SkipFlow: Improving the Precision of Points-to Analysis using Primitive Values and Predicate Edges

David Kozak
ikozak@fit.vut.cz
Oracle Labs and Brno University of Technology
Czechia

Codrut Stancu
codrut.stancu@oracle.com
Oracle Labs
Switzerland

Tomas Vojnar
vojnar@fi.muni.cz
Masaryk University and Brno University of Technology
Czechia

Christian Wimmer
christian.wimmer@oracle.com
Work done while being a member of Oracle Labs
USA

Abstract

A typical points-to analysis such as Andersen's or Steensgaard's may lose precision because it ignores the branching structure of the analyzed program. Moreover, points-to analysis typically focuses on objects only, not considering instructions manipulating primitive values. We argue that such an approach leads to an unnecessary precision loss, for example, when primitive constants `true` and `false` flow out of method calls. We propose a novel lightweight points-to analysis called SkipFlow that interprocedurally tracks the flow of both primitives and objects, and explicitly captures the branching structure of the code using predicate edges. At the same time, however, SkipFlow is as lightweight and scalable as possible, unlike a traditional flow-sensitive analysis. We apply SkipFlow to GraalVM Native Image, a closed-world solution to building standalone binaries for Java applications. We evaluate the implementation using a set of microservice applications as well as well-known benchmark suites. We show that SkipFlow reduces the size of the application in terms of reachable methods by 9% on average without significantly increasing the analysis time.

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

Keywords: points-to analysis, static analysis, pointer analysis, compiler, optimization

ACM Reference Format:
David Kozak, Codrut Stancu, Tomas Vojnar, and Christian Wimmer. 2025. SkipFlow: Improving the Precision of Points-to Analysis using Primitive Values and Predicate Edges. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3696443.3708932

1 Introduction

Points-to analysis has been applied in many areas including compiler optimizations [25], security analysis [4, 35], bug hunting [17, 32], escape analysis [37], call graph construction [1, 43], and program understanding [16, 30]. However, a typical points-to analysis such as Andersen's [3] or Steensgaard's [52] may loose a lot of precision because it ignores the branching structure of the analyzed programs [19].

A flow-sensitive analysis can mitigate the issue by computing information per program point [62], but such an analysis is known to have scalability issues [28]. This complicates usage in domains such as optimizing compilers or light-weight bug-hunting tools, where the analysis has to finish in *a few minutes* even on large programs.

Our key observation is that many branching instructions can be efficiently evaluated during the analysis and used to prune out unreachable successor branches without increasing the overhead of the analysis. To do so, we introduce the notion of *predicate edges* connecting the branching conditions with nodes representing instructions contained within the branches. In addition, we extend the domain of values commonly tracked by points-to analysis with *primitive values*, motivated by the observation that many branching instructions are based on primitive values returned from other methods.

We propose a novel predicated points-to analysis, named *SkipFlow*, that tracks the flow of both primitives and objects interprocedurally using a data structure called a *predicated value propagation graph* (PVPG). We show how the PVPG can be extracted by a sequential pass over a program and present an algorithm computing the *value states*, i.e., points-to sets, for all variables and fields.

We perform an extensive set of experiments using a set of modern microservice applications as well as the well-known benchmarking suites Renaissance [41] and DaCapo [7].

When applied on top of a context-insensitive typed-based points-to analysis [60], SkipFlow reduces the size of the

```
class Scene {
    void render(..., Display display) {
        if (display == null) {
            display = new FrameDisplay();
        }
        ...
    }
}

class BucketRenderer {
    void render(Display display) {
        ...
        display.imageBegin();
        ...
    }
}
```

Figure 1. A DaCapo Sunflow motivating example.

```
class SharedThreadContainer {
    Set<Thread> virtualThreads;
    public void onExit(Thread thread) {
        if (thread.isVirtual()) {
            virtualThreads.remove(thread);
        }
    }
}

class Thread {
    public boolean isVirtual() {
        return this instanceof BaseVirtualThread;
    }
}
```

Figure 2. A JDK motivating example.

applications in terms of reachable methods by max 9.2%, min 3.3%, avg 6.3% for microservices, max 17.2%, min 3.7%, avg 8.4% for Renaissance, and max 52.3%, min 3.5%, avg 13.3% for DaCapo without much negative impact on the analysis time. In fact, SkipFlow often reduces the analysis time because fewer methods have to be processed.

The implementation presented in this paper is based on GraalVM Native Image [59], which is written in Java, analyzes Java bytecode, and Java is used for all examples in this paper. Nevertheless, 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 to the state of the art the following:

- We propose a novel approach that increases the precision of a points-to analysis using *predicate edges*, in which flow-sensitivity is maintained for selected kinds of branching instructions only.
- We extend the points-to analysis to also track primitive constant values.
- We present an implementation for GraalVM Native Image and evaluate it using a set of modern microservice applications as well as the well-known benchmark suites Renaissance and DaCapo. The results show that SkipFlow reduces the size of the applications in terms of reachable methods by 9%, and also reduces analysis time in many cases.

2 Real-World Motivating Examples

Consider Figure 1 that is taken from the DaCapo Sunflow benchmark [7]. The method Scene.render has a parameter display that gets assigned a newly allocated FrameDisplay if being null initially. However, in the configuration of Sunflow used by the benchmark, the value of display is never null. Since this is the only place where the class FrameDisplay is instantiated, we can conclude it is not necessary for the execution at all. Eventually, the imageBegin method is called on display in the render method of the BucketRenderer class, which for a FrameDisplay transitively calls into the AWT and Swing libraries, none of which are needed. Yet, a flow-insensitive analysis is not precise enough to prove that. As it considers neither the order of statements nor the branching structure, its pointer assignment graph contains the spurious path ***new*** FrameDisplay() $\rightsquigarrow$ $\texttt{display}_{\texttt{Scene}}$ $\rightsquigarrow$ $\texttt{display}_{\texttt{BucketRenderer}}$. On the other hand, a flow-sensitive analysis can cover this case (assuming it evaluates branching conditions and tracks the *nullness* of values), but at the cost of significantly higher analysis overhead. In SkipFlow, the PVPG contains a *predicate* edge $\texttt{display}_{\texttt{Scene}} == \texttt{null}$ $\rightsquigarrow_{pred}$ ***new*** FrameDisplay(), which never triggers, thus preventing FrameDisplay from being considered as instantiated. This ensures that AWT and Swing are proven unreachable.

As a second example, consider the program in Figure 2. The top part contains the method onExit taken from the class SharedThreadContainer from the jdk.internal.vm package. The method checks whether an exited thread is a virtual thread and if so, the thread is removed from the set of virtual threads the class maintains. The actual logic of Thread.isVirtual() simply checks whether the thread is a subclass of the BaseVirtualThread class.

If the application does not use virtual threads, the body of the if statement is dead code. To prove that, we need an interprocedural analysis (since the condition and the type check are in different methods) that tracks both the flow of types (to prove that the type check always fails) and the flow of primitive values (to propagate the false value from the type check back to the caller). Furthermore, the analysis has to be at least partially flow-sensitive to utilize the information and consider the remove() call unreachable.

```
if (x instanceof T) {
    use1(x);
} else {
    use2(x);
}
```

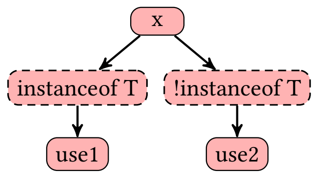


Figure 3. A type check example for filtering flows.

```
if (x > 10) {
    m();
} else {
    f();
}
```

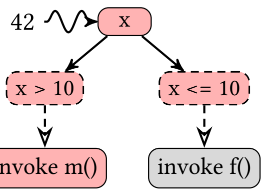


Figure 4. A predicate example. Enabled flows are coloured in red, and disabled flows in grey.

Our analysis satisfies all these requirements and successfully removes the call.

These examples are not artificial: the former comes from the well-established benchmark DaCapo Sunflow and the latter from the JDK itself. We discovered both of them when manually inspecting the results of our analysis, but they are common patterns used to dynamically configure an application. In our experiments, we show that handling such cases precisely has a significant impact on the overall precision of the analysis.

3 System Overview

This section presents a high-level overview of our analysis, informally describes the intuition behind it, and lists its key features. In the examples, we utilize statements `use(x)` in code snippets. Their purpose is to represent an arbitrary instruction resulting in a `use` dependency of a variable `x` at a given program point.

Analysis Overview. Our analysis is field-sensitive and partially flow-sensitive as explained later in this section. The analysis starts with a set of root methods, e.g., {`main`}, and then processes all transitively reachable methods until a fixed point is reached. Each reachable method is transformed into a *predicated value propagation graph* (PVPG) that represents the flow of both primitive values and objects. Nodes in the PVPG are called *flows*. Roughly, flows correspond to memory locations and instructions in the analyzed language. We defer a more precise definition of flows to Section 4 after we describe the intuition behind the key features of a PVPG. Flows can be connected by three types of edges: *use* edges for classical value propagation; *predicate* edges used for signalling when a given flow becomes *executable* (essentially unblocking value propagation through this flow); and *observe* edges to reflect that some flow-specific actions need to be performed when the value of some other flow changes (e.g., when a new type appears in a flow modeling the receiver of an invoke operation, the resolution and linking of a new target method might be needed). The graphs of individual methods are connected into an interprocedural graph by linking the actual arguments with the formal parameters and the return from the callee back to the invocation in the caller.

Filtering Flows. While SkipFlow is mostly flow-insensitive, we maintain a certain degree of flow-sensitivity using multiple approaches. First, the analysis is executed on a base language in static single assignment (SSA) form [13], which maintains flow-sensitivity for local variables. Second, we model conditional branches that involve type checks, null checks, and primitive comparisons to increase the precision. Consider the example shown in Figure 3. It is obvious from the structure of the program that the value of `x` in `use1` can only be of the type `T` (or some of its subtypes), and similarly the value of `x` in `use2` can never be of the type `T` (nor of any of its subtypes). Correspondingly, the filtering flow `instanceOf T` only passes further the type `T` (and its subtypes) for `x`, and vice versa for the other branch. When the *value state* of a variable ends up empty in a branch, the code of that branch is proven unreachable by the analysis.

Control Flow Predicates. Control flow predicates model the relationship between a condition in a branching instruction and the nodes within its branches. An instruction within a `then` branch is executable iff the flow representing the condition of the branch has a non-empty *value state*, and similarly for the `else` branch. Contrary to the typical Andersen's points-to analysis [2], flows in our analysis only propagate values if the *value state* of their predicate becomes non-empty. Consider the example in Figure 4. Predicate edges are visualized using dashed lines with empty arrow heads. If the value of `x` is known to be 42, it passes only the condition `x > 10` and thus only the flow representing the invoke `m()` is marked as executable. The condition `x <= 10` of the `else` branch filters `x` to `empty`, not enabling the flow representing the invoke `f()`. In this example, a primitive comparison acts as the predicate. Similarly, any type check or null check expressed as a filtering flow is a predicate for the beginning of its block. When using predicates, the method `f()` is not marked as reachable and therefore not analyzed.

One might think that the above approach is covered by *constant folding* commonly implemented by compilers. However, constant folding is typically implemented within the scope of a single method, therefore it covers only the case when `x` is defined as a constant locally. In the case where `x` is not a constant locally, e.g. it is a method parameter, achieving the same effect using only intraprocedural optimizations is no longer possible.

```
if (x != null) {
    y = 5;
} else {
    y = 10;
}
use(y);
```

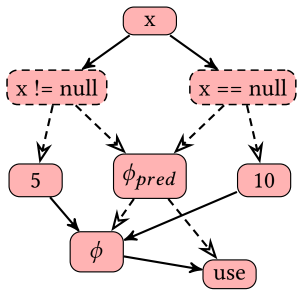


Figure 5. An example PVPG with ϕ flows.

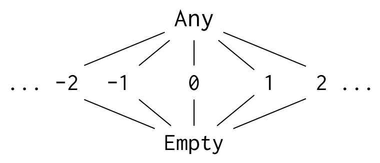


Figure 6. The lattice $\mathbb{P}$ of primitive values.

Method Invocations as Predicates. If we can prove that a method never returns, we can conclude that all the statements following the method invocation are unreachable. This happens, e.g., when a method contains an infinite loop, or when an exception is always thrown, e.g., `Assert.fail()` and similar methods. In our analysis, every method invocation is a predicate for the following statements in the block. A method with a `void` return type still returns the predicate of the ***return*** instruction as an artificial value signaling whether the ***return*** is reachable.

Joining Values using ϕ Flows. When multiple branches join in the control flow graph, it is necessary to join the incoming values in the PVPG. Consider Figure 5. The value of `y` depends on which branch was taken. In SSA form, a $y_3 \leftarrow \phi(y_1, y_2)$ instruction expresses that explicitly. In a PVPG, a ϕ flow (depicted as ϕ) is introduced to join the two definitions of `y`.

Each of the definitions of y is guarded by their corresponding predicate, the $x \neq$ `null` flow in case of $y = 5$ and the $x ==$ `null` flow in case of $y = 10$. These predicates need to be joined too, producing a predicate flow, denoted as ϕ_{pred}, for the joined value. Intuitively, the code after a control flow join is executable iff the end of any of its predecessors is. For each control flow join, a ϕ_{pred} flow is introduced, and connected via predicate edges with the latest predicates from the predecessor branches. This ϕ_{pred} becomes the new predicate in the following block, and is also the predicate of all the ϕ flows generated for joining values (see the *pred* edge going from the ϕ_{pred} to ϕ).

Abstractions for Primitive Values. As shown in the motivating example, the analysis needs to track some primitive values such as boolean constants. But to scale the analysis to hundreds of thousands of methods analyzed within a few minutes, it is necessary to provide sufficient abstractions. Therefore, we use the simple lattice $\mathbb{P}$ depicted in Figure 6 to model primitive values. We do not attempt to model intervals or sets, only concrete values, `Empty`, and `Any`. The join of two different constant values results immediately in `Any`. The boolean constants `true` and `false` are modelled as constants 1 and 0, respectively. Similarly, we do not explicitly model arithmetic operations in the PVPG. Instead, a flow is inserted into the graph that always produces the value `Any`.

Abstractions for Objects. SkipFlow is designed to work with any lattice-based representation for objects as long as type checks, null checks, and virtual method resolution can be applied on the elements of the lattice. Null references are handled as a special type that can be part of any value state. In the implementation of SkipFlow evaluated in this paper, we have chosen to represent objects by their types only. This *type abstraction* was shown useful by prior work when embedding the analysis in a compiler [60]. On top of that, the type abstraction (or a similar mechanism) is in some cases even necessary to preserve soundness for Java: For example, allocation-site information is not available when interfacing with C code via JNI where new Java objects can be created in C or passed back and forth between Java and C, or object fields can be queried in C and the results then passed back into Java.

However, we argue that our technique is applicable to both type-based and object-based points-to analysis. The decision to use the type abstraction is an implementation detail as the same analysis can be executed using a subset lattice based on allocation sites, possibly even extended with allocation context for context-sensitive analysis.

4 Predicated Value Propagation Graphs

In this section, we describe the structure of PVPGs and the rules for value propagation over them. For space reasons, we focus on a high-level description. The full formalism can be found in Appendices B and C. The input to the analysis is a Java-like managed base language in static single assignment form.

Structure of a PVPG. A PVPG models the flow of both primitive values and objects interprocedurally. Vertices in a PVPG are called **flows** to clearly distinguish them from the **instructions** in the base language. The flows represent values of method parameters, variables, and fields read or written by a particular instruction (each instruction reading or writing to a variable produces a fresh flow); method calls, which also represent the returned value in the caller; and values to be returned from a method call back to the caller. Further, flows represent values of various conditions present in the code (including their negated and inverted versions as explained later); values resulting from joining the values of other flows (results of ϕ instructions); ϕ_{pred} flows for joining predicates; and, finally, the always enabled predicate $pred^{on}$. Each flow keeps a reference to the underlying base language element it represents. The analysis computes the *value state* of each flow representing a conservative overapproximation of the values that can be assigned to the given flow during the runtime of the program.

Flows can be connected via three types of **edges** described below.

A *use* edge represents a *def-use* dependency between flows. If there is a *use* edge between flows `s` and `t`, denoted by $\texttt{s} \rightsquigarrow_{use} \texttt{t}$, the *value state* of `t` has to be at least as big as the *value state* of `s` in terms of the underlying lattice provided that s is executable and no filtering is applied (the semantics of filtering is explained later).

A *predicate* edge represents a control-flow dependency. A *predicate* edge between flows `s` and `t`, denoted by $\texttt{s} \rightsquigarrow_{pred} \texttt{t}$, defines that if `s` is executable and has a non-empty *value state*, `t` is also executable. Every flow is the target of an incoming *predicate* edge, apart from $pred^{on}$, which is always enabled and therefore does not need an incoming edge. A ϕ_{pred} flow has multiple incoming *predicate* edges. In that case, it suffices that any of its predicates has a non-empty state to enable the execution of the target flow.

An *observe* edge represents an additional flow-specific dependency. An *observe* edge between flows `s` and `t`, denoted as $\texttt{s} \rightsquigarrow_{obs} \texttt{t}$, defines that `t` has access to the *value state* of `s`, and `t` is notified when the state changes to perform some flow-specific task such as a field update or method resolution and linking (discussed more below).

Creating a PVPG. This paragraph describes how the PVPG can be created for a given method. A formalization of the algorithm is presented in Appendix B. Basic blocks of the method are processed in reverse postorder, and the instructions within each basic block are processed sequentially top to bottom. During the traversal, flows are created for the encountered instructions and memory locations.

The traversal maintains a state for each basic block consisting of: (1) A mapping from variables to previously created flows, which is used to connect flows with their dependencies, e.g. to establish a *use* edge between the flow representing `x` and the subsequently created filtering flow `x != 0`, and (2) a reference denoted as *pred* that is continuously updated to refer to the last encountered predicate. The predicate referenced by *pred* is used to establish predicate edges. At the beginning of a method, when no suitable flow is available yet, the special $pred^{on}$ flow is used, which is always enabled, i.e. the initial flows inside methods are always enabled.

When processing a branching instruction, the branching condition is handled separately for the `then` and `else` branch. For the `then` branch, the condition is used as is; for the `else` branch, it has to be negated, e.g. `x < 10` becomes `x >= 10`. These conditions are used to filter the values of their tested

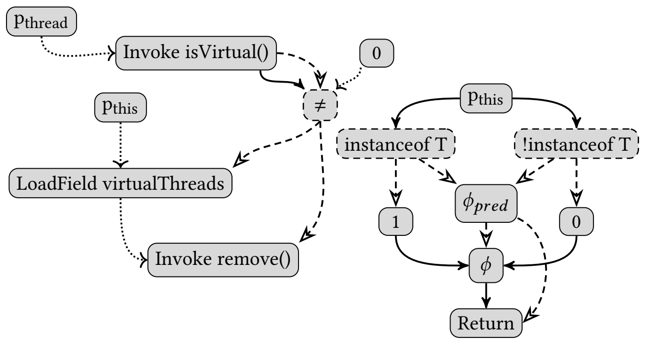


Figure 7. The PVPG for the methods `onExit` (on the left) and `isVirtual` (on the right) from the JDK motivating example in Figure 2. Full lines represent *use* edges, dashed lines represent *predicate* edges, and dotted lines represent *observe* edges. In the instanceof check, `T` stands for `BaseVirtualThread`. Note that the two p_{this} flows each represent the implicit `this` parameter in their corresponding method.

variables within the scope of the successor branches, which is done using the filtering flows described in Section 3. Inside each branch, the following happens. If the condition tests a single variable only, for example a *nullcheck* or a *typecheck*, a single filtering flow is created. If the condition is a comparison of two variables, e.g. `x < y`, two flows are created, each representing one variable[1]. The filtering flows serve as new definitions for the variables they test. In our example, we have a new flow for `x` whose value is less than `y`, and a new flow for `y`, whose value is greater than `x`.

Running Example. Figure 7 shows the PVPG for the methods `onExit` (on the left) and `isVirtual` (on the right) from the JDK motivating example in Figure 2. To make the graph more compact, we omit the always-enabled predicate $pred^{on}$. A flow that does not have any incoming *predicate* edge in the graph is predicated by $pred^{on}$. The condition that invokes `isVirtual()` and that guards the `then` branch containing the Invoke `remove()` leads to the filtering flow denoted as '$\neq$'. Recall that boolean values are represented as integers, thus the condition becomes `if (thread.isVirtual() != 0)`, which is encoded explicitly in the graph. The flow representing the Invoke `isVirtual()` is the predicate of '$\neq$' (Indeed, the method must be first invoked and only then the test may be executed), which in turn is the predicate of the flow representing the load field `virtualThreads` and the flow representing the `remove()` invocation.

Furthermore, note that p_{thread} is connected via *observe* edge with the Invoke `isVirtual()`, because the invocation may need to link new call targets every time a new type appears in the *value state* of p_{thread}. The same applies to the

[1]Note that, here, we assume a base language with conditions involving at most two variables as compound expressions can be broken into chains of simpler ones without loss of generality.

chain of *observe* edges from p_{this} through the load field of `virtualThreads` to the Invoke `remove()`. Every time a new type appears in the *value state* of p_{this}, a new field may be connected with the load field instruction, which in turn may add more types to the *value state* of the load field, possibly triggering a new method linking for the Invoke `remove()`.

The filtering flow '≠' is connected via a *use* edge with the Invoke `isVirtual()`, whose values it filters and propagates. Further, it also has an *observe* edge from the constant zero, because it needs to access its *value state* to perform the filtering (in this case, the value is constant, but it does not have to be in general).

In the `isVirtual()` method (on the right), the type check leads to two filtering flows, each of which is the predicate for their corresponding returned values.

Value propagation through PVPGs. This section describes the rules for value propagation in a PVPG. A formalization is given in Appendix C. Each flow maintains a *value state* representing the set of values that can be assigned to the corresponding base language element at runtime. Primitive values are modelled using the lattice $\mathbb{P}$ discussed in Section 3. Objects are modelled using a subset lattice over types. Following the work of Wimmer et al. [60], we do not distinguish individual allocation sites to improve scalability.

Each flow, except from $pred^{on}$, is initially disabled, not propagating any values. At the beginning of the analysis, $pred^{on}$ enables all the flows to which it is directly connected via a *predicate* edge. Once a flow becomes enabled, it propagates its own *value state* along the *use* edges. If a flow has multiple incoming *use* edges, its *value state* is the join of all the incoming values. Additionally, filtering flows filter their incoming values based on their conditions, e.g. a type check allows only subtypes to flow through it, $x < 10$ allows only values smaller than 10 to flow on, etc. Once a *value state* of any enabled flow becomes non-empty, it enables all the flows to which it is connected via *predicate* edges, possibly triggering more value propagation.

Method invocations are handled by *linking*, i.e. creating *use* edges from the arguments in the caller to the formal parameters of the callee, and from the return in the callee back to the invoke flow in the caller (the invoke flow also represents the returned value). For virtual invokes, the linking is done for every possible target method obtained by inspecting the *value state* r of the receiver (with which the invoke is connected via an *observe* edge) and performing *method resolution*[2] for each type $t \in r$.

Since the elements used in *value states* form a lattice with a finite height and all our filtering and join operators are monotone, the analysis is guaranteed to eventually reach a fixed point and terminate.

[2]The virtual method resolution is done as defined by the JVM specification [31].

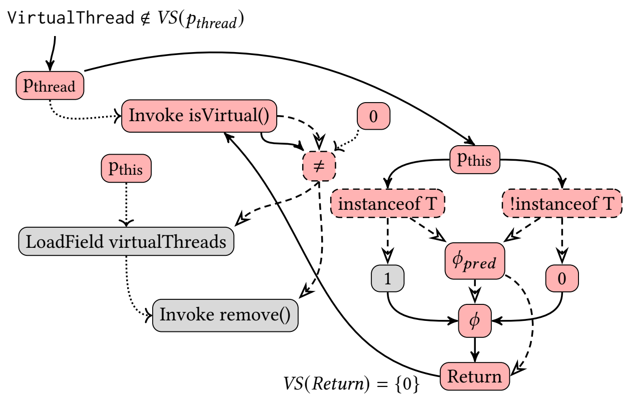


Figure 8. The PVPG for the methods `onExit` (on the left) and `isVirtual` (on the right) from the JDK motivating example in Figure 2 after the analysis reaches a fixed-point. Full lines represent *use* edges, dashed lines represent *predicate* edges, and dotted lines represent *observe* edges. Enabled flows are coloured in red, and disabled flows in grey. In the instanceof check, T stands for `BaseVirtualThread`. The method `onExit` is linked from some caller, but its p_{thread} never receives any VirtualThread (*VS* stands for *value state*). Note that the two p_{this} flows each represent the implicit `this` parameter in their corresponding method.

Running Example. Figure 8 shows the state of the PVPG for the JDK motivating example after the analysis reaches the fixed point. The following steps produce this state: First, the method `onExit()` is linked from some already reachable method `m`. Then, the PVPG of `onExit()` is created and its p_{thread} is linked with the flow in `m` representing the corresponding argument. The flows p_{thread}, Invoke `isVirtual()`, the flow representing the constant 0, and p_{this} from the `onExit()` method get immediately enabled because they are guarded by $pred^{on}$.

Once the *value state* of p_{thread} is non-empty, the Invoke `isVirtual()` is notified via its *observe* edge, which leads to a creation of the PVPG for the `isVirtual()` method and to linking the callee and the caller. During the linking, p_{thread} in `onExit()` is linked with p_{this} in `isVirtual()`, and the `Return` from `isVirtual()` is linked with the Invoke `isVirtual()` in `onExit()`, since the *value state* of the flow representing the invocation represents the returned value in the caller. So far, the *value state* of the `Return` in `isVirtual()` is still empty. In `isVirtual()`, the only enabled flows after the creation are p_{this} and both filtering flows representing the type check. Assume that no virtual thread is created in the analyzed application, so the *value state* of p_{thread} only contains non-virtual thread types. Consequently, only the filtering flow for the `else` branch in `isVirtual()` receives a non-empty *value state* after filtering, which enables the flow holding the constant 0. This value is propagated back into `onExit()` through the ϕ and the `Return` flows.

The *value state* of the Invoke `isVirtual()` is now non-empty and enables the filtering flow '$\neq$'. However, the *value state* of the Invoke `isVirtual()` contains zero only, which is filtered out by $\neq$ with another zero, so the *value state* of '$\neq$' remains empty, never enabling the rest of the flows in `onExit`. Therefore, the Invoke `remove()` is not enabled, and consequently, the `remove()` method is not processed.

5 Implementation Details

This section presents various details that are more tied to our specific implementation than the general algorithm presented above. We also discuss how we handle dynamic parts of Java including Reflection and JNI.

Handling Exceptions. While it is possible to track the propagation of exceptions from callees back into callers, we have not observed enough precision improvement in our experiments to justify the overhead. Therefore, to improve the scalability of the analysis, we assume that any instantiated exception can flow out of any exception handler with matching types.

Exception handling provides another use case for *method invokes as predicates.* Certain methods may never return and instead always throw exceptions. Such a behaviour can either be intentional, e.g., `Assert.fail()` and similar methods, or accidental, for example when the arguments passed into a method are invalid, e.g., `null` is passed as an argument that should be `non-null`. In such cases, SkipFlow can prove that the code after the method invocation is unreachable.

Boolean Values. Consistent with the JVM Specification [31], we model `boolean` values `true` and `false` as 1 and 0, respectively. Consequently, the type `boolean` does not exist from the point of view of the analysis. Recall that predicate edges are always triggered when the *value state* becomes non-empty, which is different from the *value state* becoming 0, i.e. `false`. When a *value state* becomes 0, it is non-empty, so the value propagation is triggered. Figure 7 of the running example showed how conditions are converted to comparisons with the constants 1 and 0.

Reflection, JNI, Unsafe. The Java Virtual Machine Specification [31] contains many dynamic features that are hard to analyse statically [24], for example, the Reflection API, JNI, and `Unsafe`. While soundness is in practice sometimes replaced with soundiness [33], our analysis is meant to be embedded in a compiler, thus we have to handle all of these dynamic features soundly. Otherwise, incorrect code would be generated. On the other hand, allowing any method to be called in a way opaque to the analysis defeats the purpose of the analysis, as everything would be reachable.

Therefore, we require a configuration file specifying which methods and fields are accessed via Reflection or JNI. We inherit this design decision from GraalVM Native Image [59], upon which the prototype of SkipFlow is implemented. One might argue that such an approach could lead to unsound results if incomplete configuration is provided. However, we believe that this approach is still sound with respect to the semantics of GraalVM Native Image, which relies on these configuration files anyway. We do not claim any contributions in the area of reflection handling, and SkipFlow could be easily adjusted to use a different policy.

To help creating the configuration, we provide a tool similar to [8]: a tracing agent that monitors the application at runtime. For methods designated as invoked via Reflection or JNI, we mark them as *root methods* and assume that the *value states* of their parameters can contain any instantiated subtype of their declared type. Similarly, the *value state* of any field accessed via Reflection or JNI can contain any instantiated subtype of its declared type.

For `Unsafe`, we conservatively assume that any input of any unsafe write or normal write into an unsafe-accessed field can flow out of any unsafe load.

6 Evaluation

We have implemented our approach in the Native Image [59] 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 base our branch on a recent commit in `master`[3]. In the experiments, which were done on top of Oracle GraalVM, we compare our analysis (depicted as SkipFlow), with the points-to analysis (depicted as PTA) used by default in Native Image (type-based flow-insensitive context-insensitive analysis) [60]. That is the configuration shown in [60] to be the most suitable when the analysis results are used for compilation, i.e., when a fully sound analysis is needed.

A comparison between PTA and other commonly used call graph construction algorithms, namely, Rapid Type Analysis (RTA) [5] and Variable Type Analysis (VTA) [54], was already presented in [60] (their *context-insensitive without saturation* configuration is essentially a VTA). The remaining typical call graph construction algorithm, Class Hierarchy Analysis (CHA) [14], is expected to be even less precise than RTA. Since the precision of RTA is already too low, a CHA is not implemented in GraalVM Native Image at the moment.

All the experiments were executed with Java 24[4]. We run all the benchmarks 10 times and report the average of the runs. The benchmarks 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

[3]Commit id: $d82a349401e5cfeb7d4523dd16810bfaa28b60b7$.

[4]We chose to present the data using a cutting edge release, but we have also experimented with Java 21, producing similar results.

was pinned to one of the two CPUs, and TurboBoost was disabled to avoid instability. The number of threads used by the 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 selected an old configuration similar to a current developer laptop.

We use three benchmark suites: DaCapo 9.12 [7], Renaissance 0.15.0 [41], and *microservices*. *DaCapo* is a benchmark suite that consists of client-side Java benchmarks, trying to exercise the complex interactions between the architecture, compiler, virtual machine and running application. *Renaissance* is a benchmark suite that consists of real-world, concurrent, and object-oriented workloads that exercise various concurrency primitives of the JVM. In both *Renaissance* and *DaCapo*, we use a subset of the benchmark suite because some benchmarks are not compatible with the Native Image Ahead-of-Time compilation. However, we decided to include the benchmarks *als*, *chi-square*, *dec-tree*, and *log-regression* from *Renaissance*, which are not yet fully supported in Native Image. In particular, their analysis already finishes successfully, but the compiled application does not run because class definitions at runtime are used, which are not yet fully supported.

Microservices is our own set of microservice applications. We use the three most frequently used modern frameworks for Java web services in our evaluation: Spring, Micronaut, and Quarkus. In order not to be biased towards a single framework, we selected multiple representative applications for each of them. All the applications perform different functions. Under no circumstances can our evaluation be seen as a comparison and ranking of the frameworks against each other.

In particular, the microservices benchmark suite consists of the following applications:

- *Micronaut Helloworld*: A helloworld application written with the Micronaut framework.
- *Micronaut MuShop Order, Payment, and User* [39]: Three microservices of a large demo application using the Micronaut framework.
- *Quarkus Helloworld*: A helloworld application written with the Quarkus framework.
- *Quarkus Registry* [42]: A large real-world application using the Quarkus framework. It is used to host the Quarkus extension registry.
- *Quarkus Tika*: A demo application written with the Quarkus framework.
- *Spring Helloworld*: A helloworld application written using the Spring framework.
- *Spring PetClinic* [56]: A large demo application for the Spring framework.

For each benchmark, we collect the following metrics:

- *Analysis Time*: The time it takes to run the analysis, measured in milliseconds.
- *Reachable Methods*: The number of methods marked reachable by the analysis.
- *Counter Metrics*: We count specific instructions in all reachable methods that cannot be removed or simplified using the results of the analysis. In particular, we count three types of branching instructions: *Type Checks*, *Null Checks*, *Primitive Checks*, and the number of virtual method calls that could not be devirtualized, denoted as *PolyCalls*.
- *Total Time*: The time it takes to run the whole Native Image compilation, measured in milliseconds.
- *Binary Size*: The size of the resulting binary file, measured in MB.

In the discussions below, we focus mainly on the first four metrics, which we consider *analysis-oriented*. The *total time* and *binary size* are added to demonstrate the effect on the whole Native Image compilation.

Detailed Results. Table 1 presents the results for all the benchmarks. Note that due to the rounding to seconds, the analysis time in the table sometimes looks identical, but the actual values in terms of milliseconds are still slightly different. This can be observed for example for the *DaCapo luindex* benchmark.

The first block in the table presents the *DaCapo* benchmarks. The most interesting example in this category is the *Sunflow* benchmark, in which the size of the application is reduced by 52.3%. The reason for such a radical reduction is that fact that our analysis successfully removed the `AWT` and `Swing` GUI libraries, as explained in the motivating example in Section 2. The least impacted benchmark is `lusearch`, reduced by 3.5%. `lusearch` is also the smallest benchmark in this category, thus offering fewer program points where the precision can be improved. On average, the number of reachable methods is reduced by 13.3%. The counter metrics follow a similar trend.

The most prominent case for *Microservices* is `Quarkus Tika`, whose size is reduced by 9.2%, followed by the biggest benchmark in this category, `Spring Petclinic`, which is reduced by 8.1%. The least impacted benchmark is `Micronaut Helloworld`, reduced by 3.3%, again being at the same time the smallest benchmark in this category. On average, the number of reachable methods is reduced by 6.3%.

For Renaissance, the biggest reduction is achieved on the *chi-square* benchmark, which is reduced by 17.2%. The other Spark benchmarks *als*, *dec-tree*, and *log-regression* are reduced by at least 15%. The least impacted benchmark is `reactors`, reduced by 3.7%, again being also one of the smallest in this category. In general, the whole suite is reduced by 8.4%.

Overall, we can observe that bigger applications generally offer more potential for optimizations, because they have more libraries out of which only subsets are used and thus the remainder can be removed. Nevertheless, there are some

Table 1. Results for all bench suites. For all metrics, lower is better. The best result in *reachable methods* for each bench suite is in yellow, the worst is in grey. Even for the grey rows, SkipFlow still improves over the baseline in all metrics apart from *analysis time*.

Bench Suite	Benchmark	Conguration	Analysis Time [s]		Total Time [s]		Reach. Methods		Type Checks		Null Checks		Prim Checks		Poly Calls		Binary Size [MB]	
DaCapo	fop	PTA	27		179		96.1k		39.2k		129.4k		137.8k		31.4k		136	
		SkipFlow	28	+1.3%	171	-4.6%	89.3k	-7.1%	35.9k	-8.4%	117.8k	-8.9%	125.4k	-9.0%	28.9k	-8.0%	129	-5.2%
	h2	PTA	15		96		43.3k		21.7k		49.3k		59.5k		10.3k		65	
		SkipFlow	15	0.0%	93	-3.2%	40.0k	-7.6%	19.1k	-11.8%	44.3k	-10.2%	54.6k	-8.3%	9.4k	-9.2%	61	-5.8%
	jython	PTA	24		373		74.9k		26.7k		91.2k		77.5k		33.2k		253	
		SkipFlow	22	-7.1%	365	-2.2%	70.5k	-6.0%	24.4k	-8.7%	83.4k	-8.5%	66.6k	-14.1%	31.9k	-3.8%	249	-1.6%
	luindex	PTA	8		57		31.2k		9.4k		29.7k		45.2k		6.0k		44	
		SkipFlow	8	+5.3%	56	-1.5%	30.0k	-3.9%	9.0k	-4.3%	28.1k	-5.4%	43.3k	-4.1%	5.7k	-5.8%	43	-2.6%
	lusearch	PTA	11		73		29.2k		9.0k		25.9k		42.6k		5.1k		40	
		SkipFlow	12	+4.1%	72	-1.2%	28.2k	-3.5%	8.7k	-3.9%	24.8k	-4.2%	41.0k	-3.7%	4.9k	-4.7%	39	-1.6%
	pmd	PTA	20		128		64.0k		33.6k		85.3k		95.3k		18.6k		91	
		SkipFlow	20	-0.4%	121	-4.9%	58.1k	-9.3%	29.6k	-11.8%	74.1k	-13.2%	84.1k	-11.7%	16.3k	-12.4%	85	-7.1%
	sunflow	PTA	19		124		56.7k		20.3k		72.0k		92.2k		13.4k		65	
		SkipFlow	12	-35.4%	79	-36.0%	27.1k	-52.3%	8.3k	-58.9%	25.4k	-64.7%	44.4k	-51.9%	4.4k	-67.3%	32	-50.5%
	xalan	PTA	16		109		49.0k		22.9k		64.1k		76.4k		14.9k		63	
		SkipFlow	16	-0.5%	99	-8.8%	40.6k	-17.0%	18.8k	-17.8%	50.5k	-21.2%	63.3k	-17.2%	11.3k	-23.9%	55	-13.2%
Microservices	Micronaut Helloworld	PTA	21		131		76.0k		33.7k		76.5k		79.7k		22.9k		73	
		SkipFlow	21	+2.2%	127	-3.6%	73.5k	-3.3%	32.2k	-4.5%	73.2k	-4.3%	76.2k	-4.4%	21.8k	-4.7%	70	-3.1%
	Micronaut MuShop Order	PTA	38		236		167.0k		79.1k		168.0k		139.7k		51.3k		144	
		SkipFlow	38	+0.2%	225	-4.8%	154.9k	-7.3%	73.4k	-7.3%	153.5k	-8.6%	127.2k	-8.9%	46.8k	-8.8%	136	-5.5%
	Micronaut MuShop Payment	PTA	15		96		83.0k		35.6k		78.1k		76.0k		27.0k		73	
		SkipFlow	15	+2.4%	94	-2.0%	79.5k	-4.2%	34.0k	-4.7%	73.9k	-5.4%	72.2k	-5.0%	25.6k	-5.4%	70	-3.7%
	Micronaut MuShop User	PTA	27		165		113.0k		53.6k		121.5k		111.8k		38.1k		106	
		SkipFlow	27	+0.8%	159	-3.5%	105.4k	-6.7%	49.7k	-7.2%	111.6k	-8.2%	102.7k	-8.1%	35.1k	-7.7%	100	-5.6%
	Quarkus Helloworld	PTA	18		107		59.6k		25.0k		55.1k		64.7k		15.6k		54	
		SkipFlow	18	+2.3%	105	-1.2%	56.0k	-6.0%	23.4k	-6.4%	51.5k	-6.6%	59.9k	-7.4%	14.6k	-6.9%	52	-4.0%
	Quarkus Registry	PTA	29		182		134.2k		63.4k		133.0k		107.8k		46.4k		118	
		SkipFlow	24	-18.6%	174	-4.3%	125.1k	-6.8%	59.2k	-6.7%	122.4k	-8.0%	100.2k	-7.0%	40.6k	-12.5%	112	-4.8%
	Quarkus Tika	PTA	30		181		109.1k		42.2k		111.1k		125.7k		30.3k		123	
		SkipFlow	30	-0.8%	177	-2.5%	99.1k	-9.2%	37.9k	-10.3%	99.2k	-10.8%	115.1k	-8.5%	25.4k	-16.3%	118	-4.5%
	Spring Helloworld	PTA	23		143		85.2k		39.5k		99.1k		95.0k		22.5k		81	
		SkipFlow	23	-0.7%	138	-3.3%	80.4k	-5.6%	36.7k	-7.0%	91.7k	-7.4%	89.7k	-5.6%	20.0k	-10.9%	77	-4.7%
	Spring Petclinic	PTA	44		284		210.2k		112.3k		240.1k		187.7k		73.5k		202	
		SkipFlow	44	+0.7%	273	-4.0%	193.3k	-8.1%	105.7k	-5.9%	221.5k	-7.7%	173.6k	-7.5%	66.2k	-9.9%	191	-5.3%
Renaissance	akka-uct	PTA	12		79		38.8k		10.2k		27.4k		34.7k		6.9k		31	
		SkipFlow	12	-1.1%	76	-3.4%	36.3k	-6.4%	9.0k	-12.0%	24.8k	-9.5%	31.9k	-8.0%	6.5k	-5.9%	30	-5.4%
	als	PTA	83		475		381.6k		143.4k		317.2k		220.6k		95.2k		313	
		SkipFlow	86	+3.0%	431	-9.3%	321.1k	-15.8%	121.2k	-15.4%	261.2k	-17.7%	177.6k	-19.5%	76.3k	-19.9%	269	-13.8%
	chi-square	PTA	43		286		217.8k		76.8k		192.4k		164.2k		50.0k		179	
		SkipFlow	40	-8.2%	253	-11.5%	180.3k	-17.2%	61.8k	-19.5%	156.6k	-18.6%	135.3k	-17.6%	39.1k	-21.9%	151	-16.0%
	dec-tree	PTA	86		492		385.4k		147.0k		320.4k		221.1k		96.9k		316	
		SkipFlow	90	+5.2%	446	-9.5%	324.9k	-15.7%	124.6k	-15.2%	264.3k	-17.5%	178.3k	-19.3%	77.7k	-19.7%	272	-13.8%
	finagle-chirper	PTA	22		138		94.9k		31.3k		76.0k		74.4k		24.8k		79	
		SkipFlow	20	-7.8%	127	-8.1%	82.8k	-12.7%	23.8k	-23.8%	60.3k	-20.7%	60.5k	-18.7%	19.9k	-19.7%	67	-14.2%
	finagle-http	PTA	22		138		93.9k		31.2k		76.3k		74.9k		24.7k		77	
		SkipFlow	21	-7.1%	128	-7.6%	81.9k	-12.8%	23.7k	-24.0%	60.6k	-20.7%	61.0k	-18.6%	19.8k	-19.9%	66	-15.0%
	fj-kmeans	PTA	11		64		28.0k		7.1k		19.6k		29.7k		4.5k		24	
		SkipFlow	10	-1.8%	62	-2.1%	26.4k	-5.5%	6.6k	-5.8%	18.3k	-6.5%	27.9k	-6.0%	4.2k	-8.0%	23	-3.7%
	future-genetic	PTA	10		64		28.8k		7.2k		20.1k		30.0k		4.7k		24	
		SkipFlow	10	0.0%	62	-2.5%	27.2k	-5.6%	6.8k	-6.1%	18.8k	-6.5%	28.2k	-6.1%	4.3k	-8.1%	24	-3.8%
	log-regression	PTA	90		516		394.7k		147.4k		327.5k		233.0k		99.7k		324	
		SkipFlow	87	-4.2%	457	-11.4%	334.2k	-15.3%	125.2k	-15.0%	271.5k	-17.1%	190.0k	-18.4%	80.8k	-18.9%	280	-13.5%
	mnemonics	PTA	10		63		28.2k		7.2k		20.0k		29.8k		4.7k		24	
		SkipFlow	10	+1.1%	63	+1.2%	26.6k	-5.5%	6.8k	-5.7%	18.7k	-6.3%	28.1k	-5.9%	4.3k	-7.6%	23	-3.6%
	par-mnemonics	PTA	10		63		28.2k		7.3k		20.0k		29.8k		4.7k		24	
		SkipFlow	10	+0.4%	64	+1.7%	26.7k	-5.5%	6.9k	-5.8%	18.7k	-6.3%	28.1k	-5.9%	4.3k	-7.6%	23	-3.6%
	philosophers	PTA	7		48		30.9k		8.0k		21.1k		29.2k		4.9k		25	
		SkipFlow	8	+2.4%	51	+6.2%	29.6k	-4.1%	7.7k	-4.4%	19.9k	-5.9%	27.5k	-6.1%	4.6k	-6.0%	25	-2.6%
	reactors	PTA	11		64		31.4k		7.8k		20.9k		29.0k		5.2k		26	
		SkipFlow	11	+3.1%	62	-2.2%	30.2k	-3.7%	7.5k	-4.6%	19.7k	-5.7%	27.3k	-5.9%	4.9k	-5.7%	25	-2.8%
	rx-scrabble	PTA	10		63		29.0k		7.3k		20.3k		30.2k		4.8k		33	
		SkipFlow	10	-1.0%	61	-3.5%	27.5k	-5.2%	6.9k	-5.7%	19.1k	-6.0%	28.4k	-5.9%	4.4k	-7.9%	32	-3.0%
	scala-doku	PTA	10		64		29.0k		7.4k		20.1k		30.0k		4.6k		25	
		SkipFlow	11	+2.5%	63	-1.7%	27.5k	-5.5%	7.0k	-5.7%	18.8k	-6.6%	28.2k	-5.9%	4.3k	-7.4%	24	-5.2%
	scala-kmeans	PTA	10		62		27.9k		7.0k		19.6k		29.6k		4.5k		24	
		SkipFlow	10	+1.0%	60	-2.5%	26.4k	-5.5%	6.6k	-5.8%	18.3k	-6.5%	27.8k	-6.0%	4.1k	-8.1%	23	-3.8%
	scala-stm-bench7	PTA	11		66		32.8k		8.4k		22.5k		30.0k		5.5k		27	
		SkipFlow	11	+2.7%	67	+1.6%	31.4k	-4.0%	8.0k	-4.4%	21.2k	-5.8%	28.1k	-6.1%	5.2k	-5.4%	26	-4.6%
	scrabble	PTA	10		63		28.3k		7.3k		20.0k		29.8k		4.7k		33	
		SkipFlow	10	-1.7%	64	+0.7%	26.7k	-5.5%	6.9k	-5.8%	18.7k	-6.4%	28.1k	-5.9%	4.3k	-7.6%	32	-2.6%

significant outliers, for example, the *DaCapo Sunflow* benchmark, suggesting that our analysis works especially well if specific code patterns are used, such as a method call within a provably unreachable branch or a not needed default value for an optional argument presented in Section 2.

In Figure 9, we present all the metrics normalized to the baseline points-to analysis to provide a quick overview complementing the detailed Table 1. Using the charts, we can quickly notice the already discussed *DaCapo Sunflow* outlier and also the general trend of 9% reduction in *reachable methods* across all the benchmarks.

Impact on Analysis Time. Interestingly, we can also observe that the analysis time does not increase for most benchmarks. On the contrary, the average analysis time is even reduced by 1.6%, suggesting we can actually *have a free lunch* and use a more precise analysis without negatively impacting the analysis time. Indeed, even though our analysis is more complex compared to the baseline points-to analysis, it is also more precise, leading to fewer methods marked as reachable. Fewer reachable methods mean less work for the analysis, but also for the compilation that follows, which can be seen on the *total time* metric, being reduced by 4.4% on average.

There is a tipping point after which the cost of a more precise analysis outweighs the reduction due to fewer reachable methods. We believe our analysis is a good spot in the design space, providing enough precision improvement to make a difference without impacting the analysis time.

Impact on Compiler Optimizations. There are numerous optimizations that can benefit from the facts proved by SkipFlow, including but not limited to: (1) *Dead code elimination* – flows that remain disabled until the end of the analysis correspond to instructions that can never be executed, so we can safely remove them. If a filtering flow has an empty value state after the analysis, the entire branch for which the given flow was generated is proved unreachable and can be removed; (2) *Intraprocedural constant folding* – if a method parameter is proved to be a constant value, it allows more intraprocedural constant folding after the analysis; (3) *Method inlining* – Due to the dead code elimination and constant folding, methods become smaller, making them more amendable to inlining, which can unlock further optimizations.

Our current evaluation focuses mainly on reducing the set of reachable methods. The *counter metrics* show that SkipFlow also reduces the number of type checks, null checks, primitive checks, and virtual invokes in the methods that remain reachable, but more work is needed to evaluate the impact of SkipFlow on runtime performance.

Impact on Binary Size. The reduction of reachable methods propagates to the binary size of the resulting applications. We can observe a significant reduction in this metric as well, more precisely max 5.6%, min 3.1%, avg 4.6% for microservices, max 16.0%, min 2.6%, avg 7.3% for Renaissance, max 50.5%, min 1.6% avg 11.0% for DaCapo, and 7.4% on average across all the benchmarks. In general, the binary size reduction follows a similar trend to the reduction in reachable methods.

Discussion. Overall, we conclude that while SkipFlow can also remove more type checks, null checks, primitive checks, and devirtualize more virtual calls, the biggest benefit compared to the baseline is *reducing the total number of reachable methods without increasing the overhead in terms of analysis time*, even reducing the analysis time in many cases. Reducing the number of reachable methods not only speeds up the analysis itself, but it also reduces the workload of the compilation phase that follows the analysis, which can be observed on the *total time* being 4.4% smaller on average. On top of that, fewer methods also mean less code to include in the compiled binary, thus reducing the binary size by 7.4% on average. We believe these properties make our analysis a suitable extension for a points-to analysis used in optimizing compilers.

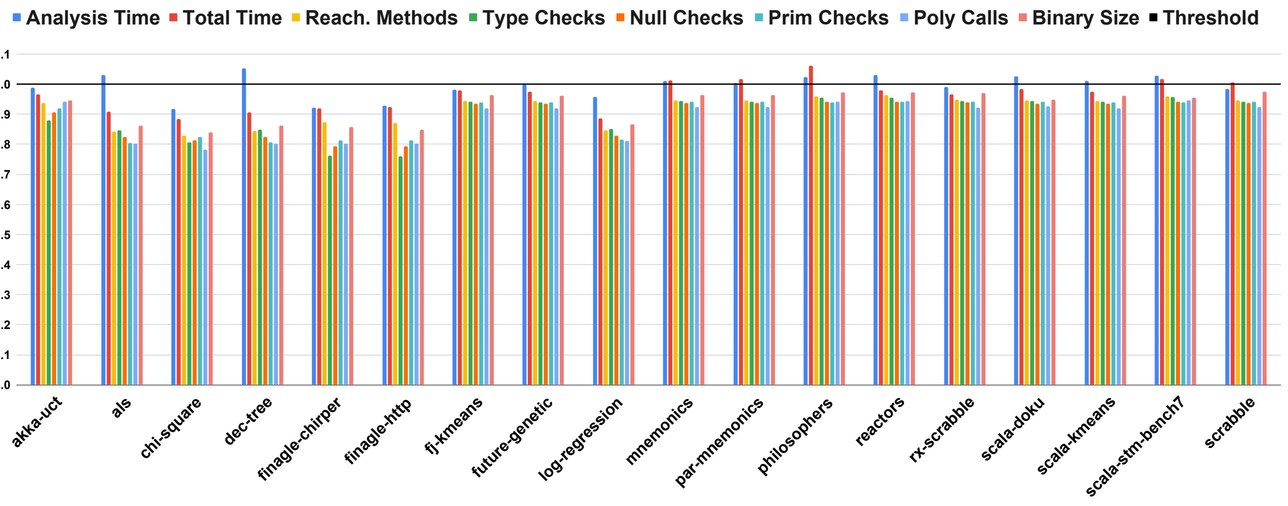
The increased precision of SkipFlow can also improve the results of any subsequent client interprocedural static analysis, such as a taint analysis, data-flow analysis or abstract interpretation, as removing spurious edges from the call graph can lead to fewer false alarms.

7 Related Work

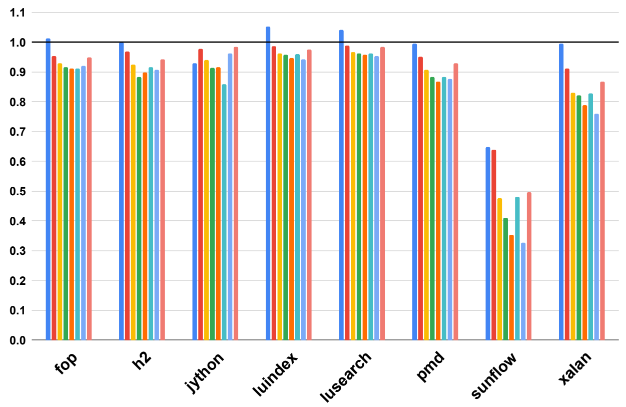
Points-to Analysis. Points-to analysis (PTA) has been applied in many areas including compilers [59], security analysis [4, 35], bug hunting [17, 32], heap allocation analysis [51], escape analysis [37], call graph construction [1, 43], and program understanding [16, 30]. Typically, PTA is implemented using inclusion-based constraint solving [2, 6], type system based unification [52], or via reduction to a graph reachability problem [28].

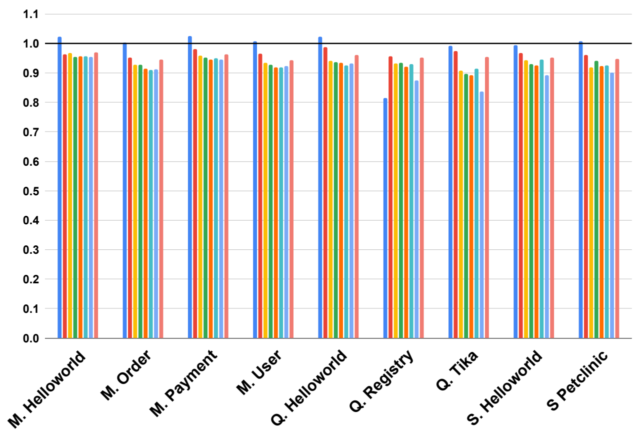
The frameworks implementing PTA essentially fall into three different categories: 1) *imperative*, e.g., Spark [26], WALA [21], and Qilin [18] (written purely in Java), 2) *declarative*, e.g., DOOP [9] (written in Datalog), 3) *hybrid*, e.g., Paddle [27] (with a declarative core in Datalog and the rest of the infrastructure in Java). SkipFlow is implemented on top of an existing imperative points-to analysis framework, is parallel, utilizes inclusion based-solving, and its formal presentation is inspired by the works of He et al. [18].

Context-sensitivity. A well-researched property of points-to analysis, and in general any interprocedural analysis, is context-sensitivity, which allows each method in the program to be analyzed under different contexts [29]. Different types of context-sensitivity have been studied in the past [22, 38, 47, 50, 55], e.g., call-site-sensitivity using method call-sites [47] and object-sensitivity using allocation sites [38].



(a) Renaissance





(b) DaCapo **(c)** Microservices

Figure 9. Normalized metrics per bench suite. Lower is better, anything below 1.0 is an improvement. SkipFlow consistently improves all the metrics apart from *analysis time*, where the results are inconclusive. Nevertheless, even for *analysis time* the average is still slightly better (-1.6%) for SkipFlow.

While a lot of effort has been made to improve the scalability of context-sensitive analysis [6, 36, 55], it still often scales poorly for large applications [60]. As the analysis time is an important metric for optimizing compilers, our implementation of SkipFlow is context-insensitive. However, the approach can be applied in a context-sensitive analysis too.

Partial Flow-sensitivity. Roy et al. proposed partial flow-sensitivity [45], which maintains flow-sensitivity only for a specific set of program points. However, their approach expected the user to provide a set of program points, requiring a manual step before the analysis and thus rendering it unsuitable for optimizing compilers.

Wei et al. [58] presented a partially flow-sensitive points-to analysis for Javascript using a State Preserving Block Graph, a transformed control-flow graph whose blocks are aggregated into region nodes according to whether or not they contain a state-update statement, i.e., a property write or delete. Nevertheless, their approach still increased the overhead of the analysis compared to the baseline analysis.

The ideas of partial flow-sensitivity are arguably close to the principles of our analysis. SkipFlow could perhaps be seen as another representative of this category. However, it is not only fully automated and capable of increasing precision but it also keeps the same overhead as a baseline flow-insensitive analysis.

Value Flow Analysis. Value flow analysis has been successfully applied to finding many types of source-sink problems including null pointer derefences [20], memory leaks [11], buffer overruns [34], the usage of uninitialized variables [61], and concurrency issues [10]. Sui et al. [53] presented a scalable and precise interprocedural static value flow analysis for C programs. Livshits et al. introduced IPSSA [34], extending SSA with def-use relationships due to pointer dereferences and procedure calls. Shi at al. presented Pinpoint [48], a sparse value flow analysis using a holistic approach in which the underlying points-to analysis is aware of the high-level properties being checked and computes only points-to sets that are relevant to checking these properties. However, such an on-demand approach cannot be directly utilized in optimizing compilers where the goal is to compute information about *all program locations*.

Most value flow analysis implementations rely on SMT solving [48, 49]. Formulas representing individual program paths are constructed and used to determine path feasibility [11]. In our approach, thanks to *filtering flows* and *predicate edges*, we can obtain better precision compared to the baseline analysis without constructing any formulas.

Dataflow Analysis. Dataflow analysis is a framework that can be utilized for a wide range of tasks, including points-to analysis or constant propagation [46]. Often, dataflow analysis can also be reduced to a graph problem [44]. Fischer et al. [15] presented predicates in dataflow analysis using *predicated lattices* which partition the program state according to a set of predicates and track a lattice element for each partition. The most distinguishing feature of SkipFlow compared to classical dataflow analysis is the usage of *predicate edges* instead of *predicate latttices*, allowing us to keep using a simple lattice.

Gated SSA form. Our approach bears a certain resemblance to the gated SSA form defined by Ottenstein et al. [40]. Contrary to the Gated SSA, we do not extend the ϕ nodes and instead use *predicate edges* to capture the impact of branching conditions.

Sparse Conditional Constant Propagation. Sparse Conditional Constant Propagation (SCCP) [12, 57] is a compiler optimization that propagates constants while taking into account the control flow of the program and the reachability of individual instructions. It has been traditionally applied only intraprocedurally on a single compilation unit (one root method and possibly many inlined callees), which leads to low precision especially when programs are built from many small functions, as is common in languages such as Java or C#. While we presented our work as a static analysis method, it can also be seen as a novel Whole Program Sparse Conditional Constant Propagation. While existing SCCP implementations also operate on a lattice of values, we are not aware of an implementation that uses a complex lattice for object values that would be comparable to a points-to-set. Within a single compilation unit, SCCP cannot handle complex operations on types. Instead, existing SCCP usually focuses on primitive values, using a lattice for primitive values that is more complicated than our simple constants-only representation of primitive values.

8 Conclusions

In this paper, we presented SkipFlow that can be viewed as an extension of a flow-insensitive points-to analysis that models primitive values and maintains flow-sensitivity for local variables and simple branching instructions only, while falling back to the flow-insensitive analysis for the rest, thus increasing the precision without significantly increasing the analysis overhead. We implemented our analysis in the GraalVM Native Image and evaluated it on a wide range of benchmarks including DaCapo, Renaissance, and a set of microservice applications. Overall, our analysis reduces the number of reachable methods by 9% on average without negatively impacting the analysis time, in fact even reducing it in many cases. We believe our analysis is a sweet spot in the design space, improving the precision of the analysis without negatively impacting analysis time.

In the future, one can think whether the amount of information considered by the analysis can be extended further without hurting the scalability.

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. This work was 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.

A Artifact Appendix

The Docker image that replicates our experiments [23], the results of which are presented in Section 6, particularly in Table 1 and Figure 9, is available on Zenodo for artifact evaluation. For more details, we refer to the artifact itself, which contains a detailed README file.

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] Lars Ole Andersen and Carsten K. Gomard. 1992. Speedup Analysis in Partial Evaluation: Preliminary results. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM'92).*

[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. *SIGPLAN Not.* 31, 10 (oct 1996), 324–341. https://doi.org/10.1145/236338.236371

[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] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 169–190. https://doi.org/10.1145/1167473.1167488

[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] Yuandao Cai, Peisen Yao, and Charles Zhang. 2021. Canary: practical static detection of inter-thread value-flow bugs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 1126–1140. https://doi.org/10.1145/3453483.3454099

[11] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 480–491. https://doi.org/10.1145/1250734.1250789

[12] Cliff Click and Keith D. Cooper. 1995. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.* 17, 2 (mar 1995), 181–196. https://doi.org/10.1145/201059.201061

[13] 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

[14] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis *(ECOOP '95)*. Springer-Verlag, Berlin, Heidelberg, 77–101.

[15] Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. 2005. Joining dataflow with predicates. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press, 227–236. https://doi.org/10.1145/1081706.1081742

[16] R Fiutem, P Tonella, G Antoniol, and E Merlo. 1999. Points-to analysis for program understanding. *Journal of Systems and Software* 44, 3 (1999), 213–227. https://doi.org/10.1016/S0164-1212(98)10058-4

[17] 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

[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] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. 1999. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.* (1999), 848–894. https://doi.org/10.1145/325478.325519

[20] David Hovemeyer and William Pugh. 2007. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM Press, 9–14. https://doi.org/10.1145/1251535.1251537

[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 Tomas Vojnar. 2024. SkipFlow: Improving the Precision of Points-to Analysis using Primitive Values and Predicate Edges - Artifact. https://doi.org/10.5281/zenodo.10900903

[24] 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

[25] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[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 19th ACM SIGSOFT Symposium and the 13th 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] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. 2023. *The Java Virtual Machine Specification, Java SE 21 Edition*. https://docs.oracle.com/javase/specs/jvms/se21/html/index.html

[32] 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

[33] 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 soundness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

https://doi.org/10.1145/2644805

[34] V. Benjamin Livshits and Monica S. Lam. 2003. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM Press, 317–326. https://doi.org/10.1145/940071.940114

[35] 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.

[36] 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

[37] Jonas Lundberg, Tobias Gutzmann, Marcus Edvinsson, and Welf Löwe. 2009. Fast and precise points-to analysis. *Information and Software Technology* 51, 10 (2009), 1428–1439. https://doi.org/10.1016/j.infsof.2009.04.012

[38] 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

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

[40] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. 1990. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM Press, 257–271. https://doi.org/10.1145/93542.93578

[41] 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

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

[43] 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

[44] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages.* ACM Press, 49–61. https://doi.org/10.1145/199448.199462

[45] Subhajit Roy and YN Srikant. 2007. Partial flow sensitivity. In *High Performance Computing–HiPC 2007: 14th International Conference, Goa, India, December 18-21, 2007. Proceedings 14.* Springer, 245–256. https://doi.org/10.1007/978-3-540-77220-0_25

[46] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167, 1 (1996), 131–170. https://doi.org/10.1016/0304-3975(96)00072-2

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

[48] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM Press, 693–706. https://doi.org/10.1145/3192366.3192418

[49] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2021. Path-sensitive sparse analysis without path conditions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM Press, 930–943. https://doi.org/10.1145/3453483.3454086

[50] 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

[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] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the International Conference on Compiler Construction.* ACM Press, New York, NY, USA, 265–266. https://doi.org/10.1145/2892208.2892235

[54] 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 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Minneapolis, Minnesota, USA) *(OOPSLA '00).* Association for Computing Machinery, New York, NY, USA, 264–280. https://doi.org/10.1145/353171.353189

[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] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* (1991), 181–210. https://doi.org/10.1145/103135.103136

[58] Shiyi Wei and Barbara G. Ryder. 2014. State-Sensitive Points-to Analysis for the Dynamic Behavior of JavaScript Objects. In *Proceedings of the European Conference on Object-Oriented Programming.* 1–26. https://doi.org/10.1007/978-3-662-44202-9_1

[59] 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

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

[61] Ding Ye, Yulei Sui, and Jingling Xue. 2018. Accelerating Dynamic Detection of Uses of Undefined Values with Static Value-Flow Analysis. In *Proceedings of the International Symposium on Code Generation and Optimization.* ACM Press, 154–164. https://doi.org/10.1145/2544137.2544154

[62] Jisheng Zhao, Michael G. Burke, and Vivek Sarkar. 2018. Parallel sparse flow-sensitive points-to analysis. In *Proceedings of the International Conference on Compiler Construction.* ACM Press, 59–70. https://doi.org/10.1145/3178372.3179517

B Predicated Value Propagation Graph

This section formalizes the *predicated value propagation graph* (PVPG), the core data structure upon which SkipFlow is executed. First, we define the base language that serves as the input to our analysis. Second, we define the lattice representing the values propagated through PVPGs. Third, we define the structure of PVPGs. Finally, we present an algorithm for creating a PVPG by a linear pass over a program written in the base language. We show how a PVPG is used in SkipFlow in Section C.

B.1 Base Language

For the formal definition of our analysis, we use the base language presented in Figure 10 with instructions for object instantiation, field access, method invocation, and control flow using ***jump*** and ***if***. The instructions within a method are separated into *blocks*.

The base language is in SSA form, i.e., all variables have one static definition, all variables are defined before their first use, and all usages are dominated by the single definition.

Each method has exactly one ***start*** $(p_0, ..., p_n)$ instruction, at the beginning of the first block. This instruction explicitly defines method parameters $p_0, ..., p_n$ where p_0 is the receiver object. Without loss of generality, each method has only a single ***return*** v instruction.

Blocks starting with a ***merge*** $[u \leftarrow \phi_1(u_1, ..., u_n), ..., v \leftarrow \phi_k(v_1, ..., v_n)]\ m$ instruction represent control flow merges and can form loops. A ***merge*** instruction has a label $m \in M$, where M is a set of labels. A ***merge*** instruction defines a unique named location m that is the target of ***jump*** instructions that use the same label. Furthermore, a ***merge*** instruction defines a set $\{\phi_1, ..., \phi_k\}$ of ϕ instructions, one for each variable that has multiple reaching definitions at the control flow merge. Every ϕ instruction has as many arguments as there are ***jump*** instructions to its ***merge***.

A ***label*** l instruction creates a unique named location $l \in L$, where L is a second set of labels disjoint from M. A ***label*** instruction marks the beginning of one of the two branches of an ***if*** instruction using the same label. A block ending with an ***if*** instruction always has two successor blocks beginning with ***label*** instructions. A block beginning with a ***label*** instruction always has a single predecessor ending with ***if***, i.e., it can never be the target of a ***jump*** instruction.

The constraints above ensure that there are no *critical edges*, i.e., control flow edges from a block with multiple successors to a block with multiple predecessors. This is without loss of generality as long as the language does not have computed jump targets.

We explicitly support three types of conditions: $v_1 = v_2$, $v_1 < v_2$, and v ***instanceof*** T. Furthermore, nullcheck is covered implicitly by a $v_1 = v_2$ check where $v_2 \leftarrow$ `null`. Boolean values are encoded as integers, and the truth test is then represented as a $v_1 = v_2$ check where $v_2 \leftarrow 1$. Other conditions are excluded from the formal base language for simplicity. Having only $<$ and $=$ as comparisons is without loss of generality because $>$, $\leq$ $\geq$, and $\neq$ can be expressed by switching the order of operands and/or the order of the ***then***/***else*** blocks. The n case in the `Expr` rule covers primitive literals. Since we do not seek to model arithmetic computations, we use the generic ***Any*** instruction instead, representing any arithmetic.

Method		::=	*Block**
Block		::=	*BlockBegin s* BlockEnd*
BlockBegin		::=	***start*** $(p_0, ..., p_n)$
		|	***merge*** $[u \leftarrow \phi_1(u_1, ..., u_n), ..., v \leftarrow \phi_k(v_1, ..., v_n)]\ m$
		|	***label*** l
Statement	s	::=	$v \leftarrow e$
		|	$v \leftarrow r.x$
		|	$r.x \leftarrow v$
		|	$v \leftarrow v_0.m(v_1, .., v_n)$
BlockEnd		::=	***return*** v
		|	***jump*** m
		|	***if*** c ***then*** l_{then} ***else*** l_{else}
Cond	c	::=	$v_1 = v_2$
		|	$v_1 < v_2$
		|	v ***instanceof*** T
Expr	e	::=	n
		|	***Any***
		|	***new*** T
		|	`null`

Figure 10. The base language considered by the analysis.

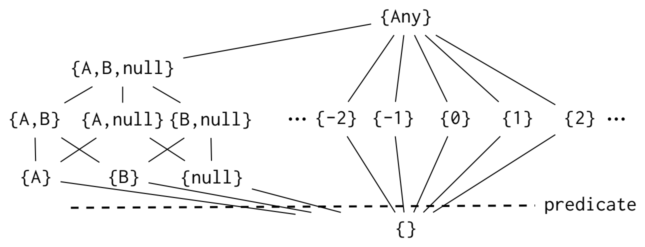


Figure 11. An example of the lattice $\mathbb{L}$ for $\mathbb{T} = \{$`A`, `B`, `null`$\}$ and all the primitive values. The line denoted as `predicate` indicates the fact that any non-empty value leads to the triggering of *predicate* edges.

B.2 The Lattice Domain of SkipFlow

In this section, we define the lattice $\mathsf{L}_{\leq}$ whose values are propagated through PVPGs. Figure 6 in Section 3 already introduced our primitive value lattice $\mathbb{P}$. To improve the scalability, we model neither sets nor intervals of values. The join of any two constant values results immediately in `Any`. Following the work of Wimmer et al. [60], objects are represented by their types only and modelled using the subset lattice $\mathbb{S} = (2^{\mathbb{T}}, \subseteq)$ over the set of program types $\mathbb{T}$. The *nullness* of a given value is modelled using the special value `null` $\in \mathbb{T}$.

Formally, $\mathbb{L}_{\leq} = (\mathbb{L}, \leq_{\mathbb{L}})$ where $\mathbb{L} = \{\{p\} \mid p \in \mathbb{P} \setminus \{\texttt{Empty}\}\} \cup 2^{\mathbb{T}}$ and $\leq_{\mathbb{L}}$ is defined as the smallest relation such that (1) $\{p\} \leq_{\mathbb{L}} \{q\} \Leftrightarrow p \leq_{\mathbb{P}} q$ for any $p, q \in \mathbb{P} \setminus \{\texttt{Empty}\}$, (2) $S_1 \leq_{\mathbb{L}} S_2 \Leftrightarrow S_1 \leq_{\mathbb{S}} S_2$ for any $S_1, S_2 \in 2^{\mathbb{T}}$, (3) $\{\} \leq_{\mathbb{L}} S \leq_{\mathbb{L}} \{\texttt{Any}\}$ for any $S \in 2^{\mathbb{T}}$, and (4) $\{\} \leq_{\mathbb{L}} \{p\} \leq_{\mathbb{L}} \{\texttt{Any}\}$ for any $p \in \mathbb{P} \setminus \{\texttt{Empty}\}$. Note that $\top_{\mathbb{L}} = \{\top_{\mathbb{P}}\} = \{\texttt{Any}\}$ and $\bot_{\mathbb{L}} = \bot_{\mathbb{S}} = \{\}$. In $\mathbb{L}_{\leq}$, the primitive values are wrapped into 1-element sets (apart from `Empty` which is replaced by the empty set), allowing us to treat all the *value states* uniformly as sets in the text that follows.

The lattice $\mathbb{L}_{\leq}$ for $\mathbb{T} = \{\texttt{A}, \texttt{B}, \texttt{null}\}$ is given in Figure 11. The line denoted as `predicate` indicates the fact that any non-empty value leads to the triggering of *predicate* edges.

In the rest of the text, since we do not have to distinguish between the lattice and its carrier set, we simply denote $\mathbb{L}_{\leq}$ as $\mathbb{L}$.

B.3 Structure of a PVPG

A PVPG models the flow of both primitive values and types interprocedurally. Vertices in a PVPG are called **flows** to clearly distinguish them from the **instructions** in the base language. In particular, the flows represent:

- values of method parameters, variables, and fields read or written by a particular instruction (each instruction reading or writing to a variable produces a fresh flow),
- method calls, which also represent the returned value in the caller,
- values to be returned from a method call back to the caller,
- values of various conditions present in the code (including their negated and inverted versions as explained later),
- values resulting from joining the values of other flows (results of ϕ instructions),
- ϕ_{pred} flows for joining predicates,
- the always enabled predicate $pred^{on}$.

Each flow keeps a reference to the underlying base language element it represents. In the algorithms and inference rules below, we use the notation $Flow(i)$ to express the creation of a new flow for a given base language element i, where i can be either an instruction as a whole, a condition, or a fresh ϕ for the joining of values. Subsequently, $f : Flow(i)$ denotes the flow f created from i.

The analysis computes the *value state* of each flow, which represents a conservative overapproximation of the values that can be assigned to the given flow during the runtime of the program. Flows can be connected via three types of **edges** described below.

A *use* edge represents a *def-use* dependency between flows. If there is a *use* edge between flows `s` and `t`, denoted by $\texttt{s} \rightsquigarrow_{use} \texttt{t}$, the *value state* of `t` has to be at least as big as the *value state* of `s` in terms of the underlying lattice provided that s is executable and no filtering is applied (the semantics of filtering is explained later).

A *predicate* edge represents a control-flow dependency. A *predicate* edge between flows `s` and `t`, denoted by $\texttt{s} \rightsquigarrow_{pred} \texttt{t}$, defines that if `s` is executable and has a non-empty *value state*, `t` is also executable. Every flow is the target of an incoming *predicate* edge, apart from $pred^{on}$, which is always enabled and therefore does not need an incoming edge. Flows that do not have any suitable predicate are assigned a *predicate* edge from $pred^{on}$. A ϕ_{pred} flow has multiple incoming *predicate* edges. In that case, it suffices that any of its predicates has a non-empty state to enable the execution of the target flow. Apart from $pred^{on}$ and ϕ_{pred}, flows representing conditions and method calls can also be the sources of predicate edges.

An *observe* edge represents an additional flow-specific dependency. An *observe* edge between flows `s` and `t`, denoted as $\texttt{s} \rightsquigarrow_{obs} \texttt{t}$, defines that `t` has access to the *value state* of `s`, and `t` is notified when the state of `s` changes to perform some flow-specific task such as a field update or method resolution and linking (discussed more below). Observe edges are created in three cases: (1) to link a flow representing a *receiver* with subsequent flows representing instructions for calling methods, (2) to link a flow representing an object with subsequent flows representing instructions for loading/storing fields, (3) to connect the second argument of a filtering flow performing a binary comparison, i.e. the argument according to whose *value state* the filtering is performed (more on that later).

B.4 Creating a PVPG

In this section, we describe how the PVPG for a given method is created by a sequential pass over the method body. The rules for value propagation are defined later in Appendix C. The PVPG is created by traversing the basic blocks in reverse postorder. The instructions within each basic block are processed sequentially from top to bottom. During the traversal, flows are created for the encountered base language elements. The traversal maintains a state for each basic block consisting of: (1) A mapping m from variables to previously created flows, which is used to connect flows with their dependencies, e.g. to establish a *use* edge between the flow representing `x` and a subsequently created filtering flow `x != 0`, (2) a reference denoted as *pred* that is continuously updated to refer to the last encountered predicate. The predicate referenced by *pred* is used to establish predicate edges. At the beginning of a method, when no suitable flow is available yet, the special $pred^{on}$ flow is used, which is always enabled, i.e. the initial flows inside methods are always enabled.

Initially, the mapping m in each basic block is empty. The *pred* of each basic block is set depending on its initial instruction. For a ***start*** instruction, *pred* is set to $pred^{on}$. For each basic block starting with a ***merge*** instruction, *pred* is set to a fresh ϕ_{pred} flow, which represents the merge of predicates at the current program point (as discussed in Section 3). Intuitively, a basic block starting with a ***merge*** is reachable iff the end of any of its predecessors is. For a ***label*** instruction, *pred* is set when processing the predecessor basic block as it depends on the branching condition.

When processing an instruction, new flows and edges are created and the basic block variables m and *pred* are updated as described in Figure 12. When processing a ***start*** instruction, flows for method parameters are created. ***merge*** and ***label*** instructions do not require any treatment as the state propagation between basic blocks is handled from the ends of predecessors. Each $v \leftarrow e$ instruction is handled by creating a flow that represents v and that is therefore stored in the mapping $b.m$. Next, each $v \leftarrow r.x$ instruction is handled similarly to $v \leftarrow e$, with an additional *observe* edge from r because the load needs to be notified every time the *value state* of the object r changes. Each $r.x \leftarrow v$ instruction has an additional *use* edge from v through which the values to be assigned to $r.x$ flow. Each $v \leftarrow v_0.m(v_1, .., v_n)$ instruction again needs an *observe* edge for the receiver v_0. Note that the call also represents the returned result v, which may be empty if the method never returns. To handle this case precisely, $v \leftarrow v_0.m(v_1, .., v_n)$ becomes the new predicate.

jump instructions need to propagate the state of the current basic block b to the successor t, which is done in the auxiliary function *propagate* presented in Figure 13. First, a *predicate* edge is established between the current predicate in b and the predicate of t, encoding the fact that if the end of b is reachable, so is the beginning of t. Second, the content of $b.m$ is propagated into $t.m$. Each variable $v \in b.m$ is handled as follows (note that we use the notation ***for*** $v \in b.m$ to denote an iteration over all the variables, i.e. keys, in the mapping m). If t does not yet have a mapping for v, it is inherited from b. Otherwise, we check if there is a collision, i.e. b and t have different flows representing v. If not, nothing needs to be done. If there is a collision, the PVPG has to be extended as follows. If the mapping for v in t is not a representation of a ϕ instruction but just a value inherited from some other predecessor, we create a new ϕ flow f and adjust the edges so that f represents the joining of the values from the predecessors. Otherwise, the ϕ flow has already been created before, and we can simply add a new *use* edge to it.

For **if** instructions, it is necessary to initialize the state of both of their successor basic blocks. To do so, the helper function *initBlock* is called, which dispatches to the *initUnary* and *initBinary* functions presented in Figure 14 based on the type of the condition c. Notice that the condition is inverted when calling *initBlock* for the `else` branch. For a unary condition (a type check), a new flow is created and inserted into the PVPG to represent the value of the checked variable x after filtering it based on c. For a binary condition, both the left variable denoted as $c.l$ and the right variable denoted as $c.r$ involved in the condition are filtered, with the resulting flows stored into $t.m[c.l]$ and $t.m[c.r]$, respectively. The filtering of $c.l$ is similar to the handling of a unary condition, with an additional *observe* edge for $c.r$ because the filtering flow needs to be notified in case any of its operands change. The filtering of $c.r$ is a mirrored version of the process for $c.l$. Note that the condition is *flipped*, not inverted, i.e. $flip(<) = >$, while $inv(<) = \geq$. This is because when filtering y with respect to $x < y$, the filtering should only allow values of y strictly greater than x. The predicates are chained so that $b.pred \rightsquigarrow_{pred} f_l \rightsquigarrow_{pred} f_r$, where f_l and f_r are the flows created for the original condition and the flipped version, respectively. Furthermore, f_r becomes the new predicate in t, ensuring that t is considered reachable only iff the end of b is reachable and the results of filtering of both x and y are not empty.

C Value Propagation through PVPGs

This section presents the core analysis algorithm based on the PVPG data structure. Given a PVPG, primitives and types are propagated from source flows along the *use* edges. Each flow has a *value state* describing the set of values that can be assigned at runtime to the code element, e.g., a variable or a field, represented by the given flow. Contrary to a typical pointer assignment analysis, values in PVPG are only propagated by flows that are *enabled* by their predicate. The values propagated through a PVPG are from the lattice $\mathbb{L}$ described in Appendix B.2.

We now proceed by defining SkipFlow through a series of inference rules precisely specifying the conditions for value propagation. Let $\mathbb{T}$, $\mathbb{M}$, $\mathbb{F}$, and $\mathbb{N}$ be pairwise disjoint sets representing the domains of types (including `null`), methods, field names, and PVPG flows, respectively. Two auxiliary functions are used: $LookUp\colon \mathbb{T} \times \mathbb{F} \rightharpoonup \mathbb{N}$, which returns the flow representing the given `field` of the given `type`, and $Resolve\colon \mathbb{T} \times \mathbb{M} \rightharpoonup \mathbb{M}$, which resolves a virtual method invocation for the given `type` and `method`[5].

The analysis computes the set of reachable methods $\mathbb{R} \subseteq \mathbb{M}$, the set of enabled flows $Enabled \subseteq \mathbb{N}$, the input *value state* of each flow $VS_{in}\colon \mathbb{N} \rightarrow \mathbb{L}$, and the output *value state* of each flow $VS_{out}\colon \mathbb{N} \rightarrow \mathbb{L}$.

Note that we introduce a separation between input and output *value states*. However, this concept is introduced purely to simplify the definition of some of the inference rules. The actual implementation uses one *value state* per flow. All the sets above as well as all the *value states* are initially empty. The analysis is started by inserting a set of *root methods* into $\mathbb{R}$ and $pred^{on}$ into *Enabled*. The set of *root methods* contains the entry points from which the analysis is started, e.g., the `main` method. Recall that $pred^{on}$ is used as the special always-enabled predicate that is assigned to the flows at the beginning of the first block in a method when

[5]Both *LookUp* and *Resolve* are partial, i.e. they only return values for valid combinations of input parameters. We assume that the base language is well-typed. Invalid code trying to access a non-existent field or calling a non-existent method would be rejected by a type system prior to running SkipFlow.

Processed Instruction i	Created Flows	Use Edges	Observe Edges	Modifications of m and $pred$
start $(p_0, ..., p_n)$	$\forall j \in [0,n] : f_j \leftarrow Flow(p_j)$			$\forall j \in [0,n] : b.m[p_j] \leftarrow f_j$
merge, ***label***				
$v \leftarrow e$	$f \leftarrow Flow(i)$			$b.m[v] \leftarrow f$
$v \leftarrow r.x$	$f \leftarrow Flow(i)$		$b.m[r] \rightsquigarrow_{obs} f$	$b.m[v] \leftarrow f$
$r.x \leftarrow v$	$f \leftarrow Flow(i)$	$b.m[v] \rightsquigarrow_{use} f$	$b.m[r] \rightsquigarrow_{obs} f$	
$v \leftarrow v_0.m(v_1, .., v_n)$	$f \leftarrow Flow(i)$		$b.m[v_0] \rightsquigarrow_{obs} f$	$b.m[v] \leftarrow f, b.pred \leftarrow f$
return v	$f \leftarrow Flow(i)$	$b.m[v] \rightsquigarrow_{use} f$		
jump m	$propagate(b, m.target)$			
if c ***then*** l_{then} ***else*** l_{else}	$initBlock(b, l_{then}, c), initBlock(b, l_{else}, inv(c))$			

Figure 12. The effect of processing each instruction i when creating a PVPG. The current basic block is denoted as b. Additionally, each flow f is assigned a predicate edge $b.pred \rightsquigarrow_{pred} f$ upon its creation. Note that *Flow* is not a function in the mathematical sense but rather a constructor creating a new flow upon every invocation. The notation $\forall j \in [0,n]$ used in the first line denotes that a set of flows indexed by integers 0 to n is handled uniformly.

```
function PROPAGATE(b,t)
  $b.pred \rightsquigarrow_{pred} t.pred$
  for $v \in b.m$ do                          ▷ Iterate over all vars, i.e. keys, in $b.m$
    if $v \notin t.m$ then                     ▷ Is there a mapping for $v$ in $t.m$?
      $t.m[v] \leftarrow b.m[v]$
    else
      $b_v \leftarrow b.m[v]$
      $t_v \leftarrow t.m[v]$
      if $b_v \neq t_v$ then   ▷ Is there a collision (two flows for the same $v$)?
        if not $isPhi(t_v)$ then                 ▷ No $\phi$ flow created yet?
          $t.m[v] \leftarrow Flow(\phi)$                ▷ Create a fresh $\phi$ flow
          $t.pred \rightsquigarrow_{pred} t.m[v]$
          $b_v \rightsquigarrow_{use} t.m[v]$
          $t_v \rightsquigarrow_{use} t.m[v]$
        else
          $b_v \rightsquigarrow_{use} t_v$        ▷ $\phi$ flow already created, just add a use edge
```

Figure 13. The *propagate* function handling ***jump*** instructions.

```
function INITBLOCK(b,t,c)
  if $isUnary(c)$ then
    $initUnary(b, t, c)$
  else
    $initBinary(b, t, c)$
function INITUNARY(b,t,c)
  $f \leftarrow Flow(c)$
  $b.pred \rightsquigarrow_{pred} f$
  $b.m[c.x] \rightsquigarrow_{use} f$
  $t.m[c.x] \leftarrow f$
  $t.pred \leftarrow f$
```

```
function INITBINARY(b,t,c)
  $f_l \leftarrow Flow(c)$
  $b.pred \rightsquigarrow_{pred} f_l$
  $b.m[c.l] \rightsquigarrow_{use} f_l$
  $b.m[c.r] \rightsquigarrow_{obs} f_l$
  $t.m[c.l] \leftarrow f_l$
  $f_r \leftarrow Flow(flip(c))$
  $f_l \rightsquigarrow_{pred} f_r$
  $b.m[c.r] \rightsquigarrow_{use} f_r$
  $b.m[c.l] \rightsquigarrow_{obs} f_r$
  $t.m[c.r] \leftarrow f_r$
  $t.pred \leftarrow f_r$
```

Figure 14. The *initBlock*, *initUnary*, and *initBinary* functions handling ***if*** instructions. In the *initBinary* function, the notation $c.l$ and $c.r$ is used to denote access to the left and the right argument of the comparison c, respectively.

no other predicate is available yet. The analysis adds transitively reachable methods to $\mathbb{R}$ until a fixed point is reached. The actual logic of the analysis is defined by the inference rules presented in Figure 15.

The Source rule specifies the behaviour of a flow representing a $v \leftarrow e$ instruction. Once enabled, the *value state* of f contains the result of the evaluation of the expression e.

The Propagate rule specifies the conditions for propagating values through a PVPG. Observe that $\leq_{\mathbb{L}}$ is established between the *value states* of f_s and f_t. This kind of formula is used throughout the rules to model any type of value propagation: Once the analysis reaches a fixed point, the *value state* of f_t has to be at least as big as the *value state* of f_s in terms of the lattice $\mathbb{L}$.

The Predicate rule defines predicate handling. A flow is enabled when its predicate is enabled and the predicate's *value state* is non-empty.

The Load and Store rules represent field manipulation. Both rules operate on the flow r representing the accessed object and lookup the flows representing fields named x on all types t in the *value state* of r, denoted as $f_{t.x}$. Recall that $Flow(v \leftarrow r.x)$ represents the joined value of all the given fields x for all the types in the *value state* of the receiver. Therefore, the Load rule establishes $\leq_{\mathbb{L}}$ between the VS_{out} of all the related $f_{t.x}$ and the VS_{in} of the $Flow(v \leftarrow r.x)$ itself. Similarly, the Store rule establishes $\leq_{\mathbb{L}}$ between the VS_{out} of $Flow(r.x \leftarrow v)$ and the VS_{in} of all the related $f_{t.x}$.

The rules TypeCheck, Cond, and PassThrough define how the VS_{in} of each flow is mapped to VS_{out}. If the given flow represents a type check c, which can be either an ***instanceof*** check or its negated version, the VS_{out} contains only the types from VS_{in} that pass the condition c. If the condition c is a comparison, the auxiliary function *Compare* is used to compute the VS_{out}. If a flow does not have any condition, its input is passed directly to its output without any filtering.

The auxiliary *Compare* function uses an operator *cond*, and v_l, v_r as operands, returning the content of v_l filtered with respect to c and v_r. If at least one of the operands is empty, we return an empty set because both operands are needed to perform the filtering. If the operator is '=' and at least one of the operands contains Any, then the result of the

$$\frac{f\colon \mathit{Flow}(v \leftarrow e),\ f \in \mathit{Enabled}}{\mathit{VS}_{in}(f) = \begin{cases} n & \text{for } v \leftarrow n \\ \texttt{Any} & \text{for } v \leftarrow \textbf{\textit{Any}} \\ T & \text{for } v \leftarrow \textbf{\textit{new}}\ T \\ \textbf{\textit{null}} & \text{for } v \leftarrow \texttt{null} \end{cases}}\ \texttt{[Source]} \qquad \frac{f_s \in \mathit{Enabled},\ f_s \rightsquigarrow_{use} f_t}{\mathit{VS}_{out}(f_s) \leq_{\mathbb{L}} \mathit{VS}_{in}(f_t)}\ \texttt{[Propagate]} \qquad \frac{f_s \in \mathit{Enabled},\ \mathit{VS}_{out}(f_s) \neq \{\},\quad f_s \rightsquigarrow_{pred} f_t}{f_t \in \mathit{Enabled}}\ \texttt{[Predicate]}$$

$$\frac{f\colon \mathit{Flow}(v \leftarrow r.x),\ f \in \mathit{Enabled} \quad t \in \mathit{VS}_{out}(r),\ f_{t.x} = \mathit{LookUp}(t, x)}{\mathit{VS}_{out}(f_{t.x}) \leq_{\mathbb{L}} \mathit{VS}_{in}(f)}\ \texttt{[Load]} \qquad \frac{f\colon \mathit{Flow}(r.x \leftarrow v),\ f \in \mathit{Enabled} \quad t \in \mathit{VS}_{out}(r),\ f_{t.x} = \mathit{LookUp}(t, x)}{\mathit{VS}_{out}(f) \leq_{\mathbb{L}} \mathit{VS}_{in}(f_{t.x})}\ \texttt{[Store]}$$

$$\frac{f\colon \mathit{Flow}(v \leftarrow v_0.m(v_1, .., v_n)) \quad f \in \mathit{Enabled},\ \ t \in \mathit{VS}_{out}(v_0) \quad r = \mathit{Resolve}(t, m)}{r \text{ added to } \mathbb{R},\ \forall i \in [0, n]\colon v_i \rightsquigarrow_{use} p_i^r \quad \mathit{ret}_r \rightsquigarrow_{use} f}\ \texttt{[Invoke]}$$

$$\frac{f\colon \mathit{Flow}(c),\ f \in \mathit{Enabled},\ \mathit{isTypeCheck}(f) \quad v_{out} = \{t \mid t \in \mathit{VS}_{in}(f) \wedge c(t)\}}{v_{out} \leq_{\mathbb{L}} \mathit{VS}_{out}(f)}\ \texttt{[TypeCheck]} \qquad \frac{f\colon \mathit{Flow}(c),\ f \in \mathit{Enabled},\ \mathit{isComparison}(f) \quad v_{out} = \mathit{Compare}(c, \mathit{VS}_{out}(c.l), \mathit{VS}_{out}(c.r))}{v_{out} \leq_{\mathbb{L}} \mathit{VS}_{out}(f)}\ \texttt{[Cond]}$$

$$\frac{\neg(\mathit{isTypeCheck}(f)), \quad \neg(\mathit{isComparison}(f)), \quad f \in \mathit{Enabled}}{\mathit{VS}_{in}(f) \leq_{\mathbb{L}} \mathit{VS}_{out}(f)}\ \texttt{[PassThrough]}$$

$$\mathit{Compare}(\mathit{cond}, v_l, v_r) = \begin{cases} \{\} & \text{if } v_l = \{\} \vee v_r = \{\} \\ \mathit{min}_{\mathbb{L}}(v_l, v_r) & \text{else if } \mathit{cond}\ \textbf{\textit{is}}\ \text{'='} \wedge\ (\texttt{Any} \in v_l \vee \texttt{Any} \in v_r) \\ v_l \cap v_r & \text{else if } \mathit{cond}\ \textbf{\textit{is}}\ \text{'='} \\ v_l \setminus v_r & \text{else if } \mathit{cond}\ \textbf{\textit{is}}\ \text{'}\neq\text{'} \\ v_l & \text{else if } \texttt{Any} \in v_l \vee \texttt{Any} \in v_r \\ \{l \mid l \in v_l \wedge r \in v_r \wedge \mathit{cond}(l, r)\} & \text{otherwise} \end{cases}$$

Figure 15. Inference rules used by SkipFlow. In the rules, the meaning of the notation $f\colon \mathit{Flow}(v \leftarrow e)$ denotes that the flow f *was* created from the base language instruction $v \leftarrow e$ and similarly for the other instructions. Likewise, we use the notation $f\colon \mathit{Flow}(c)$ to denote a fact that a flow *was* created from a condition c. The helper functions *isTypeCheck* and *isComparison* return `true` iff the flow was created from a type check or a comparison operator, respectively.

filtering is the lower value in terms of the Lattice $\mathbb{L}$, e.g.:

$$\mathit{Compare}(\text{'='}, \{\texttt{Any}\}, \{5\}) = \{5\},$$
$$\mathit{Compare}(\text{'='}, \{\texttt{Any}\}, \{\texttt{Any}\}) = \{\texttt{Any}\}.$$

If the operator is '=', but none of the operands contains Any, the result is the set intersection of its arguments, e.g.:

$$\mathit{Compare}(\text{'='}, \{\mathsf{A}, \mathsf{B}\}, \{\mathsf{B}, \mathsf{C}\}) = \{\mathsf{B}\},$$
$$\mathit{Compare}(\text{'='}, \{3\}, \{3\}) = \{3\},$$
$$\mathit{Compare}(\text{'='}, \{3\}, \{5\}) = \{\}.$$

If the operator is '≠', the result is a set difference:

$$\mathit{Compare}(\text{'}\neq\text{'}, \{\mathsf{A}, \mathsf{B}\}, \{\mathsf{B}, \mathsf{C}\}) = \{\mathsf{A}\}.$$

This again works on primitives as well:

$$\mathit{Compare}(\text{'}\neq\text{'}, \{0\}, \{0\}) = \{\},$$
$$\mathit{Compare}(\text{'}\neq\text{'}, \{5\}, \{3\}) = \{5\}.$$

The remaining cases are '<', '≤', and other relational operators apart from '=' and '≠' handled above. These operators are defined on primitives only, and so we know that v_l and v_r are the 1-element sets containing some primitive value, or Any. If one of the operands contains Any, we cannot do any useful filtering, regardless of whether Any is in v_l or v_r. If Any $\in v_l$, we could only reduce v_l to an interval (assuming Any $\notin v_r$), which we have decided not to use for scalability purposes. If Any $\in v_r$, we do not have enough information to filter v_l. Therefore, we simply return v_l in both cases. Finally, if none of the operands contains Any, we return the value from v_l if it passes the *cond* with respect to v_r; otherwise, we return the empty set:

$$\mathit{Compare}(\text{'<'}, \{3\}, \{5\}) = \{3\},$$
$$\mathit{Compare}(\text{'<'}, \{3\}, \{1\}) = \{\}.$$

Finally, the `Invoke` rule specifies how methods are linked. Every time a new type is added to the *value state* of the receiver v_0, it is passed to the `Resolve` function to determine the callee r. Each callee r is added to the set of reachable methods $\mathbb{R}$ and subsequently linked by creating a *use* edge from the argument flows v_i in the caller to the parameter flows p_i^r of the callee, and from the return flow ret_r of the callee to the invoke f itself, which represents the returned value in the caller.

Received 2024-09-09; accepted 2024-11-04