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 ~20% average speedup (max: 110%, median: 18%).

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:

\begin{align*}
p &= q; \\
r &= p; \\
r &= q; &// redundant
\end{align*}

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.)

\begin{align*}
r &= q; &p.f &= q; \\
p.f &= r; &r &= p.f; \\
// redundant &r &= q; &// redundant
\end{align*}

**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 Door framework for Java points-to analysis by Bravenboer et al. [4]. 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 [21, 22] 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 about 20% over all analyses, with significantly higher numbers (up to 110%) for specific analyses and input programs.

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. [5] 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. [20].) The technique of Nasre [14] 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 [16] and Hardekopf and Lin’s [8]. Hardekopf and Lin have also applied similar ideas in a hybrid online/offline setting [7], 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 [5, 8, 14, 16] 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 [4, 6, 11, 15, 23, 24] 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,
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 \( \text{from} \) is a subset of the points-to set for variable \( \text{to} \) if the program contains a Move instruction between \( \text{from} \) and \( \text{to} \). 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 [6] has very analogous rules:

\[
\text{PTSTO}(v, h) \leftarrow \text{Alloc}(v, h).
\]

\[
\text{PTSTO}(v1, h) \leftarrow \text{PTSTO}(v2, h), \text{Move}(v1, v2).
\]

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. [10].)

\[
\text{VarPointsTo}(\text{var}, \text{ctx}, \text{heap}, \text{htctx}) \leftarrow \text{Alloc}(\text{var}, \text{heap}).
\]

\[
\text{VarPointsTo}(\text{to}, \text{ctx}, \text{heap}, \text{htctx}) \leftarrow \text{Move}(\text{to}, \text{from}),
\quad \text{VarPointsTo}(\text{from}, \text{ctx}, \text{heap}, \text{htctx}).
\]

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 \( S \subseteq T \land T \subseteq U \implies S \subseteq U \). 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:

\[
p = q;
r = p;
r = q; \quad // \text{redundant}
\]

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:

\[
\text{VarPointsTo}(\text{to}, \text{heap}) \leftarrow \text{Move}(\text{to}, \text{from}),
\quad \text{VarPointsTo}(\text{from}, \text{heap}).
\]

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 inferable 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: \textit{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 Door 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.

\[
\begin{align*}
r &= q; \\
p.f &= r; \\
p.f &= q; & \text{// redundant}
\end{align*}
\]

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

\[
\begin{align*}
r &= q; \\
C.f &= r; & \text{\(C\) is a class} \\
C.f &= q; & \text{// redundant}
\end{align*}
\]

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

\[
\begin{align*}
r &= q; \\
q &= p.f; \\
r &= p.f; & \text{// redundant}
\end{align*}
\]

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:

\[
\begin{align*}
r &= q; \\
q &= C.f; & \text{\(C\) is a class} \\
r &= C.f; & \text{// redundant}
\end{align*}
\]

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

\[
\begin{align*}
p &= q; \\
r &= p; \\
r &= q; & \text{// redundant}
\end{align*}
\]

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

\[
\begin{align*}
p.f &= q; \\
r &= p.f; \\
r &= q; & \text{// redundant}
\end{align*}
\]

And similarly for static fields:

\[
\begin{align*}
C.f &= q; & \text{\(C\) is a class} \\
r &= C.f; \\
r &= q; & \text{// redundant}
\end{align*}
\]

**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 \(\text{arr}[1]\) to be the same
abstract location \(\text{arr}[\ast]\)—an approach often called array
insensitive. Thus, for instance, we can eliminate array loads:

\[
\begin{align*}
r &= q; \\
q &= \text{arr}\[\ast]\; \\
r &= \text{arr}\[\ast]; & \text{// redundant}
\end{align*}
\]

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

\[
\begin{align*}
\text{arr}[\ast] &= q; \\
r &= \text{arr}\[\ast]; \\
r &= q; & \text{// redundant}
\end{align*}
\]

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

\[
\begin{align*}
r &= q; \\
q &= p.m(); \\
r &= p.m(); & \text{// redundant}
\end{align*}
\]

The above is not limited to no-argument methods, but the
arguments need to be identical local variables for the trans-
formation to apply.\(^3\)

As usual, analogous transformations apply to static meth-
ods:

\[
\begin{align*}
r &= q; \\
q &= C.m(); & \text{\(C\) is a class} \\
r &= C.m(); & \text{// redundant}
\end{align*}
\]

It is somewhat surprising that method calls can be elimi-
nated, 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
[10, 19], 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 iden-
tical then the two calls receive the same information from the

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

**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 Door 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:

```plaintext
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 \([5, 8, 14, 16]\). 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:

- The variables belong in the same strongly-connected component of the constraint graph.
- The variables have identical in-flows. E.g.,
  ```plaintext
  q = p.f;
r = p.f;
q,r receive no other assignments
  ```

The above are the most profitable special cases of the general approach of Hardekopf and Lin \([8]\) 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 \([14]\), which is also handled by the reasoning in Hardekopf and Lin’s approach \([8]\).

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 \([14]\):

```plaintext
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;
```

```
\[ \begin{align*}
&\quad\quad p \\
&\quad\quad q \\
&{\downarrow{}} &{\downarrow{}} \\
&\quad\quad r
\end{align*} \]
```

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 Hardekopf and Lin’s approach [8]. 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:

\[
\begin{align*}
  r &= q; \\
  q &= p.m(a); \\
  r &= s.m(b);
\end{align*}
\]

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 [21, 22]. Running our analysis determines that several of the local variables are redundant and the method body can be significantly simplified. (\( r_0 \) is cloning @this; \( r_1 \) is cloning @param0; \( r_4 \) and \( r_5 \) are cloning \( r_3 \); \( r_7, r_8, r_10, \) and \( r_11 \) are cloning \( r_6 \).) The result of the transformation can be seen in Figure 2.  

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.

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 \( r_1 \), 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 [9]). 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 \( r_1 \), a client analysis will find the variable that replaced \( r_1 \) 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 Door framework for Java points-to analysis by Bravenboer et al. [4]. Door 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 Door 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 opti-
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;
if $r4 == null goto label0;

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

$r0.<java.util.TreeMap: java.util.TreeMap$Entry root> = $r2;
goto label3;
label1: $r9 = $r6.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right>;
if $r9 != $r1 goto label2;

$r10 = $r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = $r1;
$r10.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right> = $r2;
goto label3;
label2: $r11 = $r1.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = $r1;
$r11.<java.util.TreeMap$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>;
$r3 = @param0.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right>;
@param0.<java.util.TreeMap$Entry: java.util.TreeMap$Entry left> = $r3;
if $r6 == null goto label0;

$r6 = @param0.<java.util.TreeMap$Entry: java.util.TreeMap$Entry right> = @param0;
$r6.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = @param0;
r2.<java.util.TreeMap$Entry: java.util.TreeMap$Entry parent> = $r6;
if $r9 != 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> = @param0;
goto label3;
label2: $r6.<java.util.TreeMap$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.
mization over the entire program (as if) in parallel. 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.

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):

\[ \text{RedundantLoad}(\_r, \_p, \_f) \leftarrow \]
\[ \text{Load}(\_r, \_p, \_f), \]
\[ \text{Load}(\_q, \_p, \_f), \]
\[ \text{Move}(\_r, \_q), \]
\[ \_r \neq \_q, \]
\[ \text{!TransitiveFlow}(\_r, \_q). \]

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. 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)

\(^6\) Currently there is no real parallel computation, although this is entirely up to the Datalog evaluation engine.

\(^7\) Our implementation can be found in http://doop.program-analysis.org/ and mainly in files logic/transform.logic and logic/transform-delta.logic.
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 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 Door 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 [17, 18] (1call, 1call+H), object-sensitive [12, 13] (1obj, 1obj+H, 2obj+H), and type-sensitive [19] (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 [10].)

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 [21, 22]. 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, 19]: jython and hsqldb are analyzed with reflection disabled and hsqldb has its entry point set manually in a special harness.

Our set-based pre-processing runs in an average of 49sec (max: 96, min: 31) over all benchmarks and the standard library in the Jimple representation, with an average of 47sec (max: 93, min: 30) for the SSA representation. As discussed in the previous section, this time is incurred only once per program or library and is influenced heavily by low-level engineering considerations that we did not try to optimize.

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 110%, 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.** Tables 5 and 6 (for the Jimple and SSA IR, respectively) show different instruction types and the impact of set-based pre-processing on them. (The instruction types are renamed to be mostly self-explanatory.)

As expected, local assignments (Move instructions) are drastically reduced: close to 90% of them for the SSA IR and about 97% for the Jimple IR are eliminated! Nevertheless, this does not impact performance significantly in our setting: Move instructions are handled very efficiently in the Door implementation. Other kinds of instructions that are reduced...
Table 3. Number of reachable (local) variables for the Jimple intermediate language representation. Maximum and minimum reduction percentages per row are shown in bold.

<table>
<thead>
<tr>
<th>Libraries</th>
<th>antlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>jython</th>
<th>luindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>91.642</td>
<td>95.196</td>
<td>144.580</td>
<td>86.577</td>
<td>92.942</td>
<td>79.450</td>
<td>69.513</td>
<td>74.513</td>
<td>80.615</td>
<td>92.463</td>
<td>92.463</td>
</tr>
<tr>
<td>set-based</td>
<td>65.407</td>
<td>66.558</td>
<td>101.850</td>
<td>61.196</td>
<td>63.698</td>
<td>54.995</td>
<td>48.973</td>
<td>52.060</td>
<td>56.114</td>
<td>64.203</td>
<td>64.203</td>
</tr>
<tr>
<td>reduction</td>
<td><strong>28.63%</strong></td>
<td><strong>29.08%</strong></td>
<td><strong>29.55%</strong></td>
<td><strong>31.46%</strong></td>
<td><strong>29.46%</strong></td>
<td><strong>28.78%</strong></td>
<td><strong>28.78%</strong></td>
<td><strong>27.88%</strong></td>
<td><strong>27.44%</strong></td>
<td><strong>30.56%</strong></td>
<td><strong>30.56%</strong></td>
</tr>
<tr>
<td>original</td>
<td>90.029</td>
<td>93.523</td>
<td>141.628</td>
<td>84.780</td>
<td>92.035</td>
<td>78.710</td>
<td>67.764</td>
<td>72.771</td>
<td>78.947</td>
<td>90.775</td>
<td>90.775</td>
</tr>
<tr>
<td>set-based</td>
<td>64.315</td>
<td>65.438</td>
<td>99.889</td>
<td>59.982</td>
<td>63.072</td>
<td>54.484</td>
<td>47.771</td>
<td>50.859</td>
<td>54.991</td>
<td>63.064</td>
<td>63.064</td>
</tr>
<tr>
<td>reduction</td>
<td><strong>28.56%</strong></td>
<td><strong>30.03%</strong></td>
<td><strong>29.47%</strong></td>
<td><strong>31.17%</strong></td>
<td><strong>30.78%</strong></td>
<td><strong>30.80%</strong></td>
<td><strong>30.80%</strong></td>
<td><strong>30.11%</strong></td>
<td><strong>30.34%</strong></td>
<td><strong>30.53%</strong></td>
<td><strong>30.53%</strong></td>
</tr>
<tr>
<td>original</td>
<td>89.774</td>
<td>93.233</td>
<td>-</td>
<td>84.145</td>
<td>91.210</td>
<td>78.048</td>
<td>72.361</td>
<td>72.361</td>
<td>78.303</td>
<td>90.408</td>
<td>90.408</td>
</tr>
<tr>
<td>set-based</td>
<td>64.122</td>
<td>65.226</td>
<td>-</td>
<td>59.545</td>
<td>62.454</td>
<td>53.983</td>
<td>47.578</td>
<td>50.553</td>
<td>54.633</td>
<td>62.800</td>
<td>62.800</td>
</tr>
<tr>
<td>reduction</td>
<td><strong>28.57%</strong></td>
<td><strong>30.04%</strong></td>
<td><strong>29.24%</strong></td>
<td><strong>31.53%</strong></td>
<td><strong>30.83%</strong></td>
<td><strong>30.92%</strong></td>
<td><strong>30.92%</strong></td>
<td><strong>30.14%</strong></td>
<td><strong>30.41%</strong></td>
<td><strong>30.54%</strong></td>
<td><strong>30.54%</strong></td>
</tr>
<tr>
<td>original</td>
<td>88.201</td>
<td>91.588</td>
<td>113.387</td>
<td>82.038</td>
<td>-</td>
<td>-</td>
<td>65.911</td>
<td>70.718</td>
<td>76.680</td>
<td>88.196</td>
<td>88.196</td>
</tr>
<tr>
<td>set-based</td>
<td>62.978</td>
<td>64.048</td>
<td>80.997</td>
<td>58.020</td>
<td>-</td>
<td>-</td>
<td>46.413</td>
<td>49.353</td>
<td>53.291</td>
<td>61.135</td>
<td>61.135</td>
</tr>
<tr>
<td>reduction</td>
<td><strong>28.58%</strong></td>
<td><strong>30.07%</strong></td>
<td><strong>28.57%</strong></td>
<td><strong>31.53%</strong></td>
<td><strong>30.83%</strong></td>
<td><strong>30.92%</strong></td>
<td><strong>30.92%</strong></td>
<td><strong>30.14%</strong></td>
<td><strong>30.41%</strong></td>
<td><strong>30.54%</strong></td>
<td><strong>30.54%</strong></td>
</tr>
<tr>
<td>original</td>
<td>88.401</td>
<td>91.928</td>
<td>119.598</td>
<td>82.392</td>
<td>89.782</td>
<td>75.631</td>
<td>66.111</td>
<td>70.918</td>
<td>76.927</td>
<td>88.496</td>
<td>88.496</td>
</tr>
<tr>
<td>set-based</td>
<td>63.132</td>
<td>64.281</td>
<td>85.079</td>
<td>58.283</td>
<td>61.438</td>
<td>52.149</td>
<td>46.567</td>
<td>49.507</td>
<td>53.488</td>
<td>61.358</td>
<td>61.358</td>
</tr>
<tr>
<td>reduction</td>
<td><strong>28.58%</strong></td>
<td><strong>30.07%</strong></td>
<td><strong>28.86%</strong></td>
<td><strong>31.57%</strong></td>
<td><strong>31.05%</strong></td>
<td><strong>29.56%</strong></td>
<td><strong>29.56%</strong></td>
<td><strong>30.19%</strong></td>
<td><strong>30.47%</strong></td>
<td><strong>30.67%</strong></td>
<td><strong>30.67%</strong></td>
</tr>
<tr>
<td>original</td>
<td>90.264</td>
<td>93.748</td>
<td>141.415</td>
<td>84.851</td>
<td>92.283</td>
<td>78.921</td>
<td>68.135</td>
<td>73.142</td>
<td>79.202</td>
<td>91.012</td>
<td>91.012</td>
</tr>
<tr>
<td>set-based</td>
<td>64.470</td>
<td>65.587</td>
<td>99.791</td>
<td>60.022</td>
<td>63.240</td>
<td>54.621</td>
<td>48.036</td>
<td>51.124</td>
<td>55.158</td>
<td>63.221</td>
<td>63.221</td>
</tr>
<tr>
<td>reduction</td>
<td><strong>28.58%</strong></td>
<td><strong>30.04%</strong></td>
<td><strong>29.43%</strong></td>
<td><strong>31.47%</strong></td>
<td><strong>30.79%</strong></td>
<td><strong>29.50%</strong></td>
<td><strong>29.50%</strong></td>
<td><strong>30.10%</strong></td>
<td><strong>30.36%</strong></td>
<td><strong>30.53%</strong></td>
<td><strong>30.53%</strong></td>
</tr>
</tbody>
</table>

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.

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 analysis and has the advantage of being impervious to plat-
form 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.

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 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

---

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].
<table>
<thead>
<tr>
<th>Move</th>
<th>amlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>python</th>
<th>lunindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>117905</td>
<td>59909</td>
<td>135342</td>
<td>59797</td>
<td>128252</td>
<td>141059</td>
<td>59108</td>
<td>39108</td>
<td>134639</td>
<td>128690</td>
</tr>
<tr>
<td>removed reduction</td>
<td>104245</td>
<td>188665</td>
<td>86865</td>
<td>89170</td>
<td>87999</td>
<td>87929</td>
<td>89289</td>
<td>8822</td>
<td>115154</td>
<td>87929</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return</th>
<th>amlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>python</th>
<th>lunindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>76662</td>
<td>47208</td>
<td>79979</td>
<td>39199</td>
<td>82965</td>
<td>86623</td>
<td>36849</td>
<td>36649</td>
<td>82812</td>
<td>75835</td>
</tr>
<tr>
<td>removed reduction</td>
<td>5411</td>
<td>13013</td>
<td>5768</td>
<td>2209</td>
<td>5414</td>
<td>5216</td>
<td>1939</td>
<td>1939</td>
<td>5875</td>
<td>4977</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LoadArray</th>
<th>amlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>python</th>
<th>lunindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>4261</td>
<td>2887</td>
<td>4894</td>
<td>3017</td>
<td>5384</td>
<td>3125</td>
<td>2389</td>
<td>2389</td>
<td>4655</td>
<td>4857</td>
</tr>
<tr>
<td>removed reduction</td>
<td>1813</td>
<td>1410</td>
<td>2108</td>
<td>1329</td>
<td>2283</td>
<td>2193</td>
<td>997</td>
<td>997</td>
<td>2018</td>
<td>1932</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LoadField</th>
<th>amlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>python</th>
<th>lunindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>56377</td>
<td>28575</td>
<td>61513</td>
<td>25794</td>
<td>61870</td>
<td>62057</td>
<td>26215</td>
<td>26215</td>
<td>62412</td>
<td>57428</td>
</tr>
<tr>
<td>removed reduction</td>
<td>26415</td>
<td>11403</td>
<td>27403</td>
<td>10879</td>
<td>28809</td>
<td>28257</td>
<td>11178</td>
<td>11178</td>
<td>29938</td>
<td>26312</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LoadSField</th>
<th>amlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>python</th>
<th>lunindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>17526</td>
<td>8760</td>
<td>21480</td>
<td>8161</td>
<td>19081</td>
<td>20832</td>
<td>8401</td>
<td>8401</td>
<td>20022</td>
<td>18262</td>
</tr>
<tr>
<td>removed reduction</td>
<td>6411</td>
<td>3770</td>
<td>7811</td>
<td>3144</td>
<td>7390</td>
<td>7991</td>
<td>3504</td>
<td>3504</td>
<td>8286</td>
<td>6909</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StoreArray</th>
<th>amlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>python</th>
<th>lunindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>11630</td>
<td>3715</td>
<td>13781</td>
<td>3528</td>
<td>13472</td>
<td>12824</td>
<td>3568</td>
<td>3568</td>
<td>13183</td>
<td>13113</td>
</tr>
<tr>
<td>removed reduction</td>
<td>212</td>
<td>89</td>
<td>221</td>
<td>103</td>
<td>283</td>
<td>257</td>
<td>94</td>
<td>94</td>
<td>252</td>
<td>224</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StoreField</th>
<th>amlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>python</th>
<th>lunindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>13588</td>
<td>7130</td>
<td>15744</td>
<td>7052</td>
<td>14688</td>
<td>15198</td>
<td>7266</td>
<td>7266</td>
<td>15599</td>
<td>14731</td>
</tr>
<tr>
<td>removed reduction</td>
<td>94</td>
<td>47</td>
<td>102</td>
<td>43</td>
<td>110</td>
<td>113</td>
<td>54</td>
<td>54</td>
<td>159</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StoreSField</th>
<th>amlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>python</th>
<th>lunindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>4674</td>
<td>1825</td>
<td>5574</td>
<td>1945</td>
<td>4927</td>
<td>5727</td>
<td>2025</td>
<td>2025</td>
<td>5095</td>
<td>4822</td>
</tr>
<tr>
<td>removed reduction</td>
<td>0.06%</td>
<td>0.05%</td>
<td>0.05%</td>
<td>0.10%</td>
<td>0.08%</td>
<td>0.05%</td>
<td>0.05%</td>
<td>0.05%</td>
<td>0.06%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VirMethCall</th>
<th>amlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>python</th>
<th>lunindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>13632</td>
<td>71755</td>
<td>147917</td>
<td>64302</td>
<td>144532</td>
<td>144718</td>
<td>61360</td>
<td>61360</td>
<td>144805</td>
<td>134451</td>
</tr>
<tr>
<td>removed reduction</td>
<td>14400</td>
<td>7584</td>
<td>15289</td>
<td>5937</td>
<td>12302</td>
<td>12313</td>
<td>5656</td>
<td>5656</td>
<td>13094</td>
<td>11786</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SpecMethCall</th>
<th>amlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>python</th>
<th>lunindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>52727</td>
<td>29750</td>
<td>59781</td>
<td>29039</td>
<td>56784</td>
<td>60228</td>
<td>29811</td>
<td>29811</td>
<td>60926</td>
<td>54618</td>
</tr>
<tr>
<td>removed reduction</td>
<td>1269</td>
<td>27</td>
<td>1483</td>
<td>807</td>
<td>1437</td>
<td>1727</td>
<td>1169</td>
<td>1169</td>
<td>2694</td>
<td>1317</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StatMethCall</th>
<th>amlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>python</th>
<th>lunindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>26680</td>
<td>14421</td>
<td>30266</td>
<td>15188</td>
<td>30237</td>
<td>32616</td>
<td>14738</td>
<td>14738</td>
<td>29736</td>
<td>28070</td>
</tr>
<tr>
<td>removed reduction</td>
<td>3657</td>
<td>1806</td>
<td>4359</td>
<td>1693</td>
<td>4098</td>
<td>431</td>
<td>1701</td>
<td>1701</td>
<td>4649</td>
<td>3773</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total</th>
<th>amlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>python</th>
<th>lunindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>620181</td>
<td>322721</td>
<td>687747</td>
<td>309164</td>
<td>672651</td>
<td>702366</td>
<td>302566</td>
<td>302566</td>
<td>684479</td>
<td>638250</td>
</tr>
<tr>
<td>removed reduction</td>
<td>163934</td>
<td>83594</td>
<td>184552</td>
<td>79316</td>
<td>174979</td>
<td>186470</td>
<td>79065</td>
<td>79065</td>
<td>185759</td>
<td>170487</td>
</tr>
</tbody>
</table>

| 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

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 only 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

---

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

<table>
<thead>
<tr>
<th>library</th>
<th>antlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>python</th>
<th>luindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>10,988</td>
<td>9,819</td>
<td>15,340</td>
<td>6,275</td>
<td>5,398</td>
<td>4,706</td>
<td>4,792</td>
<td>5,073</td>
<td>5,544</td>
<td>6,553</td>
</tr>
<tr>
<td>set-based</td>
<td>10,038</td>
<td>8,412</td>
<td>13,503</td>
<td>5,628</td>
<td>4,739</td>
<td>4,139</td>
<td>4,382</td>
<td>4,614</td>
<td>5,024</td>
<td>5,832</td>
</tr>
<tr>
<td>reduction</td>
<td>6.85%</td>
<td>14.33%</td>
<td>11.98%</td>
<td>10.31%</td>
<td>12.21%</td>
<td>12.05%</td>
<td>8.56%</td>
<td>9.05%</td>
<td>9.38%</td>
<td>11%</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>library</th>
<th>antlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>python</th>
<th>luindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>10,988</td>
<td>9,819</td>
<td>15,340</td>
<td>6,275</td>
<td>5,398</td>
<td>4,706</td>
<td>4,792</td>
<td>5,073</td>
<td>5,544</td>
<td>6,553</td>
</tr>
<tr>
<td>set-based</td>
<td>10,038</td>
<td>8,412</td>
<td>13,503</td>
<td>5,628</td>
<td>4,739</td>
<td>4,139</td>
<td>4,382</td>
<td>4,614</td>
<td>5,024</td>
<td>5,832</td>
</tr>
<tr>
<td>reduction</td>
<td>6.85%</td>
<td>14.33%</td>
<td>11.98%</td>
<td>10.31%</td>
<td>12.21%</td>
<td>12.05%</td>
<td>8.56%</td>
<td>9.05%</td>
<td>9.38%</td>
<td>11%</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>library</th>
<th>antlr</th>
<th>bloat</th>
<th>chart</th>
<th>eclipse</th>
<th>hsqldb</th>
<th>python</th>
<th>luindex</th>
<th>lusearch</th>
<th>pmd</th>
<th>xalan</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>10,988</td>
<td>9,819</td>
<td>15,340</td>
<td>6,275</td>
<td>5,398</td>
<td>4,706</td>
<td>4,792</td>
<td>5,073</td>
<td>5,544</td>
<td>6,553</td>
</tr>
<tr>
<td>set-based</td>
<td>10,038</td>
<td>8,412</td>
<td>13,503</td>
<td>5,628</td>
<td>4,739</td>
<td>4,139</td>
<td>4,382</td>
<td>4,614</td>
<td>5,024</td>
<td>5,832</td>
</tr>
<tr>
<td>reduction</td>
<td>6.85%</td>
<td>14.33%</td>
<td>11.98%</td>
<td>10.31%</td>
<td>12.21%</td>
<td>12.05%</td>
<td>8.56%</td>
<td>9.05%</td>
<td>9.38%</td>
<td>11%</td>
</tr>
</tbody>
</table>

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.

transparency 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.

### 7. Conclusions

In 2000, Rountev and Chandra wrote, regarding their off-line optimization technique [16]:

*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 Door 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 offline 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.

Acknowledgments

We gratefully acknowledge funding by the European Union under a Marie Curie International Reintegration Grant and a European Research Council Starting/Consolidator grant; and by the Greek Secretariat for Research and Technology under an Excellence (Aristeia) award.

References


on Principles of Programming Languages (POPL), pages 81–95, New York, NY, USA, 2000. ACM.


