Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity

Yue Li
Aarhus University
yueli@cs.au.dk

Tian Tan
Aarhus University
tiantan@cs.au.dk

Anders Møller
Aarhus University
amoeller@cs.au.dk

Yannis Smaragdakis
University of Athens
smaragd@di.uoa.gr

ABSTRACT

Context-sensitivity is important in pointer analysis to ensure high precision, but existing techniques suffer from unpredictable scalability. Many variants of context-sensitivity exist, and it is difficult to choose one that leads to reasonable analysis time and obtains high precision, without running the analysis multiple times.

We present the Scaler framework that addresses this problem. Scaler efficiently estimates the amount of points-to information that would be needed to analyze each method with different variants of context-sensitivity. It then selects an appropriate variant for each method so that the total amount of points-to information is bounded, while utilizing the available space to maximize precision.

Our experimental results demonstrate that Scaler achieves predictable scalability for all the evaluated programs (e.g., speedups can reach 10x for 2-object-sensitivity), while providing a precision that matches or even exceeds that of the best alternative techniques.

CCS CONCEPTS

• **Theory of computation → Program analysis**;

KEYWORDS

static analysis, points-to analysis, Java

ACM Reference Format:
Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236041

1 INTRODUCTION

Pointer analysis is a family of static analysis techniques that provide a foundation for many other analyses and software engineering tasks, such as program slicing [36, 39], reflection analysis [19, 31], bug detection [13, 26], security analysis [1, 23], program verification [8, 27], and program debugging and comprehension [5, 21]. The goal of pointer analysis is to statically compute a set of objects (abstracted as their allocation sites) that a program variable may point to during run time. Although stating this goal is simple, it is challenging to produce precise analysis results without sacrificing scalability [12, 30, 35]. Thus, for decades, researchers have continued to develop sophisticated pointer analysis techniques [2, 14–16, 18, 22, 24, 25, 32, 33, 37, 38].

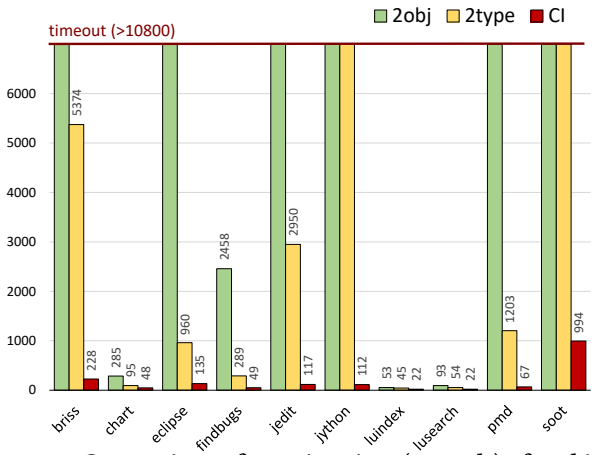


Figure 1: Comparison of running time (seconds) of 2-object sensitivity, 2-type sensitivity, and context-insensitive analyses. The y-axis is truncated to 7000 seconds for readability, and all truncated cases reach the time budget, 10800 seconds.

One of the key mechanisms for achieving high analysis precision is *context sensitivity*, which allows each program method to be analyzed differently according to the context it is used in [17]. Context sensitivity has different variants, depending on the kind of context information used. For Java programs, object-sensitivity [25] and type-sensitivity [32] have proven to be quite effective. The former is strictly more precise but less efficient than the latter [15, 37]. However, with any available variant, although the analysis can gain in precision, scalability is known to be *unpredictable* [33, 38], in the sense that programs very similar in size and other complexity metrics may have completely different scalability profiles.

Figure 1 shows time spent analyzing 10 real-world Java programs[1] under 2-object-sensitivity (`2obj`) [25], which is among the most precise variants of context sensitivity, 2-type-sensitivity (`2type`) [32], and context-insensitivity (CI). We observe that

- `2obj` is not scalable for 6 out of 10 programs within 3 hours, while it can finish running for 3 programs within 5 minutes;
- program size is far from a reliable predictor—for example, jython (12 718 methods) is smaller than `briss` (26 582 methods), however, `2type` is not scalable for the former but scalable for the latter;

[1] These are all popular open-source applications, including the heaviest (`jython` and `eclipse`) of the DaCapo benchmarks [3].

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236041

- context insensitivity exhibits very good and stable scalability for all programs (but it is much less precise).

Scalability of context-sensitivity is not only unpredictable, but also tends to be bimodal [32]: when analyzing a given program with existing context-sensitivity techniques, usually the analysis either terminates relatively quickly (we say that the analysis is *scalable* in this case), or it blows up and does not terminate even with very high time limits.

Consider a scenario where the task is to analyze a set of programs within a given time budget, for example as part of a large-scale security analysis. Should one pick a precise context-sensitive pointer analysis and take the risk that it may not scale for several programs, or pick a scalable, context-insensitive, analysis that sacrifices precision for all programs? One answer is *introspective analysis* [33], which tunes context sensitivity according to the results of a first-pass, context-insensitive analysis. However, that approach is, as the authors put it, an "if all else fails" analysis that should only be used if traditional context-sensitive algorithms fail. Thus, computing resources are wasted, because one has to wait until the context-sensitive analysis reaches a timeout, before the introspective analysis is run as a fallback.

In this paper, we present a pointer analysis framework named Scaler that has the following desirable properties.

(i) Users only have to apply Scaler once for each program, without a need to experiment with different variants of context sensitivity.

(ii) Scaler prioritizes scalability. Given a reasonable time budget, Scaler can be expected to finish analyzing any given program P within the budget. More specifically, if a context-insensitive pointer analysis is scalable for P, users can confidently expect that Scaler will also scale for P.

(iii) Scaler is able to achieve precision comparable to, or better than that of the most precise context-sensitivity variant that is scalable for P. That is, the user does not need to manually pick a context-sensitivity variant a priori, but can expect to effectively match the best option that one could have picked, on a single analysis run. Experimentally, this precision is much greater than prior introspective analyses [33].

The key insight of Scaler is that the size of context-sensitive points-to information (or, equivalently, the total memory consumed) is the critical factor that determines whether analysis of a given program using a particular context-sensitivity variant is scalable or not, and that it is possible to estimate the size of context-sensitive points-to information produced by different kinds of context sensitivity without conducting the context-sensitive analysis. This estimate can be computed using a cheap, context-insensitive pre-analysis, by leveraging the notion of *object allocation graphs* [37].

Scaler is parameterized by a number called the *total scalability threshold* (TST), which can be selected based on the available memory. For a given TST, Scaler automatically adapts different kinds of context sensitivity (e.g., object-sensitivity, type-sensitivity) and context insensitivity, at the level of individual methods in the program, to stay within the desired bound and thereby retain scalability, while utilizing the available space to maximize precision.

In summary, this paper makes the following contributions:

- We propose the Scaler pointer analysis framework that, given a total scalability threshold, automatically selects an appropriate variant of context sensitivity for each method in the program being analyzed (Section 3). The approach relies on the concept of *scalability-critical methods* that helps explain why context-sensitive pointer analysis is sometimes unscalable.
- We present a novel technique to efficiently estimate the amount of points-to information that would be needed to analyze each method with different variants of context sensitivity, using object allocation graphs (Section 4).
- We describe our open-source implementation (Section 5).
- We conduct extensive experiments by comparing Scaler with state-of-the-art pointer analyses (Section 6). The evaluation demonstrates that Scaler achieves predictable scalability for all the evaluated programs in one shot, while providing a precision that matches or even exceeds that of the best alternatives. As an example, the `jython` benchmark from DaCapo is known to cause problems for context-sensitive pointer analysis [15, 38]: `1type` is the most precise conventional pointer analysis that is scalable for `jython`, taking around 33 minutes on an ordinary PC, while `2obj` and `2type` do not complete within 3 hours. In comparison, Scaler achieves significantly better precision than both `1type` and the state-of-the-art introspective analysis [33] and takes under 8 minutes.

2 BACKGROUND

Pointer analysis typically consists of computing the points-to sets of variables in the program text.[2] Points-to sets form a relation between variables and abstract objects, i.e., a subset of

$$Var \times Obj$$

with *Var* being the set of program variables and *Obj* the set of abstract objects. Abstract objects are typically represented as *allocation sites*, i.e., instructions that allocate objects (e.g., `new` in Java) [6]. An allocation site stands for all the run-time objects it can possibly allocate. This representation by nature loses significant precision. A program variable corresponds to many run-time incarnations during program execution—not just for executions under different inputs but also for different instances of local variables during distinct activations of the same method.

To combat the loss of precision, *context-sensitive* pointer analysis enhances the computed relations to maintain a more precise abstraction of variables and objects [30, 35]. Variables get qualified with contexts, to distinguish their different incarnations. The analysis effectively computes a subset of

$$Ctx \times Var \times Obj$$

where *Ctx* is a set of contexts for variables.[3] This precision is valuable for intermediate analysis computations, even though the final analysis results get collapsed in the easily-understandable $Var \times Obj$ relation: distinguishing the behavior of a much-used variable or

[2] A second formulation is that of *alias analysis*, which computes the pairs of variables or expressions that can be aliased. For most published algorithms, computing points-to sets and computing alias sets are equivalent problems: one can be mapped to the other without affecting the algorithm's fundamental precision or scalability.

[3] For simplicity of exposition, we ignore context sensitivity for abstract objects (a.k.a., *heap cloning*) which also qualifies *Obj* with a set of heap contexts, *HCtx*, in much the same way as qualifying variables.

object (e.g., in a library method) according to the context in which it is used helps the precision at all use sites of the common code.

Two observations on context sensitivity will be important in our subsequent discussion. First, contexts qualify variables but are generally chosen *per method*. The main use of context sensitivity is to distinguish different activations of the same method (e.g., "stack frames" for the same procedure, in traditional stack-based languages), which create fresh instances of all local variables at run time. Therefore, all local variables of the same method have the same set of contexts.

Second, the worst-case complexity of a context-sensitive pointer analysis is much higher than that of a context-insensitive one: the number of computed points-to facts, and hence the complexity of the analysis, increases in the worst case multiplicatively by the size of the set *Ctx*. However, in the common case, the precision of context sensitivity often compensates, reducing the analysis complexity: different contexts divide the points-to sets of variables into non-overlapping subsets. In the ideal case, if a context-insensitive analysis computes a points-to relation $pt \subseteq Var \times Obj$, a context-sensitive analysis would compute a relation $pt' \subseteq Ctx \times Var \times Obj$ that is not greater in cardinality than $Var \times Obj$. The extra precision of the *Ctx* information would be enough to split the original points-to sets into disjoint subsets. This observation also hints at why context sensitivity has unpredictable scalability: When it works well to maintain precision, its cost is modest to none. When it fails to do so, however, the cost is orders-of-magnitude higher.

The actual definitions of the set *Ctx* can vary widely. Three main categories are call-site sensitivity, object sensitivity, and type sensitivity:

- *Call-site sensitivity* has *Ctx* be a sequence of call sites, i.e., instructions that call the method in which the qualified variable has been declared.
- *Object sensitivity* has *Ctx* be a sequence of abstract objects: they are the receiver object of the method containing the qualified variable, the receiver object of the method that allocated the previous receiver object, etc.
- *Type sensitivity* keeps the same information as object sensitivity, but objects used as contexts are collapsed not per allocation instruction but per class that contains it.

We refer the reader to surveys that discuss the options in full detail [30, 35].

3 THE SCALER FRAMEWORK

In this section, we present an overview of the Scaler approach. We first describe the idea of *scalability thresholds*, which is the key to predictable scalability (Section 3.1). Next, we explain how Scaler automatically chooses a suitable scalability threshold for each program, based on a single parameter called the *total scalability threshold* that depends only on the available space for storing points-to information (Section 3.2). These ideas are then collected in the overall Scaler framework (Section 3.3).

3.1 Scalability Thresholds

We begin by introducing the concept of *scalability-critical methods* (Section 3.1.1), and we then leverage this concept to address unscalability (Section 3.1.2).

3.1.1 Scalability-Critical Methods. The scalability of a context-sensitive pointer analysis is closely linked to the amount of points-to information it produces, i.e., the size of relation $pt' \subseteq Ctx \times Var \times Obj$. The challenge that Scaler addresses is to closely upper bound the size of pt' without performing a context-sensitive analysis, given only the result, $pt \subseteq Var \times Obj$, of a context-insensitive analysis. Prior approaches have made similar attempts to obtain scalability, but without a concrete model to predict scalability effectively, as explained in Section 7.

By then distinguishing which parts of the program can contribute disproportionately to the total size of pt', Scaler can adjust its context sensitivity at different methods, to yield higher precision only when this does not endanger scalability.

To achieve this, we introduce the concept of *scalability-critical methods*. These are methods that are likely to yield very large amounts of context-sensitive points-to information. Whether a method is scalability-critical depends on the context sensitivity in question. For example, a method may be scalability-critical under `2obj` (a 2-object-sensitive analysis) but not under `2type` (a 2-type-sensitive analysis).

Definition 3.1 (Scalability-Critical Methods). A method m is *scalability-critical* (or, for brevity, just *critical*) under context sensitivity c, if the value of the product $\texttt{\#ctx}^c_m \cdot \texttt{\#pts}_m$ exceeds a given *scalability threshold* $\texttt{ST}_p$, where

- $\texttt{\#ctx}^c_m$ is an estimate of the number of contexts for m if using context-sensitivity variant c, computed using a fast context-insensitive analysis,
- $\texttt{\#pts}_m$ is the sum of the sizes of the points-to sets for all local variables in m, computed using the same context-insensitive analysis, and
- $\texttt{ST}_p$ is a value that can be given by users (Section 3.1.2) or computed automatically (Section 3.2) for program p.

That is, we upper-bound the potential contribution of a method to the context-sensitive points-to set ($pt' \subseteq Ctx \times Var \times Obj$) by computing the number of potential combinations of possible abstract objects (i.e., a subset of *Obj*) and contexts (i.e., a subset of *Ctx*) that may pertain to the method. The individual numbers are guaranteed to be upper bounds, since they are computed by a less precise (and, thus, conservatively over-approximate) context-insensitive analysis. Accordingly, their product is guaranteed to be an upper bound of the number of potential combinations.

The intuition behind Definition 3.1 is that a method m may cause scalability problems for different reasons: (1) m is being analyzed in too many contexts, (2) too much points-to information is computed within m, or (3) although the individual numbers of contexts and points-to facts for m are not very large, their product is. For this reason, the product of the estimates $\texttt{\#ctx}^c_m$ and $\texttt{\#pts}_m$ gives an indication of potential scalability problems.

It makes sense to perform this reasoning at the method level, since (as discussed in Section 2) decisions on context sensitivity are made per-method, i.e., for all local variables of a method together.

The number $\texttt{\#pts}_m$ can be obtained directly from the context-insensitive points-to relation: $\texttt{\#pts}_m = \sum_{v \in m} |pt(v, \_)|$. Computing $\texttt{\#ctx}^c_m$ is less straightforward. This is one of the technical novelties of the approach, and we postpone its discussion until Section 4.

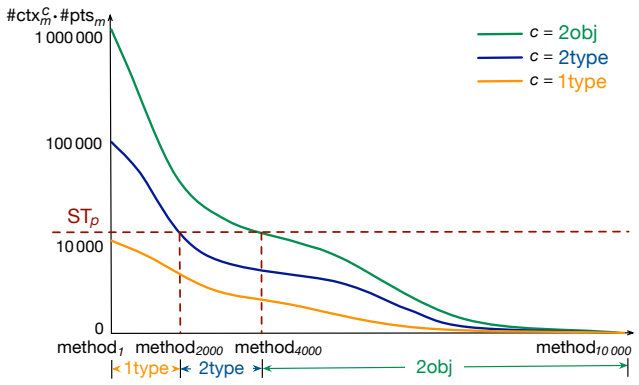


Figure 2: Choosing a context-sensitivity strategy c for the methods of program p based on a scalability threshold, $\mathtt{ST}_p$.

3.1.2 Choosing Context Sensitivity Strategies. Given a scalability threshold $\mathtt{ST}_p$, Scaler classifies each method m as scalability-critical or not, relative to a context sensitivity variant. It then ensures scalability for critical methods by selecting a cheaper (less precise) context sensitivity variant for them. As a result, different methods will be analyzed with different context sensitivity variants.

Assume there is a set of context sensitivity variants

$$\mathbb{C} = \{\, c_1,\ c_2,\ \ldots,\ c_n \,\}$$

where c_i is typically more precise (but less efficient) than c_{i-1}. For example, this set could be $\mathbb{C} = \{\mathtt{1type}, \mathtt{2type}, \mathtt{2obj}\}$.

Figure 2 illustrates (imaginary) distributions of $\mathtt{\#ctx}_m^c\mathtt{\cdot\#pts}_m$ values (log-scale y-axis) for a 10 000-method program, with each method m (x-axis) ranked by decreasing value of $\mathtt{\#ctx}_m^c\mathtt{\cdot\#pts}_m$, under different context sensitivity variants $c \in \mathbb{C}$. Assume a suitable $\mathtt{ST}_p$ value has been chosen (we explain in Section 3.2 how this can be done). According to Definition 3.1, the first 4 000 methods are scalability-critical under `2obj`, and the first 2 000 are scalability-critical also under `2type`. All 10 000 methods are non-scalability-critical under `1type`.

For each method m, Scaler selects the most precise context-sensitivity variant $c_i \in \mathbb{C}$ for which m is not scalability-critical, and context-insensitivity (*CI*) if none exists. That is, method m is analyzed with context-sensitivity $\mathit{SelectCtx}(m, \mathtt{ST}_p)$:

$$\mathit{SelectCtx}(m, \mathtt{ST}_p) = \begin{cases} c_n & \text{if } m \text{ is non-critical under } c_n \text{ for } \mathtt{ST}_p \\ c_k & \text{if } \exists c_k :\ m \text{ is non-critical under } c_k \\ & \quad \wedge\ m \text{ is critical under } c_{k+1} \text{ for } \mathtt{ST}_p \\ \mathit{CI} & \text{otherwise} \end{cases}$$

For example, in Figure 2, methods 1 to 1 999 will be analyzed using `1type`, methods 2 000 to 3 999 using `2type`, and methods 4 000 to 10 000 using `2obj`.

The remaining issue, which we address in the following section, is how to choose an appropriate scalability threshold for a given program to be analyzed.

3.2 Total Scalability Thresholds

As discussed earlier, the overall cost of a context-sensitive pointer analysis is closely linked to the size of the points-to relation being computed. An analysis that fails to terminate within realistic time limits often does so because the space needed to represent the points-to relation exhausts the available RAM. The scalability of pointer analysis is therefore related to the features of the analysis environment, most importantly the memory size.

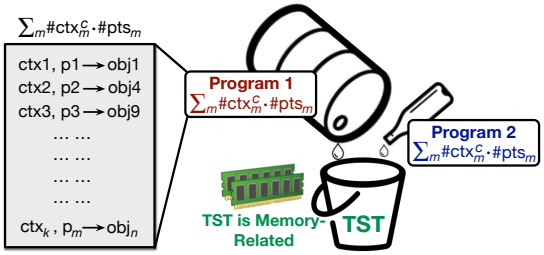


Figure 3: Context-sensitive points-to information and TST.

We now show how Scaler determines a scalability threshold for a given program, based on the concept of a *total scalability threshold* that represents the analysis capacity and can be selected based on, for example, the available memory, independently of the program being analyzed.

Definition 3.2 (Total Scalability Threshold). A *total scalability threshold* (TST) is a number that satisfies the inequality $\mathcal{E}(\mathtt{ST}_p) \leq \mathtt{TST}$ where $\mathcal{E}(\mathtt{ST}_p)$ is the estimated cost for the given value of $\mathtt{ST}_p$, computed as the sum of the sizes of the points-to relations (cf. Definition 3.1) for all the methods in the program:

$$\mathcal{E}(\mathtt{ST}_p) = \textstyle\sum_m \mathtt{\#ctx}_m^{\mathit{SelectCtx}(m, \mathtt{ST}_p)} \cdot \mathtt{\#pts}_m$$

Figure 3 illustrates the concept. The complexity of a pointer analysis (running on a given program) is closely related to the size of the points-to information, which is upper-bounded by $\mathcal{E}(\mathtt{ST}_p)$. The TST of the environment can be seen as analogous to the volume of a container. The volume of the points-to information computed by the analysis has to fit in the TST volume for the analysis to scale, i.e., $\mathcal{E}(\mathtt{ST}_p) \leq \mathtt{TST}$. This effectively normalizes the analysis capacity for all programs, regardless of the number of methods they contain, or the variants of context sensitivity employed. We discuss in Section 6 the actual TST values used in our experiments.

Importantly, in the TST inequality, $\mathtt{\#ctx}_m^{\mathit{SelectCtx}(m, \mathtt{ST}_p)}$ depends on the choice of context sensitivity, which can vary based on the approach described in Section 3.1. The *self tuning* approach of Scaler consists of computing, given the TST, an appropriate value of $\mathtt{ST}_p$ for a program p that will be used to assign appropriate context-sensitivity variants to p's methods.

To maximize precision while ensuring scalability, Scaler needs to pick the *largest* $\mathtt{ST}_p$ that satisfies the TST inequality. In this way, as many methods as possible are analyzed with the most precise context sensitivity that the total analysis capacity can afford. That is, Scaler needs to pick $max\{\mathtt{ST}_p \mid \mathcal{E}(\mathtt{ST}_p) \leq \mathtt{TST}\}$.

Figure 4 illustrates the mechanism using the distribution of Figure 2. This time, $\mathtt{ST}_p$ is not given by the user in advance. Instead, its value is computed to satisfy the inequality shown in Figure 4. For a candidate $\mathtt{ST}_p$ value, each method is classified as detailed in Section 3.1.2: it is assigned the most precise context sensitivity that keeps it from being scalability-critical.

In the case of Figure 4, the value of $\mathcal{E}(\mathtt{ST}_p)$ for this program corresponds to the sum of the areas of `A1`, `A2`, and `A3`. If `A1` + `A2` + `A3` under the candidate $\mathtt{ST}_p$ is below TST, then $\mathtt{ST}_p$ is viable, but

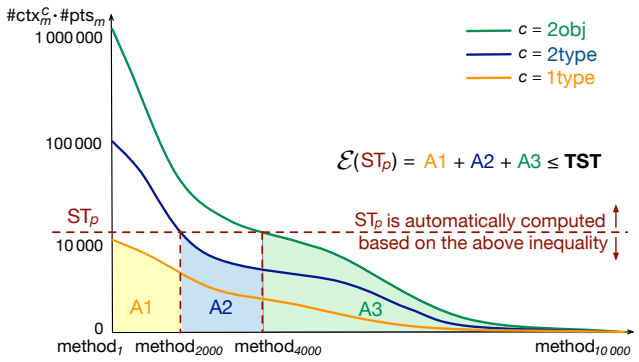


Figure 4: Selecting the scalability threshold (ST$_p$) based on the total scalability threshold (TST).

higher ST$_p$ values may also be viable. The maximum such ST$_p$ is computed using binary search in a range between 0 and the maximum value of $\texttt{\#ctx}_m^{SelectCtx(m,\mathrm{ST}_p)} \cdot \texttt{\#pts}_m$ for all m.

3.3 Overall Approach

Figure 5 shows the overall structure of the Scaler framework. Scaler is a stand-alone system that automatically selects a per-method context-sensitivity variant that can subsequently be used to configure an existing pointer analysis tool like Doop, Wala, or Chord.

Given a program p, a fast (but imprecise) context-insensitive pointer analysis is run first to produce some basic points-to information that is used for estimating $\texttt{\#pts}_m$ (Section 3.1.1) and $\texttt{\#ctx}_m^c$ (Section 4). The core of Scaler consists of one component that decides the scalability threshold based on a given total scalability threshold (Section 3.2) and another component that chooses context sensitivity strategies for all methods in the program (Section 3.1.2).

The framework relies on the collection $\mathbb{C}$ of context-sensitivity variants. All that is needed for each variant is a mechanism for estimating the number of contexts, as presented in Section 4 for object-sensitivity and type-sensitivity.

4 ESTIMATING THE NUMBER OF CONTEXTS

In Section 3.1 we postponed the discussion of how to compute an upper bound on the number of contexts ($\texttt{\#ctx}_m^c$), for a given variant of context sensitivity c, using only context-insensitive analysis results. This computation is one of the key elements of Scaler and distinguishes it from prior approaches that have also tried to adapt context sensitivity on a per-method basis [14, 33, 41].

The computation of the possible contexts for the context-sensitive analysis of a method, from only context-insensitive analysis results, is relatively easy for simple variants of context sensitivity, such as call-site sensitivity [28]: the possible contexts are call sites, readily identifiable in the program code. This computation is nontrivial for object- and type-sensitivity[24, 32], however: the contexts of a method are abstract objects, determined by the analysis mechanism. We are not aware of any prior work that performs a similar computation of possible contexts for object-sensitive or type-sensitive analyses, without running the analyses themselves.

Scaler performs this computation by leveraging the *object allocation graph* (OAG) structure proposed by Tan et al. [37]. With the OAG, the context computation problem can be formulated as a graph traversal problem. For any program, based on an OAG derived from pre-analysis (context-insensitive pointer analysis), Scaler computes the number of contexts ($\texttt{\#ctx}_m^c$) of every method for each kind of context sensitivity c used (`2obj`, `2type` and `1type` in our setup) by enumerating all contexts of the method.[4]

The OAG of a program is a directed graph. A node of the OAG represents an abstract object, which is identified by its allocation site in the program. An edge of the OAG, say $o_1 \rightarrow o_2$, represents an object-allocation relation between o_1 and o_2, i.e., o_1 is a receiver object of the method that contains the allocation site of o_2. Scaler leverages the pre-analysis to build the OAG for the given program. The OAG provides a graphical perspective of object- (and type-) sensitivity, i.e., a *k-depth* context in object-sensitivity corresponds to a *k-node* path in the OAG [37]. Thus, to compute k-object-sensitive contexts of a method m, Scaler simply enumerates *k-node* paths in the OAG, leading to the receiver objects of m.

Figure 6 illustrates the mechanism with a simple example. The allocation sites are labeled `B1`, `B2`, and `C1`, respectively. Suppose we compute `2obj` contexts for method `m()` (line 12) and its receiver object is `C1` (allocated at line 16). Further, `B1` and `B2` are two allocator objects of `C1` according to k-object-sensitivity [25, 32]. The possible (`2obj`) contexts of `m()` are `[B1,C1]` and `[B2,C1]`. The corresponding OAG is given in Figure 6, which shows two 2-node paths that are exactly the `2obj` contexts for `m()`. Since a context-insensitive analysis over-approximates the fully precise OAG, some edges may be spurious. However, this is fine, since we only need an upper bound of the number of possible contexts, to establish scalability.

Type sensitivity is an isomorphic approximation of object sensitivity [30], thus we can easily derive the contexts for type sensitivity with the OAG. Following the definition [32], Scaler computes the contexts for type sensitivity by merely replacing objects in contexts (as computed from the OAG) by the types that contain the allocation sites of the objects. For example, to compute the contexts for method `m()` (Figure 6) under `2type`, Scaler first obtains the `2obj` contexts, i.e., `[B1,C1]` and `[B2,C1]`, then replaces `B1` and `B2` by type `A`, and `C1` by type `B`. As a result, there is only one context of `m()` under `2type`, i.e., `[A,B]`.

5 IMPLEMENTATION

We have implemented Scaler as a stand-alone open-source tool in Java, available at http://www.brics.dk/scaler/. Scaler is designed to work with various pointer analysis frameworks such as Doop [4], Wala [7], Chord [26], and Soot [40]. To demonstrate its effectiveness, we have integrated Scaler with Doop [4], a state-of-the-art pointer analysis framework for Java.

In practice, we found that the $\texttt{\#ctx}_m^c \cdot \texttt{\#pts}_m$ values of some Java collection methods under package `java.util.*` are very large, because (1) the collection methods are frequently called, thus their $\texttt{\#ctx}_m^c$ values are large, and (2) many objects are passed to the

[4] We do not consider call-site sensitive analyses (`1call`, `2call`) or 1-object sensitivity (`1obj`) in our evaluation, as these analysis variants are typically both less precise and less scalable than at least one of the analyses in our setup. The Scaler approach can be adapted to any collection of context sensitivity variants, as long as a relative ordering in both increasing precision and cost is possible.

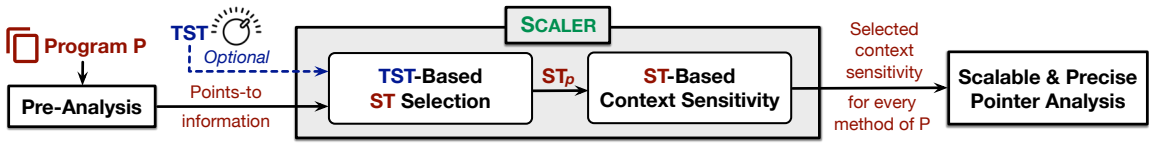


Figure 5: Overview of the SCALER framework.

```
1  class A {
2    void foo() {
3      B b1 = new B();//B1
4      b1.bar();
5    }
6    void goo() {
7      B b2 = new B();//B2
8      b2.bar();
9    }
10 }
11 class C {
12   void m() {...}
13 }
14 class B {
15   void bar() {
16     C c = new C();//C1
17     c.m();
18   }
19 }
```

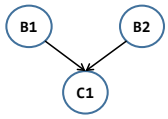


Figure 6: An example illustrating $\#\texttt{ctx}^{c}_{m}$ computation.

collection methods, thus their $\#\texttt{pts}_m$ values are large. This would affect $\texttt{ST}_p$-based context sensitivity, which may make SCALER pick less precise context sensitivity for these methods. However, collection methods are important to the precision of pointer analysis, thus should be analyzed as precisely as possible by SCALER. To address this problem, SCALER treats methods under package `java.util.*` specially, explicitly assigning them to be analyzed by the most precise context sensitivity (i.e., `2obj` in our settings).

6 EVALUATION

In the evaluation, we investigate the following research questions:

RQ1. Does SCALER consistently achieve scalability, while matching or exceeding the precision of the most precise conventional pointer analyses [25, 32] that use the same variant of context-sensitivity for the entire program?

RQ2. How does SCALER fare against introspective analysis [33] that also applies context-sensitivity selectively for the different methods of the program?

RQ3. What is the overhead of running SCALER? What are the computed values of $\texttt{ST}_p$ and what are the resulting distributions of *SelectCtx* in practice?

RQ4. How does SCALER perform with different `TST` values and different memory sizes?

Experimental Setup. All pointer analyses were performed using DOOP [29] (with the version published in the artifact of [33], which contains the exact setup for different analyses). The time budget for all analyses is 3 hours (10 800 seconds). To demonstrate the generality of SCALER's scalability and precision, we consider 10 real-world Java programs that cover different levels of program complexity. Five of these programs come from the DaCapo 2006 benchmark suite and include the toughest-to-analyze programs (`jython`, `eclipse`). The rest are well-known open-source applications. Concretely, as shown in Table 1, 6 programs (`jython`, `soot`, `pmd`, `briss`, `jedit` and `eclipse`) represent complex applications for which `2obj` is not scalable within 3 hours; 2 programs (`findbugs` and `chart`) represent medium-complexity programs for which `2obj` costs thousands or hundreds of seconds; 2 programs (`luindex` and `lusearch`) represent simple programs for which `2obj` costs only dozens of seconds. These programs are all analyzed with a large Java library: Open JDK 1.6.0_24.

SCALER uses a default `TST` value of 30M (million) which exhibits uniform scalability on all machines in our experimental setup. To answer RQ1–RQ3 (Section 6.1–6.3), we run the experiments on our default machine with a Xeon E5-2697A 2.6GHz CPU and 48GB of RAM, with the default `TST` (30M). To answer RQ4, different `TST` values are considered, and the experiments are also run on machines with smaller and larger memory sizes (Section 6.4).

6.1 Scalability and Precision of SCALER-Guided Pointer Analysis

In this section, we examine the scalability and precision of SCALER-guided pointer analysis (SCALER for short) by comparing it with conventional context-sensitive pointer analyses for Java: object-sensitivity [25] and type-sensitivity [32], which are the mainstream variants adopted in recent analysis clients. For example, object-sensitivity is widely adopted in recent Android static analysis frameworks [1, 9] and type-sensitivity is used in recent reflection analysis [20] and security analysis [10] tools.

Table 1 shows the results for all analyses. Each program has five rows of data, respectively representing context-insensitive (CI), conventional context-sensitive, SCALER, and two introspective (IntroA and IntroB) pointer analyses. (The last two analyses will be discussed in Section 6.2.) For conventional context-sensitive pointer analyses, we consider `2obj`, `2type`, and `1type`. Call-site-sensitivity and `1obj` are not considered as the former is not effective for Java programs [15, 17, 37] and the latter is usually both less efficient and less precise than `2type` [15, 32]. As it is virtually impossible to predict the scalability of a precise context-sensitive pointer analysis in advance, to get the most precise conventional analysis results we run `2obj`, `2type`, and `1type` in the given order (from the most to the least precise one) until one of them terminates within 3 hours. So in Table 1, the second row for each program shows the results of the most precise conventional pointer analysis that is scalable.

6.1.1 Scalability of SCALER-Guided Pointer Analysis. In Table 1, the third column for each program demonstrates SCALER's extremely good scalability. `2obj` is not scalable for the first six programs, and even the fast `2type` also fails to scale for `jython` and `soot`. However, SCALER scales for all of them, with its one-shot principle. In addition, for the first seven complex and large programs, SCALER runs (sometimes significantly) faster than the most precise conventional pointer analysis that is scalable, even if we add SCALER's pre-analysis time (CI).

For example, according to past literature and extensive experience [15, 32, 37, 38], `jython` is considered the most troublesome program in terms of scalability, among the DaCapo benchmarks [3].

Table 1: Efficiency and precision metrics for all programs and pointer analyses. In all cases, lower is better.

Program	Analysis	Time (seconds) 3h=10 800s	Precision metrics #may-fail casts	#poly calls	#reach methods	#call edges	Program	Analysis	Time (seconds) 3h=10 800s	Precision metrics #may-fail casts	#poly calls	#reach methods	#call edges
jython	CI	112	2 234	2 778	12 718	114 856	eclipse	CI	135	4 190	9 197	20 862	161 222
	2obj+ 2type+ 1type	>3h+>3h+1 997	2 117	2 577	12 430	111 834		2obj+ 2type	>3h+960	3 401	8 542	20 314	145 210
	Scaler	452	1 852	2 500	12 167	107 410		Scaler	652	3 211	8 486	20 374	145 953
	IntroA	381	2 202	2 632	12 663	114 095		IntroA	1 369	4 175	8 889	20 853	160 139
	IntroB	>3h	–	–	–	–		IntroB	528	3 640	8 539	20 491	149 980
soot	CI	994	16 570	16 532	32 459	415 476	findbugs	CI	49	2 508	2 925	13 036	77 370
	2obj+ 2type+ 1type	>3h+>3h+3 812	16 212	15 511	32 308	386 840		2obj	2 458	1 409	2 182	12 657	65 836
	Scaler	1 236	10 549	14 822	31 982	374 877		Scaler	272	1 452	2 195	12 676	66 177
	IntroA	1 295	16 503	15 947	32 390	413 083		IntroA	188	2 271	2 422	12 960	73 681
	IntroB	4 850	15 474	14 895	32 222	319 431		IntroB	397	2 024	2 372	12 882	70 725
pmd	CI	67	2 948	4 183	15 254	104 457	chart	CI	48	1 810	1 852	12 064	63 453
	2obj+ 2type	>3h+1 203	2 317	3 577	14 863	92 885		2obj	285	883	1 378	11 330	52 374
	Scaler	705	2 176	3 536	14 895	92 775		Scaler	254	976	1 402	11 530	53 198
	IntroA	356	2 820	3 823	15 117	101 762		IntroA	128	1 580	1 613	11 952	61 323
	IntroB	2 986	2 524	3 694	15 006	96 565		IntroB	189	1 236	1 497	11 518	55 594
briss	CI	228	4 904	6 297	26 582	176 785	luindex	CI	22	734	940	6 670	33 130
	2obj+ 2type	>3h+5 374	3 589	5 208	25 478	150 735		2obj	53	297	675	6 256	29 021
	Scaler	1 194	3 428	5 323	25 652	152 761		Scaler	53	297	675	6 256	29 021
	IntroA	497	4 889	6 076	26 507	175 565		IntroA	45	617	802	6 600	32 370
	IntroB	>3h	–	–	–	–		IntroB	48	450	714	6 316	29 835
jedit	CI	117	3 398	4 769	21 232	120 309	lusearch	CI	22	844	1 133	7 352	36 343
	2obj+ 2type	>3h+2 950	2 507	3 971	20 620	98 819		2obj	93	299	850	6 904	31 811
	Scaler	1 769	2 397	4 012	20 726	99 536		Scaler	93	299	850	6 904	31 811
	IntroA	300	3 110	4 429	21 075	116 745		IntroA	94	681	981	7 277	35 531
	IntroB	6 942	2 609	4 088	20 730	105 116		IntroB	96	462	891	6 970	32 656

To get the most precise results for a conventional pointer analysis, 23 597 seconds (3h + 3h + 1 997s) are spent before one discovers that `1type` is scalable for `jython` on our machine; however, Scaler only costs 452 seconds for `jython`, and with better precision (in terms of all precision metrics) as described in the next section.

6.1.2 Precision of Scaler-Guided Pointer Analysis. To measure precision, we use four independently useful client analyses that are often (although rarely all together) used as precision metrics in existing literature [14, 15, 32, 33, 38]. Together they paint a fairly accurate picture of analysis precision, as it impacts clients. The clients are: a cast-resolution analysis (metric: the number of cast operations that may fail—#may-fail casts), a devirtualization analysis (metric: the number of virtual call sites that cannot be disambiguated into monomorphic calls—#poly calls), a method reachability analysis (metric: the number of reachable methods—#reach methods), and a call-graph construction analysis (metric: the number of call graph edges—#call edges). In Table 1, columns (4–7) for each program list precision results. In all cases, lower is better. We find that, overwhelmingly, Scaler achieves comparable, and typically better, precision than the most precise conventional pointer analysis that is comparably scalable (e.g., `1type` for `jython`, `2type` for `pmd` and `2obj` for `luindex`). In the case of `findbugs`, the precision of Scaler is marginally lower than that of `2obj`, but the running time is an order of magnitude lower. The `chart` benchmark is the only one for which Scaler is slightly less precise than `2obj` without also being much faster, yet Scaler still attains most of the precision gains of `2obj` relative to a context-insensitive analysis. Moreover, as shown in Section 6.4, the precision of Scaler can improve by simply increasing the `TST` value.

> Answer to **RQ1**: Scaler-*guided pointer analysis exhibits extremely good and uniform scalability while matching or even exceeding the precision of the most precise conventional pointer analysis that is scalable.*

6.2 Comparison with Introspective Analyses

The closest relative of our work in past literature is introspective analysis [33]: a technique that also attempts to tune context sensitivity per-method based on a pre-analysis. Introspective analysis uses heuristics (such as "total points-to information") that do not, however, have the upper-bound guarantee, scalability emphasis, or context-number-estimation ability of Scaler.

There are two published heuristics leading to different variants of introspective analyses, IntroA and IntroB. Generally, IntroA is faster but less precise than IntroB. Like Scaler, introspective analysis also relies on a context-insensitive analysis (CI) as its pre-analysis. Unlike Scaler, which decides on each method what context-sensitivity it needs (e.g., `2obj`, `2type`, `1type` or CI), introspective analysis decides on each method whether it needs contexts. Despite the difference, the computation time of producing the context selection information is very similar (a few seconds for each program on average). As a result, the overhead of both analyses' decision making can be considered similar.

In Table 1, the last three rows for each program show the comparison results. In most programs (except `soot` and `eclipse`), IntroA

Table 2: Performance of Scaler.

Program	jython	soot	pmd	briss	jedit	eclipse	findbugs	chart	luindex	lusearch	avg.
Scaler time (seconds)	35.0	1.8	0.8	2.6	1.4	1.0	0.5	0.4	0.2	0.3	4.4

runs faster than Scaler thus also exhibiting good scalability; however, Scaler's precision is significantly better than IntroA's (Scaler wins in precision in all the precision metrics of all the programs). IntroB exhibits better precision than IntroA in all cases (when it is scalable) but it is still less precise than Scaler in all the cases except #call-edges for `soot` and #reach-methods for `chart`. Regarding efficiency, Scaler runs faster than IntroB for most programs (except `eclipse`, `chart`, and `luindex`); in addition, IntroB is not scalable for two programs (`jython` and `briss`).

Answer to **RQ2**: *Comparing with state-of-the-art adaptive analyses, introspective analyses IntroA and IntroB, both* Scaler *and IntroA exhibit extremely good scalability while* Scaler*'s precision is significantly better than IntroA's; and* Scaler*'s scalability is better than IntroB's while being more precise in most cases.*

6.3 Scaler's Effectiveness as a Pre-Analysis

This section answers RQ3 by examining the overhead of Scaler's adaptivity as well as the $\mathtt{ST}_p$ values selected by Scaler for each program, and the corresponding distribution of different kinds of context sensitivity based on the selected $\mathtt{ST}_p$.

6.3.1 Overhead of Scaler. The overall overhead of the Scaler framework (Figure 5) consists of two components: (1) a context-insensitive pointer analysis CI, which provides points-to information to Scaler, and (2) Scaler logic itself, which performs TST-based $\mathtt{ST}_p$ selection and $\mathtt{ST}_p$-based context-sensitivity. Scaler's pre-analysis time (CI) is given in Table 1, and Table 2 shows the overhead of Scaler itself. The average overall overhead of Scaler for each program is 183.8 seconds, of which CI pre-analysis costs account for the vast majority (179.4 seconds), while Scaler logic costs only 4.4 seconds. Considering the significant scalability improvements achieved by Scaler, its overhead is negligible.

Scaler spends 35 seconds for `jython`, which is markedly longer than for other programs (Table 2). The reason is that `jython` is especially complicated in the sense that many of its methods have enormous numbers of contexts, thus Scaler spends much time on $\#\mathtt{ctx}^c_m$ computation. For instance, Scaler's $\#\mathtt{ctx}^c_m$ computation shows that 130 methods of `jython` have more than 1 000 000 contexts under `2obj`. (For comparison, the maximal $\#\mathtt{ctx}^c_m$ value of a single method under `2obj` in `soot`, the largest program in our evaluation, is only 88 473.) In this way, Scaler also reveals the reason why many context-sensitive pointer analyses fail to analyze `jython` scalably as reported in previous work [15, 32, 37, 38]. Scaler avoids the problem due to its $\mathtt{ST}_p$-based context-sensitivity design (Section 3.1.2).

6.3.2 ST Values and Context-Sensitivity Distributions. Figure 7 gives the $\mathtt{ST}_p$ values selected by Scaler for each program according to the default TST (30M), and the distribution of different kinds of context sensitivity over the methods of the programs based on the selected $\mathtt{ST}_p$. Generally, given the same TST, Scaler automatically selects small $\mathtt{ST}_p$ values for complex programs (e.g., `soot` and `briss`), medium $\mathtt{ST}_p$ values for medium-complexity programs (`findbugs` and `chart`), and large $\mathtt{ST}_p$ values for simple programs (`luindex` and `lusearch`).

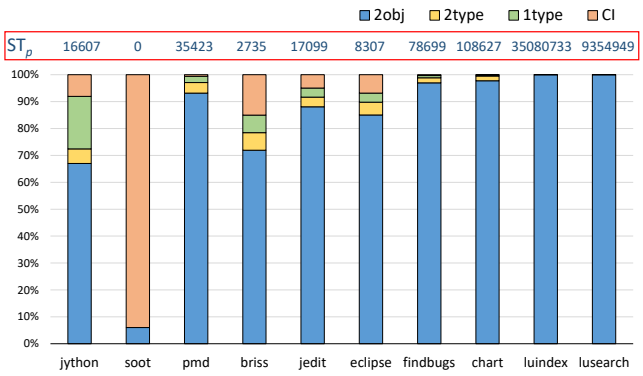


Figure 7: The $\mathtt{ST}_p$ value (on top of each bar) computed by Scaler for each program and the distribution of different kinds of context-sensitivities selected according to each $\mathtt{ST}_p$.

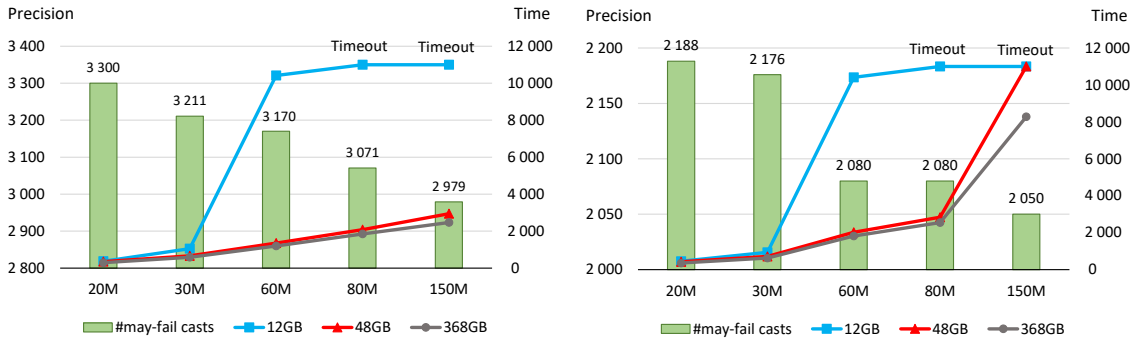
Scaler automatically selects `2obj`, the most precise context-sensitivity in our experiments, for most methods (80.6% per program on average). This is the reason why Scaler-guided pointer analysis achieves very good precision as shown in Table 1. This also demonstrates that our insight of *scalability-critical methods* (Section 3.1.1) holds in practice: in most cases, only a small set of scalability-critical methods make the whole analysis unscalable.

We next discuss two outlier cases, at both ends of the spectrum, since they are informative of Scaler's limit behavior.

`soot`. Scaler selects 0 as $\mathtt{ST}_p$ value for `soot`, so most of its methods are analyzed with context-insensitivity (CI). The reason is that `soot` is very large. The pre-analysis of Scaler reports that the total size of points-to information of `soot` (i.e., the sum of $\#\mathtt{pts}_m$ of all methods) is 110 901 529, which already exceeds our default TST (30M). As a result, Scaler automatically selects `2obj` for only 1 792 methods in package `java.util.*` (as explained in Section 5) and CI for all other methods according to the definition of *SelectCtx* in Section 3.1.2, to ensure the scalability of pointer analysis.

`luindex` *and* `lusearch`. Scaler selects two very large $\mathtt{ST}_p$ values for `luindex` and `lusearch` (the two simplest programs in our evaluation). Accordingly, it assigns `2obj` for all methods in the two programs which are all classified as non-scalability-critical methods under `2obj`. There is only one exception: the method `<java.lang.Object: void <init>()>`. In Java's type hierarchy, `Object` is the supertype of all classes; thus its `<init>` method will be called whenever a constructor is called, yielding too many contexts ($\#\mathtt{ctx}^c_m$), and a `this` variable that abstractly points to all the objects created in the program, which makes the method's $\#\mathtt{pts}_m$ value very large.

As a result, the $\#\mathtt{ctx}^c_m \cdot \#\mathtt{pts}_m$ value for this `<init>` method exceeds the selected $\mathtt{ST}_p$ value, so Scaler selects less precise context



(a) eclipse (b) pmd

Figure 8: Scaler-guided pointer analysis time (efficiency) and the corresponding number of may-fail casts (precision) with different TST values (20M, 30M, 60M, 80M and 150M) on the machines under different memory sizes (12GB, 48GB, and 368GB).

sensitivity for it. Since `java.lang.Object`'s constructor `<init>` is an empty method, analyzing it with or without `2obj` does not affect the precision of pointer analysis. Thus Scaler-guided pointer analysis achieves exactly the same precision as the conventional `2obj` analysis for `luindex` and `lusearch` (see Table 1). This again demonstrates the self-tuning ability of Scaler which can automatically obtain maximal precision for simple programs.

Answer to **RQ3**: Scaler *automatically selects appropriate* $\mathtt{ST}_p$ *values and context-sensitivity to ensure the scalability of pointer analysis for programs of varying complexity. In addition, the cost of such good adaptivity is very low.*

6.4 Scaler-Guided Pointer Analysis with Different TSTs and Memory Sizes

As real-world analysis settings may differ widely, we conduct experiments by running Scaler with different TST and memory sizes, to further evaluate the adaptiveness and usability of Scaler in practice. We run Scaler for the top six costliest-to-analyze programs in Table 1 (`jython`, `soot`, `pmd`, `briss`, `jedit`, and `eclipse`), with TST values (20M, 30M, 60M, 80M, and 150M) on three machines with different memory sizes: 12GB, 48GB, and 368GB, representing typical memory sizes of a personal laptop, a commodity server, and a large server, respectively.

Conventional pointer analyses that are not scalable for these six programs in Table 1 (with memory size 48GB) are also all unscalable on another machine with a much larger memory size, 368GB. However, under Scaler's default TST of 30M (as used in previous experiments), Scaler-guided pointer analysis still scales for all six programs even on a machine with only 12GB memory. This result further demonstrates the effectiveness of Scaler in making pointer analysis scalable in practice.

Since the six programs under different TST and memory settings exhibit similar trends, for space reasons we only show two representative programs, `eclipse` and `pmd`, in Figure 8. We illustrate precision changes via the #may-fail casts metric, which is arguably the most common precision metric in Java pointer-analysis literature [14, 15, 17, 32–34, 37, 38].

Scaler-*Guided Pointer Analysis under Different TSTs.* Figure 8 encodes a lot of information: it shows how both precision and scalability vary as a function of TST values, for three different RAM configurations. As the figure demonstrates, there is nothing special about Scaler's default 30M TST: the system adapts appropriately for both larger and smaller values. Users can simply increase or decrease the TST value to achieve better precision (up to the level the analysis setup can support) or better efficiency, respectively. This simple design with TST being the only tuning knob is able to drive complicated pointer analyses to adapt to different programs to achieve their preferred scalability and precision.

Scaler-*Guided Pointer Analysis with Different Memory Sizes.* As shown in Figure 8, with the largest TST value of 150M, Scaler-guided pointer analysis is not scalable for `pmd` with 12GB or 48GB memory, but it is scalable if using 368GB memory. With the same TST, Scaler-guided pointer analysis running on a machine with a larger memory is more likely to be scalable. For example, with a TST of 150M, Scaler-guided pointer analysis is not scalable with 12GB memory for any of the 6 programs, it is scalable for 4 programs if using 48GB memory, and it is scalable for all 6 programs with 368GB memory. This result indirectly demonstrates that scalability (and, thus, the choice of TST) is tied to memory size.

Answer to **RQ4**: *Better precision or better efficiency of* Scaler*-guided pointer analysis can be achieved by simply increasing or decreasing the* TST *value. The scalability is related to memory sizes: with the same* TST, Scaler*-guided pointer analysis is more likely to scale under larger memory; thus a small (large)* TST *is recommended for a small (large) memory size for good scalability (precision).*

7 RELATED WORK

Context sensitivity, despite bringing great precision benefits, introduces many uncertainties to scalability, which may render a pointer analysis useless in practice. We focus on related work that leverages pre-analysis to achieve good efficiency and precision trade-offs for context-sensitive pointer analysis.

Introspective analysis [33] attempts to achieve precision and scalability trade-offs by refining a context-sensitive analysis while avoiding its worst-case cost. Similar to Scaler, it first performs a pre-analysis (context-insensitive pointer analysis) to extract necessary information to guide later pointer analysis. Unlike Scaler, it relies on a set of six manually-selected metrics (e.g., the maximum

field points-to set over all fields) to define two heuristics, resulting in two introspective analyses, IntroA and IntroB, which are compared with Scaler in Section 6.2. Benefiting from the new insight of Scaler (TST-based self-tuning context sensitivity), Scaler outperforms IntroB on both precision and scalability and achieves the same level of scalability as IntroA, while being significantly more precise. Moreover, the six different metrics in introspective analysis need appropriate values to be set in advance to produce effective analysis results; Scaler's insights enable its methodology to be quite simple: it only needs one value (TST) for users to achieve better precision or better efficiency as desired, resulting in better usability in practice. Finally, introspective analysis is not one-shot: using it will always incur a cost in precision, even if the program could be analyzed more precisely. Therefore, its user will deploy it only after first attempting a precise analysis and failing.

Hassanshahi et al. [11] leverage similar metrics as introspective analysis to guide selective object-sensitive pointer analysis for large codebases. However, their pre-analysis involves several phases (that need different metrics and heuristics): a context-insensitive analysis is first performed to extract the program kernel where a context-insensitive or fixed object-sensitive analysis is still not sufficiently precise; then a fixed (heavy) object-sensitive pointer analysis is applied to the extracted (smaller) kernel to determine appropriate context depth for each selected object. After these pre-analyses, the selected object-sensitive information is used to guide the main pointer analysis which is demonstrated to work well for the OpenJDK library. However, unlike introspective analysis [33] and Scaler, the overhead of the pre-analysis is uncertain, as it heavily relies on the complexity of the extracted kernel, which further depends on various metric values selected by users. Thus, it is unclear if the technique can exhibit general effectiveness for arbitrary Java programs in practice.

Both of the above approaches [11, 33] involve metrics and heuristics that are defined manually. An alternative is to use machine learning techniques, as in the two approaches we describe next.

Wei and Ryder [41] present an adaptive context-sensitive analysis for JavaScript. They first use a machine learning algorithm to obtain the relationship between some user-defined method characteristics (extracted from a pre-analysis) and context-sensitivity choice (1-call-site-, 1-object-, or 1-parameter-sensitivity), and express the results as a decision tree. Based on domain knowledge, the decision tree is further manually adjusted to produce heuristics that are easy to interpret while the classifications can still maintain good accuracy. Finally, based on the heuristics, methods are analyzed with different context sensitivity, resulting in better precision achieved than single context-sensitive analysis.

Jeong et al. [14] propose a data-driven approach to guiding context sensitivity for Java. Unlike Scaler, where various kinds of context sensitivity with different lengths are applied to different methods, only a single kind of context sensitivity is applied to the program, and each method is finally assigned an appropriate context length, including zero (i.e., context insensitivity). As deep contexts are finally properly applied to only a subset of the methods, more efficient context-sensitive analysis can be achieved with still good precision. To select a context length for each method, 25 metrics (atomic features) are selected and a machine learning approach is used to learn heuristics based on these metrics. However, unlike Scaler's lightweight pre-analysis, the learning phase is heavy and costs 54 hours in the Jeong et al. experimental setting.

Generally, a machine learning approach is sensitive to the training on input programs, and its learned results are usually difficult to explain, for example to discern why the learning algorithm selects a given context for a method. Instead, Scaler is a principled, rigorously-modeled, approach derived from simple insights; thus its guided results are tractable and interpretable, leading to more stable and uniform effectiveness.

Unlike conventional context-sensitive pointer analysis, which uses consecutive context elements for each context, the Bean approach by Tan et al. [37] identifies and skips the redundant context elements that are useless for improving the precision of context-sensitive analysis. As a result, the saved space allows more precision-useful context elements to be involved to distinguish more contexts, making the analysis more precise with a small efficiency overhead. Precision is the focus of Bean while Scaler's is scalability. In addition, as explained in Section 4, rather than identifying redundant precision-useless context elements, Scaler leverages the OAG from Tan et al. [37] to compute the context numbers in advance.

Mahjong [38], a recent heap abstraction for pointer analysis of Java, is also based on a cheap pre-analysis, like Scaler. It enables an allocation-site-based pointer analysis to run significantly faster while achieving nearly the same precision for type-dependent clients, such as call graph construction. Differently, Scaler works for general pointer analysis, including alias analysis (i.e., not just type-dependent clients) which cannot be handled by Mahjong effectively. In addition, Scaler is able to scale for trouble programs such as `jython` where even the very fast Mahjong analysis fails.

8 CONCLUSIONS

Good scalability is hard to achieve for precise context-sensitive pointer analysis. To tackle this problem, we have introduced the Scaler framework, which automatically chooses a suitable context-sensitivity variant for each method in the given program, based on a fast, context-insensitive pre-analysis. The key insight is that it is possible to efficiently identify *scalability-critical methods* and that scalability can be predicted using the ideas of *scalability thresholds* and *total scalability thresholds*. The focus of Scaler is scalability, but at the same time it aims to maximize precision, relative to a given total scalability threshold that can be selected based on the available memory.

The experimental evaluation of Scaler demonstrates that it is able to achieve extremely good scalability while producing highly precise points-to results, in one shot, regardless of the programs being analyzed. This may directly benefit many other program analyses and software engineering tools that require scalable and precise pointer analysis. Moreover, we expect the ideas behind Scaler may help other kinds of static analyses to become more scalable with good precision for real-world programs.

ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC) under the FP7 and Horizon 2020 research and innovation programs (grant agreements 307334 and 647544).

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269. https://doi.org/10.1145/2594291.2594299

[2] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie J. Hendren, and Navindra Umanee. 2003. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, Ron Cytron and Rajiv Gupta (Eds.). ACM, 103–114. https://doi.org/10.1145/781131.781144

[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, Peri L. Tarr and William R. Cook (Eds.). ACM, 169–190. https://doi.org/10.1145/1167473.1167488

[4] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 243–262. https://doi.org/10.1145/1640089.1640108

[5] Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 363–374. https://doi.org/10.1145/1542476.1542517

[6] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. 1990. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, Bernard N. Fischer (Ed.). ACM, 296–310. https://doi.org/10.1145/93542.93585

[7] Julian Dolby et al. 2018. WALA: T. J. Watson Libraries for Analysis. http://wala.sf.net.

[8] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2 (2008), 9:1–9:34. https://doi.org/10.1145/1348250.1348255

[9] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information flow analysis of Android applications in DroidSafe. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society. https://www.ndss-symposium.org/ndss2015/information-flow-analysis-android-applications-droidsafe

[10] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. *PACMPL* 1, OOPSLA (2017), 102:1–102:28. https://doi.org/10.1145/3133926

[11] Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017*, Karim Ali and Cristina Cifuentes (Eds.). ACM, 13–18. https://doi.org/10.1145/3088515.3088519

[12] Michael Hind. 2001. Pointer analysis: haven't we solved this problem yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, John Field and Gregor Snelting (Eds.). ACM, 54–61. https://doi.org/10.1145/379605.379665

[13] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings (Lecture Notes in Computer Science)*, Jens Palsberg and Zhendong Su (Eds.), Vol. 5673. Springer, 238–255. https://doi.org/10.1007/978-3-642-03237-0_17

[14] Sehun Jeong, Minseok Jeon, Sung Deok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *PACMPL* 1, OOPSLA (2017), 100:1–100:28. https://doi.org/10.1145/3133924

[15] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 423–434. https://doi.org/10.1145/2462156.2462191

[16] Ondrej Lhoták and Laurie J. Hendren. 2003. Scaling Java points-to analysis using SPARK. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science)*, Görel Hedin (Ed.), Vol. 2622. Springer, 153–169. https://doi.org/10.1007/3-540-36579-6_12

[17] Ondrej Lhoták and Laurie J. Hendren. 2006. Context-sensitive points-to analysis: Is it worth it?. In *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings (Lecture Notes in Computer Science)*, Alan Mycroft and Andreas Zeller (Eds.), Vol. 3923. Springer, 47–64. https://doi.org/10.1007/11688839_5

[18] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 343–353. https://doi.org/10.1145/2025113.2025160

[19] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-inferencing reflection resolution for Java. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science)*, Richard E. Jones (Ed.), Vol. 8586. Springer, 27–53. https://doi.org/10.1007/978-3-662-44202-9_2

[20] Yue Li, Tian Tan, and Jingling Xue. 2015. Effective soundness-guided reflection analysis. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings (Lecture Notes in Computer Science)*, Sandrine Blazy and Thomas Jensen (Eds.), Vol. 9291. Springer, 162–180. https://doi.org/10.1007/978-3-662-48288-9_10

[21] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program tailoring: Slicing by sequential criteria. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 15:1–15:27. https://doi.org/10.4230/LIPIcs.ECOOP.2016.15

[22] Donglin Liang and Mary Jean Harrold. 1999. Efficient points-to analysis for whole-program analysis. In *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings (Lecture Notes in Computer Science)*, Oscar Nierstrasz and Michel Lemoine (Eds.), Vol. 1687. Springer, 199–215. https://doi.org/10.1007/3-540-48166-4_13

[23] Benjamin Livshits and Monica S. Lam. 2005. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*, Patrick D. McDaniel (Ed.). USENIX Association.

[24] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, Phyllis G. Frankl (Ed.). ACM, 1–11. https://doi.org/10.1145/566172.566174

[25] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (2005), 1–41. https://doi.org/10.1145/1044834.1044835

[26] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 308–319. https://doi.org/10.1145/1133981.1134018

[27] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. 2012. Statically checking API protocol conformance with mined multi-object specifications. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 925–935. https://doi.org/10.1109/ICSE.2012.6227127

[28] Micha Sharir and Amir Pnueli. 1981. *Two approaches to interprocedural data flow analysis.* Prentice-Hall, Chapter 7, 189–234.

[29] Yannis Smaragdakis et al. 2018. Doop: Framework for Java pointer analysis. http://doop.program-analysis.org.

[30] Yannis Smaragdakis and George Balatsouras. 2015. Pointer analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69. https://doi.org/10.1561/2500000014

[31] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More sound static handling of Java reflection. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings (Lecture Notes in Computer Science)*, Xinyu Feng and Sungwoo Park (Eds.), Vol. 9458. Springer, 485–503. https://doi.org/10.1007/978-3-319-26529-2_26

[32] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 17–30. https://doi.org/10.1145/1926385.1926390

[33] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 485–495. https://doi.org/10.1145/2594291.2594320

[34] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 387–400. https://doi.org/10.1145/1133981.1134027

[35] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 196–232. https://doi.org/10.1007/978-3-642-36946-9_8

[36] Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. 2007. Thin slicing. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 112–122. https://doi.org/10.1145/1250734.1250748

[37] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science)*, Xavier Rival (Ed.), Vol. 9837. Springer, 489–510. https://doi.org/10.1007/978-3-662-53413-7_24

[38] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 278–291. https://doi.org/10.1145/3062341.3062360

[39] Frank Tip. 1995. A survey of program slicing techniques. *J. Prog. Lang.* 3, 3 (1995). http://compscinet.dcs.kcl.ac.uk/JP/jp030301.abs.html

[40] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, Stephen A. MacKay and J. Howard Johnson (Eds.). IBM, 13. https://doi.org/10.1145/781995.782008

[41] Shiyi Wei and Barbara G. Ryder. 2015. Adaptive context-sensitive analysis for JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs)*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 712–734. https://doi.org/10.4230/LIPIcs.ECOOP.2015.712