a.getClass() \neq T.class\}$
$x_1.equals(x_2)$	$pt(x_1) = \{a \in pt(x_1) \mid \neg(a.class^* == Object.class) \vee (\exists b \in pt(x_2) : a == b)\}$	<none>
x instanceof T	$pt(x) = \{a \in pt(x) \mid a$ instanceof $T\}$	$pt(x) = \{a \in pt(x) \mid \neg(a$ instanceof $T)\}$

perform a small-scale SSA construction, and, thus, a certain degree of flow-sensitivity is introduced into the flow-insensitive analysis, this is not the case, as a filtered variable depends on its unfiltered counterpart and therefore its analysis values may be altered by assignments that strictly appear after the filter operation. Further, the analysis of the unfiltered variable is influenced by the filtered variable.

C. Implementation for Memory SSA based Program Representations

The universal approach discussed before requires the manipulation of the intermediate representation that is not yet in SSA form, as well as adaptation of the SSA based program representation. In the following, we discuss an alternative implementation approach that makes do with adapting only the SSA based program representation.

```
class A {
 Object x,y;
 void foo(A a) {
  Object m = x;
  if (a.x != m) {
   y = m;
  }
 }
}
```

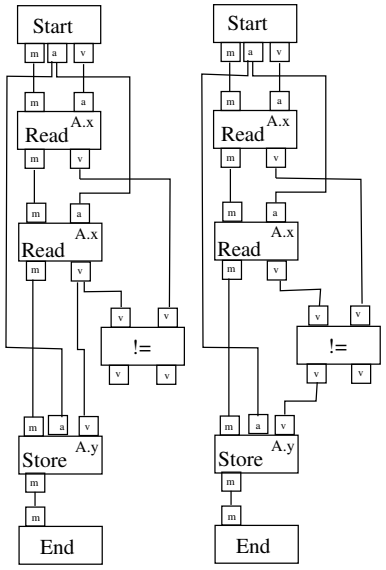

Fig. 5. Transformation on Memory SSA Graphs

Instead of introducing additional nodes within Memory SSA, the existing nodes – in our case, *instanceof*- and *EQ*-nodes – take over the computations of the filtered points-to sets, we thus modify the nodes' transfer functions. Uses of the operands in the according branches – computed again by domination relation as described in the previous section – are replaced by the filtered values computed by these operations.

Figure 5 shows an example of this transformation. Note that, for simplicity, the basic block structure, which is required for the computation of the dominator relations, as well operations related to intra-procedural control flow are not included in the graph. The left of the two graphs shows a regular Memory SSA graph with the relevant equality operation retained. Now, the equality operation is turned into a filter operation and, since it clearly dominates the store operation to $A.y$, that operation's input value is replaced by the filtered value, as can be seen in the right graph.

While this approach reduces the implementation effort, it has a drawback. Due to flow sensitivity, we could gain more precision than for the flow insensitive approach when a variable has been filtered individually on two distinct paths which then coalesce. Using the general approach, the two filtered values would be merged by a ϕ-operation. The implementation technique presented here cannot achieve this, thus, the unfiltered variable will be used on the merged path.

Depending on programming style, multiple read accesses to the same field may occur in sequence. In Figure 6(a), we have rewritten the previous example such that we do not create a local variable to which we copy the contents of the field x. Now, an additional *Read* node is inserted. The filter operation is now performed for the first read operation in line 4, while the assignment in line 5 will use the second read operation which has not been filtered.

Performing an *Available Expressions* analysis and removing superfluous memory read accesses solves the problem; however, we use the less complex algorithm described in Figure 7. The algorithm works as follows: It searches the memory dependencies of read accesses upwards and checks if it discovers an equal (same field, same address) read access. If so, the two read accesses are merged. It stops the upward traversal if it encounters an operation that potentially changes the memory with respect to the referenced field. Such operations include write accesses to the same field, method invocations, and memory-ϕ operations.

Besides improving the analysis precision of our presented approach, this optimization has also a positive effect on performance, as identical and therefore superfluous operations are removed.

IV. EVALUATION

In this section, we describe four different setups of our points-to analysis which we then compare by means of five different metrics as well as performance measurements.

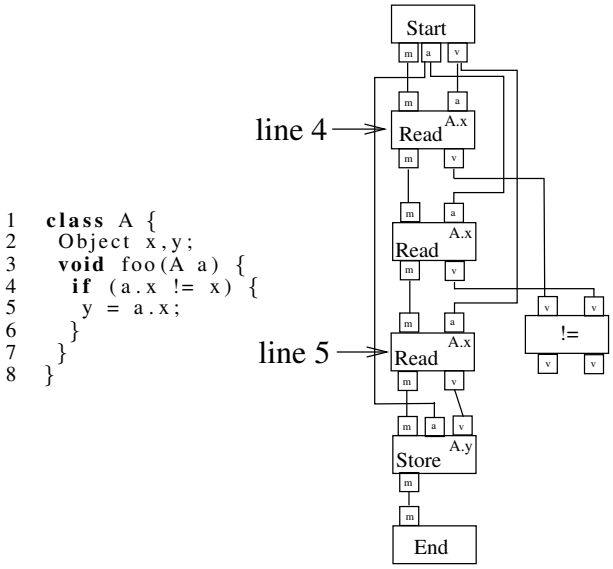

Fig. 6. Sequential Read Accesses to Same Field

```
procedure REMOVEDUPLICATEREADS(graph)
    for all n ∈ graph.nodes do
        if n instanceof Read then
            in = getMemoryInNode(n)
            while doesNotChangeMemFor(n, in) do
                if in instanceof Read
                        ∧ accessSameField(n, in)
                        ∧ useSameAddr(n, in) then
                            n.replaceWith(in)
                    break
                else
                    in = in.getMemoryInNode(n)
                end if
            end while
        end if
    end for
end procedure
```

Fig. 7. Removing Redundant Read Operations

A. Metrics

We use two abstract metrics which have been proposed before to assess the precision of a points-to analysis [19], as well as three metrics at source code level. All our metrics aim at *application methods*: An application method is one that belongs to the core of the analyzed program, and is identified by checking if the name of the package it belongs to starts with a certain character string. This ensures that we anticipate remeasuring the effects different analysis optimizations have on the Java runtime library.

Our two abstract metrics are:

- OEdge: The Application Object Call Graph (AOCG) is a fine-grained call graph. The graph consists of nodes $[o, m]$, where o are abstract objects and m methods, and edges $[o_i, m_p] \rightarrow [o_j, m_q]$ in between them. OEdge is the number of edges, i.e., call relations, in an AOCG, where at least the caller or callee is an application object member.
- Heap: The number of abstract objects referenced by object fields. That is, the sum of the set sizes for all points-to values stored in all application object fields. Stored *null*-values are not counted for this metric.

Our source level metrics are:

- Safe casts: A type cast is *safe* when it is guaranteed not to fail at runtime, i.e., the input set of the cast operation equals the output set. We measure the percentage of safe casts that occur in application methods.
- Dead code: An *access* is a field read or write operation on an application field, or a method invocation originating from an application method. A *dead access* is one which cannot be reached, i.e., its *this*-pointer has an empty points-to set. We measure dead code in percentage of dead accesses compared to total accesses. We determine the total number of accesses by going through all application methods which are in our intermediate program representation, thus, accesses within unreachable methods count in to the number total accesses.
- Safe accesses: A *safe access* is an access which is guaranteed to not raise a null pointer exception at runtime. We count in dead accesses to remove the effect that a more precise analysis may actually show less safe accesses due to safe accesses now being identified as dead.

B. Setup

We use a reimplementation of the points-to analysis described in [18] which is locally flow-sensitive and partly interprocedurally flow-sensitive. Our reimplementation is more flexible in terms of graph construction, but the underlying analysis has not yet been optimized for performance and memory consumption. Currently, no exception handling and stubs for only few native methods are supported. We have implemented three of the filter operations we proposed in Section III-A: *Reference equality*, *null check*, and *instanceof*.

We combine our baseline analysis and our analysis using the filter operation each in combination with a context insensitive and 1-Object-Sensitivity (1-Obj-Sens) [10], [13], thus having four different setups. The optimization for redundant read-operations, which we presented in Section III-C, is always applied to ensure comparability in terms of performance.

All benchmarks were run on a Dell PowerEdge 1850, 6GB RAM, Dual Intel Xeon 3.2GHz computer, in a 32-bit Java Virtual Machine, version 1.6.0-b105, with 2GB heap space. The benchmark programs are all freely available. They are either taken from the well-known *DaCapo benchmark suite*[1], or their version numbers are specified. *Obfuscator* is a tool coming along with the *Recoder* metaprogramming framework[2].

C. Results

We now present the results of our measurements, beginning with a discussion of the abstract metrics, followed by the source level metrics. We then discuss performance issues.

[1] http://www.dacapobench.org
[2] http://recoder.sourceforge.net

TABLE II
ABSTRACT METRICS

	ConIns				ObjSens			
	PathIns		PathSens		PathIns		PathSens	
Program	OEdges	Heap	OEdges %	Heap %	OEdges %	Heap %	OEdges %	Heap %
AntLR	31129	16578	99.85	100.00	38.26	30.43	38.11	30.43
Chart	122973	903030	99.13	99.71	83.91	40.62	83.47	40.60
Emma 2.0	329835	594215	99.63	99.89	46.36	23.69	46.19	23.66
JavaCC 3.2	193629	12824	99.99	99.77	30.26	69.70	30.25	69.47
JavaC 1.3	317699	152660	100.00	100.00	70.27	41.48	70.27	41.48
Jython	2121374	308583	72.32	91.16	90.21	36.46	66.31	29.11
Obfuscator 0.73	223900	69529	99.63	100.18	73.16	76.35	72.52	76.01
PMD	117046	55455	97.95	99.51	71.24	58.67	69.40	58.17
SableCC	1078027	283697	100.00	100.00	35.32	35.32	34.06	34.06
average			96.50	98.91	59.89	45.86	56.73	44.78

1) Abstract Metrics: Table II shows the benchmark results for our two abstract metrics, *OEdges* and *Heap*. *ConIns* and *ObjSens* label the results for the context-insensitive and context-sensitive setup, respectively, and *PathIns* and *PathSens* the results of path-insensitive and path-sensitive setup.

Neither in the context-insensitive, nor in the context-sensitive approach do the benchmarks show any major improvements when adding path-sensitivity. *Jython* makes a formidable exception here, for which we shortly discuss the reasons: Figure 8 shows an excerpt from the Jython source code. The first two *instanceof* operations in line 1 and 3 on the local variable *im_func* do not affect the object creations sites in lines 2 and 4, respectively, with respect to the abstract metrics, as those *instanceof* operations merely guard the cast expressions. However, both *instanceof* operations act as filters for the object creation site in line 6, which greatly reduces the input set of the second argument. This kind of code appears at several locations in Jython, but not in the other benchmark programs used, which explains the discrepancy.

One test case, *Obfuscator*, shows a very slight precision decrease in our metric *Heap*. This can be explained as follows: The analysis metric itself does not benefit from the filter operations, but the stabilization process is affected. Since our analysis is partly inter-procedurally flow-sensitive, variations in the stabilization process, which cause this anomaly, occur.

2) Source Level Metrics: Table III shows the benchmark results for our three source level metrics. The percentage of safe casts increases by more than nine percentage points each in the context-insensitive and the context-sensitive setup when adding path-sensitivity, corresponding a gain of 25.5 to 31.6 percent. On average, 0.72 to 0.75 more percentage points (a gain of 3.56 to 5.01 percent) of dead code are discovered, and 2.56 to 2.71 more percentage points (a plus of 3.13 to 3.35 percent) of accesses can be considered safe, respectively. While the large number of null-pointer checks commonly found in programs would suggest a larger improvement in this metric, the results of this metric are already on a very high level. The mass of the remaining unsafe accesses likely stem from array usage. We do not perform any strong updates on arrays but always initialize a newly created array with null, as required by [20].

	ConIns			1-Obj-Sense		
Program	Orig *(h.m:s)*	PathIns *(h.m:s)*	PSens %	Orig *(h.m:s)*	PathIns *(h.m:s)*	PSens %
AntLR	0:03	0:30	112	0:09	1:17	117
Chart		53:55	114		9.12:27	121
Emma 2.0	2:45	32:44	119	1.36:59	1.43:39	109
JavaCC 3.2	0:04	1:35	106	1:28	42:19	102
JavaC 1.3	0:27	10:28	115	10:21	53:29	119
Jython		7:12	91		3.27:02	79
Obfusc. 0.73	0:15	5:44	88	1:27	22:38	79
PMD		0:59	100		8:08	101
SableCC	0:08	2:35	132	0:21	19:31	109
average			109			104

TABLE IV
EXECUTION TIMES

At no surprise comes the fact that the number of safe cast operations greatly improves – on average, by 9.46 to 9.92 percentage points or by 25 to 31 percent – with our proposed optimization.

The precision gain remains quite constant with the different context sensitivities, which suggests that the filter nodes target different room for improvement than other variation points like, e.g., context-sensitivity.

3) Performance Considerations: As stated before, our implementation has not been tuned for performance and memory usage. Thus, the focus in the discussion lies on relative performance. [19] shows that the analysis can be performed efficiently.

For the construction process, we currently use a straightforward algorithm for computing dominance relations that runs in $O(n^2)$ and is not further tuned for performance, and our graph transformations run on a graph library that is designed for flexibility, not speed. However, fast algorithms for computing dominator relations are known [21], [22]. Inserting the filter operations into the control flow graphs and transforming the SSA graphs, respectively, should be possible with fast, optimized algorithms, as patterns that are to be matched always consist of few nodes, and branch nodes are necessary indicators for the presence of such patterns.

Table IV lists the analysis times for three analysis: Here, *Orig* labels the execution times of the original implementation

```
1  if (im_func instanceof PyFunction)
2    return new PyMethod(container, (PyFunction)im_func, im_class);
3  else if (im_func instanceof PyReflectedFunction)
4    return new PyMethod(container, (PyReflectedFunction)im_func, im_class);
5  else
6    return new PyMethod(container, im_func, im_class);
```

Fig. 8. Source Code Excerpt From PyMethod.java

TABLE III
SOURCE LEVEL METRICS

	ConIns						1-Obj-Sens					
	PathIns %			PathSens %			PathIns %			PathSens %		
Program	Casts	Dead C.	SafeAcc.	Casts	Dead C.	SafeAcc.	Casts	Dead C.	SafeAcc.	Casts	Dead C.	SafeAcc.
AntLR	43.14	19.25	84.14	49.02	19.54	85.34	44.12	24.72	85.09	49.02	25.01	86.23
Chart	46.35	22.23	79.11	55.06	24.80	84.19	48.00	29.65	80.02	56.47	32.18	84.65
Emma 2.0	48.26	12.73	82.87	49.57	13.25	86.49	52.61	19.03	84.58	53.91	19.51	88.08
JavaCC 3.2	10.41	2.46	68.55	33.93	2.65	70.94	10.41	2.84	68.56	32.14	3.03	70.95
JavaC 1.3	3.58	1.63	67.82	7.67	1.71	69.11	6.13	6.87	68.49	10.22	6.93	69.75
Jython	64.63	28.27	90.82	77.57	29.54	93.03	66.61	37.09	91.17	69.45	38.22	93.28
Obfuscator 0.73	37.83	17.85	79.08	66.00	18.40	83.93	37.83	21.02	79.79	66.00	21.58	84.11
PMD	38.63	14.88	92.87	51.50	16.10	93.89	42.92	17.45	93.27	55.79	18.66	94.28
SableCC	17.88	15.44	83.75	18.60	15.50	86.44	24.66	23.36	84.34	25.38	23.42	87.02
average	34.52	14.97	81.00	45.44	15.72	83.71	37.03	20.23	81.70	46.49	20.95	84.26
gain				31.63	5.01	3.35				25.55	3.56	3.13

of Points-to SSA, which also supports exception handling, as published in [19]. Some of the benchmark program – *Chart* and *Jython* – were not tested in that paper, and *PMD* was tested in a different version. Thus, we omit the execution times here. The results serve as comparative numbers and show that the analysis can be performed efficiently. *PathIns* and *PSens* are the execution times for our reimplementation; the former is the absolute amount of time required to run the baseline analysis, while the latter shows the relative time factor when running the analysis with the filter operations.

For the context-insensitive setup, the average slowdown is 9%, for the context-sensitive setup 4%. Two of the benchmark programs, *Jython* and *Obfuscator*, are analyzed faster than without the optimizations. For these two test cases, the number of repetitions of analyzing methods – we do not list these numbers explicitly, as they are not of major importance – drastically reduces due to improved analysis precision[3] [**Welf: list??**] For these two benchmark programs, intra-procedural overhead of our path-sensitive approach is outweighed by the positive side-effects on the inter-procedural analysis.

However, the relative performance results are preliminary as the analysis in its entirety is rather slow at its current implementation state. Once the performance of our reimplementation adjusts to that of our original implementation, these results will have to be reevaluated.

V. RELATED WORK

In this section, we present current research related to points-to analysis of object-oriented programs. For brevity, we focus our efforts on works explicitly dealing with the analysis of object-oriented programs. However, it should be noted that most works targeting object-oriented programs have an "imperative counterpart" which often predates the object-oriented work. People interested in more general reviews of the area should take a look at [2], [6], [23], [24].

A. Path-Sensitivity

An analysis is *path-sensitive* if it makes use of the expressions used in control statements to restrict the possible values that might enter the different branches of the statement. In this paper, we focus on Boolean expressions involving reference variables and use that information to filter out "impossible" abstract objects from different branches in iterative and selective control statements. This approach reminds of the so-called *Gated SSA* formalism [25], [26]. In Gated SSA, ϕ-nodes are extended to γ-nodes which are annotated with the corresponding branching conditions. These may then be used to make statements about taken paths after control flow meets. In contrast, our approach inserts statements at the beginning of distinct paths.

If-Conversion, e.g., [27], [28], is a compiler optimization aiming at reducing the number of conditional branches used in many state-of-the-art compilers. Here, control flow is also converted to dataflow by inlining the guard expressions of *if* statements. Multiple paths can be merged into dataflow operations, and the information is then used to select best fitting – in terms of runtime efficiency – machine operations. The approach of If-Conversion to convert control flow into dataflow differs from our approach in that If-Conversion aims at primitive data types.

Many approaches deal with the *meet over all paths (MOP)* dataflow problem, e.g., [29]. Since the number of paths is,

[3]An improved analysis result does not necessarily lead to such a lowered number of analyzed methods; however, a thorough discussion of this topic goes beyond the scope of this paper.

in general, unbounded, approaches narrow down the set of paths, e.g., by finding correlations between branch conditions [30]. Xie *et al.* [31] use path-sensitive analysis in their array access checker ARCHER. Their approach to path-sensitivity selects a set of execution paths – both a super- and subset of legal paths – and eliminate infeasible paths based on branching conditions. A different approach limits the number of paths to investigate by selecting interesting paths based on dynamic analysis, e.g., [32]. Our approach differs from these in that it does not rule out infeasible paths but limits the points-to sets of variables on certain paths.

B. Flow-Sensitivity

We only know of three papers that report on flow-sensitive approaches to points-to analysis for object-oriented programs [33]–[35].

The major theoretical obstacle in a flow-sensitive analysis is the question when it is safe to perform strong updates. These are only permitted when we are sure about the ordering of the reads and writes of a given variable/field.

Both [33] and [34] use weak updates for every assignment involving fields and method parameters. The more recent of the two works [34] reports increased precision at a reasonable cost. A more ambitious approach is taken in [35]. They compute inter-procedural def-use information, which is later used to decide whether strong updates are safe or not. They report high precision (but unacceptable cost).

Our dataflow analysis technique, first presented in [18], is an abstract interpretation of the program. It 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 later when the analysis of the called methods is completed. The resulting analysis is flow-sensitive 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. This statement was verified by experiments in [18].

C. Context-Sensitivity

In a context-sensitive analysis, a method is separately analyzed for each different call context [6], [36], [37]. Approaches differ in the context definitions. The two traditional approaches to defining contexts are referred to as the *call string* approach and the *functional* approach [36]. The call string approach uses the top sequence of the call stack at each call site to define a context. In this so-called k-CFA family of algorithms, the precision of an analysis is denoted by the length k of the top sequence of the call stack [37]. The functional approach uses some abstractions of the call site's actual parameters to distinguishing different contexts [6], [36]. Both the call string and the functional approaches were evaluated, and put into a common framework, by Grove *et al.* in [6].

VI. Conclusions and Future Work

In this paper, we suggest a new way to achieve *path-sensitivity* as a new orthogonal variation of points-to analysis. A points-to analysis is called path-sensitive if it exploits control flow information leading to an execution path when analyzing the operations on that path. Our approach retains control flow information by inserting path-specific filter operations in the dataflow. During analysis, these filter operations can be treated as ordinary program entities of a program representation: they are assigned a transfer function, which is updated iteratively until a fixed point is reached. We have identified five such filter operation types corresponding to different types of branching conditions in the control flow. In theory, a path-sensitive points-to analysis is more precise than its path-insensitive counterpart.

Furthermore, we implemented our approach with two baseline variants of points-to analysis: both based on a sparse SSA representation of the programs, both flow-sensitive, and context-insensitive and context-sensitive, respectively. Using abstract precision metrics, we confirmed the expected increase in precision of the path-sensitive variants in practice.

However, the precision of points-to analysis is not a value in itself. It has to be measured in terms of improved precision of a client analysis using this points-to information as its input. We selected three source code analyses for assessment: cast safety, dead code, and null-pointer analysis. We assessed the precision of these client analyses when path-insensitive and path-sensitive, respectively, points-to analysis is used as the basis. In the case of cast safety analysis, improvements are considerable, e.g., 31% on the average when applied in the context-insensitive setting. For the other two client analyses, improvements are not similarly impressive but still noticeable: on average 5% and 3,5%, respectively, in the context-insensitive setting. In the context-sensitive setting, the improvements are in the same order of magnitude but somewhat smaller.

Additional effort in analysis for achieving higher precision usually reduces the analysis performance. On average, we only measured a slow-down in performance of less than 10%; we even observed a speed-up in 4 of our 18 scenarios.

However, performance is work in progress. In order to find appropriate filters, we implemented a flexible analysis framework and chose straightforward implementations of intermediate analyses, e.g., for determining insertion points of filters. All this leads to an unacceptably low absolute performance of our implementations of both the baseline and the path-sensitive analysis. Future work will have to bring this performance up to an acceptable level. We have no doubt that this is possible since our original implementation of the baseline analysis (path-insensitive, flow-sensitive, context-insensitive and -sensitive) executes in the range of a few seconds [19]. However, the relative performance when adding

path-sensitivity might then change and, hence, needs to be assessed again.

Future work also includes identifying and implementing more types of filter operations.

Finally, the effects of path-sensitive points-to analysis should also be evaluated for flow-insensitive points-to analysis variants.

REFERENCES

[1] 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.

[2] 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.

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

[4] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 1999.

[5] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers - Principles, Techniques and Tools*. Addison-Wesley, 1986.

[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] M. Streckenbach and G. Snelting, "Points-to for Java: A general framework and an empirical comparison," Lehrstuhl für Softwaresysteme,Universität Passau, Germany, Tech. Rep., November 2000.

[8] D. Liang, M. Pennings, and M. Harrold, "Extending and evaluating flow-insensitive and context-insensitive points-to analysis for Java," in *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, June 2001, pp. 73–79.

[9] A. Rountev, A. Milanova, and B. G. Ryder, "Points-to analysis for Java based on annotated constraints," in *Proceedings of the Conference on Object-oriented Programmimg, Systems, Languages, and Applications (OOPSLA'01)*, October 2001.

[10] 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.

[11] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using Spark," in *Proceedings of the International Conference on Compiler Construction (CC'03)*, April 2003, pp. 153–169.

[12] 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.

[13] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for Java," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 1, pp. 1–41, 2005.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1989, pp. 25–35.

[15] M. Trapp, "Optimierung Objektorientierter Programme," Ph.D. dissertation, Universität Karlsruhe, December 1999.

[16] G. Lindenmaier, "libfirm – a library for compiler optimization research implementing firm," Fakultät für Informatik, Universität Karlsruhe, Germany, Tech. Rep. 2002-5, Sep 2002.

[17] G. Lindenmaier, M. Beck, B. Boesler, and R. Geiß, "Firm, an intermediate language for compiler research," Fakultät für Informatik, Universität Karlsruhe, Germany, Tech. Rep. 2005-8, 3 2005.

[18] 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)*, to appear in 2007.

[19] J. Lundberg, M. Edvinsson, and W. Löwe, "Fast and precise points-to analysis," The School of Mathematics and Systems Engineering, Växjö University, Växjö, Sweden, Tech. Rep., 2007.

[20] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[21] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 1, pp. 121–141, 1979.

[22] K. Cooper, T. Harvey, and K. Kennedy, "A simple, fast dominance algorithm," 2001.

[23] 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.

[24] J. Palsberg, "Object-oriented type inference," in *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, July 2001, pp. 20–27.

[25] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *Proceedings of Symposium on Principles of Programming Languages (POPL'88)*. New York, NY, USA: ACM Press, 1988, pp. 1–11.

[26] P. Havlak, "Construction of thinned gated single assignment form," in *1993 Workshop on Languages and Compilers for Parallel Computing*, no. 768. Portland, Ore.: Berlin: Springer Verlag, 1993, pp. 477–499.

[27] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1983, pp. 177–189.

[28] C. Mallon, "If-Konversion auf SSA," Studienarbeit, Universität Karlsruhe (TH), 2007.

[29] R. Bodík and S. Anik, "Path-sensitive value-flow analysis," in *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1998, pp. 237–251.

[30] M. Das, S. Lerner, and M. Seigle, "ESP: Path-sensitive program verification in polynomial time," in *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 2002, pp. 57–68.

[31] Y. Xie, A. Chou, and D. Engler, "ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors," in *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM Press, 2003, pp. 327–336.

[32] G. Ammons and J. R. Larus, "Improving data-flow analysis with path profiles," in *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 1998, pp. 72–84.

[33] A. Diwan, J. E. B. Moss, and K. S. McKinley, "Simple and effective analysis of statically typed object-oriented programs," in *Procedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, October 1996.

[34] J. Whaley and M. S. Lam, "An efficient inclusion-based points-to analysis for strictly-typed languages," in *Proceedings of the Static Analysis Symposium (SAS'02)*, 2002.

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

[36] 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.

[37] O. Shivers, "Control-flow analysis of higher-order languages," Ph.D. dissertation, Carnegie-Mellon University, 1991.