Set-Based Pre-Processing for Points-To Analysis

Yannis Smaragdakis George Balatsouras George Kastrinis

Department of Informatics
University of Athens, 15784, Greece
{smaragd,gbalats,gkastrinis}@di.uoa.gr

Abstract

We present set-based pre-analysis: a virtually universal optimization technique for flow-insensitive points-to analysis. Points-to analysis computes a static abstraction of how object values flow through a program's variables. Set-based pre-analysis relies on the observation that much of this reasoning can take place at the set level rather than the value level. Computing constraints at the set level results in significant optimization opportunities: we can rewrite the input program into a simplified form with the same essential points-to properties. This rewrite results in removing both local variables and instructions, thus simplifying the subsequent value-based points-to computation. Effectively, set-based pre-analysis puts the program in a normal form optimized for points-to analysis.

Compared to other techniques for off-line optimization of points-to analyses in the literature, the new elements of our approach are the ability to eliminate statements, and not just variables, as well as its modularity: set-based pre-analysis can be performed on the input just once, e.g., allowing the pre-optimization of libraries that are subsequently reused many times and for different analyses. In experiments with Java programs, set-based pre-analysis eliminates 30% of the program's local variables and 30% or more of computed context-sensitive points-to facts, over a wide set of benchmarks and analyses, resulting in a 24% average speedup (max: 103%, median: 20%).

1. Introduction

Points-to analysis consists of computing a static abstraction of all the data that a pointer variable (and, by extension, any pointer expression) can point to during program execution. In modern languages, points-to analysis forms the substrate of practically any other static analysis: any use of static analysis (e.g., for optimization, bug detection, program comprehension, online programming assistance) needs to discover the true value of an expression involving pointers (or "references", in Java and C#).

In a points-to analysis, objects are represented by their allocation sites, possibly qualified with a "context" for more precision. These object representations are the *values* of the analysis. Analysis algorithms operate at the value level, distinguishing between different values as the program requires. For instance, in a simple Java code fragment, such as the one below, the analysis needs to distinguish the different allocation sites and to reason about the flow of such values throughout the program.

```
1  p = new A();
2  p.f = new B();
3  q = new A();
4  q.f = new C();
5  r = p.f;
6  r.foo();
```

Even though lines 1 and 3 allocate an object of the same type, the allocations are distinguished (even in a *flow-insensitive* analysis, which considers all statements in *any* order). Such value-based reasoning is key: since the two objects are used in different ways throughout the fragment (e.g., their `f` field receives objects of different dynamic type) the distinction is kept and leads to higher analysis precision. Eventually, the fact that `r` can only be assigned the `B` object allocated in line 2 is essential for deciding what method will be called in line 6, due to dynamic dispatch.

Our *set-based reasoning/pre-analysis* technique is based on the observation that value-based reasoning is not always essential in the course of executing a points-to analysis algorithm. Instead, some reasoning can be performed entirely at the *set* level, i.e., by considering the entire set of values that a variable may hold as a black-box. This is a very general observation, practically applicable to any flow-insensitive points-to analysis algorithm. For a simple example, consider the three-statement pattern below[1] and their effects on a points-to analysis:

[1] For all such statement patterns throughout the paper, we consider the statements to appear *in any order and amidst any other program instructions*, as long as they are in the same procedure.

[Copyright notice will appear here once 'preprint' option is removed.]

```
1  p = q;
2  r = p;
3  r = q; // redundant
```

Regardless of what values flow into variables p, q, and r, the assignment in line 3 can be eliminated without affecting the results of the points-to analysis. The reasoning for this is entirely at the set level: the set of values flowing from q to r (due to the assignment of line 3) is a subset of the set of values flowing from p to r via the assignment of line 2, because the set of values flowing into q is a subset of those flowing into p (due to line 1). As we show later in detail, similar reasoning also applies to several cases of value flow through fields, establishing the redundancy of field-read, field-write, or pointer assignment instructions, e.g., in the two patterns below. (Again, statements may actually appear in any order, but they have to be in the same procedure.)

`r = q;`	`p.f = q;`
`p.f = r;`	`r = p.f;`
`p.f = q; // redundant`	`r = q; // redundant`

Contributions. The main contributions of our work are as follows:

- We introduce "set-based pre-analysis" as the idea of set-based reasoning and program transformation for points-to analysis, and demonstrate its potential via numerous optimizations. Set-based pre-analysis introduces a set-based (abstract) reasoning phase, to complement existing value-based (concrete) reasoning in points-to analysis.
- We implement set-based points-to reasoning as a pre-analysis and pre-transformation step over the Doop framework for Java points-to analysis by Bravenboer et al. [3]. This allows us to transparently apply set-based transformation to more than 20 different flow-insensitive points-to analysis algorithms and two different intermediate representations—the default Jimple representation of the Soot framework [19, 20] and a Static Single Assignment (SSA) version. None of the closest comparable past techniques have had such wide applicability.
- We evaluate the impact of set-based pre-analysis and pre-transformation over several large Java programs and the standard library. (Notably, no past work on pre-processing constraints has been applied and evaluated in the context of OO languages—our technique is intraprocedural and fully compatible with dynamic dispatch/on-the-fly-callgraph discovery.) In all, 30% of the program's local variables and the same amount or more of the context-sensitive points-to facts can be safely eliminated. This results in space savings and an average speedup of 24% over all analyses, with significantly higher numbers (up to 103%) for specific analyses and inputs.

The rest of the paper places our approach amongst its closest relatives in the literature (Section 2), introduces the base reasoning for points-to analysis and connects it to our optimization approach (Section 3), describes our patterns (Section 4), details our implementation (Section 5), presents experimental results (Section 6), and concludes (Section 7).

2. Placement of the Work

The literature on points-to analysis and its optimizations is extensive and covers intriguing breadth and depth. Therefore, it is useful to place our approach in relation to others early on, to make clear its similarities to the closest past work and its novelty.

The closest past work to our approach consists of techniques to establish that two variables are *clones*, i.e., that a variable's points-to set is identical to that of another. Such clone detection has been explored in the context of flow-insensitive C-language analyses, by techniques that are based on the concept of the *constraint graph*: a graph with nodes denoting pointer variables and an edge between nodes p and q denoting flow (e.g., a direct assignment) from variable p to variable q. Online cycle elimination by Fändrich et al. [4] detects cycles in the constraint graph and collapses all nodes in a cycle into a representative node, since such nodes will have identical points-to information. (An enhancement, which does not change the essence of online cycle elimination, is offered by the *projection merging* technique of Su et al. [**?**].) The technique of Nasre [13] extends such constraint graph reasoning based on the observation that if two nodes have the same dominator in the constraint graph, then they are clones: the values flowing to them are (only) those of the dominator node. Even more closely related to our approach are constraint-graph-based techniques that are applied off-line (i.e., before the points-to analysis runs). Prime examples of such techniques are Rountev and Chandra's [15] and Hardekopf and Lin's [7]. (Hardekopf and Lin have also applied similar ideas in a hybrid online/offline setting [6], but for the purposes of our work the offline technique is a closer comparable.) Both of these techniques perform an off-line detection of equivalent points-to sets and use this knowledge to eliminate redundant work in subsequent points-to computations. Hardekopf and Lin's approach is impressively general, computing hash codes that encode all the logical processing of a points-to set that is induced by the current program and, thus, detecting equivalent points-to sets even through complex program patterns.

Set-based pre-analysis has two benefits compared to all such past work:

- Generality: Our reasoning is not centered around points-to sets but around instructions, offering more opportunities for optimization. In constraint-graph terms, our approach can also eliminate edges of the graph, whereas past approaches could only eliminate entire nodes (which was the only way to eliminate their incident edges).
- Modularity: Past approaches applied such optimizations as an integral part of the analysis. The optimization was

tied to the representation structure (constraint-graph) used for the analysis implementation itself. In contrast, our approach can be applied as a local pre-processing step.

Specifically, set-based pre-analysis generalizes past approaches because it does not need to establish equivalence (i.e., that two variables are clones) in order to reap optimization benefits. All our earlier examples are applicable even when the variables involved are not clones of each other. For instance, consider the program pattern:

```
r = q;
p.f = r;
p.f = q; // redundant
```

Past approaches could avoid the computation of the last line but only if points-to sets r and q[2] (or q and p.f) were shown equivalent. Our observation is that merely knowing that points-to set r is a superset of points-to set q is sufficient for establishing the redundancy of the last instruction, when taken in conjunction with the second line. Thus, the above pattern is applicable even when the program contains other assignments to r and to p.f.

At the same time, set-based pre-analysis is more modular and orthogonal. Specifically, we apply set-based pre-analysis entirely intraprocedurally, i.e., without considering the subset relationships between points-to sets of local variables that occur in different methods. This allows set-based pre-analysis to be expressed as a *local* program transformation: every set-based pre-analysis optimization is a rewrite pattern that, once triggered, performs a simplification of the program used as input to the subsequent points-to analysis. Such simplifications consist of eliminating instructions or variables from the input program (together with renamings of use-sites of eliminated variables). E.g., in all our examples, the instruction labeled "redundant" is removed. This produces a *reduced* input program that condenses program behavior before the real points-to analysis starts. The result is tantamount to introducing a *new intermediate language*, optimized for the subsequent value-based points-to analysis. Different points-to analysis algorithms can then run on the reduced program and will yield results equivalent to applying them on the original input program.

In contrast, it is not always possible to express the closest comparable techniques in past literature [4, 7, 13, 15] as local program transformations: if two points-to sets are equivalent but the sets correspond to local variables at different scopes (e.g., a formal argument in a callee function and an actual in a caller), there is no local program transformation to express the elimination of one variable and its replacement by the other at every use site: each of the variables is out of scope at the use-site of the other. Our application of set-based reasoning only intra-procedurally means that the rewrite approach is always possible. This yields important modularity benefits:

1. When the program-under-analysis does not vary (but multiple analyses need to be performed) the program can be reduced once-and-for-all via set-based reasoning. Then, all points-to analyses can be performed over the reduced program.
2. Even when the program-under-analysis varies, much of the analysis complexity is due to combining the program with a large standard library. By using our purely-intraprocedural technique, we can pre-process large libraries once-and-for-all, and subsequently analyze them with any input program.
3. Our optimizations are easy to illustrate and understand, since they are pattern-based program transformations. The optimizations are also orthogonal to other complex reasoning in a points-to analysis (e.g., past off-line techniques do not work with online call-graph construction).

3. Set-Based Pre-Analysis and Points-To Analysis Via Subset Constraints

We next give background on points-to analysis from an angle that demonstrates the applicability of our set-based pre-analysis idea. As outlined in the Introduction, set-based pre-analysis is based on the observation that many points-to analysis inferences can be performed at the set level and not the value level. This insight is very general and applies to essentially any analysis (although different transformations may be valid for different kinds of analyses, as we discuss in Section 4).

The generality of the idea of set-based pre-analysis can be seen by first considering how different points-to analyses can be expressed in a unified setting. One of the most popular ways to express points-to analysis algorithms [3, 5, 10, 14, 21, 22] is via *subset constraints*. Subset constraints effectively state which value set has to be a subset of which other, with these value sets being sets of constants, the points-to sets of local variables, the field-points-to sets of object expressions, and more. Finding minimal sets that satisfy all the subset constraints produces the output of the points-to analysis.

Such analyses often leverage the Datalog programming language for their implementation. Datalog directly encodes recursive subset constraints. This means that the program under analysis is first encoded as data tables that represent all the program information. For instance, there is typically an input table representing each kind of program instruction in an intermediate language (e.g., tables Move, Alloc, Store, etc.). Then, the analysis is performed via rules that encode the subset constraints. For instance, the typical handling of Alloc and Move instructions (i.e., direct assignment of a newly allocated object to a variable and assignment between local variables, respectively) is via the following rules:

[2] More accurately: "the points-to sets for variables r and q", but we overload the terminology throughout, when no confusion should arise.

```
VarPointsTo(var, heap) <- Alloc(var,heap).
VarPointsTo(to, heap) <- Move(to, from),
                         VarPointsTo(from, heap).
```

The first of these rules states that the set of data in the `Alloc` table (which is an input table, whose rows encode the corresponding program instructions) is a subset of the `VarPointsTo` data for the entities (a variable and a heap object, i.e., a unique identifier of an allocation site) participating in the `Alloc` instruction. The second rule states that the points-to set for variable `from` is a subset of the points-to set for variable `to` if the program contains a `Move` instruction between `from` and `to`. Other rules can be used to introduce more subset constraints and eventually implement arbitrarily complex analyses. The computation performed by the analysis consists of successively enlarging the sets in order to satisfy all the subset constraints.

An interesting observation is that the variability between points-to analyses is usually not affecting the structure of the main rules for handling program instructions. Virtually all points-to analyses expressed in Datalog will have rules much like the above for handling `Alloc` and `Move` instructions. For instance, the GateKeeper analysis of Guarnieri and Livshits [5] has very analogous rules:

```
PTSTO(v, h) <- Alloc(v, h).
PTSTO(v1, h) <- PTSTO(v2, h), Move(v1, v2).
```

Our own context-sensitive analysis framework has rules that just add "context" variables to the above. (The context variables are used to vary the precision and performance of the analysis, but the mechanism for doing so is not important for our current discussion. A concise 9-rule model sufficient to express a large variety of points-to analyses can be found in Kastrinis et al. [9].)

```
VarPointsTo(var, ctx, heap, hctx) <-
  Alloc(var,heap).
VarPointsTo(to, ctx, heap, hctx) <-
  Move(to, from),
  VarPointsTo(from, ctx, heap, htcx).
```

The common structure of the rules in all these different analyses means that it is possible to make simplifications of the input program in a way that these simplifications apply to a multitude of analyses. Set-based pre-analysis is based on the observation that some subset constraints are always implied by others and can therefore be eliminated. In other words, all set-based pre-analysis patterns that we are going to examine in this paper are instances of the implication $S \subseteq T \land T \subseteq U \Rightarrow S \subseteq U$. This simple pattern can be applied to the constraints induced by different rules of existing points-to analyses, e.g., rules handling local assignments, field loads and stores, static fields, etc. For illustration, consider one of our earlier examples of a high-level program:

```
  p = q;
  r = p;
  r = q; // redundant
```

This example consists entirely of `Move` instructions at the intermediate language level. To show that the third instruction is redundant, consider that its use inside a subset-constraint-based points-to analysis is in a rule such as:

```
VarPointsTo(to, heap) <- Move(to, from),
                         VarPointsTo(from, heap).
```

For the third instruction, the rule is instantiated with `to` equal to program variable "`r`" and `from` equal to program variable "`q`". Thus, the rule's effect is to state that the points-to set of variable `q` is a subset of the points-to set of variable `r`—a constraint already inferrable by applying the same Datalog rule to the first two instructions.

In summary, the idea of set-based pre-analysis is highly general. Since virtually all points-to analysis algorithms can be expressed via subset constraints, applying common set-based reasoning on these constraints can determine that several of them are redundant. A key element is that such reasoning can be applied to simplify the program alone, independently of the analysis, under the expectation that all analyses of the same general family treat program features via similar rules.

4. Set-Based Pre-Analysis and Optimizations

We next present a collection of set-based pre-analysis instances as well as a general discussion on how these transformations are applied.

4.1 Set-Based Pre-Analysis Patterns

Our set-based pre-analysis instances are expressed as intra-procedural program transformations. Although each transformation may have specific pre-conditions of applicability, all of the transformations share a general structure: they consist of multiple program statements whose presence enables removing one or more other (redundant) statements. Therefore all transformations share their main applicability pre-condition: *the transformations are applicable to* flow-insensitive *points-to analyses (for which the statements of a procedure are considered to execute in any order) as long as the enabling statements appear anywhere in the same procedure.*

(Some of our ideas can be adapted to apply to a flow-sensitive setting, but, in that setting, points-to sets are kept per-instruction, so set-based reasoning is likely to be superseded by normal updates of points-to sets per-statement. Our implementation setting—the Doop analysis framework—only contains flow-insensitive points-to analyses.)

The optimizations can apply up to fixpoint, since applying one of them may enable others. Several of the patterns below do not often appear verbatim in practice but arise once variables start getting merged, other instructions eliminated, etc. We discuss this topic further in Section 4.2. Additionally, application of transformations should be done in a way that the enabling statements are not themselves eliminated by application of two transformations at the same time—an issue discussed in detail in Section 5.

Store statement elimination. Our first transformations eliminate store statements, i.e., assignments to pointer-indexed memory.

```
r = q;
p.f = r;
p.f = q; // redundant
```

Via standard subset-based reasoning, the third statement is redundant if the first two are present. The same applies to static store statements:

```
r = q;
C.f = r; // C is a class
C.f = q; // redundant
```

Load statement elimination. The next two transformations eliminate load statements, i.e., reads from pointer-indexed memory.

```
r = q;
q = p.f;
r = p.f; // redundant
```

Again, any value flowing to `r` through the load from `p.f` is redundant, since it also flows through the move from `q` (given that `q` also loads `p.f`). The same applies to static loads:

```
r = q;
q = C.f; // C is a class
r = C.f; // redundant
```

Move statement elimination. The next patterns eliminate move statements, i.e., copies between local variables. The first is our earlier example:

```
p = q;
r = p;
r = q; // redundant
```

However, the same flow of values can occur through assignments to pointer-indexed memory:

```
p.f = q;
r = p.f;
r = q; // redundant
```

And similarly for static fields:

```
C.f = q; // C is a class
r = C.f;
r = q; // redundant
```

Handling of array accesses. All of the earlier patterns also apply to load and store statements involving arrays instead of local objects. This is doubly interesting since points-to analyses often have very approximate handling of arrays, e.g., considering all array locations `arr[i]` to be the same abstract location `arr[*]`—an approach often called *array insensitive*. Thus, for instance, we can eliminate array loads:

```
r = q;
q = arr[*];
r = arr[*]; // redundant
```

Similarly, we can use array loads to eliminate move statements (and in general can adapt all earlier patterns to array statements):

```
arr[*] = q;
r = arr[*];
r = q; // redundant
```

Method call elimination. An interesting observation is that method call statements can also be eliminated using set-based reasoning:

```
r = q;
q = p.m();
r = p.m(); // redundant
```

The above is not limited to no-argument methods, but the arguments need to be identical local variables for the transformation to apply.[3]

As usual, analogous transformations apply to static methods:

```
r = q;
q = C.m(); // C is a class
r = C.m(); // redundant
```

It is somewhat surprising that method calls can be eliminated, as above. After all, the usual semantics of imperative languages dictate that identical method calls cannot be merged since they can have different effects on state. A flow-insensitive points-to analysis, however, computes an over-approximation of all executions of a program, assuming that every reachable method is executed an unbounded number of times. That is, upon encountering a method, the analysis takes into account not just a single execution of the method but the maximal effects that *any number* of executions might have on the points-to information. Thus, points-to analyses do not model state changes performed by a method in a way that repeated equivalent actions make a difference.

Similarly, the above transformation is valid even when the analysis adds context, of any usual kind. For instance, as can be seen in models of various kinds of context-sensitivity [9, 18], new contexts are created at method call sites and the context remains the same throughout the method body. This means that local variables of the same calling method have the same context throughout the method body, i.e., hold the same values as far as the analysis is concerned. Therefore, two calls to a method `m` may be analyzed in different contexts but if their arguments (and receiver object) are lexically identical then the two calls receive the same information from the outside world. Therefore, the information computed for the body of the called method, `m`, will be identical under both contexts. Thus, analyzing the function twice has no effect on its callers or on the heap model—the only difference is the (undesirable) replication of identical information in the analysis of the method itself.

[3] Since methods can cause exceptions to be thrown, in the case of precise exception handling [**?**] an extra requirement is that no exception handler starts or ends between the two equivalent method calls.

Duplicate statement elimination. An "obvious" use of set-based reasoning is to eliminate duplicate statements (e.g., two instances of "`p = q;`", "`q = p.f;`" or "`q = p.m();`"). For many kinds of points-to analyses, duplicate statements are not even represented in the analysis input. (For our implementation setting—the Doop framework—the only duplicate statements represented in the input are method calls. Thus, this transformation has been largely applied to the input implicitly even before our work.)

Interestingly, program transformation patterns can allow some variability when detecting duplicate statements. For instance, consider:

```
q = p.m();
p.m(); // redundant
```

The second call to `p.m()` is redundant even though its form is not identical to the first call—the return value is ignored. In practice, this detection pattern needs to be specified separately, for most intermediate languages. Note also that the first statement is not made redundant by the existence of the second: it defines variable `q`, while the second does not.

Duplicate variable elimination. An important optimization in set-based reasoning imitates the variable elimination logic of past work that builds on the constraint graph abstraction [4, 7, 13, 15]. The constraint graph is a graph with nodes denoting pointer variables and expressions, and edges between them denoting value flow. The graph encodes all known subset relations between points-to sets. For instance, an assignment "`p = q;`" implies an edge from points-to set `q` to points-to set `p` in the constraint graph, as well as an edge from set `q.f` to set `p.f`, etc. Similarly, an assignment "`q = p.f;`" implies an edge from set `p.f` to set `q` in the constraint graph. Similar edges are induced by other program constructs (e.g., store statements) and the transitivity property is applied. On the resulting graph, two local variables are clones of each other whenever *any* of the following conditions apply:[4]

- The variables belong in the same strongly-connected-component of the constraint graph.
- The variables have identical in-flows. E.g.,

  ```
  q = p.f;
  r = p.f;
  q,r receive no other assignments
  ```

  or:

  ```
  q = p.m();
  r = p.m();
  q,r receive no other assignments
  ```

  The above are the most profitable special cases of the general approach of Hardekopf and Lin [7] for identifying equivalent flows.
- The variables have the same dominator in the constraint graph: the values flowing to them are the same since they are (only) those of the dominator node. This is Nasre's insight [13], which is also handled by the reasoning in Hardekopf and Lin's approach [7].

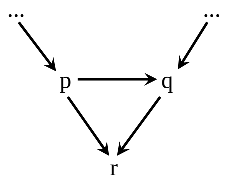
When clone variables are detected, one of them can be eliminated. All def-sites of the eliminated variable are removed from the program and all use-sites are renamed to use the other clone variable.

4.2 How Transformations Are Applied

Set-based pre-analysis separates set-based (abstract) from value-based (concrete) reasoning in points-to analysis. Effectively the approach normalizes programs into a normal form suitable for quick points-to analysis execution. It is important that this normalization is performed in an intra-procedural setting, so that the results of the transformation can be reused independently of other changes to the code. For instance, large libraries can be transformed once and the result of the transformation can be reused for any program using the library.

The transformations we just saw can be applied up to fix-point. The reason is that there is a synergy between the two kinds of transformations: statement-elimination can enable variable-elimination and vice-versa. For an example of the former direction, consider (in the program fragment below, together with its associated constraint subgraph) the rule that two variables are equivalent when they have the same dominator [13]:

```
p = new Object();
    // or any other in-flow, e.g., calls
q = new Object();
    // or any other in-flow, e.g., calls
q = p;
r = q;
r = p;
```



Due to the external flow to `p` and `q`, there is no dominance relationship among any of the three nodes in this (sub)graph. Yet the assignment "`r = p;`" is redundant, as established via set-based reasoning. Consequently, if we remove the `p`-to-`r` edge, the resulting constraint graph has `q` as the dominator of `r`: any value flowing to `r` has to go through `q`. Indeed, this is an instance where set-based pre-analysis generalizes past approaches. The above example is handled by a special algorithm (called the *HU algorithm*) in

[4] Note that this reasoning only applies to local variables. Field expressions cannot be eliminated with such local examination. Recall that all our optimizations are entirely intraprocedural and are expressed as local program transformations.

Hardekopf and Lin's approach [7]. The HU algorithm aims precisely at exploiting subset relationships in the course of determining the equivalence of other points-to sets. (Recall, however, that Hardekopf and Lin's approach, just like other past constraint-graph-based approaches, only yields benefit when it discovers equivalent variables to eliminate, whereas our approach can also eliminate redundant constraint graph edges/instructions regardless of whether the points-to sets involved end up being equivalent or not.)

Examples of the converse direction are even more common: eliminating a variable can trigger any of the statement-eliminating optimizations. For instance, the program may contain statements:

```
r = q;
q = p.m(a);
r = s.m(b);
```

If earlier steps establish that p and s are clones, and that a and b are clones, then normalizing the use-sites of all clone variables reveals that the last statement is redundant.

4.3 Illustration

To see the simplification that set-based pre-processing can introduce, consider its application to an example method from the JDK, `java.util.TreeMap.rotateRight`. Figure 1 shows the full body of the original method in the Jimple intermediate language of the Soot framework [19, 20]. Running our analysis determines that several of the local variables are redundant and the method body can be significantly simplified. (`r0` is cloning `@this`; `r1` is cloning `@param0`; `r4` and `r5` are cloning `r3`; `r7`, `r8`, `r10`, and `r11` are cloning `r6`.) The result of the transformation can be seen in Figure 2. [5] As can be seen, the reduced form is much shorter than the original and eliminates internal complexity. It also allows us to illustrate some important points.

First, the reduced form of the bytecode is obtained under the assumption of flow-insensitive points-to analysis and, thus, is *not* guaranteed to be equivalent to the original, or even legal (e.g., may violate conventions of the intermediate language). For instance, the reduced program may be using variables that are only assigned in different flows of control. Nevertheless, the simplified method body is a faithful substitute of the original as far as flow-insensitive points-to analysis is concerned. Even if we were to apply the transformations with a flow-sensitive analysis in mind (e.g., only apply transformations that eliminate duplicate actions), the result would not be guaranteed equivalent. For instance, in normal execution, reading the same field twice or calling the same method twice is not the same as reading the field or calling the method just once.

[5] This result was produced manually, since our implementation does not affect the textual form of the intermediate language. However, the manual mapping depicts the actual simplifications of the input program as performed by our implementation.

Also note that the reduction of the program may be affecting externally visible elements. For most client analyses this is not the case—e.g., analyses finding reachable methods or computing a connectivity graph of heap objects are unaffected by set-based pre-processing. Yet other analyses may have a concept of internal elements, such as exact calling instructions or temporary local variables. For instance, the user of the analysis may request the points-to set of eliminated local variable `r1`, or the target methods of an eliminated call-site. Computing this information is a mere matter of post-processing and does not affect the inherent precision or correctness of the analysis. That is, the analysis on the reduced program is fully equivalent to that on the original, yet the output information can be viewed as being in a condensed form. Similar no-loss condensed representations are common in points-to analysis algorithms (e.g., for exception object merging [8]). It is straightforward to reproduce the original output, if so desired by the client analysis, and this can also be done lazily, upon request. For instance, instead of querying the points-to set of variable `r1`, a client analysis will find the variable that replaced `r1` and query *its* points-to set. To simulate the full original points-to set for all variables in the program, we only need to combine the condensed points-to information with the information of which local variables were replaced by which others. As we discuss in Section 6, this per-analysis post-processing has virtually zero cost.

5. Implementation

Our implementation of set-based pre-analysis is in the context of the Doop framework for Java points-to analysis by Bravenboer et al. [3]. Doop expresses a large variety of flow-insensitive points-to analyses declaratively, using the Datalog language. Our set-based pre-analysis implementation applies transparently to all Doop analyses as a pre-processing step over their normal input. Although this step could be implemented in some other language, we chose to use Datalog in our implementation for reasons of convenience and engineering uniformity.

Specifically, set-based pre-analysis is a pre-computation over all input tables of the regular points-to analysis. As mentioned earlier, such tables represent all syntactic constructs, i.e., instruction types, of the intermediate language. Since our set-based pre-analysis implementation is declarative, it cannot alter the input tables in-place. Instead, the optimization is performed as a sequence of alternating phases. First, a *detection* phase discovers all opportunities for optimization over the entire program (as if) in parallel.[6] Then, a *transformation* phase produces simplified new input tables by computing the result of the optimizations over the original tables. These two steps repeat until no more optimization is profitable.

[6] Currently there is no real parallel computation, although this is entirely up to the Datalog evaluation engine.

```
private void rotateRight(java.util.TreeMap$Entry)
        java.util.TreeMap r0;
        java.util.TreeMap$Entry r1, r2, $r3, $r4, $r5, $r6, $r7, $r8, $r9, $r10, $r11;

        r0 := @this: java.util.TreeMap;
        r1 := @param0: java.util.TreeMap$Entry;
        if r1 == null goto label4;

        r2 = r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry left>;
        $r3 = r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right>;
        r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry left> = $r3;
        $r4 = r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right>;
        if $r4 == null goto label0;

        $r5 = r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right>;
        $r5.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = r1;
label0: $r6 = r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent>;
        r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = $r6;
        $r7 = r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent>;
        if $r7 != null goto label1;

        r0.<java.util.TreeMap: java.util.TreeMap$Entry root> = r2;
        goto label3;
label1: $r8 = r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent>;
        $r9 = $r8.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right>;
        if $r9 != r1 goto label2;

        $r10 = r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent>;
        $r10.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right> = r2;
           goto label3;
label2: $r11 = r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent>;
        $r11.<java.util.TreeMap$Entry: java.util.TreeMap$Entry left> = r2;
label3: r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right> = r1;
        r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = r2;
label4: return;
```

Figure 1. Original JDK method in Jimple form (manually reformatted for space).

```
private void rotateRight(java.util.TreeMap$Entry)
        java.util.TreeMap$Entry r2, $r3, $r6, $r9;

        if @param0 == null goto label4;

        r2 = @param0.<java.util.TreeMap$Entry: java.util.TreeMap$Entry left>;
        $r3 = r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right>;
        @param0.<java.util.TreeMap$Entry: java.util.TreeMap$Entry left> = $r3;
        if $r3 == null goto label0;

        $r3.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = @param0;
label0: $r6 = @param0.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent>;
        r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = $r6;
        if $r6 != null goto label1;

        @this.<java.util.TreeMap: java.util.TreeMap$Entry root> = r2;
        goto label3;
label1: $r9 = $r6.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right>;
        if $r9 != @param0 goto label2;

        $r6.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right> = r2;
        goto label3;
label2: $r6.<java.util.TreeMap$Entry: java.util.TreeMap$Entry left> = r2;
label3: r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right> = @param0;
        @param0.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = r2;
label4: return;
```

Figure 2. Reduced JDK method. Labels remain and should help in matching with the corresponding regions of Figure 1.

The above scheme introduces some subtleties. The first concerns the detection phase: Discovering the potential for all optimizations in parallel means that we have to manually ensure to never optimize away an instruction that may enable another optimization. This requires all of our declarative rules to have *disabling conditions*. For instance, consider our usual pattern for eliminating load instructions:

```
  r = q;
  q = p.f;
  r = p.f; // redundant
```

This pattern is implemented by the Datalog rule below (simplified, with variables renamed for easy correspondence with the example):

```
RedundantLoad(_r, _p, _f) <-
  Load(_r, _p, _f),
  Load(_q, _p, _f),
  Move(_r, _q),
  _r != _q,
  !TransitiveFlow(_r, _q).
```

The last two conditions of the rule ensure that logical variables _r and _q represent different program variables (which is a necessary precondition in any setting) and that there is no flow (i.e., no direct or indirect assignments) from _r to _q. The latter condition is included only because of the declarative evaluation of the rules, which evaluates them as-if-in-parallel. Its result is to guard against removing one of the first two lines of the pattern. Without it, the same pattern would match twice in code such as:

```
  z = y;
  y = z;   // could be more complex flow
  z = x.f; // redundant
  y = x.f; // also redundant but not both
```

Matching the pattern twice would remove both load instructions (i.e., both of the last two lines) which is erroneous. Some of our rules have numerous disabling conditions as a result of similar reasoning—for instance, the rule to compute redundant `Move` instructions has six extra conditions, to avoid accidental cycles as well as overlap with other patterns that would invalidate the rewrite.

A second subtlety concerns the transformation phase: Although multiple optimizations may be legitimately enabled, there may be overlap in their application. For a declarative implementation, where the rules can be applied in any order, we need to explicitly disambiguate what happens in case of such overlap. For instance, consider the program statement `p.f = q;`. Our rules distinguish the following cases of enabled transformations:

- The entire statement is redundant and, thus, eliminated.
- The statement is not redundant, but variable `p` alone is a redundant copy of some other variable `s`. The above "use" of variable `p` should be replaced by `s`.
- The statement is not redundant, but variable `q` alone is a redundant copy of some other variable `r` (with a similar transformation as before).
- The statement is not redundant, but both variables `p` and `q` are redundant copies of variables `s` and `r`, respectively. Thus, both variable uses should be replaced.

The above complications are typical of rule-based rewrite systems and should also apply to other implementations that employ pattern-based program transformation for set-based pre-processing. Note that the correctness of the above transformations (especially due to the potential of removing statements that enable other transformations performed in parallel) is left to the implementor of the transformations. For our implementation, we have followed a highly stylized pattern that orders optimizations for every statement kind from stronger to weaker, as in the example of the bullet points above. Additionally, every optimization transformation includes extra conditions to ensure that its enabling statements are not themselves eliminated. Of course, human error can always creep in. In practice we found that it is quite easy to debug our rewrite rules, since they produce semantically equivalent input programs from the perspective of subsequent points-to analysis. Observing a single detailed metric that remains invariant by our optimizations is typically enough to detect even rare errors in our logic. (An excellent such metric has been the context-sensitive "instance field points-to" value, which sums together the sizes of points-to sets of all heap objects. This value typically has 6 or more significant digits and is very sensitive to any semantic change in the input program.)

Our implementation of set-based analysis and transformation currently consists of approximately 150 Datalog rules, some of which are fairly involved.[7] Recall that, in addition to transformation patterns, these rules implement algorithms for strongly-connected components, dominance, etc., over the constraint-graph abstraction. This logic usually takes well under a minute to apply to the programs of our benchmark set together with the full JDK library. The running time can be significantly reduced further—e.g., we can enable only the most profitable transformations for roughly half the cost. Furthermore, the only inevitable time cost is that of the detection phase, i.e., of identifying all the sites where the optimization will take place. This typically takes 10secs or less. The rest of the time consists of program transformation and is bloated due to low-level engineering considerations. (E.g., on every transformation our declarative engine incrementally adjusts the results of the analysis even though they will not be needed.) We did not try to improve the speed of detection and transformation, at the cost of complicating our implementation, because this speed is largely irrelevant. As discussed earlier, our analysis cost is

[7] Our implementation can be found in http://doop.program-analysis.org/ and mainly in files logic/transform.logic and logic/transform-delta.logic.

one-off: the program is transformed once and for all, and can be reused in its reduced form for any number of further analyses or queries. Additionally, the vast majority of the cost is not concerning the program but the library. The above (sub-minute average) times include the set-based analysis and transformation of the entire JDK 1.6 (not just its parts reachable by the current program under analysis). Therefore, even in a setting where one wants to re-analyze the program regularly (e.g., because it is under current development) the library can be analyzed and transformed only once, with the resulting reduced library re-used for every program.

6. Experiments

We evaluated the impact of set-based pre-analysis on 7 representative analyses from the Doop framework. The analyses span a wide range of precision and performance, comprising a context-insensitive Andersen-style points-to analysis (*insens*), and context-sensitive variants both with and without a context-sensitive heap abstraction (a.k.a. heap cloning) for different kinds of context-sensitivity: call-site-sensitive [16, 17] (*1call*, *1call+H*), object-sensitive [11, 12] (*1obj*, *1obj+H*, *2obj+H*), and type-sensitive [18] (*2type+H*). (Comparing the precision of these known algorithms is outside the scope of this work, but such measurements for our exact setting can be found in past literature [9].)

We used two different intermediate representations (IRs) in our evaluation. The first is the default form of the Jimple intermediate language of the Soot framework [19, 20]. The second is a Static Single Assignment (SSA) version of an otherwise similar intermediate language, also supported by the Soot framework. The reason for trying both intermediate languages was to see whether the impact of set-based pre-analysis would be significantly greater on a representation that is profligate with local variables (SSA) vs. a representation that was designed merely as a convenient intermediate language of a major compiler framework (Jimple).

Our evaluation setting uses the LogicBlox Datalog engine, v.3.9.0, on a Xeon X5650 2.67GHz machine with only one thread running at a time and 24GB of RAM (i.e., ample for the analyses studied). We analyze the DaCapo benchmark programs under JDK 1.6.0_37. We use the same settings as earlier published work [1, 18]: jython and hsqldb are analyzed with reflection disabled and hsqldb has its entry point set manually in a special harness.

Tables 1-8 show the results of our experiments. Missing entries correspond to analyses that did not terminate in 90mins.

Running time. Tables 1 and 2 (for the Jimple and SSA IR, respectively) present the running times of all analyses and compute the speedup afforded by set-based optimization. (All running time numbers given are medians of three runs.)

Set-based pre-analysis has a significant impact on the running time of almost every analysis, with its highest impact on call-site-sensitive analyses. There is virtually no program that does not consistently benefit from set-based optimization and we see overall speedups that are as high as 103%, with averages around 20%. Although some programs clearly benefit more than others, the result is not particularly pronounced: note how the maximum and minimum speedup entries (in bold) are distributed over several columns and are often not far from numbers for other benchmarks over the same analysis.

The choice of intermediate representation does not affect the effectiveness of our technique much, either. Although the SSA form introduces more local variables, the difference in speedup is small and mostly within noise levels. This shows that even a human-designed intermediate representation (Jimple) offers enough opportunities for set-based optimization. Recall that the elimination of variables and IR statements by set-based optimization is not something that a regular intermediate language could replicate: the optimization is valid only for the purposes of points-to analysis, not for the purposes of program execution.

Variables eliminated. Tables 3 and 4 (for the Jimple and SSA IR, respectively) show the numbers of reachable local variables (as computed by the points-to analysis itself) both with and without set-based optimization. This is a useful metric for seeing how much of the program (together with reachable code in the JDK libraries) is distilled away when applying set-based optimization. Importantly, this measure is static: it counts eliminated variables in the program text. As we can see, this does not correlate well with the speedup numbers from the earlier tables. Indeed, the reduction percentage for local variables is remarkably steady over all benchmarks, at roughly 30%. The number also does not vary much over analyses, but this is quite expected: the only impact the analysis has on the number of variables eliminated is because of changes in reachable code. (More precise analyses, e.g., 2obj+H, have fewer reachable variables than imprecise ones, e.g., insens. Still, this variance hardly affects the average reduction in reachable variables due to set-based pre-analysis.)

On these tables we can see a little more clearly the effect of IR choice. The SSA form has consistently higher variable counts than the Jimple form. The reduction percentages are also consistently (but very slightly) higher, but not nearly as much as the difference in absolute variable counts between Jimple and SSA. This shows more vividly that the speedup of set-based optimization is not due to eliminating variables that would be redundant in the IR anyway.

Instructions eliminated.

Context-sensitive facts eliminated. Tables 7 and 8 (for the Jimple and SSA IR, respectively) show the effect of set-based optimization on perhaps the most important internal complexity metric of a points-to analysis: the cumulative size of context-sensitive points-to sets. This is a metric that correlates very well with the memory requirements of the

		antlr	bloat	chart	eclipse	hsqldb	jython	luindex	lusearch	pmd	xalan	**AVG**
insens	original	69.30	59.59	123.44	47.74	57.05	55.96	40.97	42.07	59.57	64.85	
	set-based	62.13	50.81	105.88	42.04	49.62	49.36	36.73	37.57	53.61	57.05	
	speedup	11.54%	**17.28%**	16.58%	13.56%	14.97%	13.37%	11.54%	11.98%	**11.12%**	13.67%	13.56%
1obj	original	166.26	373.93	1240.59	117.66	218.33	119.96	76.61	89.18	135.84	189.40	
	set-based	148.18	314.23	1188.23	105.71	178.31	108.62	68.39	82.05	121.84	168.55	
	speedup	12.20%	19.00%	**4.41%**	11.30%	**22.44%**	10.44%	12.02%	8.69%	11.49%	12.37%	12.44%
1obj+H	original	810.71	1593.31	-	555.22	4335.92	832.50	240.08	262.64	332.01	803.66	
	set-based	637.04	1253.69	-	478.98	3295.88	749.98	205.87	226.77	282.76	641.49	
	speedup	27.26%	27.09%	-	15.92%	**31.56%**	**11.00%**	16.62%	15.82%	17.42%	25.28%	20.88%
2obj+H	original	217.53	5060.39	896.04	532.55	-	-	131.01	183.31	167.71	4521.01	
	set-based	182.77	3670.61	712.25	464.14	-	-	107.83	159.04	139.84	4150.37	
	speedup	19.02%	**37.86%**	25.80%	14.74%	-	-	21.50%	15.26%	19.93%	**8.93%**	20.38%
2type+H	original	108.13	142.85	211.91	152.45	194.73	731.41	75.22	76.29	114.48	168.16	
	set-based	92.83	116.45	179.59	123.79	153.24	616.92	64.95	65.94	98.86	142.07	
	speedup	16.48%	22.67%	18.00%	23.15%	**27.08%**	18.56%	15.81%	**15.70%**	15.80%	18.36%	19.16%
1call	original	110.09	186.30	288.43	81.42	90.49	88.29	59.36	63.62	89.97	108.70	
	set-based	83.93	117.55	228.08	64.63	68.54	72.16	48.88	52.48	73.69	87.04	
	speedup	31.17%	**58.49%**	26.46%	25.98%	32.03%	22.35%	21.44%	**21.23%**	22.09%	24.89%	28.61%
1call+H	original	366.16	1351.58	957.15	478.31	332.63	401.10	171.59	186.63	245.95	470.74	
	set-based	241.75	670.30	701.25	317.24	196.89	265.98	126.00	132.44	171.69	371.35	
	speedup	51.46%	**101.64%**	36.49%	50.77%	68.94%	50.80%	36.18%	40.92%	43.25%	**26.76%**	50.72%

Table 1. Execution time (in seconds) for a variety of analyses on various benchmarks using the Jimple intermediate language. Maximum and minimum speedups per row are shown in bold.

		antlr	bloat	chart	eclipse	hsqldb	jython	luindex	lusearch	pmd	xalan	**AVG**
insens	original	69.23	59.49	124.16	47.49	56.70	55.33	41.16	41.95	60.12	65.90	
	set-based	62.32	50.78	105.87	42.26	49.92	49.43	36.83	37.78	53.55	57.94	
	speedup	11.09%	17.15%	**17.28%**	12.38%	13.58%	11.94%	11.76%	**11.04%**	12.27%	13.74%	13.22%
1obj	original	166.39	358.83	1256.02	118.34	223.60	119.45	76.27	89.28	134.19	190.18	
	set-based	147.11	305.15	1111.43	104.34	178.63	106.17	67.83	80.96	121.58	169.02	
	speedup	13.11%	17.59%	13.01%	13.42%	**25.17%**	12.51%	12.44%	**10.28%**	10.37%	12.52%	14.04%
1obj+H	original	815.78	1460.25	-	625.52	4349.26	843.57	245.65	261.08	340.82	793.59	
	set-based	650.36	1251.08	-	471.16	3143.83	708.67	210.27	223.44	288.38	710.26	
	speedup	25.44%	16.72%	-	32.76%	**38.34%**	19.04%	16.83%	16.85%	18.18%	**11.73%**	21.76%
2obj+H	original	223.09	4621.55	920.44	535.00	-	-	131.41	139.95	168.02	4621.02	
	set-based	184.40	3533.01	684.62	429.30	-	-	108.21	115.17	140.98	4246.40	
	speedup	20.98%	30.81%	**34.45%**	24.62%	-	-	21.44%	21.52%	19.18%	**8.82%**	20.38%
2type+H	original	108.17	138.99	226.27	151.12	197.54	726.93	76.37	76.76	115.65	172.52	
	set-based	93.94	114.91	182.41	124.22	154.86	609.35	65.37	66.44	99.36	143.88	
	speedup	**15.15%**	20.96%	24.04%	21.66%	**27.56%**	19.30%	16.83%	15.53%	16.39%	19.91%	19.73%
1call	original	108.97	186.63	281.24	81.88	90.54	88.24	59.68	64.30	90.39	110.65	
	set-based	83.71	117.58	220.83	63.98	67.86	71.51	49.64	52.26	73.07	87.14	
	speedup	30.18%	**58.73%**	27.36%	27.98%	33.42%	23.40%	**20.23%**	23.04%	23.70%	26.98%	29.50%
1call+H	original	360.12	1419.06	896.61	442.65	308.10	377.58	171.58	183.68	244.42	461.27	
	set-based	239.75	698.93	664.32	307.61	187.22	256.70	122.87	130.26	169.69	337.39	
	speedup	50.21%	**103.03%**	**34.97%**	43.90%	64.57%	47.09%	39.64%	41.01%	44.04%	36.72%	50.52%

Table 2. Execution time (in seconds) for a variety of analyses on various benchmarks using an SSA version of the intermediate language. Maximum and minimum speedups per row are shown in bold.

analysis and has the advantage of being impervious to platform and implementation fluctuations: an optimization that speeds up execution could do so by taking advantage of the specifics of the environment—e.g., peculiarities of our Datalog execution engine. Improvement in context-sensitive points-to set sizes, however, is a change that transfers well to completely different implementation settings.[8]

[8] It is telling that analysis implementations that use binary decision diagrams (BDDs) try hard to minimize this metric in order to achieve peak performance [2].

		antlr	bloat	chart	eclipse	hsqldb	jython	luindex	lusearch	pmd	xalan	**AVG**
insens	original	87,906	91,859	138,286	83,269	88,809	76,164	66,649	71,330	77,653	87,983	
	set-based	63,050	64,377	97,760	59,149	61,214	52,982	47,173	50,063	54,228	61,308	
	reduction	**28.28%**	29.92%	29.31%	28.97%	**31.07%**	30.44%	29.22%	29.81%	30.17%	30.32%	29.75%
1obj	original	86,409	90,302	135,458	81,228	87,891	75,383	65,032	69,716	76,101	86,403	
	set-based	62,019	63,318	95,893	57,751	60,580	52,438	46,047	48,937	53,166	60,224	
	reduction	**28.23%**	29.88%	29.21%	28.90%	**31.07%**	30.44%	29.19%	29.81%	30.14%	30.30%	29.71%
1obj+H	original	86,154	90,012	-	80,613	87,076	74,738	64,777	69,306	75,666	86,042	
	set-based	61,826	63,106	-	57,331	59,970	51,949	45,854	48,631	52,816	59,965	
	reduction	**28.24%**	29.89%	-	28.88%	**31.13%**	30.49%	29.21%	29.83%	30.20%	30.31%	29.80%
2obj+H	original	84,605	88,387	108,722	78,545	-	-	63,203	67,689	73,886	83,879	
	set-based	60,699	61,941	78,004	55,833	-	-	44,706	47,450	51,509	58,332	
	reduction	28.26%	29.92%	**28.25%**	28.92%	-	-	29.27%	29.90%	30.29%	**30.46%**	29.41%
2type+H	original	84,805	88,725	114,663	78,872	85,672	72,354	63,403	67,889	74,128	84,175	
	set-based	60,853	62,172	81,913	56,073	58,972	50,140	44,860	47,604	51,701	58,553	
	reduction	**28.24%**	29.93%	28.56%	28.91%	**31.17%**	30.70%	29.25%	29.88%	30.25%	30.44%	29.73%
1call	original	86,684	90,567	135,361	81,372	88,181	75,637	65,427	70,111	76,396	86,678	
	set-based	62,204	63,497	95,852	57,846	60,779	52,608	46,327	49,217	53,363	60,410	
	reduction	**28.24%**	29.89%	29.88%	28.91%	**31.07%**	30.45%	29.19%	29.80%	30.15%	30.31%	29.79%
1call+H	original	86,684	90,567	135,230	81,372	88,181	75,637	65,427	70,111	76,396	86,678	
	set-based	62,204	63,497	95,748	57,846	60,779	52,608	46,327	49,217	53,363	60,410	
	reduction	**28.24%**	29.89%	29.20%	28.91%	**31.07%**	30.45%	29.19%	29.80%	30.15%	30.31%	29.72%

Table 3. Number of reachable (local) variables for the Jimple intermediate language representation. Maximum and minimum reduction percentages per row are shown in bold.

		antlr	bloat	chart	eclipse	hsqldb	jython	luindex	lusearch	pmd	xalan	**AVG**
insens	original	91,642	95,196	144,580	86,577	92,942	79,450	69,513	74,513	80,615	92,463	
	set-based	65,407	66,558	101,850	61,196	63,698	54,995	48,973	52,060	56,114	64,203	
	reduction	**28.63%**	30.08%	29.55%	29.32%	**31.46%**	30.78%	29.55%	30.13%	30.39%	30.56%	30.05%
1obj	original	90,029	93,523	141,628	84,780	92,035	78,710	67,764	72,771	78,947	90,775	
	set-based	64,315	65,438	99,889	59,982	63,072	54,484	47,771	50,859	54,991	63,064	
	reduction	**28.56%**	30.03%	29.47%	29.25%	**31.47%**	30.78%	29.50%	30.11%	30.34%	30.53%	30.04%
1obj+H	original	89,774	93,233	-	84,145	91,210	78,048	67,509	72,361	78,503	90,408	
	set-based	64,122	65,226	-	59,545	62,454	53,983	47,578	50,553	54,633	62,800	
	reduction	**28.57%**	30.04%	-	29.24%	**31.53%**	30.83%	29.52%	30.14%	30.41%	30.54%	30.09%
2obj+H	original	88,201	91,588	113,387	82,038	-	-	65,911	70,718	76,680	88,196	
	set-based	62,978	64,048	80,997	58,020	-	-	46,413	49,353	53,291	61,135	
	reduction	28.60%	30.07%	**28.57%**	29.28%	-	-	29.58%	30.21%	30.50%	**30.68%**	29.69%
2type+H	original	88,401	91,928	119,598	82,392	89,782	75,631	66,111	70,918	76,927	88,496	
	set-based	63,132	64,281	85,079	58,283	61,438	52,149	46,567	49,507	53,488	61,358	
	reduction	**28.58%**	30.07%	28.86%	29.26%	**31.57%**	31.05%	29.56%	30.19%	30.47%	30.67%	30.03%
1call	original	90,264	93,748	141,415	84,851	92,283	78,921	68,135	73,142	79,202	91,012	
	set-based	64,470	65,587	99,791	60,022	63,240	54,621	48,036	51,124	55,158	63,221	
	reduction	**28.58%**	30.04%	29.43%	29.26%	**31.47%**	30.79%	29.50%	30.10%	30.36%	30.54%	30.07%
1call+H	original	90,264	93,748	141,251	84,851	92,283	78,921	68,135	73,142	79,202	91,012	
	set-based	64,470	65,587	99,665	60,022	63,240	54,621	48,036	51,124	55,158	63,221	
	reduction	**28.58%**	30.04%	29.44%	29.26%	**31.47%**	30.79%	29.50%	30.10%	30.36%	30.53%	30.07%

Table 4. Number of reachable (local) variables for an SSA version of the intermediate language. Maximum and minimum reduction percentages per row are shown in bold.

As seen on the tables, set-based optimization has significant impact on the sizes of context-sensitive points-to sets. More than 30% of points-to facts on average never need to be inferred in the optimized version of the input. This difference affects the complexity of the analysis itself but not its outcome: the final, context-insensitive points-to sets for the same variable or object field will be identical, since our transformation is semantics-preserving relative to the points-to analysis.

Post-processing. Set-based pre-analysis leaves the output of points-to analysis in a condensed form, as far as certain further analyses are concerned. This is the case for client

		antlr	bloat	chart	eclipse	hsqldb	jython	luindex	lusearch	pmd	xalan
	original	97342	50499	112656	49593	105074	116649	49373	49373	111388	105636
Move	removed	94300	49012	109104	48099	101775	113265	47930	47930	107967	102208
	reduction	96.87%	97.05%	**96.84%**	96.98%	96.86%	**97.09%**	97.07%	97.07%	96.92%	96.75%
	original	76662	42708	79794	39199	82965	86623	36849	36849	82812	75835
Return	removed	5290	3103	5655	2194	5276	5087	1925	1925	5747	4850
	reduction	6.90%	**7.26%**	7.08%	5.59%	6.35%	5.87%	**5.22%**	**5.22%**	6.93%	6.39%
	original	11908	6159	13849	5381	12653	14108	5060	5060	13715	12545
Cast	removed	17	10	23	9	17	17	12	12	23	27
	reduction	0.14%	0.16%	0.16%	0.16%	0.13%	**0.12%**	**0.23%**	**0.23%**	0.16%	0.21%
	original	4249	2861	4882	3009	5372	5104	2383	2383	4643	4575
LoadArray	removed	1775	1379	2067	1309	2221	2144	978	978	1979	1887
	reduction	41.77%	**48.19%**	42.33%	43.50%	41.34%	42.00%	**41.04%**	**41.04%**	42.62%	41.24%
	original	56357	26562	61493	25780	61839	62033	26202	26202	62342	57407
LoadField	removed	26190	11297	27184	10767	28558	28027	11058	11058	29630	26088
	reduction	46.47%	42.53%	44.20%	**41.76%**	46.18%	45.18%	42.20%	42.20%	**47.52%**	45.44%
	original	17509	8758	21450	8159	19064	20810	8399	8399	20002	18243
LoadSField	removed	6239	3694	7411	3058	7037	6988	3388	3388	7765	6581
	reduction	35.63%	**42.17%**	34.55%	37.48%	36.91%	**33.58%**	40.33%	40.33%	38.82%	36.07%
	original	11648	3714	13778	3528	13470	12818	3568	3568	13180	13111
StoreArray	removed	217	90	226	104	288	261	95	95	258	230
	reduction	1.86%	2.42%	1.64%	**2.94%**	2.13%	2.03%	2.66%	2.66%	1.95%	**1.75%**
	original	13581	7126	15737	7048	14679	15191	7260	7260	15587	14364
StoreField	removed	90	44	98	41	106	109	49	49	155	96
	reduction	0.66%	0.61%	0.62%	**0.58%**	0.72%	0.71%	0.67%	0.67%	**0.99%**	0.66%
	original	4674	1825	5574	1943	4927	5727	2025	2025	5091	4820
StoreSField	removed	3	1	3	1	4	3	1	1	3	3
	reduction	0.06%	0.05%	0.05%	0.05%	**0.08%**	0.05%	**0.04%**	**0.04%**	0.05%	0.06%
	original	136362	71755	147917	64302	144532	144718	61360	61360	144805	134451
VirtMethCall	removed	14447	7670	15394	6005	12406	12380	5725	5725	13149	11858
	reduction	10.59%	**10.68%**	10.40%	9.33%	8.58%	**8.55%**	9.33%	9.33%	9.08%	8.81%
	original	26680	14421	30266	15188	30237	32616	14738	14738	29736	28070
StatMethCall	removed	3645	1818	4341	1705	4070	4294	1712	1712	4638	3756
	reduction	13.66%	12.60%	14.34%	**11.22%**	13.46%	13.16%	11.61%	11.61%	**15.59%**	13.38%
	original	52727	29750	59778	29039	56784	60228	29411	29411	60426	54618
SpecMethCall	removed	1262	795	1478	807	1424	1764	1164	1164	2698	1312
	reduction	**2.39%**	2.67%	2.47%	2.77%	2.50%	2.92%	3.95%	3.95%	**4.46%**	2.40%
	original	599547	313278	664975	298922	649383	677861	292799	292799	661095	615111
Total	removed	153475	78913	172984	74099	163182	174339	74037	74037	174012	158896
	reduction	25.59%	25.18%	26.01%	24.78%	**25.12%**	25.71%	25.28%	25.28%	**26.32%**	25.83%

Table 5. Number of instructions (per instruction type) for the Jimple intermediate language representation. Maximum and minimum reduction percentages per row are shown in bold.

analyses that have knowledge of program internals, such as temporary variables or call-sites, which may have been optimized away. As mentioned in Section 4.3, this information can be retrieved via post-processing. Such post-processing is typically specific to the client analysis: the analysis will post-process the information it cares about to add back missing elements. To enable post-processing, our implementation offers maps from eliminated variables and call-sites to equivalent ones. For instance, it is easy to post-process the tables that depict our final points-to information for every program variable, by augmenting the existing logic with an extra case:

```
InsensVarPointsTo(var, heap) <-
  VarPointsTo(var2, _, heap, _), DupCopies(var, var2).
```

The `DupCopies` table, above, stores the fact that `var` is replaced by the equivalent variable `var2`. The rule causes the final output of the analysis, table `InsensVarPointsTo`, to also integrate facts for eliminated variables, thus replicating exactly the analysis results *without* set-based pre-analysis.

Such post-processing incurs virtually zero cost. For example, for a 2obj+H analysis, the above post-processing adds roughly 1sec to the query reporting the `InsensVarPointsTo` results.[9]

[9] Although in theory the output of a points-to analysis is the points-to information for local variables, most of the time points-to analysis is *not* performed with the purpose of producing results for all local variables. Instead, points-to analysis may be required in order to compute reachable

		antlr	bloat	chart	eclipse	hsqldb	jython	luindex	lusearch	pmd	xalan
	original	117905	59909	135342	59797	128252	141059	59108	59108	134639	128690
Move	removed	104245	53586	120005	53170	112849	124026	52772	52772	118791	113154
	reduction	88.41%	**89.44%**	88.66%	88.91%	87.99%	**87.92%**	89.28%	89.28%	88.22%	**87.92%**
	original	76662	42708	79794	39199	82965	86623	36849	36849	82812	75835
Return	removed	5415	3103	5768	2209	5414	5216	1939	1939	5875	4977
	reduction	7.06%	**7.26%**	7.22%	5.63%	6.52%	6.02%	**5.26%**	**5.26%**	7.09%	6.56%
	original	4261	2867	4894	3017	5384	5125	2389	2389	4655	4587
LoadArray	removed	1813	1410	2108	1329	2283	2193	997	997	2018	1932
	reduction	42.54%	**49.18%**	43.07%	44.05%	42.40%	42.79%	**41.73%**	**41.73%**	43.35%	42.11%
	original	56377	26575	61513	25794	61870	62057	26215	26215	62412	57428
LoadField	removed	26415	11403	27403	10879	28809	28275	11178	11178	29938	26312
	reduction	46.85%	42.90%	44.54%	**42.17%**	46.56%	45.56%	42.63%	42.63%	**47.96%**	45.81%
	original	17526	8760	21480	8161	19081	20832	8401	8401	20022	18262
LoadSField	removed	6411	3770	7811	3144	7390	7991	3504	3504	8286	6909
	reduction	36.57%	**43.03%**	**36.36%**	38.52%	38.72%	38.35%	41.70%	41.70%	41.38%	37.83%
	original	11650	3715	13781	3528	13472	12824	3568	3568	13183	13113
StoreArray	removed	212	89	221	103	283	257	94	94	252	224
	reduction	1.81%	2.39%	**1.60%**	**2.91%**	2.10%	2.00%	2.63%	2.63%	1.91%	1.70%
	original	13588	7130	15744	7052	14688	15198	7266	7266	15599	14371
StoreField	removed	94	47	102	43	110	113	54	54	159	100
	reduction	0.69%	0.65%	0.64%	**0.60%**	0.74%	0.74%	0.74%	0.74%	**1.01%**	0.69%
	original	4674	1825	5574	1945	4927	5727	2025	2025	5095	4822
StoreSField	removed	3	1	3	2	4	3	1	1	3	3
	reduction	0.06%	0.05%	0.05%	**0.10%**	0.08%	0.05%	**0.04%**	**0.04%**	0.05%	0.06%
	original	136362	71755	147917	64302	144532	144718	61360	61360	144805	134451
VirtMethCall	removed	14400	7584	15289	5937	12302	12313	5656	5656	13094	11786
	reduction	**10.56%**	**10.56%**	10.33%	9.23%	**8.51%**	8.50%	9.21%	9.21%	9.04%	8.76%
	original	52727	29750	59778	29039	56784	60228	29411	29411	60426	54618
SpecMethCall	removed	1269	795	1483	807	1437	1772	1169	1169	2694	1317
	reduction	**2.40%**	2.67%	2.48%	2.77%	2.53%	2.94%	3.97%	3.97%	**4.45%**	2.41%
	original	26680	14421	30266	15188	30237	32616	14738	14738	29736	28070
StatMethCall	removed	3657	1806	4359	1693	4098	4311	1701	1701	4649	3773
	reduction	13.70%	12.52%	14.40%	**11.14%**	13.55%	13.21%	11.54%	11.54%	**15.63%**	13.44%
	original	620181	322721	687747	309164	672651	702366	302566	302566	684479	638250
Total	removed	163934	83594	184552	79316	174979	186470	79065	79065	185759	170487
	reduction	26.43%	25.90%	26.83%	**25.65%**	26.01%	26.54%	26.13%	26.13%	**27.13%**	26.71%

Table 6. Number of instructions (per instruction type) for an SSA version of intermediate language representation. Maximum and minimum reduction percentages per row are shown in bold.

The reason that post-processing is virtually cost-free is dual. First, post-processing only adjusts information in the final output table, and not in all other tables involved in intermediate computations. Second, post-processing can avoid changing the context-sensitive facts computed by an analysis and instead affect the final context-insensitive facts, as in the above example query.

Summary. In all, we see that set-based reasoning has a significant impact on points-to analysis, and that this applies transparently to a very wide variety of analyses, without any need to change the analysis at all. For practical usability it is also important to recall that this impact is modular: a library (or other invariant code) can be optimized once and the results reused in conjunction with any other client program, and for different points-to analyses.

methods, points-to information for heap objects, object type connectivity graphs, etc. Thus, post-processing is relatively rarely required in practice.

7. Conclusions

In 2000, Rountev and Chandra wrote, regarding their off-line optimization technique [15]:

> *While we have concentrated on reducing the cost of Andersen's analysis, we conjecture that such precomputation can be helpful in other points-to analyses as well.*

Although subsequent work advanced the area of off-line optimization of points-to analysis, it has not achieved this conjectured generality and independence from the analysis algorithm. In the work we presented here, we fulfill this promise by expressing the optimization as a pre-processing step that is largely orthogonal to the subsequent points-to analysis. We applied our approach to 7 different points-to analysis algorithms for demonstration purposes, and it transparently applies to any other points-to analysis in the Doop framework. There is virtually no other comparable optimization mechanism of such wide applicability to different points-to analyses in the literature—algorithmic improvements in this area are usually analysis-specific.

Furthermore, the intraprocedural nature of our approach means that it can be applied once-and-for-all to libraries and have the results be reused, and that it works well with points-to analysis algorithms employing on-the-fly call-graph construction (in languages with dynamic dispatch). Finally, our approach is also more general than past techniques for off-line optimization, because it allows removing individual redundant program statements instead of just collapsing variables. We believe that the generality and orthogonality of our set-based pre-analysis will render it a valuable weapon in the arsenal of the static analysis programmer for years to come.

References

[1] K. Ali and O. Lhoták. Application-only call graph construction. In J. Noble, editor, *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, volume 7313 of *Lecture Notes in Computer Science*, pages 688–712. Springer Berlin Heidelberg, 2012.

[2] M. Berndl, O. Lhoták, F. Qian, L. J. Hendren, and N. Umanee. Points-to analysis using bdds. In *PLDI*, pages 103–114. ACM, 2003.

[3] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA '09: 24th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2009. ACM.

[4] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 85–96, New York, NY, USA, 1998. ACM.

[5] S. Guarnieri and B. Livshits. GateKeeper: mostly static enforcement of security and reliability policies for Javascript code. In *Proceedings of the 18th USENIX Security Symposium*, SSYM'09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.

[6] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI'07: Proc. ACM SIGPLAN conf. on Programming Language Design and Implementation*, pages 290–299, New York, NY, USA, 2007. ACM.

[7] B. Hardekopf and C. Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *In International Static Analysis Symposium (SAS)*, pages 265–280, 2007.

[8] G. Kastrinis and Y. Smaragdakis. Efficient and effective handling of exceptions in java points-to analysis. In *Compiler Construction*, Mar. 2013.

[9] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Conf. on Programming Language Design and Implementation (PLDI)*. ACM, June 2013.

[10] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *PODS '05: Proc. of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2005. ACM.

[11] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 1–11, New York, NY, USA, 2002. ACM.

[12] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[13] R. Nasre. Exploiting the structure of the constraint graph for efficient points-to analysis. In *Proceedings of the 2012 international symposium on Memory Management*, ISMM '12, pages 121–132, New York, NY, USA, 2012. ACM.

[14] T. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196. Kluwer Academic Publishers, 1994.

[15] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 47–56, New York, NY, USA, 2000. ACM.

[16] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*, pages 189–233, Englewood Cliffs, NJ, 1981. Prentice-Hall, Inc.

[17] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[18] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity (the making of a precise and scalable pointer analysis). In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 17–30. ACM Press, Jan. 2011.

[19] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

[20] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[21] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog with binary decision diagrams for program analysis. In K. Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.

[22] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proc. of the ACM SIGPLAN 2004 conf. on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM.

		antlr	bloat	chart	eclipse	hsqldb	jython	luindex	lusearch	pmd	xalan	**AVG**
insens	original	10,988	9,819	15,340	6,275	5,398	4,706	4,792	5,073	5,544	6,553	
	set-based	10,038	8,412	13,503	5,628	4,739	4,139	4,382	4,614	5,024	5,832	
	reduction	8.65%	**14.33%**	11.98%	10.31%	12.21%	12.05%	**8.56%**	9.05%	9.38%	11%	10.75%
1obj	original	14,301	21,927	62,502	9,353	13,955	8,671	5,435	6,218	7,987	15,449	
	set-based	10,003	14,462	48,332	6,718	9,249	6,177	3,978	4,560	5,807	11,573	
	reduction	30.05%	**34.04%**	**22.67%**	28.17%	33.72%	28.76%	26.81%	26.66%	27.29%	25.09%	28.33%
1obj+H	original	82,899	81,797	-	58,271	193,882	101,621	25,707	26,885	30,558	97,004	
	set-based	56,858	57,255	-	41,433	126,778	69,461	18,359	19,194	21,754	72,472	
	reduction	31.41%	30%	-	28.90%	**34.61%**	31.65%	28.58%	28.61%	28.81%	**25.29%**	29.76%
2obj+H	original	19,917	153,469	67,608	44,638	-	-	11,143	13,182	13,202	166,641	
	set-based	13,642	99,222	48,102	30,823	-	-	7,487	9,176	9,001	120,744	
	reduction	31.51%	**35.35%**	28.85%	30.95%	-	-	32.81%	30.39%	31.82%	**27.54%**	31.15%
2type+H	original	5,354	11,446	13,319	13,552	13,660	52,015	4,108	4,204	4,550	10,205	
	set-based	3,846	7,510	9,328	9,290	8,586	34,514	2,783	2,858	3,075	6,912	
	reduction	**28.17%**	34.39%	29.96%	31.45%	**37.14%**	33.65%	32.25%	32.02%	32.42%	32.27%	32.37%
1call	original	16,093	32,946	49,649	12,264	9,601	10,430	7,839	8,763	11,369	14,499	
	set-based	10,337	18,111	34,245	8,251	6,238	6,988	5,380	6,002	7,812	10,129	
	reduction	35.77%	**45.03%**	31.03%	32.72%	35.03%	33%	31.37%	31.51%	31.29%	**30.14%**	33.69%
1call+H	original	54,844	150,516	120,865	61,524	39,783	50,633	26,107	28,525	35,945	59,872	
	set-based	29,697	73,972	78,952	38,998	24,348	31,251	16,937	18,200	23,372	38,553	
	reduction	45.85%	**50.84%**	**34.68%**	36.61%	38.80%	38.28%	35.12%	36.20%	34.98%	35.61%	38.70%

Table 7. Number of context-sensitive VarPointsTo entries (measured in thousands) for the Jimple representation. Maximum and minimum reduction percentages per row are shown in bold.

		antlr	bloat	chart	eclipse	hsqldb	jython	luindex	lusearch	pmd	xalan	**AVG**
insens	original	11,032	9,750	15,209	6,226	5,402	4,674	4,756	5,033	5,493	6,523	
	set-based	10,028	8,323	13,328	5,560	4,707	4,088	4,335	4,564	4,962	5,771	
	reduction	9.1%	**14.64%**	12.37%	10.70%	12.87%	12.54%	**8.85%**	9.32%	9.67%	11.53%	11.16%
1obj	original	15,319	22,299	63,017	9,649	15,328	8,875	5,630	6,367	8,202	16,155	
	set-based	10,583	14,650	48,453	6,848	9,880	6,266	4,082	4,624	5,908	11,951	
	reduction	30.92%	34.30%	**23.11%**	29.03%	**35.54%**	29.40%	27.50%	27.38%	27.97%	26.02%	29.12%
1obj+H	original	89,220	84,823	-	61,463	219,442	103,896	27,138	28,032	32,259	106,986	
	set-based	60,233	58,887	-	42,807	138,527	70,326	19,040	19,647	22,527	78,634	
	reduction	32.49%	30.58%	-	30.35%	**36.87%**	32.31%	29.84%	29.91%	30.17%	**26.50%**	31%
2obj+H	original	21,435	149,871	73,953	48,425	-	-	11,702	12,480	13,907	182,049	
	set-based	14,521	98,137	51,984	32,919	-	-	7,829	8,380	9,472	130,912	
	reduction	32.26%	**34.52%**	29.71%	32.02%	-	-	33.10%	32.85%	31.89%	**28.09%**	31.80%
2type+H	original	5,607	11,170	13,942	14,556	14,737	54,314	4,318	4,410	4,773	10,826	
	set-based	3,998	7,332	9,676	9,858	9,139	35,787	2,913	2,978	3,213	7,272	
	reduction	**28.70%**	34.36%	30.60%	32.28%	**37.99%**	34.11%	32.54%	32.47%	32.68%	32.83%	32.86%
1call	original	16,256	33,020	48,119	12,373	9,743	10,427	7,840	8,724	11,364	14,642	
	set-based	10,294	18,007	33,174	8,223	6,187	6,913	5,317	5,908	7,730	10,097	
	reduction	36.68%	**45.47%**	31.06%	33.54%	36.50%	33.70%	32.18%	32.28%	31.98%	**31.04%**	34.44%
1call+H	original	55,508	150,685	118,214	62,299	40,667	50,873	26,151	28,526	36,075	61,156	
	set-based	29,599	75,764	77,077	38,886	24,213	30,952	16,742	17,975	23,185	38,846	
	reduction	46.68%	**49.72%**	**34.80%**	37.58%	40.46%	39.16%	35.98%	36.99%	35.73%	36.48%	39.36%

Table 8. Number of context-sensitive VarPointsTo entries (measured in thousands) for the SSA representation. Maximum and minimum reduction percentages per row are shown in bold.