Scaling Type-Based Points-to Analysis with Saturation

CHRISTIAN WIMMER, Oracle Labs, USA
CODRUT STANCU, Oracle Labs, Switzerland
DAVID KOZAK, Brno University of Technology, Czechia and Oracle Labs, Czechia
THOMAS WÜRTHINGER, Oracle Labs, Switzerland

Designing a whole-program static analysis requires trade-offs between precision and scalability. While a context-insensitive points-to analysis is often considered a good compromise, it still has non-linear complexity that leads to scalability problems when analyzing large applications. On the other hand, rapid type analysis scales well but lacks precision. We use *saturation* in a context-insensitive type-based points-to analysis to make it as scalable as a rapid type analysis, while preserving most of the precision of the points-to analysis. With saturation, the points-to analysis only propagates small points-to sets for variables. If a variable can have more values than a certain threshold, the variable and all its usages are considered saturated and no longer analyzed.

Our implementation in the points-to analysis of GraalVM Native Image, a closed-world approach to build standalone binaries for Java applications, shows that saturation allows GraalVM Native Image to analyze large Java applications with hundreds of thousands of methods in less than two minutes.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**.

Additional Key Words and Phrases: points-to analysis, static analysis, pointer analysis, Java, GraalVM

ACM Reference Format:
Christian Wimmer, Codrut Stancu, David Kozak, and Thomas Würthinger. 2024. Scaling Type-Based Points-to Analysis with Saturation. *Proc. ACM Program. Lang.* 8, PLDI, Article 187 (June 2024), 24 pages. https://doi.org/10.1145/3656417

1 INTRODUCTION

Whole-program points-to analysis [20, 46] has many diverse usages, for example, call graph construction [1, 42], security analysis [4, 33], auto-parallelization [43], bug analysis [14, 31], heap allocation analysis [51], and program debugging and understanding [30]. Unfortunately, it often scales poorly for large applications. While a lot of effort has been put into improving the scalability of context-sensitive analysis [6, 35, 55] to make it scale similarly to a context-insensitive analysis, it is often overlooked that even a standard context-insensitive points-to analysis has scalability problems because its complexity is quadratic in practice [50]. With increasing application size, there are both more variables v and more heap abstractions n. Any analysis with a quadratic complexity $O(v * n)$ has scalability problems for a sufficiently large application.

When points-to analysis is too slow, rapid type analysis [5, 53] is often used as an alternative. In contrast to a points-to analysis, it does not compute which values a certain variable can have,

Authors' addresses: Christian Wimmer, Oracle Labs, Redwood Shores, USA, christian.wimmer@oracle.com; Codrut Stancu, Oracle Labs, Zurich, Switzerland, codrut.stancu@oracle.com; David Kozak, Brno University of Technology, Brno, Czechia and Oracle Labs, Brno, Czechia, ikozak@fit.vut.cz; Thomas Würthinger, Oracle Labs, Zurich, Switzerland, thomas.wuerthinger@oracle.com.

ACM 2475-1421/2024/6-ART187
https://doi.org/10.1145/3656417

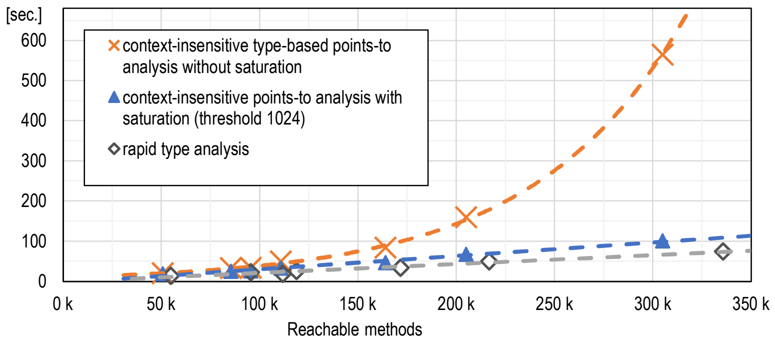

Fig. 1. Analysis time for the benchmarks presented in Section 7. The trend lines illustrate the quadratic vs. linear growth of the analysis time without vs. with saturation.

but only computes which methods are reachable. This can be done with linear time and memory complexity with regard to the size of the application, but leads to a significant loss in precision.

In this paper, we focus on a type-based points-to analysis, which uses types as heap abstractions instead of allocation sites. To improve the scalability of such a points-to analysis, we propose to use *saturation* to prune the inter-procedural pointer assignment graph data structure while the analysis is running. Saturation removes variables for which the analysis already found more values than an arbitrary but fixed threshold. The intuition is simple: if a particular result of the analysis cannot be used to improve any metric later on, it is not worth spending time to prove that result. Saturation replaces the precise points-to state for a variable with declared-type information, and removes the variable and all its usages from the pointer assignment graph. We show that a context-insensitive points-to analysis that uses saturation is nearly as fast as a rapid type analysis, but still nearly as precise as a points-to analysis without saturation. We also show that the actual saturation threshold value does not impact the analysis time much, i.e., a large range of thresholds lead to a precise yet scalable analysis.

Scalability issues of a context-insensitive points-to analysis can go unnoticed because they are only visible in large applications, i.e., many standard benchmarks are too small to observe the problems. Our evaluation shows, for example, that a benchmark with about 100 thousand reachable methods can be analyzed in about 30 seconds without saturation, and saturation reduces the analysis time only by about 25%. But for a benchmark with 300 thousand reachable methods, the analysis without saturation takes more than 9 minutes, while with saturation it takes only 100 seconds—a more than 5x difference and a linear growth in analysis time.

Figure 1 illustrates this growth of analysis time for the benchmarks evaluated later on in Section 7. We compare only three configurations here: a context-insensitive type-based points-to analysis without saturation, the same analysis with saturation (saturation threshold 1024), and a rapid type analysis. The x-axis shows the number of reachable methods for the 7 benchmarks. The number is not the same for the configurations: rapid type analysis has a noticeable loss in precision, therefore all benchmarks are to the right of the corresponding points-to analysis numbers. The y-axis shows the analysis time. The trend lines interpolate between the individual benchmark results.

Our implementation, which is based on the Native Image component of GraalVM [38], is written in Java, analyzes Java bytecode, and Java is used for all examples in this paper. However, our approach is not limited to Java or languages that compile to Java bytecode. It can be applied to all managed languages that are amenable to points-to analysis, such as C# or other languages of the .NET framework.

In summary, this paper contributes the following:

- We argue that a type-based analysis, in contrast to the more commonly used allocation site abstraction, does not significantly reduce the precision of a sound analysis for Java.
- We propose to use *saturation* to limit the size of states that are propagated by a type-based points-to analysis.
- We argue that saturation allows a type-based context-insensitive points-to analysis to scale as well as a rapid type analysis, but preserves most of the useful precision.
- Saturation is used in the production-quality points-to analysis of GraalVM Native Image. We evaluate it with various saturation thresholds, and also compare it with a rapid type analysis as well as a context-sensitive analysis to show linear scalability and high precision.

2 SYSTEM OVERVIEW

Points-to analysis is always a trade-off between time spent to do the analysis, the precision of the analysis results, and soundness. A context-insensitive points-to analysis is usually chosen when analysis time is relevant, but to increase precision, many different forms of context-sensitive analyses have been proposed [37, 48, 55]. In our experimental evaluation with Java application frameworks, context-sensitivity however does not show enough benefits to justify the analysis time. Therefore, our contributions are targeted in the opposite direction: we want to improve the scalability of a context-insensitive sound analysis as much as possible, so that the analysis time only grows linearly with the application size.

To improve the scalability, we propose to use a type-based abstraction and further optimize that abstraction using *saturation*. Saturation removes nodes from the pointer assignment graph that exceed a certain threshold of elements in the points-to set of a variable. This allows the points-to analysis to scale like a rapid type analysis, which only identifies all reachable code but does not track individual variables. But in contrast to a rapid type analysis, variables with a small points-to set still have precise information. These are the variables that matter when using analysis results for e.g., de-virtualization of method calls or removing unnecessary type checks.

2.1 Points-to Analysis

This section introduces our points-to analysis. It can be configured to be context-insensitive or context-sensitive, with several different strategies to define the context. It is flow-insensitive for fields but flow-sensitive for local variables. It starts with a set of root methods and then identifies and processes all transitively reachable methods until a fixed point is reached. It analyzes Java bytecode up to the latest Java version 21, and is sound with regards to all details of the Java specification as well as real-world usages of Java. It is imperative and written in Java.

Each reachable method is parsed from Java bytecode into a *pointer assignment graph*. For each instruction that operates on an object, a node in the pointer assignment graph is created. In addition to these nodes representing instructions, there are nodes representing actual and formal parameters as well as return values of methods. The nodes are connected via directed *use* edges.

Each node maintains a *points-to set* containing all values that can reach the variable. Nodes representing allocation instructions and nodes representing object constants act as sources that produce values, which are then propagated along the use edges. Once a value is added to a points-to set, it is never removed. Thus, the size of all sets can only grow.

In addition to type checks, Java applications also contain a high number of null checks: For memory safety, every variable needs to be checked to not be `null` before accessing memory or using it as the receiver of a method call. Our points-to analysis therefore also tracks nullness information in the points-to set. If a set contains a dedicated null-marker value, the value can be null and an explicit null check is necessary in the code. If the marker is not in the set, the variable is known to be non-null and null checks on the variable can be removed.

The pointer assignment graphs of methods are connected into a single inter-procedural graph covering the whole application. For that, nodes producing the actual arguments of method calls are connected with formal parameters nodes of the target methods, and return nodes from the target methods are connected back into the invocation nodes in the callers. The input edges of invocation nodes are thus not used for regular value propagation but rather for steering the interconnection of sources of arguments with the formal-parameter nodes (and of the appropriate return node with the invocation node).

For static methods, this linkage happens when the pointer assignment graph of the caller is created. For virtual method calls, the linkage happens dynamically during the analysis: Every time a new type is added into the points-to set of a receiver of a method call, it is used to resolve the concrete method to be linked.

The results of the points-to analysis are useful not only to identify reachable elements but also for many code optimizations. They can be used to remove unnecessary casts, null checks, and dead branches; simplify conditions that are always true or false; exclude fields that are never accessed; and to optimize virtual calls with a limited number of receiver types. Knowing the set of receivers and their types allows de-virtualization of method calls with only one possible target method, as well as polymorphic method inlining when there are a few target methods only.

2.2 Type-based Analysis

The kind of elements contained in the points-to sets depends on the configuration of the analysis. In the most precise case of a context-sensitive points-to analysis, the elements are allocation sites of objects qualified by the context how that allocation site was reached. In many context-insensitive points-to analysis implementations, elements are also the allocation sites of objects but without a context [9, 18, 22].

In this section, we argue that using a type abstraction does not reduce precision significantly in a sound points-to analysis for Java bytecode, as long as the main goal of the analysis is to find the reachable elements and to provide information for code optimizations. In the type abstraction, a points-to set is the set of types that the variable can get assigned. Several aspects of Java, both details of the Java specification and real-world usages of Java, limit the benefits of allocation site information. We present some examples here:

In Java bytecode, the allocation of an object with the newinstance bytecode is immediately followed by an invocation of the constructor. Each constructor must invoke a constructor from the superclass, until the root constructor of java.lang.Object is reached. In a context-insensitive analysis, the allocation site allows to distinguish the places of the newinstance bytecode. However, due to the lack of context, the constructor is already shared between all allocation sites of a particular type, so all field stores in the constructor coalesce. Similarly, direct field accesses using the getfield and putfield bytecode can be attributed to a particular allocation site, but many field accesses in Java are wrapped in get/set accessor methods shared between all allocation sites.

The Java Native Interface (JNI) allows calls between Java and C code. It is possible, and commonly done in the JDK class library, to allocate objects as well as access arrays and fields in C code, and pass the results to Java code. Any Java method that might be invoked from C via JNI can only use declared-type information for its parameters (unless the Java points-to analysis is co-designed with a C points-to analysis). To maintain soundness, a points-to analysis with an allocation-site abstraction must treat such opaque values conservatively and assume that they can represent any object of the declared type, i.e., use a points-to set with all allocation sites of the declared type and its subtypes.

The Java class `Unsafe` allows the allocation of objects without static knowledge of the type. For such an allocation site, the points-to analysis cannot track precise information, but can only rely on the types that the allocated object is eventually cast to. In addition, `Unsafe` allows field accesses by offset, without static knowledge of which field is accessed.

While at first it seems that objects affected by the imprecision of JNI and `Unsafe` are rare, the design of the Java type system for arrays leads to the rapid spread of such objects to nearly all places of an application: values are stored into object arrays, and objects arrays are stored in instance fields (for example in collections like `ArrayList`) and other object arrays. Without context information, many usages see all allocation sites of a particular type.

It would be possible to increase precision by not handling certain details of JNI, `Unsafe`, or other parts of the Java specification, but this would make the analysis unsound. While an analysis with a small amount of unsoundness is still useful for domains like security analysis or program understanding [32], it is out of scope for this paper to explore.

2.3 Saturation of the Pointer Assignment Graph

Points-to analysis computes the set of objects that each variable in an application might point to. The smaller the set is, the more useful the information is for later optimization stages. Smaller sets are also cheap to propagate around. The most frequent operations on points-to sets, union and filtering, are usually proportional to the set size. Larger sets on the other hand are expensive to compute and provide little useful information for later optimization stages. We therefore set a maximum size, and when a points-to set exceeds that size we call it *saturated*.

For such saturated pointer assignment nodes, the points-to analysis no longer tracks exact type information. Instead, it falls back to declared-type information, i.e., a saturated value can have any reachable subtype of its declared type. This is the same information that a rapid type analysis is using. When new types are found as reachable, the subtype set grows without any further analysis effort. Once a pointer assignment node gets saturated, we remove it from the pointer assignment graph. Saturation of one node also triggers saturation of all its usages. This greatly reduces the size of the inter-procedural graph while the analysis is running, reducing not only analysis time but also the memory footprint.

Without saturation, the size of the pointer assignment graph has quadratic complexity: in a program with v variables and n types, the complexity of the memory size to store the type states is $O(v * n)$. While in reality only few variables have the full size of n types, that number of variables still grows when the size of the application is growing, keeping the complexity quadratic. With an arbitrary but fixed saturation threshold of 100, the maximum size of the points-to sets is $O(v * 100) = O(v)$, i.e., the complexity is linear.

2.4 Saturation of Call Sites

When the receiver of a virtual method invocation gets saturated, the pointer assignment graph node for the invocation is replaced with a canonical *saturated invocation node* for the invoked method. When new subtypes of the receiver type are registered as reachable, only the single saturated flow needs to observe the new subtype and possibly link a new target method. Saturation of call sites is crucial to achieve linear time complexity. Without saturation, the analysis time for handling such invokes has quadratic time complexity: with i invocations and n classes that override a method, each of the i invocations could need to be linked to n callees. This leads to the quadratic complexity of $O(i * n)$, both for the computational effort of points-to set propagation and the memory to store the information. With saturation, the complexity of updating the single saturated invoke is reduced to $O(1 * n) = O(n)$. With an arbitrary but fixed saturation threshold of 100, updating the remaining non-saturated invokes has a maximum complexity of $O(i * 100) = O(i)$. Combining both parts, saturation reduces the complexity from $O(i * n)$ to $O(i + n)$, i.e., from quadratic to linear.

Saturation of call sites does not reduce the precision of the analysis further compared to saturation of the pointer assignment graph. It simply avoids redundant linking of the same callees to different invokes that in the end all get the maximal callee set.

The canonical example are methods declared in `java.lang.Object` like `toString`: when an application grows, both new call sites of `toString` and new implementations of `toString` are added, i.e., more invocations can reach more implementation methods. With saturation, all invocations of `toString` with a saturated receiver are replaced with the single saturated invocation of `Object.toString`, and only that single pointer assignment graph node needs to link new implementations of `toString`.

When an invocation gets saturated, the return value of that invocation is not automatically saturated. Since all implementations of `Object.toString` return a `String` (which cannot be subclassed in Java), the points-to set of the value returned by the saturated invocation only contains the single type `String`. In addition to the four methods in `Object` that are frequently invoked and overridden (`toString`, `equals`, `hashCode`, and `clone`), saturation removes also analysis bottlenecks in commonly used interface methods like `Comparable.compareTo`, `Function.apply`, or the Java collections framework.

2.5 Impact of Saturation

The main configuration parameter of saturation is the saturation threshold, i.e., the maximum size that a points-to set can reach before it is considered saturated. While it would be possible to have heuristics that avoid a single fixed value, our evaluation shows that there is a wide range of threshold values that lead to similar results. For the discussion here it is sufficient to focus on a single saturation threshold for the whole pointer assignment graph. Low saturation thresholds are not overly useful in practice.

A saturation threshold of 0 effectively degrades the points-to analysis to a rapid type analysis. As soon as a single value can reach a variable, the value is no longer tracked but instead the variable is saturated. There is no benefit of using such a saturated points-to analysis configuration instead of a rapid type analysis.

A saturation threshold of 1 only tracks variables that have a single exact value assigned. Such variables have the highest optimization potential: all type checks on such variables can be decided statically based on their single exact type, and all virtual calls can be de-virtualized because with only one receiver type there can also be only one invoked method. As confirmed by our evaluation later on, the change from threshold 0 to 1 is the biggest individual increase in analysis precision.

Further small increases of the threshold to 2, and then up to about 16, lead to substantial increases in precision: For such a low number of precisely known types, it is still likely that type checks can be decided statically. And even if a small number of types reach the receiver type of a call, all these types can resolve to the same invoked method, i.e., the method call can be de-virtualized. A more precise analysis also means that fewer methods are found as reachable, and fewer reachable methods mean less work to do for the analysis. A saturation threshold of 0 or 1 is not fastest analysis configuration. Instead, the fastest analysis uses a threshold that is low enough so that points-to set propagation is not a bottleneck yet, but high enough to achieve a good analysis precision.

Larger increases of the threshold into the hundreds and thousands of types shows only minor increases in precision: If a variable can have a hundred or a thousand types, it is already unlikely that a type check can be decided or that there is only a single implementation method for a polymorphic call (if there is only a single implementation method overall in the whole application, then the call

```java
static void main() {
  Object o1 = sourceOne();
  ... = o1.toString();

  Object o2 = sourceTwo();
  ... = o2.toString();
}
```

```java
static Object sourceOne() {
  return new A();
}

class A {
  String toString() {...}
}
```

```java
static Object sourceTwo() {
  Object[] array = ...
  int index = ...
  return array[index];
}
```

Fig. 2. Java source code for the example.

is de-virtualized even when the call site is saturated). The analysis time is increasing because there are now operations on sets with hundreds or thousands of elements.

If the saturation threshold approaches the total number of types reachable in the application, saturation becomes ineffective and the analysis converges into a traditional points-to analysis without saturation.

In addition to the list of types, saturation also loses "non-null" information, i.e., a null check of a saturated value is treated as if it can contain `null`. Since pointer assignments that get saturated are removed from the pointer assignment graph, the non-null information can also no longer be propagated around. Therefore, the number of null checks that can be proven by the analysis is affected similarly to the number of type checks.

3 EXAMPLE

We use the Java source code shown in Figure 2 to illustrate saturation and its benefits. The method `main` invokes `toString` on two different values, returned by the helper methods `sourceOne` and `sourceTwo`. Assume that `sourceOne` returns always a new instance of class `A`, while `sourceTwo` returns a value loaded from an `Object[]` array. Both invocations of `toString` are virtual.

Figure 3 shows the inter-procedural pointer assignment graph of our example. For space reasons, we show the pointer assignment graph before and after the analysis in the same figure: when the pointer assignment graph is created, all nodes shown in the figure are created, including the "removed by saturation" nodes shown with dashed lines. Saturation removes those nodes from the graph while the points-to analysis is running, as explained later. When creating the pointer assignment graph for a method, all instructions that operate on object values lead to pointer assignment nodes. In Java, many instructions also lead to additional operations required to ensure memory safety. In our example, both virtual invokes are preceded by a receiver null check.

Nodes in the pointer assignment graph perform a "union" operation: the points-to set of a node is the union of all states referenced by incoming edges. In addition, some nodes like Filter also perform a filtering operation, e.g., the points-to set of a Filter: nonNull is always non-null even if a predecessor is null; and the points-to set of a FormalReceiver is implicitly filtered with the declaring type of the method. Filter nodes are not created for a branch instruction, but for its successor blocks. In our example, the null check before the first invoke leads to a Filter: alwaysNull that tracks the state of variable `o1` when it is `null`; and to a Filter: nonNull that tracks the state when the variable `o1` is not `null`. This approach mimics the flow of values during actual program execution.

A method invocation is translated into multiple nodes: the Invoke node that links to the callees; an ActualParam node for each method parameter that propagates points-to sets into the callees; and one ActualReturn node that accumulates the values returned by the callees.

In order to link callers with callees, InvokeVirtual nodes observe the receiver type, i.e., the points-to set of their ActualParam: this. The target method is resolved for each type in that points-to set, yielding the list of callee methods. In our example, the first invoke node InvokeVirtual: toString in method `main` observes that type `A` is added to the points-to set of its ActualParam: this node. The

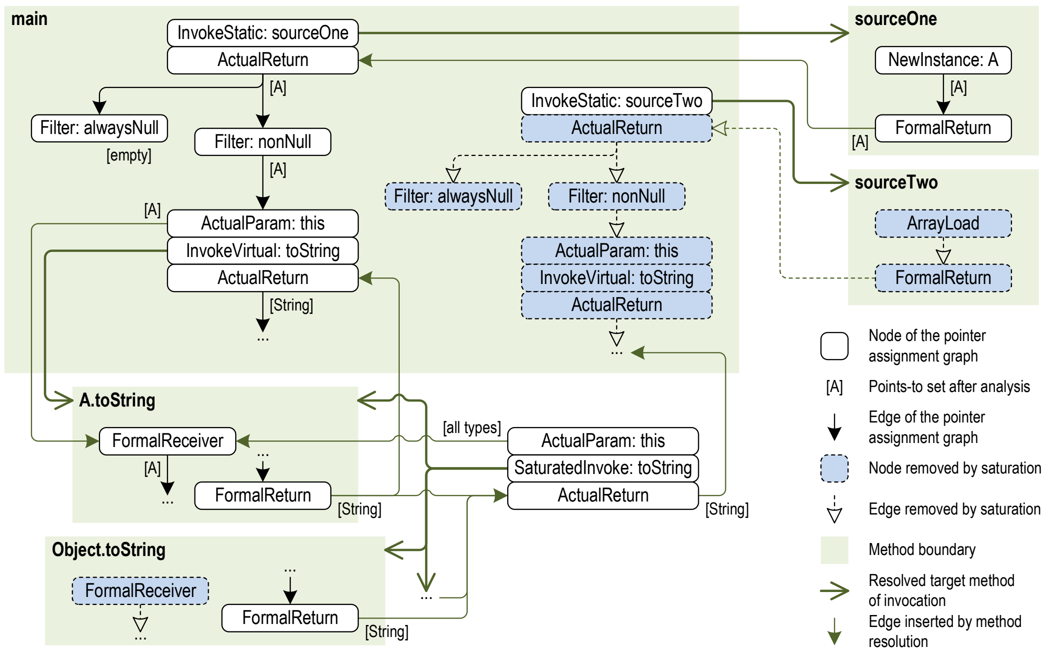

Fig. 3. Inter-procedural pointer assignment graph of the example.

target method `Object.toString` is resolved to the callee `A.toString`. Since this is the first time that `A.toString` is seen as a callee, it is marked as "reachable", its bytecode is parsed and translated into a pointer assignment graph. Then the pointer assignment graph is linked by connecting the ActualParam: this of the caller with the FormalReceiver of the callee and the FormalReturn of the callee with the ActualReturn of the caller. These flow graph edges propagate the points-to set into and out of the callee.

The first half of our example invokes the method `sourceOne` and then invokes `toString` on the result. In the pointer assignment graph, the return value of method `sourceOne` is a NewInstance node, whose points-to set contains only the type `A`. This points-to set is propagated via the formal and actual return nodes to the Filter: alwaysNull and Filter: nonNull nodes of the null check. Since the incoming points-to set is already known to be non-null, the points-to set of Filter: alwaysNull is *empty* and the Filter: nonNull does not need to filter anything. The analysis has proven that the null check is unnecessary and can be removed. The InvokeVirtual of method `Object.toString` is resolved to the single callee `A.toString`. The analysis has proven that the virtual invoke can be de-virtualized. The points-to analysis proves valuable facts for this half of our example. It also reaches the fixed point quickly because all involved points-to sets are small.

The second half of our example invokes the method `sourceTwo` and then invokes `toString` on the result. The return value of method `sourceTwo` is a load from an `Object[]` array. Tracking the exact points-to set of an array load is often futile: it is going to be close to the set of all reachable types. Saturation of pointer assignments ensures that the points-to analysis does not spend time attempting to optimize such values. First, the ArrayLoad node gets saturated. This triggers saturation of the formal return, the actual return, the non-null filter, and finally the InvokeVirtual node. Saturated nodes are removed from the pointer assignment graph, reducing the graph size while the analysis is running.

NEW	$l : x \leftarrow \textbf{\textit{new}}\ T$	CALL	$l : x \leftarrow a_0.f(a_1, ..., a_n)$	LOAD	$l : x \leftarrow r.f$
NULL	$l : x \leftarrow \textbf{\textit{null}}$	RETURN	$l : \textbf{\textit{return}}\ ret$	STORE	$l : r.f \leftarrow x$
MERGE	$l : x \leftarrow merge(x_1, ..., x_n)$				
FILTER	$l : x \leftarrow filter_{cond}(y), cond \in \{T, !T, null, !null\}$				

Fig. 4. Base language processed by the analysis.

$$\begin{array}{l} \textbf{if } x \ \textbf{\textit{is}}\ T \textbf{ then} \\ \quad \langle \text{body\_1} \rangle \\ \textbf{else} \\ \quad \langle \text{body\_2} \rangle \end{array} \quad \Longrightarrow \quad \begin{array}{l} \textbf{if } x \ \textbf{\textit{is}}\ T \textbf{ then} \\ \quad x_1 \leftarrow filter_T(x) \\ \quad \langle \text{body\_1 using } x_1 \rangle \\ \textbf{else} \\ \quad x_2 \leftarrow filter_{!T}(x) \\ \quad \langle \text{body\_2 using } x_2 \rangle \end{array}$$

Fig. 5. FILTER example.

Saturation of the virtual invocation replaces the InvokeVirtual node with the single canonical saturated node for `Object.toString`. Only this saturated invoke node needs to resolve all implementations of `Object.toString`, which includes `Object.toString` itself, `A.toString`, and all other `toString` implementations in the application. The FormalReceiver of `Object.toString` is saturated too because class `Object` has many subclasses. But the FormalReceiver of `A.toString` is not saturated, because in our example class `A` does not have any subclasses so the filtering done by the FormalReceiver node leads to a small points-to set. The points-to analysis cannot prove valuable facts for this half of our example. But it reaches the fixed point quickly because saturation prunes the pointer assignment graph early.

In summary, our example is analyzed as follows:

- Variable `o1` has the exact type `A` and can never be `null`. The virtual invoke `o1.toString` is de-virtualized, and its preceding implicit null check is removed.
- Saturation reduces analysis time by pruning the variable `o2` from the pointer assignment graph and replacing the invocation `o2.toString` with the saturated invocation.

4 FORMAL DESCRIPTION

This section provides the formal description of our analysis. The formalism is inspired by the works of He et al. [18]. The analysis takes as input a base language in the form of a control flow graph with instructions for object instantiation, object access, and method invocation. Arrays are modelled as objects with one field representing all indices merged. The base language is in static single assignment (SSA) form, i.e., all variables have one static definition [10]. Furthermore, we assume (for simplicity and without loss of generality) that all variable names are unique and every method has one return statement. Figure 4 enumerates the instructions of the base language. Each instruction is identified by a label l.

The control flow graph together with the SSA form enable flow-sensitivity for local variables. We utilize conditions to locally improve the information about a variable `x` used by a type check or null check. Other conditions do not improve the precision of the analysis and are therefore not in the formalism. As shown in Figure 5, we introduce pseudo-instructions `FILTER` at the beginning of the `if` and `else` blocks, which create new local variables whose points-to set are filtered versions according to the condition, and replace all usages of `x` dominated by the branch with the new variables. A `FILTER` instruction can take four different forms (type check, null check, and their negations) depending on the condition and the side of the branch.

$$\frac{l\colon x \leftarrow \textbf{\textit{new}}\ T,\quad o = \mathit{HeapAbs}(l, T)}{o \rightsquigarrow x,\quad \forall T' \in \mathit{SuperTypes}(T)\colon o \rightsquigarrow T'}\,\texttt{[New]} \qquad \frac{l\colon x \leftarrow \textbf{\textit{null}}}{\textbf{\textit{null}} \rightsquigarrow x}\,\texttt{[Null]}$$

$$\frac{l\colon x \leftarrow r.f,\quad o \rightsquigarrow^* r}{o.f \rightsquigarrow x}\,\texttt{[Load]} \qquad \frac{l\colon r.f \leftarrow x,\quad o \rightsquigarrow^* r}{x \rightsquigarrow o.f}\,\texttt{[Store]}$$

$$\frac{l\colon x \leftarrow \mathit{merge}(x_1, ..., x_n)}{\forall i \in [1, n]\colon x_i \rightsquigarrow x}\,\texttt{[Merge]} \qquad \frac{l\colon x \leftarrow \mathit{filter}_{cond}(y),\quad \mathit{fn} = \mathit{filterNode}(cond)}{y \rightsquigarrow \mathit{fn} \rightsquigarrow x}\,\texttt{[Filter]}$$

$$\frac{l\colon x \leftarrow a_0.f(a_1, ..., a_n),\quad o \rightsquigarrow^* a_0,\quad m = \mathit{Resolve}(o, f)}{\forall i \in [0, n]\colon a_i \rightsquigarrow p_i^m,\quad ret^m \rightsquigarrow x}\,\texttt{[Call, Return]}$$

Fig. 6. Inference rules for the base language.

Let $\mathbb{V}$, $\mathbb{H}$, $\mathbb{T}$, $\mathbb{M}$, $\mathbb{F}$, $\mathbb{L}$, and $\mathbb{Q}$ be the domains for representing sets of variables, heap abstractions (including a special `null` marker), types, methods, field names, instructions (identified by their labels), and filter nodes, respectively. The *pointer assignment graph* (PAG) is a directed graph whose nodes are from the domain $\mathbb{N} = \mathbb{V} \cup (\mathbb{H} \times \mathbb{F}) \cup \mathbb{T} \cup \mathbb{Q}$. The set of PAG edges is initially empty.

During the analysis, edges are added to the PAG and points-to sets PTS_{in}, and PTS_{out}, formally represented as functions $\mathbb{N} \longrightarrow \mathcal{P}(\mathbb{H})$, are computed. We separate points-to sets into input and output parts for each node, in order to introduce saturation into the formalism (the actual implementation only stores the output sets). In addition to variables and fields, types act as nodes in our PAG too and have points-to sets: The points-to set of a type `t` contains all generated heap abstractions of `t` as well as all subtypes of `t`. These points-to sets are used for saturation.

The following auxiliary functions are used:

- *HeapAbs*: $\mathbb{L} \times \mathbb{T} \longrightarrow \mathbb{H}$ returns the heap abstraction of the given object.
- *SuperTypes*: $\mathbb{T} \longrightarrow \mathcal{P}(\mathbb{T})$ returns all supertypes, including the type itself.
- *TypeOf*: $\mathbb{N} \longrightarrow \mathbb{T}$ returns the static type of a variable or field represented by the PAG node.
- *Resolve*: $\mathbb{H} \times \mathbb{M} \longrightarrow \mathbb{M}$ performs method resolution for the given heap abstraction and method.

The analysis can be parameterized by the *HeapAbs* function. For a standard context-insensitive points-to analysis, it is defined as $\mathit{HeapAbs}(l, T) = l : T$ where an object is represented by its allocation site and type. For the type-based points-to analysis, it is $\mathit{HeapAbs}(l, T) = T$ where an object is represented by its type only. Since there are no individual objects in the type-based analysis, the instance fields are represented as fields of the declared type and all arrays are aliased into one.

The inference rules shown in Figure 6 present how PAG edges are added based on the processed instructions in so-far discovered reachable methods. The notation $x \rightsquigarrow y$ indicates a PAG edge from x to y. The notation $o \rightsquigarrow^* t$ indicates a path in PAG from o to t that respects the filter nodes, i.e., o has to pass the conditions of all filter nodes along the path.

Rule `New` states that for each allocation a heap abstraction o is generated, and a PAG edge is created from o to the target variable x. To maintain information about all generated heap abstractions per type, a PAG edge is also created from o to all supertypes of T, including T itself. Rules `Null`, `Load`, `Store`, and `Merge` (for phi functions of the SSA form) directly express the semantics of their corresponding instructions. Rule `Filter` creates a filter node *fn* between x and y that filters objects based on its condition. Rule `Call`, which also handles `Return`, describes the linking of callers with callees in method invocations. Actual parameters a_i are connected with formal parameters p_i (including the receiver) and the return value is connected with the target variable x in the caller. The function *Resolve* marks the returned method as reachable, i.e., method resolution increases the set of reachable methods.

$$\frac{o \in \mathbb{H}}{PTS_{in}(o) = \{o\}}\texttt{[Init]} \qquad \frac{n_1 \rightsquigarrow n_2}{PTS_{out}(n_1) \subseteq PTS_{in}(n_2)}\texttt{[Propagate]}$$

$$\frac{n \in \mathbb{N}, \quad n \notin \mathit{Saturated}}{\{o \mid o \in PTS_{in}(n) \wedge \mathit{check}(o, n)\} \subseteq PTS_{out}(n)}\texttt{[UpdateNotSaturated]}$$

$$\frac{n \in \mathbb{N}, \quad |PTS_{out}(n)| > N}{n \in \mathit{Saturated}}\texttt{[Saturation]} \qquad \mathit{check}(o, n) = \begin{cases} \mathit{cond}(o) & \text{if } n \textbf{ is } \mathit{filterNode}(\mathit{cond}) \\ \mathit{true} & \text{otherwise} \end{cases}$$

$$\frac{n \in \mathbb{N}, \quad n \in \mathit{Saturated}, \quad T = \mathit{TypeOf}(n)}{PTS_{out}(n) = PTS_{out}(T)}\texttt{[UpdateSaturated]}$$

Fig. 7. Inference rules for updating points-to sets.

The rules shown in Figure 7 describe how points-to sets are computed. Rule `Init` states that the points-to set of each heap abstraction o (including `null`) is constant and contain only o. Rule `Propagate` ensures that the output PTS_{out} of any PAG node is propagated to the input PTS_{in} of all its usages. During the computation, the *Saturated* set (initially empty) containing saturated PAG nodes is gradually expanded. The existence of the *Saturated* set is only for the formalism, while the actual implementation does the opposite: the implementation maintains only non-saturated nodes and removes the saturated ones. Rule `UpdateNotSaturated` describes the propagation from PTS_{in} to PTS_{out} for non-saturated nodes. It states that all heap abstractions from the input that pass the *check* are in the output. The *check* is an auxiliary function that returns *true* for non-filter nodes and uses the *cond* predicate for the filter nodes. Rule `Saturation` defines when a node becomes saturated. To disable saturation, it is sufficient to set $N = \infty$. Finally, rule `UpdateSaturated` defines the update of saturated nodes. The points-to sets of these nodes no longer maintain precise information and instead just reuse the value from the node of their declared type.

5 CORRECTNESS

In this section, we sketch a correctness proof for our points-to analysis with saturation. The starting point is a baseline points-to analysis without saturation, i.e., we assume that such an analysis is sound and always terminates.

For soundness, all points-to sets that are not saturated are unchanged when comparing to the baseline. Since saturation of one points-to set leads to saturation of all usages of that points-to set, any non-saturated points-to set is built and updated exactly as in the baseline analysis. If a points-to set is saturated, the precise points-to set is replaced with declared-type information, i.e., the set of the declared type and all its subtypes. Since Java bytecode is a type-safe language, no points-to analysis can ever add a type to a precise points-to set that is not assignable to the declared type of the variable. This means that the declared-type information is always a superset of the precise type information. As long as the precise type information is sound, a superset of the precise information is sound too.

For termination, a context-insensitive points-to analysis is guaranteed to terminate because points-to sets only grow, i.e., in the worst case the sets of all variables grow to contain all sources of values and then the analysis terminates. Saturation can only reduce the number of variables that need a points-to set, because the application itself is not growing. The maximum size of a points-to set is only decreased by saturation: instead of the total number of types in the application, the maximum size is now the minimum of the (arbitrary but fixed) saturation threshold and the number of types in the application. In summary, saturation does not increase the number of variables and does not increase the maximum size of a points-to set, so termination is still guaranteed.

6 IMPLEMENTATION DETAILS

This section presents implementation details that are more tied to our implementation in GraalVM Native Image than the general concepts presented in the previous sections.

6.1 Data Structures

The operations performed on large points-to sets are often the bottleneck of a points-to analysis. To make union and filtering operations on points-to sets efficient, we represent them as dense bit sets where each type has a unique id that is used as the index into the bit set. This makes operations on large points-to sets fast, which occur frequently when saturation is not used.

When saturation is enabled, a sparse bit-set representation could provide benefits. But profiling showed that the points-to operations are not a bottleneck anymore with saturation, so a more optimized representation of small points-to sets was not necessary yet.

6.2 Heap Snapshotting

The input of GraalVM Native Image is Java bytecode, compiled from any language that compiles to Java bytecode such as Java, Scala, or Kotlin. The application, its libraries, the JDK, and VM components are all analyzed by the points-to analysis and then AOT compiled. The result is a native executable for a specific operating system and architecture.

Java bytecode is difficult to analyze when looking at code only. Data objects are used pervasively for things that are primitive values in other languages. For example, `enum` values are not primitive integer values but objects. Design patterns like factories as well as dependency injection approaches that decouple interfaces from their implementations lead to a static state that is initialized once and then remains constant throughout the whole run of an application. A points-to analysis that only looks at code therefore needs to be overly conservative: it needs to be ready for any implementation class to be injected at run time, instead of the one fixed implementation class that is configured for a particular application.

To overcome this limitation, we use *application initialization at build time* [57]: the application is initialized as much as possible already before running the points-to analysis. The amount of initialization is up to the user, but ideally all configuration and dependency injection is already done at build time. This leads to Java objects being allocated at build time. *Heap snapshotting* builds the object graph of such objects, which are stored in the data section of the application and form the initial Java heap when the application starts.

Heap snapshotting is done as part of the points-to analysis: whenever the analysis marks a field as reachable, heap snapshotting processes the field and follows its references. As part of heap snapshotting, new types can be marked as reachable, which also makes new methods reachable. These methods are then processed by the points-to analysis. Therefore, the fixed point of the points-to analysis not only requires a fixed point in the points-to set propagation, but also requires a fixed point of heap snapshotting.

6.3 Exception Handling

In Java, every method can throw exceptions, either implicitly as part of, e.g., field and array accesses, or explicitly using `throw`. So-called *checked exceptions* need to be listed in the method signature, while subclasses of `RuntimeException` and `Error` do not need to be listed. It would be possible to track exception types the same way as method return values in the points-to analysis, i.e., propagate the possible exception types from callees to callers in a points-to set. In our experiments, that did not pay off: nearly every method can throw a long list of implicit exceptions. Also the checked exceptions cannot be trusted, because the requirement that they are listed only applies to the

Java programming language, not the Java bytecode specification. Instead of propagating separate points-to sets for exceptions that a method can throw across the call graph, we treat the exception object of every exception handler as a node in the pointer assignment graph that can have any subclass of `java.lang.Throwable` (the base class of all exceptions in the Java type system).

6.4 Handling of Reflection and JNI

The Java reflection API provides class lookup by name, as well as access of the methods, constructors, and fields of a class. Reflection complicates points-to analysis because it is not apparent which classes, methods, and fields are accessed [25]. A conservative analysis that assumes everything is accessed using reflection would defeat the goals of a points-to analysis: we cannot determine on a fine-grained level which classes, methods, and fields are used by an application if all of them are possibly accessed via reflection. We require the developer to specify at build time which classes, methods, and fields should be visible to reflection. Only the listed elements are then processed by the points-to analysis. To help collecting reflection information, we provide a tool similar to [8]: a tracing agent that records reflection usage of the application during development.

The Java Native Interface (JNI) is handled in the same way as reflection: JNI allows C code to look up classes, methods, and fields by name and then invoke the methods or access the fields. The developer needs to specify at build time which elements are visible to JNI.

6.5 Handling of `invokedynamic`

The Java bytecode `invokedynamic` allows the invocation of a method that is not fixed yet in the bytecode, but computed before the first execution of the bytecode. A so-called *bootstrap method* is invoked that returns a `MethodHandle`, i.e., a representation of the to-be-invoked method that is in most cases similar to a reflection method reference. Initialization at build time allows us to optimize most dynamic method invocations: When the bootstrap method can be executed at build time, we convert the dynamic call site into a regular method call. The points-to analysis is not aware that the call was dynamic, and analyzes it like any other call site. This handling is possible for the most common usages of `invokedynamic` in Java, like lambda expressions, string concatenation, and `VarHandle` usages.

If a bootstrap method does not return a constant method handle, the analysis sees an invocation of the method handle interpreter, i.e., an interpreter that determines at run time which method is invoked. This reduces analysis precision significantly, because the return value of the dynamic invocation is no longer precise. But our handling is always sound and complete.

6.6 Handling of `Unsafe` Memory Access

The Java classes `sun.misc.Unsafe` and `jdk.internal.misc.Unsafe` allow direct memory access, i.e., access of fields and array elements just by providing a base object and a memory offset. Even though the usage of `Unsafe` is discouraged, it is widely used in real-world Java code. Similarly to reflection, we require a manual configuration at build time, i.e., all fields that are accessed via `Unsafe` need to be listed. For such fields, the precision of the points-to analysis is greatly reduced. All such fields must be unified to the same points-to set, i.e., all values flowing into any unsafe store affect all fields registered for `Unsafe` access.

7 EVALUATION

Our approach is used by the Native Image component of GraalVM, which produces standalone binaries for Java applications that contain the application along with all its dependencies, as well as the necessary runtime components such as the garbage collector. We evaluate with Oracle GraalVM for JDK 21, which is based on build 21.0.1+12.1 of Java 21.

We run all benchmarks 10 times and report the average of the runs (except for the context-sensitive configuration, which for time reasons we run only once). The benchmarks (except for the context-sensitive configuration) were executed on a dual-socket Intel Xeon E5-2630 v3 running at 2.40 GHz with 8 physical / 16 logical cores per socket, 128 GByte main memory, running Oracle Linux Server release 7.3. The benchmark execution was pinned to one of the two CPUs, and TurboBoost was disabled to avoid instability. The number of threads used by the points-to analysis was set to 16. We use a server configuration for benchmarking because that allows us to provide stable and reproducible numbers. However, we deliberately select an old configuration that is similar to a current developer laptop. For the context-sensitive configuration, the analysis takes hours instead of seconds/minutes for the larger benchmarks, so we use a variety of large modern servers with more cores and more memory.

We use the following benchmarks for evaluation:

- *Spring PetClinic* [56]: A large demo application for the Spring framework.
- *Micronaut MuShop Order* [39]: A large demo application using the Micronaut framework. The Order service is the largest service of MuShop.
- *Quarkus Registry* [41]: A large real-world application using the Quarkus framework. It is used to host the Quarkus extension registry.
- *Renaissance als*: A benchmark from the Renaissance suite that runs the ALS algorithm from the Spark ML library.
- *Renaissance finagle-chirper*: The largest benchmark of the Renaissance suite that does not include the Spark library.
- *columba* and *soot* are two of the largest benchmarks used in related work. We took the benchmarks from the artifact provided by [36]. Note that *columba* cannot be executed on Java 21 because it relies on internals of the JDK that have changed in Java 9, but a manual inspection of the static analysis results show that the list of reachable methods reasonably reflects the classes of the application.

We use the three most frequently used modern frameworks for Java web services in our evaluation: Spring, Micronaut, and Quarkus. In order to not be biased towards a single framework, we selected a large application from each framework. All applications perform different functions. Under no circumstance can our evaluation be seen as a comparison and ranking of the frameworks against each other. In addition, we use two of the largest benchmarks from the Renaissance suite version 0.14.1 [40]. Reporting all benchmark results from the Renaissance suite would not bring additional insights: all of the Renaissance benchmarks that use Spark are variations of the same large underlying framework, and the other Renaissance benchmarks are small in terms of reachable methods. Similarly, we are not using any benchmarks from the DaCapo or SpecJVM 2008 suites because these benchmarks are too small in terms of reachable methods to show any scalability issues when using a context-insensitive points-to analysis without saturation.

7.1 Precision and Analysis Time

We compare the precision and analysis time of several different static analysis configurations. Our context-insensitive type-based (CI) points-to analysis without saturation can be seen as a baseline: it produces reasonably precise analysis results, and can analyze medium-sized applications usually in under one minute. However, for applications with hundreds of thousands of reachable methods, scalability problems appear. When enabling saturation, a saturation threshold needs to be chosen. Figure 8 evaluates the context-insensitive points-to analysis with 1024 as an example of a large saturation threshold, and 16 as an example of a small saturation threshold. Figure 9 later on evaluates more saturation thresholds for selected benchmarks.

	Configuration	Analysis time	Reach. methods	Reach. types	Type checks	Null checks	Poly. calls	Call edges
Spring PetClinic	CI without saturation	160 sec.	205 k	39 k	100 k	231 k	71 k	3,055 k
	CI with saturation (threshold 1024)	67 sec.	205 k	39 k	102 k	234 k	71 k	3,243 k
	CI with saturation (threshold 16)	62 sec.	206 k	39 k	114 k	238 k	73 k	4,047 k
	CI always saturate (threshold 0)	66 sec.	210 k	39 k	190 k	311 k	114 k	10,204 k
	rapid type analysis	49 sec.	217 k	41 k				10,468 k
	CS (2-obj 1-heap)	20 h.	205 k	39 k	99 k	230 k	70 k	2,975 k
Micronaut MuShop Order	CI without saturation	84 sec.	164 k	29 k	72 k	159 k	50 k	3,608 k
	CI with saturation (threshold 1024)	46 sec.	164 k	29 k	73 k	161 k	50 k	3,703 k
	CI with saturation (threshold 16)	44 sec.	165 k	29 k	80 k	165 k	51 k	4,110 k
	CI always saturate (threshold 0)	46 sec.	170 k	29 k	128 k	217 k	78 k	6,778 k
	rapid type analysis	34 sec.	172 k	30 k				6,881 k
	CS (2-obj 1-heap)	5 h.	164 k	29 k	71 k	158 k	49 k	3,503 k
Quarkus Registry	CI without saturation	50 sec.	111 k	20 k	51 k	106 k	34 k	1,074 k
	CI with saturation (threshold 1024)	33 sec.	111 k	20 k	51 k	107 k	34 k	1,116 k
	CI with saturation (threshold 16)	33 sec.	111 k	20 k	55 k	109 k	35 k	1,349 k
	CI always saturate (threshold 0)	35 sec.	117 k	21 k	93 k	151 k	57 k	3,203 k
	rapid type analysis	26 sec.	119 k	22 k				3,261 k
	CS (2-obj 1-heap)	141 min.	111 k	20 k	50 k	104 k	33 k	1,010 k
Renaissance Als	CI without saturation	564 sec.	305 k	57 k	111 k	266 k	82 k	7,396 k
	CI with saturation (threshold 1024)	100 sec.	305 k	57 k	113 k	268 k	82 k	7,547 k
	CI with saturation (threshold 16)	105 sec.	313 k	57 k	128 k	272 k	84 k	18,964 k
	CI always saturate (threshold 0)	114 sec.	324 k	58 k	186 k	349 k	120 k	31,841 k
	rapid type analysis	74 sec.	336 k	58 k				31,642 k
	CS (2-obj 1-heap)	62 h.	303 k	57 k	105 k	264 k	80 k	7,059 k
Renaissance finagle-chirper	CI without saturation	33 sec.	85 k	16 k	26 k	68 k	22 k	572 k
	CI with saturation (threshold 1024)	25 sec.	85 k	16 k	26 k	68 k	22 k	578 k
	CI with saturation (threshold 16)	26 sec.	87 k	16 k	29 k	70 k	22 k	881 k
	CI always saturate (threshold 0)	27 sec.	94 k	17 k	49 k	98 k	36 k	2,293 k
	rapid type analysis	23 sec.	96 k	17 k				2,345 k
	CS (2-obj 1-heap)	45 min.	85 k	16 k	25 k	67 k	21 k	530 k
columba	CI without saturation	33 sec.	96 k	15 k	39 k	126 k	25 k	627 k
	CI with saturation (threshold 1024)	26 sec.	96 k	15 k	40 k	127 k	25 k	644 k
	CI with saturation (threshold 16)	25 sec.	96 k	15 k	42 k	130 k	26 k	804 k
	CI always saturate (threshold 0)	26 sec.	101 k	16 k	66 k	174 k	49 k	1,688 k
	rapid type analysis	19 sec.	112 k	18 k				1,982 k
	CS (2-obj 1-heap)	38 min.	95 k	15 k	39 k	125 k	24 k	598 k
soot	CI without saturation	20 sec.	51 k	7 k	26 k	61 k	18 k	387 k
	CI with saturation (threshold 1024)	19 sec.	51 k	7 k	26 k	61 k	18 k	394 k
	CI with saturation (threshold 16)	19 sec.	51 k	7 k	29 k	64 k	19 k	652 k
	CI always saturate (threshold 0)	19 sec.	55 k	8 k	47 k	91 k	34 k	1,399 k
	rapid type analysis	14 sec.	55 k	8 k				1,295 k
	CS (2-obj 1-heap)	5 min.	51 k	7 k	26 k	60 k	17 k	378 k

Fig. 8. Benchmark results comparing context-insensitive type-based (CI) points-to analysis without saturation; saturation thresholds of 1024, 16, and 0; rapid type analysis; and context-sensitive (CS) points-to analysis. For all numbers, smaller is better because all analysis configurations are sound, i.e., we do not trade soundness for an increase in precision.

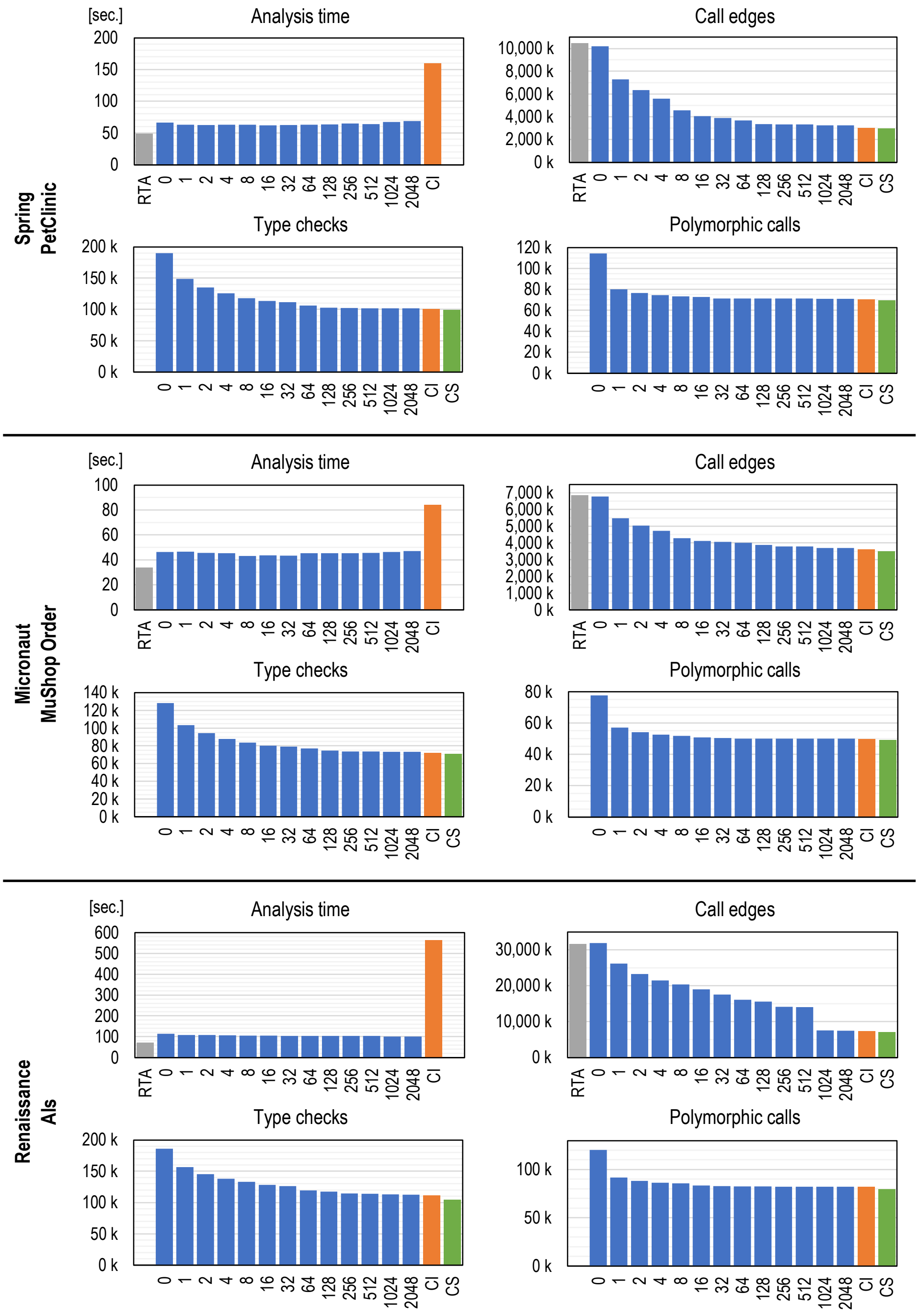

Fig. 9. Detailed results for the three largest benchmarks. RTA: rapid type analysis; numbered columns: context-insensitive points-to analysis with saturation and the saturation threshold set to that number; CI: context-insensitive type-based points-to analysis without saturation; CS: context-sensitive points-to analysis with 2-obj 1-heap context. For all numbers, smaller is better.

Next, we set the saturation threshold to 0. As explained in Section 2.5, this effectively degrades the precision of the points-to analysis to that of a rapid type analysis, while still doing some analysis work that a rapid type analysis would not need to do. For comparison, we also include the results of an actual rapid type analysis implementation that is available in GraalVM Native Image [24]. In contrast to all the other configurations, this is a separate analysis that does not share any code with the points-to analysis implementations. Finally, we present numbers for a context-sensitive (CS) points-to analysis with the context definition that was shown to work best for Java applications: 2-object-sensitive with a 1-context-sensitive heap and call-site-sensitivity for static calls [22]. This configuration uses a traditional allocation site abstraction, i.e., is not type-based like our context-insensitive configurations. While a context-insensitive configuration with an allocation site abstraction could be evaluated too, it is not reported in this paper because it must be strictly slower than our type-based context-insensitive analysis, and must be strictly less precise than our 2-obj 1-heap context-sensitive analysis. Considering the already small precision differences reported in Figure 8, adding another context-insensitive configuration would not provide further insights.

While our context-insensitive analysis (with and without saturation) is production-quality and has been optimized to reduce analysis time, our context-sensitive analysis is not as optimized. Running the context-sensitive analysis on our benchmarks takes several hours on large servers for the larger benchmarks. This slow analysis time could likely be improved by further optimizations of the data structures. We do not claim that a context-sensitive analysis is always that much slower than a context-insensitive analysis. However, the number of reachable methods and other analysis counters are valid. We therefore do claim that our configuration of a context-sensitive analysis is not able to provide meaningful benefits on the number of reachable methods and the precision of the analysis results for our benchmarks.

All configurations evaluated are sound. While it is usually possible to increase precision by giving up some soundness, our use case in GraalVM Native Image requires a sound analysis. Otherwise wrong code would be generated based on the unsound static analysis results, i.e., the binaries produced by GraalVM Native Image would crash.

Figure 8 presents the detailed results for all benchmarks. In addition to the number of reachable methods, we also show the number of reachable types. A type is reachable if at least one of its methods is reachable, if its `java.lang.Class` constant is used explicitly, or if it is a supertype or superinterface of a reachable type.

The column *Type checks* presents the number of type check operations remaining after the points-to analysis, i.e., the number of type checks that cannot be proven as “always passing” or “always failing”. We likely cover more source of type checks than related work: in addition to the `instanceof` and `checkcast` bytecode in Java that are explicit type checks, several bytecode in Java perform an implicit type check. For example, array stores require a store check, and interface calls require a type check for the receiver object. All sources of type checks mandated by the Java specification are included in our results.

The column *Null checks* presents the number of null checks not proven by the points-to analysis. This is a metric rarely evaluated in related work, but important for the optimization of Java applications: If a null check cannot be proven, it must be executed to ensure memory safety in Java. Like with type checks, we also include implicit sources of null checks such as field accesses, array accesses, and instance method invocations. The column *Polymorphic calls* reports the number of calls that could not be de-virtualized by the analysis, i.e., the number of method call sites where more than one method could be invoked at run time.

The number of *Type checks*, *Null checks*, and *Polymorphic calls* is not reported by the rapid type analysis that we are using for comparison. But we can reasonably expect that the number would be similar to our points-to analysis results with saturation threshold 0. For the metrics that are available, there is a small but measurable precision difference between rapid type analysis and saturation threshold 0. Saturation with threshold 0 still tracks "empty" points-to sets, i.e., it is able to track variables that cannot have a value and whose definition therefore must be dead code. This leads to fewer reachable methods, because with the rapid type analysis such code and its transitive closure is seen as reachable.

Finally, the column *Call edges* shows the number of call graph edges between the reachable methods. Even though this metric is frequently used in related work, we do not consider it as valuable as the other metrics. First, it is not "actionable", i.e., it cannot be used to optimize code. But more importantly, it mixes linear values (each de-virtualized call site is counted as one edge) with quadratic values (for each polymorphic call site all possible target methods are counted).

Figure 8 validates our claims that saturation improves scalability with little impact on precision: While for the smaller benchmarks with less than 100 thousand reachable methods the analysis time in all non-context-sensitive configurations is similar, the analysis time for the largest benchmark *als* is more than 5 times longer without saturation. The relative difference between the context-insensitive analysis with and without saturation increases when the number of reachable methods increases, which is a clear sign of the quadratic time complexity of the context-insensitive analysis that is resolved by saturation. The rapid type analysis is generally a bit faster than the points-to analysis with saturation, but both grow linearly with the number of reachable methods.

The precision of the points-to analysis with saturation threshold 1024 is nearly indistinguishable from the precision of the context-insensitive analysis without saturation. Even for the smaller saturation threshold 16, the number of type checks and the number of polymorphic calls is not degraded much, while saturation threshold 0 has far worse precision.

7.2 Impact of the Saturation Threshold

To further study the impact of the saturation threshold, Figure 9 shows results for the three largest benchmarks with saturation thresholds that are exponentially increased up to 2048.

The results show that the actual value of the saturation threshold has little impact on the analysis time. Within the small variations of analysis time, the threshold leading to the fastest analysis is benchmark-specific, but it is never a small threshold like 0, 1, or 2: with such a small saturation threshold, the precision of the analysis is impacted so much that the number of reachable methods increases, and a higher number of reachable methods means that the analysis needs to do more work and is therefore slower.

On the other hand, the precision of the analysis improves monotonically when increasing the saturation threshold. For the number of type checks, the improvement is more gradual, while for the number of polymorphic calls there is a distinctive precision improvement between saturation threshold 0 and 1.

The *als* benchmark shows a significant drop in the number of call edges when going from saturation threshold 512 to 1024: the number drops from 14 million with threshold 512 to 7.6 million with threshold 1024. This anomaly is an example of quadratic effects that can make the number of call edges a misleading metric. The Spark library used by the *als* benchmark is written in Scala, and Scala makes heavy use of functional interfaces. In *als* there are 10,248 implementation classes of the interface `scala.Function1`, and 2,028 call sites of `Function1.apply()`. With saturation threshold 512, 1,248 of the call sites are saturated, while with saturation threshold 1024 only 629 of the call sites are saturated. The difference of 1,248 - 629 = 619 call sites have around 80 callees without saturation. That number is still too high to have any meaningful impact on any optimization, therefore other metrics like the number of type checks and the number of polymorphic calls do not show any difference between the two saturation thresholds. But reducing the callee count of 619 call sites from 10,248 to 80 removes (10,248 - 80) * 619 = 6.3 million edges from the call graph. A single method is responsible for nearly all of the difference in the call graph size.

In summary, we see that saturation is highly effective in reducing the analysis time even when the saturation threshold is rather large like 1024. For such a large threshold the precision is nearly indistinguishable from a context-insensitive analysis without saturation, which means it is always beneficial to use saturation.

7.3 Comparison with the Evaluation in Related Work

The benchmarks *columba* and *soot* are the two benchmarks from [36] with the longest analysis time when they are analyzed with the *Tai-e* framework [54]. The authors report a context-insensitive analysis time of 117 seconds for *columba* and 107 seconds for *soot*. Our type-based context-insensitive analysis is significantly faster, requiring 33 seconds and 20 seconds, respectively. But more importantly, these benchmarks are still too small to show any scalability issues, i.e., our saturation does not decrease analysis time much because there is no performance problem to solve for such small applications.

In our evaluation, the two benchmarks are significantly larger compared to the related work. This has several reasons:

- Our numbers include a full Java runtime system that is written in Java too and analyzed as well as optimized together with the application. So our number included, e.g., the garbage collector too. A simple one-line Java "Hello world" application in our system has about 20 thousand reachable methods, most of which are not included in the related work.
- We run with a much newer Java version. The related work uses Java 6, which was released in 2006, and the JDK has increased significantly in size since then. It is also not clear how much of the JDK is included in the static analysis results of the related work, while we always include all reachable JDK methods.
- The difference for the number of type checks is even larger than the difference for the other metrics. We suspect that casts originating indirectly from instructions other than `instanceof` and `checkcast` are not included in the related work, while our numbers include all direct and indirect sources of type checks that are necessary for memory-safe execution of Java.

8 RELATED WORK

In the literature, we can find many different approaches on how to formulate and subsequently implement points-to analysis. The most popular technique is inclusion-based constraint solving, where an analysis is formulated as repeatedly resolving constraints between points-to sets of program variables [6]. This technique was proposed by Andersen [2]. Among other approaches, we can name for example type-system based analyses that utilize unification proposed by Steensgaard [52], and analyses based on a value flow graph in which the analysis can be reduced to a graph reachability problem, as presented by Li et al. [28]. Since our analysis is based on inclusion-based solving, we focus more on different approaches used within this area.

The frameworks implementing points-to analysis essentially fall into three different categories: 1) *imperative*, e.g., Spark [26], WALA [21], and Qilin [18] (written purely in Java), 2) *declarative*, e.g., DOOP [9] (written in Datalog), or 3) *hybrid*, e.g., Paddle [27] (with a declarative core in Datalog and the rest of the infrastructure in Java). Our implementation is *imperative*.

Bravenboer et al. introduced the Doop framework which implements the analysis as a declarative Datalog program where the pointer relations are encoded as datalog rules [9]. Doop achieved a full order-of-magnitude speedup in runtime when compared to Lhotak and Hendren's Paddle framework [27]. Another declared benefit is modularity and configurability which stem from the declarative nature of the tool. Many types of static analysis tools were implemented on top of Doop, for example [13, 22, 48].

Recently, He et al. introduced Qilin, which utilizes an imperative worklist-based algorithm and outperforms the declarative datalog-based implementation [18]. Our core analysis is similar to theirs with the exception that our implementation is parallel. Instead of putting new tasks into the worklist, they are scheduled in a thread pool.

An important property of points-to analysis, and in general any interprocedural analysis, is context-sensitivity, which allows each program method to be analyzed under different contexts [29]. Different types of context-sensitivity have been studied in the past [22, 37, 45, 48, 55]. Call-site-sensitivity uses method call-sites as the context, essentially simulating the runtime call stack [45]. In contrast, object-sensitivity uses object allocation sites as context elements [37]. Kastrinis et al. suggested that while these two approaches are incomparable in precision, interesting points exist in the design space when choosing different contexts for different program elements, for example, static vs. virtual calls [22]. The precision improvement depends on how well the context abstraction partitions the flow facts across the various created contexts [55].

A lot of effort has been put into studying so-called fine-grained context-sensitivity. He et al. presented a new points-to analysis framework [18] that supports fine-grained context-sensitivity. It is both selective [34, 49] (with only a subset of methods analyzed context-sensitively) and partial [35] (with only a subset of variables/objects in a method analyzed context-sensitively).

Fine-grained context-sensitivity significantly increases the state space of configuration options for points-to analysis. Consequently, the task of selecting which variables to analyze context-sensitively has been formed [17, 29, 35]. He et al. introduced Turner, whose goal is to select which variables in a program under k-object context should be analyzed context-sensitively [17]. Ideally, only so-called precision-critical variables should be considered. However, computing this set exactly is undecidable. Turner represents a sweet spot between two previous approaches Eagle (precision-preserving by reasoning about the Context Free Language reachability in the program) [35] and Zipper (examining the value flows heuristically) [29]. Turner uses the object containment and an intra-procedural object reachability analysis to decide which variables/objects in the program should be context-sensitive.

Another direction for scaling object-sensitive points-to analysis is context-debloating [19]. This approach aims to find so-called context-dependent objects. An object `o` is context-dependent if running the analysis without using context for `o` leads to precision loss. Consequently, context-sensitive analysis can be applied only to a subset of objects where it improves precision. Recently, He at al. introduced DebloaterX [16], a container-usage-based approach that identifies context-dependent objects based on three common container patterns.

While fine-grained context-sensitivity and context-sensitivity in general can be beneficial, our experiments suggest that a context-insensitive analysis is already precise enough especially for the use case of optimizing code. Our concept of saturation can be perceived as a step in the opposite direction. Rather than trying to increase the precision for the performance-critical variables, we aim at reducing the analysis time by decreasing the precision for variables that cannot be optimized anyways.

Ma et al. introduced a novel cut-shortcut approach [36] for increasing the precision of points-to analysis for Java that does not rely on context-sensitivity. The key observation is that an imprecision occurs for context-insensitivity when object flows from multiple callers are merged in a method `m` and subsequently flow outside of `m`. The proposed algorithm cuts edges that introduce the imprecision and adds shortcut edges instead that go directly from precise sources to the targets circumventing the intermediary nodes within `m`. Our approach using saturation is orthogonal to cut-shortcut. We already independently discovered and implemented their *Local Flow Pattern* to optimize method arguments returned unchanged by a method. The other patterns could be added to our analysis with saturation too to make our analysis more precise without harming scalability.

Smaragdakis et al. introduced the idea of simplifying programs for points-to analysis using set-based reasoning as opposed to value-based reasoning [47]. The authors argue that a significant portion of local fields and statements of the analyzed program can be removed by a simple intraprocedural analysis that takes into account the subset constraints for individual variables. Such optimizations can be performed offline as a preprocessing step, for example to preprocess a library, and the output can then be reused by multiple different points-to analyses. The authors also argue that similar preprocessing can be introduced for other static analysis methods.

Barbar et al. introduced optimizations of data structures used to represent points-to sets in an inclusion-based set constraint solving [6]. The main idea is to choose object identifiers in such a way as to reduce the number of empty words in bit-vectors that represent the points-to sets. The authors present several approaches how to generate the identifiers using a staged analysis, where an auxiliary cheaper analysis is used as a preprocessing step to bootstrap the main expensive analysis. The same authors also presented the idea of using hash consing from the functional programming community to represent points-to sets [7]. Another approach for scaling points-to analysis is removing cycles in the constraint graphs, explored for example by Hardekopf et al. [15] and Fähndrich et al. [11]. We solve the problem of expensive set operations by introducing saturation. The number of values in points-to sets in our implementation never exceeds a given low threshold.

Another direction on how to improve the scalability of points-to analysis was introduced by Anastasios et al. [3] as well as Fegade et al. [12]. The key observation utilized by that approach is that certain data structures, such as the collections in Java, can be represented using simpler replacement models (called *sound-modulo-analysis* by Anastasios and *semantic models* by Fegade). Models reduce the complex internal implementation while preserving necessary facts for points-to analysis. Using these models can increase the precision with a manageable performance cost.

Schubert et al. presented ModAnalyzer [44], a novel approach that performs an integrated compositional analysis for callgraph, points-to, and context-sensitive data-flow analysis. ModAnalyzer uses lossless summaries that can be persisted and reused. Utilizing reusable summaries could reduce the analysis time for our analysis as well. However, ModAnalyzer is based on C++ and the authors mention that using the same approach for languages using more virtual calls such as Java or C# could increase the portion of partial summaries, which would degrade the overall performance of ModAnalyzer.

9 CONCLUSIONS

Scalability of a static analysis is important for use cases where large real-world applications need to be analyzed as a whole, i.e., where it is not possible to split a whole-program optimization into individual analysis of smaller modules. Unfortunately, even context-insensitive points-to analysis has quadratic complexity both for execution time and memory, which until now only left rapid type analysis as a viable and scalable analysis option. However, a rapid type analysis comes with a significant loss in analysis precision.

Our type-based context-insensitive points-to analysis with saturation solves this problem: It scales linearly like a rapid type analysis, but produces analysis results that are nearly indistinguishable from a context-insensitive points-to analysis that does not use saturation. This leads to a vastly reduced analysis time for large applications, and still meaningful improvements in analysis time for small applications. With these properties, we are confident that our points-to analysis with saturation can replace and improve any current regular context-insensitive points-to analysis as well as most current usages of a rapid type analysis.

10 DATA AVAILABILITY STATEMENT

The published artifact [23] is a Docker image that contains Oracle GraalVM for JDK 21 [38], the benchmarks, and a script to run the benchmarks in the evaluated configurations.

ACKNOWLEDGMENTS

We thank all members of the GraalVM team at Oracle Labs and the Institute for System Software at the Johannes Kepler University Linz for their support and contributions. We also thank the anonymous reviewers for their feedback and suggestions to improve the paper. This work has been supported by the Czech Science Foundation project 23-06506S and the FIT BUT internal project FIT-S-23-8151. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

REFERENCES

[1] Karim Ali and Ondřej Lhoták. 2012. Application-Only Call Graph Construction. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer-Verlag, 688–712. https://doi.org/10.1007/978-3-642-31057-7_30

[2] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Ph. D. Dissertation. University of Copenhagen.

[3] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static Analysis of Java Enterprise Applications: Frameworks and Caches, the Elephants in the Room. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM Press, 794–807. https://doi.org/10.1145/3385412.3386026

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM Press, 259–269. https://doi.org/10.1145/2594291.2594299

[5] David F. Bacon and Peter F. Sweeney. 1996. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications.* ACM Press, 324–341. https://doi.org/10.1145/236337.236371

[6] Mohamad Barbar and Yulei Sui. 2021. Compacting Points-to Sets through Object Clustering. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 159. https://doi.org/10.1145/3485547

[7] Mohamad Barbar and Yulei Sui. 2021. Hash Consed Points-To Sets. In *Static Analysis: International Symposium.* Springer-Verlag, 25–48. https://doi.org/10.1007/978-3-030-88806-0_2

[8] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the International Conference on Software Engineering.* ACM Press, 241–250. https://doi.org/10.1145/1985793.1985827

[9] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications.* ACM Press, 243–262. https://doi.org/10.1145/1640089.1640108

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (1991), 451–490. https://doi.org/10.1145/115372.115320

[11] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. 1998. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM Press, 85–96. https://doi.org/10.1145/277650.277667

[12] Pratik Fegade and Christian Wimmer. 2020. Scalable Pointer Analysis of Data Structures Using Semantic Models. In *Proceedings of the International Conference on Compiler Construction.* ACM Press, 39–50. https://doi.org/10.1145/3377555.3377885

[13] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2018. Shooting from the Heap: Ultra-Scalable Static Analysis with Heap Snapshots. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM Press, 198–208. https://doi.org/10.1145/3213846.3213860

[14] Samuel Z. Guyer and Calvin Lin. 2005. Error checking with client-driven pointer analysis. *Science of Computer Programming* 58, 1 (2005), 83 – 114. https://doi.org/10.1016/j.scico.2005.02.005

[15] Ben Hardekopf and Calvin Lin. 2007. The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.*

ACM Press, 290–299. https://doi.org/10.1145/1250734.1250767

[16] Dongjie He, Yujiang Gui, Wei Li, Yonggang Tao, Changwei Zou, Yulei Sui, and Jingling Xue. 2023. A Container-Usage-Pattern-Based Context Debloating Approach for Object-Sensitive Pointer Analysis. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 30. https://doi.org/10.1145/3622832

[17] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. 2021. Accelerating Object-Sensitive Pointer Analysis by Exploiting Object Containment and Reachability. In *Proceedings of the European Conference on Object-Oriented Programming (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*. Leibniz-Zentrum für Informatik, 16:1–16:31. https://doi.org/10.4230/LIPIcs.ECOOP.2021.16

[18] Dongjie He, Jingbo Lu, and Jingling Xue. 2022. Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis. In *Proceedings of the European Conference on Object-Oriented Programming (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*. Leibniz-Zentrum für Informatik, 30:1–30:29. https://doi.org/10.4230/LIPIcs.ECOOP.2022.30

[19] Dongjie He, Jingbo Lu, and Jingling Xue. 2023. IFDS-Based Context Debloating for Object-Sensitive Pointer Analysis. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 44 pages. https://doi.org/10.1145/3579641

[20] 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*. ACM Press, 54–61. https://doi.org/10.1145/379605.379665

[21] IBM. 2020. WALA: T.J. Watson Libraries for Analysis. http://wala.sourceforge.net/

[22] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-Sensitivity for Points-to Analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 423–434. https://doi.org/10.1145/2491956.2462191

[23] David Kozak, Stancu Codrut, Christian Wimmer, and Thomas Würthinger. 2024. Scaling Points-to Analysis Using Saturation - Artifact. https://doi.org/10.5281/zenodo.10961908

[24] David Kozak, Vojin Jovanovic, Codrut Stancu, Tomáš Vojnar, and Christian Wimmer. 2023. Comparing Rapid Type Analysis with Points-To Analysis in GraalVM Native Image. In *Proceedings of the ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. ACM Press, 129–142. https://doi.org/10.1145/3617651.3622980

[25] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society, 507–518. https://doi.org/10.1109/ICSE.2017.53

[26] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Proceedings of the International Conference on Compiler Construction*. Springer-Verlag, 153–169. https://doi.org/10.1007/3-540-36579-6_12

[27] Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the Benefits of Context-Sensitive Points-to Analysis Using a BDD-Based Implementation. *ACM Transactions on Software Engineering and Methodology* 18, 1, Article 3 (oct 2008). https://doi.org/10.1145/1391984.1391987

[28] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the Performance of Flow-Sensitive Points-to Analysis Using Value Flow. In *Proceedings of the ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering*. ACM Press, 343–353. https://doi.org/10.1145/2025113.2025160

[29] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-Guided Context Sensitivity for Pointer Analysis. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 141. https://doi.org/10.1145/3276511

[30] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program Tailoring: Slicing by Sequential Criteria. In *Proceedings of the European Conference on Object-Oriented Programming (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*. 15:1–15:27. https://doi.org/10.4230/LIPIcs.ECOOP.2016.15

[31] Bozhen Liu and Jeff Huang. 2018. D4: Fast Concurrency Debugging with Parallel Differential Analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 359–373. https://doi.org/10.1145/3192366.3192390

[32] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (2015), 44–46. https://doi.org/10.1145/2644805

[33] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the USENIX Security Symposium*. USENIX.

[34] Jingbo Lu, Dongjie He, and Jingling Xue. 2021. Selective Context-Sensitivity for k-CFA with CFL-Reachability. In *Static Analysis: International Symposium*. Springer-Verlag, 261–285. https://doi.org/10.1007/978-3-030-88806-0_13

[35] Jingbo Lu and Jingling Xue. 2019. Precision-Preserving yet Fast Object-Sensitive Pointer Analysis with Partial Context Sensitivity. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 148. https://doi.org/10.1145/3360574

[36] Wenjie Ma, Shengyuan Yang, Tian Tan, Xiaoxing Ma, Chang Xu, and Yue Li. 2023. Context Sensitivity without Contexts: A Cut-Shortcut Approach to Fast and Precise Pointer Analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press. https://doi.org/10.1145/3591242

[37] 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 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM Press, 1–11. https://doi.org/10.1145/566172.566174

[38] Oracle. 2023. GraalVM. https://www.graalvm.org/

[39] Oracle. 2023. Micronaut MuShop. https://github.com/oracle-quickstart/oci-micronaut/

[40] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 31–47. https://doi.org/10.1145/3314221.3314637

[41] Quarkus. 2023. Extension Registry Application. https://github.com/quarkusio/registry.quarkus.io

[42] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. 2016. Call Graph Construction for Java Libraries. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press, 474–486. https://doi.org/10.1145/2950290.2950312

[43] Radu Rugina and Martin Rinard. 1999. Automatic Parallelization of Divide and Conquer Algorithms. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, 72–83. https://doi.org/10.1145/301104.301111

[44] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2021. Lossless, Persisted Summarization of Static Callgraph, Points-To and Data-Flow Analysis. In *Proceedings of the European Conference on Object-Oriented Programming (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*. Leibniz-Zentrum für Informatik, 2:1–2:31. https://doi.org/10.4230/LIPIcs.ECOOP.2021.2

[45] Micha Sharir, Amir Pnueli, et al. 1978. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences.

[46] 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

[47] Yannis Smaragdakis, George Balatsouras, and George Kastrinis. 2013. Set-Based Pre-Processing for Points-to Analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 253–270. https://doi.org/10.1145/2509136.2509524

[48] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, 17–30. https://doi.org/10.1145/1926385.1926390

[49] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-Sensitivity, across the Board. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 485–495. https://doi.org/10.1145/2594291.2594320

[50] Manu Sridharan and Stephen J. Fink. 2009. The Complexity of Andersen's Analysis in Practice. In *Static Analysis: International Symposium*. Springer-Verlag, 205–221. https://doi.org/10.1007/978-3-642-03237-0_15

[51] Codruţ Stancu, Christian Wimmer, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Safe and Efficient Hybrid Memory Management for Java. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*. ACM Press, 81–92. https://doi.org/10.1145/2754169.2754185

[52] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, 32–41. https://doi.org/10.1145/237721.237727

[53] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical Virtual Method Call Resolution for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 264–280. https://doi.org/10.1145/353171.353189

[54] Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM Press, 1093–1105. https://doi.org/10.1145/3597926.3598120

[55] Manas Thakur and V. Krishna Nandivada. 2020. Mix Your Contexts Well: Opportunities Unleashed by Recent Advances in Scaling Context-Sensitivity. In *Proceedings of the International Conference on Compiler Construction*. ACM Press, 27–38. https://doi.org/10.1145/3377555.3377902

[56] The Spring PetClinic Community. 2023. Open Source sample applications based on the Spring stack. https://spring-petclinic.github.io/

[57] Christian Wimmer, Codruţ Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 184. https://doi.org/10.1145/3360610

Received 2023-11-16; accepted 2024-03-31