Compact Representation and Interleaved Solving for Scalable Constraint-Based Points-to Analysis

Ramya Kasaraneni
Dept of CSE, IIT Madras
Chennai, TN, India
raka@cse.iitm.ac.in

V. Krishna Nandivada
Dept of CSE, IIT Madras
Chennai, TN, India
nvk@iitm.ac.in

Abstract

Constraint-based points-to analysis using Andersen-style inclusion constraints is widely used for its convenience, generality, and precision in modeling complex program behaviors. Typically, such analyses generate constraints and resolve them by computing the transitive closure of a constraint graph. However, traditional constraint modeling often introduces redundancy during constraint generation, solving, or both. In the context of points-to analysis for object-oriented languages like Java, this redundancy primarily stems from the modeling of virtual calls and heap operations. As a result, the analysis produces redundant constraints and inflated constraint graphs, thereby increasing analysis time.

To address these limitations, we propose a novel constraint representation and solving system *PInter*, which extends the traditional inclusion constraint model. It presents a novel technique to represent the constraints in compact and expressive way. Further, it defers generation of constraints until relevant objects are discovered, enabling demand-driven and interleaved constraint generation and solving. We present a proof of correctness of *PInter*. We used *PInter* to implement two different Java points-to analyses, within the Soot framework, and evaluated it on 12 applications drawn from the DaCapo benchmark suite. As is standard, we used Tamiflex to handle dynamic features of Java benchmarks and performed a soundy evaluation. Our results show that, compared to traditional methods, *PInter* reduced the constraint count by 96% (geomean) and the analysis time by 78% (geomean). We also evaluated *PInter* against Soot's Spark and the flow-, context-insensitive analysis in Doop, and found that *PInter* achieved significantly faster performance.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → *Automated static analysis*; • **Mathematics of computing** → *Solvers.*

Keywords: program analysis, inclusion-based constraints, parametric-constraints

ACM Reference Format:
Ramya Kasaraneni and V. Krishna Nandivada. 2026. Compact Representation and Interleaved Solving for Scalable Constraint-Based Points-to Analysis. In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction (CC '26), January 31 – February 1, 2026, Sydney, NSW, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3771775.3786280

CC '26, Sydney, NSW, Australia

ACM ISBN 979-8-4007-2274-5/2026/01
https://doi.org/10.1145/3771775.3786280

1 Introduction

Points-to analysis is a fundamental technique in static program analysis, enabling numerous optimizations, and correctness guarantees [9, 18, 19, 24]. Among various approaches, inclusion-constraint based points-to analysis [1, 7, 23] has gained widespread adoption due to its generality and ability to model complex program behaviors accurately.

Despite their utility, traditional approaches to generating and solving inclusion constraints [15, 21, 23] face performance and scalability bottlenecks, especially in languages like Java with pervasive virtual method calls. These approaches typically suffer from redundancy in constraint generation or/and solving. They model heap operations and virtual method calls by generating constraints for all potential targets upfront, often based on initial approximations like class hierarchy analysis [8].

For instance, Spark [15], the inclusion constraint-based points-to analysis in the Soot framework [29], pre-builds a Pointer Assignment Graph (PAG) upfront for all possible target methods even when the call-graph is constructed on-the-fly (called partly on-the-fly [16]). This upfront approach generates a large number of constraints from the start, many of which may be unnecessary. Similarly, for heap operations like stores (e.g., `x.f = y`), or loads (e.g., `z = x.f`), Spark adds edges to the PAG, such as `y` → `x.f`, or `x.f` → `z`. And when points-to sets are updated during the constraint solving phase (say, `o1` is added to the points-to set of `x`), Spark creates field reference nodes for `o1`, creating edges `y` → `o1.f` or `o1.f` → `z`. However, as points-to sets evolve, revealing new objects or aliases, the solving scheme of Spark revisits and reprocesses all load and store edges. This is because the pre-built PAG ties field references to variables (e.g., `x.f`) rather than specific objects (e.g., `o1.f`), making it hard to pinpoint which loads or stores are affected by changes.

Similar issues can also be observed in the constraint-based type inference approach [21], which generates constraints for all methods and constructs a trace graph to model type flows. It has an incremental solving approach which reduces redundant processing, but still requires generating constraints upfront for all potential methods, limiting scalability for large programs. Another popular approach, the annotated constraint-based points-to analysis [23], extends Andersen's analysis to Java using annotated inclusion constraints, where field annotations track flows through specific fields and method annotations model virtual calls with on-the-fly call graph construction. Constraints are generated incrementally for statements in reachable methods, discovered via reachability from entry points, and solved using local closure rules on a sparse, inductive-form graph until a fixed point is reached. This avoids global reprocessing of loads and stores by propagating points-to sets locally based on matching annotations. However, it does not use type-based filtering [15] during analysis, applying declared types only post-analysis, potentially leading to larger points-to sets and more reachable methods. It also introduces fresh variables for load and store constraints, increasing the constraint graph size. We further illustrate some of these challenges using an example.

Consider the example code snippet in Fig. 1, featuring a main class `M` and four classes: `A`, `B`, `C`, and `D`, where `B`, `C`, and `D` inherit from `A`. Each class defines an overridden method `foo()`. An object created at line `n` is denoted by `On`. For a virtual call like `r.foo()` at Line 16, traditional static analysis [15, 21] generates constraints for all possible implementations of `foo` based on the class hierarchy (`A::foo`, `B::foo`, `C::foo`, `D::foo`). These constraints can be redundant depending on the type of the analysis being performed, and gives us an idea of scope of improvement: Performing a flow and context-insensitive points-to analysis will result in `r` pointing to {`O4`,`O10`,`O13`}, thereby requiring only a subset of the generated constraints (`A::foo`, `B::foo`). With a type-based filtering during the analysis, the potential targets for the call would only be `B::foo`. Similarly, for heap operations, such as the field store `a.f=b` at Line 7, traditional approaches [15] tie field references to the base variable `a`. This requires revisiting all loads and stores whenever the points-to set of the base variable changes. Alternatively, introducing fresh variables [23] to model heap operations reduces the number of revisits but creates large sized constraint graphs.

To address these limitations, we propose a novel approach using parameterized constraints. Instead of eagerly generating constraints for all potential targets, we associate a single parameterized constraint with each operation (e.g., field access or virtual call). Such a constraint is instantiated only when a new object flows into the base variable. This demand-driven approach interleaves constraint generation and solving, incrementally producing constraints as points-to sets evolve. By aligning constraint generation closely with the points-to set propagation, our method reduces the computational cost of constraint generation and solving.

```
1  class M{
2   public static void main(..)
3    {
4      A a = new A(); //O4
5      A b = new B(); //O5
6      A c = new A(); //O6
7      a.f = b;
8      c.f = a;
9      if(..){
10       a = new A(); //O10
11     }
12     else{
13       a = new B(); //O13
14     }
15     B r = (B)a;
16     r.foo();
17 }
18 class A{
19   A f;
20   void foo(){...}
21 }
22 class B extends A{
23   void foo(){...}
24 }
25 class C extends A{
26   void foo(){...}
27 }
28 class D extends A{
29   void foo(){...}
30 }
```

Figure 1. Example code snippet.

This paper proposes a novel constraint representation and solving scheme called *PInter* with the following key contributions that enhance the traditional inclusion constraint-based framework.

1. We propose a compact representation of Andersen-style conditional constraints that speeds up the solving process and effectively reduces memory consumption.
2. To better handle calls to overridden methods—a primary source of constraint explosion in object-oriented analysis—we introduce a new constraint type called the "FunctionCall" constraint. This specialized constraint encodes virtual calls in a manner that avoids generating redundant constraints typically associated with method overriding, thus streamlining the resolution of dynamic dispatch in the analysis.
3. We present a novel, efficient interleaved constraint generation and solving scheme that ensures that we only generate the necessary constraints on demand and solve them.
4. We present a proof of correctness for the proposed scheme.
5. We implemented flow-insensitive and flow-sensitive points-to analyses for Java using *PInter* as well as the traditional constraints scheme [21] in Soot framework [29], and evaluated on 12 benchmarks from DaCapo [4] benchmark suite. We show that *PInter* generates 96% (geomean) less number of constraints and takes 78% (geomean) less time compared to the traditional constraints scheme. We also show that *PInter* takes significantly less time compared to the flow, context-insensitive analysis implementations available in Spark [15] and Doop [6].

2 The Interleaved Constraint Generation and Solving Algorithm

Considering the complexity and redundancy in the generation and solving of traditional constraint system, we propose the parameterization of conditional constraints and an interleaved constraint generation and solving scheme, which we name the *P*arameterized *Inter*leaved scheme (*PInter*).

Fig. 2 shows the working of our interleaved analysis. The *PInterDriver* algorithm takes as input a Java application and the entry point (main method) and generates and solves constraints. It is a worklist based algorithm that interleaves the processing of each method: constraint generation, and constraint solving. For all the methods present in the current worklist, the *PInter* algorithm first invokes the processMethod function to generate the relevant constraints for each method. Then it solves the generated constraints for all these methods together. Solving of these constraints may add more methods (based on the current points-to information) to the worklist. In this algorithm, a method is processed at most once. This iterative process is summarized in Fig. 3.

In Section 2.1, we discuss our constraint system and the constraints for points-to analysis and in Section 2.2 we discuss our constraint solving algorithm.

2.1 Constraint Generation

2.1.1 Traditional Constraint System. In program analysis, constraints are formal rules that model relationships between program entities, such as variables, objects, or types, based on program semantics like assignments or method calls. Constraint-based program analyses construct and solve these rules to infer properties, such as points-to sets in points-to analysis, or type sets in type inference, ensuring properties like type safety or valid pointer targets. For example, in points-to analysis, constraints define how objects flow through variables. Drawing from Oxhøj et al. [21], constraints are categorized as: (i) member constraints of the form $x \in A$, ensuring specific element x is in a set A; (ii) propagation constraints of the form $X \subseteq Y$, ensuring that elements of set X flow to set Y; and (iii) conditional constraints of the form $a \in A \Rightarrow X \subseteq Y$, modeling dependencies along program paths indicating a constraint $X \subseteq Y$ holds only under a certain condition (if $a \in A$). These constraints are typically solved using a fixed-point iteration over a constraint graph. For each statement, constraints are generated based on how the statement affects the points-to information.

2.1.2 Modifications to the Traditional Constraint System . We generate four types of constraints: Member, Propagation, Conditional, FunctionCall as shown in Fig. 4. The member and propagation constraints are the same as the traditional ones [21]. Unlike traditional conditional constraints, which associate a static condition with a constraint, our framework defines parametric conditional constraints that explicitly depend on parameters.

Traditional [21] conditional constraints are of the form $(O_x \in A \Rightarrow X \subseteq Y)$, where O_x is a possible member of set A. The RHS constraint $(X \subseteq Y)$ is evaluated whenever the condition becomes true. In contrast, in our concise representation, membership conditions are of the form $r \in A$, where r is a parameter ranging over all possible members of A. This avoids listing of large number of (often redundant) individual constraints for each possible member element of A. To distinguish these concise constraints from traditional conditional constraints, we represent them as $(\!|cond_r, C_r|\!)$, where $cond_r$ is a parameterized membership condition, and. C_r can be any arbitrary constraint that may have r as a free variable. The idea is that C_r is to be evaluated whenever $cond_r$ is true.

In addition to these membership based conditional constraints, to support strong update [25] in store statements, we also support cardinality based constraints of the form $(\!|cond, C, elseC|\!)$, where *cond* is a cardinality condition of the form $|A| == 1$, and C and *elseC* are arbitrary constraints. In such a conditional constraint, if A is non-empty, the constraint C holds if the condition *cond* evaluates to true; otherwise *elseC* holds. Our system requires that C and *elseC* have the following *monotonic discipline*: (i) solving of either of the constraints should add elements to the same target set. (ii) the set of values added by *elseC* must be $\supseteq$ the set of values added by C. This discipline helps in our correctness argument (see Section 4).

Example: For example, consider a store statement `x.f = y`. Assume, the maps *varPts* and *fieldPts* store the points-to information of variables and heap respectively. Then, the conditional constraint, $(\!|r \in varPts(\mathtt{x}), varPts(\mathtt{y}) \subseteq fieldPts(r, \mathtt{f})|\!)$, indicates that for each reference 'r' that flows into $varPts(\mathtt{x})$ (the points-to set of x), $varPts(\mathtt{y})$ will flow into $fieldPts(r, \mathtt{y})$.

In general, we believe that in *PInter*, $cond_r$ can encode a variety of conditions defined over the set A, enabling the expression of different semantic properties. We leave it as a future work to explore the same.

In addition to the three types of standard constraints, to avoid generating redundant constraints due to method call statements, we maintain an additional type of constraint called FunctionCall-constraint. A FunctionCall-constraint is of the form $[\![cond_r; S_1]\!]$, where the $cond_r$ can only be a membership condition on the receiver variable. For example, consider a statement S_c of the form $x.bar()$ and a FunctionCall constraint $[\![o \in varPts(S_c, x); S_c]\!]$. This constraint indicates that the call-site S_c should be processed by considering each object (o) that the receiver x may point to.

2.1.3 Points-to Analysis using *PInter*. Points-to analysis statically computes the set of objects that variables or fields of objects, may point to at runtime. Constraint-based points-to analysis models these relationships using set-inclusion constraints [1] which are formal rules capturing

```
1  Function PInterDriver :
2      workList := {mainMethod};
3      while (workList is not empty) do
4          consList = genConstraints(workList);
5          solveConstraints(consList);
6  Function genConstraints :
7      while workList is not empty do
8          m = workList.remove();
9          consList = consList ⋃ processMethod(m);
10     return consList
11 Function solveConstraints :
12     for c in consList do
13         solve(c); // May add to workList.
```

Figure 2. Details of the *PInterDriver* algorithm.

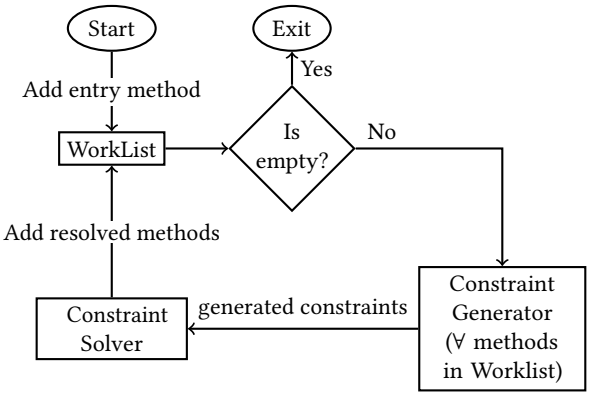

Figure 3. Block diagram showing the working of the interleaved approach, described in Fig. 2. The process is iterated till the worklist returned by the constraint-solver is empty.

Constraint type	Notation	Description
Member	$[\![M \in A]\!]$	Ensures that an element M belongs to a set A.
Propagation	$[\![A \subseteq B]\!]$	Ensures that set A is a subset of set B.
Conditional	$(\!\|cond_r,\ C_r, [elseC_r]\|\!)$	Applies constraint C_r if condition $cond_r$ is satisfied. Otherwise, optionally applies an $elseC_r$
FunctionCall	$[\![cond_r; S_c]\!]$	Resolves function in S_c based on condition $cond_r$.

Figure 4. Constraint types. Notation used: A and B indicate flow-sets, C indicates constraint; M indicates any element in the flow-set; r indicates temporary variables used by our scheme (not application variables); $cond_r$, C_r, indicate conditions and constraints that may be referring to a temporary r; S_c indicates a call statement.

subset relationships, or element membership based on program semantics. We now present instantiations of *PInter* to perform popular flow-insensitive and flow-sensitive (context-insensitive) points-to analyses.

For each pair of statements (S_1, S_2) in the program, where S_2 is a control-flow successor of S_1 [20], we define constraints for both flow-sensitive and flow-insensitive points-to analyses. We store points-to information in two maps: *varPts* for variables and *fieldPts* for fields. In flow-insensitive analysis, points-to maps are global, and their keys exclude statements. In flow-sensitive analysis, points-to maps are maintained at each statement, and we demonstrate how the maps at S_2 are influenced by the semantics of S_1.

1. *Allocation*: For an allocation statement $a = \text{new } X()$, let o denote the abstract object created at S_1. The following constraints are added:
   - Flow-insensitive: $[\![o \in varPts(a)]\!]$
   - Flow-sensitive: $[\![o \in varPts(S_2, a)]\!]$
2. *Assignment*: For an assignment statement $a = b$, the points-to set of b is transferred to a. The constraints are:
   - Flow-insensitive: $[\![varPts(b) \subseteq varPts(a)]\!]$
   - Flow-sensitive: $[\![varPts(S_1, b) \subseteq varPts(S_2, a)]\!]$
3. *Load*: For a load statement $a = b.f$, the constraints are:
   - Flow-insensitive:
     $(\!|r \in varPts(b), [\![fieldPts(r, f) \subseteq varPts(a)]\!]|\!)$
   - Flow-sensitive:
     $(\!|r \in varPts(S_1, b), [\![fieldPts(S_1, r, f) \subseteq varPts(S_2, a)]\!]|\!)$
4. *Store*: For a store statement $a.f = b$, the constraints are shown below. The flow-sensitive variant performs strong update [25] in case $varPts(S_1, a)$ is singleton set, and the object in it is guaranteed to represent a unique object at run time. Otherwise, it performs a weak update which is given as the else-constraint. Assume that using a simple prepass, the set of unique abstract objects in the program are stored in *uniqueObjs* set and the non-unique abstract objects are stored in *nonUniqueObjs*. The constraint (ii) depicts the usage of a cardinality based conditional constraint.
   - Flow-insensitive:
     $(\!|r \in varPts(a), [\![varPts(b) \subseteq fieldPts(r, f)]\!]|\!)$
   - Flow-sensitive:
   (i) $(\!|r \in varPts(S_1, a) \cap nonUniqueObjs,$
   $[\![varPts(S_1, b) \cup fieldPts(S_1, r, f) \subseteq fieldPts(S_2, r, f)]\!]|\!)$
   (ii) $(\!|r \in varPts(S_1, a) \cap uniqueObjs,$
   $(\!|\,|varPts(S_1, a)| == 1,$
   $[\![varPts(S_1, b) \subseteq fieldPts(S_2, r, f)]\!],$
   $[\![varPts(S_1, b) \cup fieldPts(S_1, r, f) \subseteq fieldPts(S_2, r, f)]\!]|\!)|\!)$

5. *Call*: For a method call statement $a = b.\text{bar}(\text{args})$, if the call is virtual, we generate a FunctionCall constraint:
- Flow-insensitive: $[\![r \in varPts(b); S_1]\!]$
- Flow-sensitive: $[\![r \in varPts(S_1, b); S_1]\!]$
6. *Return*: For a method return statement '`return` a' in a method `meth`, we generate a propagation constraint, where *retVal* is a map storing the set of references a method may return:
- Flow-insensitive: $[\![varPts(a) \subseteq retVal(\texttt{meth})]\!]$
- Flow-sensitive: $[\![varPts(S_1, a) \subseteq retVal(\texttt{meth})]\!]$

The constraints described above apply to serial execution. When analyzing parallel programs, our constraints are modular enough to integrate concurrency information to accurately model the effects of concurrent heap operations. The presence of concurrency requires that heap updates be propagated to concurrently executing statements, as these updates can occur in any order at runtime. Therefore, in the flow-sensitive points-to constraints, if S_1 is a load or store statement, then besides propagating points-to information to its control-flow successors S_2, we also propagate the points-to maps of heap to all statements that may execute concurrently with S_1.

We believe that *PInter* can also be naturally extended to incorporate context sensitivity, which we leave as future work. Context can be added as an extra parameter in a context-sensitive instantiation of *PInter*, analogous to the statement parameter used in flow-sensitive analysis. Our decision to focus on flow-sensitivity in this work is motivated by the novel cardinality condition we developed, which naturally applies to strong updates within the flow-sensitive setting. Adding context-sensitivity on top of flow-sensitivity would significantly impact scalability, as is well known [26, 27]. Hence, we limited our approach to flow-sensitive but context-insensitive analysis.

2.2 Constraint Solving

We now present our novel constraint solving approach, which interleaves with the constraint generation, as shown in Fig. 3. Thus, the constraint solver may be invoked iteratively many times. In each iteration, it handles all the constraints that are generated in that iteration.

The solving algorithm that we propose is an extension of constraint solving algorithm of Oxhøj et al. [21]. Unlike their constraint solving algorithm, our algorithm needs to handle parametric conditional constraints. Further, as discussed in Section 2.1.2, we support a new type of constraint called FunctionCall constraint to avoid redundant constraints. These are specialized constraints that avoid creating redundant conditional constraints while processing function calls and instead help generate additional constraints, as and when required. For example, consider a function call `x.foo(...)`, where the static type of `x` is `X`, and classes `Y` and `Z` extend `X`, and also override the method `foo`. If, during the analysis, initially `x` points to an object of type `Y`, then we will add constraints to propagate the information from the call-site to the parameters of `Y.foo` and add `Y.foo` to a worklist of methods if `Y.foo` is not already processed. And while solving the constraints, if we find that another object of type `Z` also flows into `x` then we generate additional constraints to propagate the information from the call-site to the parameters of `Z.foo`, and add `Z.foo` to worklist if it is not already processed. (details explained in Section 2.2.3). Consequently, our constraint generation and solving procedures interleave (unlike a single pass approach of Oxhøj et al.). We also represent the flow sets as bit vectors for efficiency; though for ease of exposition, we use sets in the algorithm discussed in this section. Our solving algorithm is shown in Fig. 5.

2.2.1 Solving Member and Propagation Constraints.

We use the standard approach in solving the member and propagation constraints; we store the propagation relations in a graph *propDep* (see Lines 3-6 in Fig. 5).

2.2.2 Solving Conditional Constraint.

Unlike the traditional approach [21], since our conditional constraints are parametric in nature, our handling of these constraints differ. Consider a membership based conditional constraint $(\!|cond_r,\ C_r|\!)$ in which $cond_r$ is a membership condition of the form $r \in A$. We first store C_r in $condDep(A)$, which stores the constraints to be solved when a new member gets added to A; For every member M of the set A, we instantiate C_r by replacing the occurrences of r by M and then solve it (see Lines 8-11 in Fig. 5).

Consider a cardinality based conditional constraint of the form $(\!|cond,\ C, elseC|\!)$, we invoke the *solve* function on C or $elseC$, based on the truth value of *cond* (see Lines 13-14 in Fig. 5).

2.2.3 Solving FunctionCall Constraint.

Consider a FunctionCall constraint c of the form $[\![cond_r; S_c]\!]$, where S_c is a statement having a function call and $cond_r$ is of the form $r \in A$ (assuming A is the points-to set of the receiver in the function call). Similar to the solving of conditional constraints, we first store S_c in $funcDep(A)$, which stores the call-statements that need to be revisited when a new member gets added to A. For each M in A, we invoke *handleMethod* (see Lines 15-18).

The *handleMethod* function first identifies the method to be called for the receiver of type M. It then generates the following constraints to handle the impact of the call (see Lines 15-18). We show below the flow-sensitive constraints. The corresponding flow-insensitive constraints can be intuitively derived. Say, S_c is of the form $k = x.bar(...)$, and L_1 is the first statement in $M : bar$, and $func(S_c)$ be *foo* (call statement).

① Argument copy: $(\!|v \in varPts(S_c, x),\ [\![varPts(S_c, a_i) \subseteq varPts(L_1, p_i)]\!]|\!), \forall i \in \{0, ..., n\}$, where $\{p_0, p_2, .., p_n\}$ are the formal arguments of function *bar* in M.

```
1  Function solve(c) : // Solves the input constraint c and any
     other constraints generated in the process
2    switch type of c do
3      case Member constraint of the form ⟦M ∈ A⟧ do
         propagate(M, A) ;
4      case Propagation constraint of the form ⟦A ⊆ B⟧ do
5        add edge A ⟶ B to propDep;
6        foreach M ∈ A do propagate(M, B) ;
7      case Conditional constraint of the form ⟪cond_r, C_r⟫
         do
8        add C_r into condDep(A);
9        foreach M ∈ A do
10         C' = C_r[r/M]; // Replace free
             occurrences of r with M
11         solve(C');
12     case Conditional constraint of the form
         ⟪cond, C, [elseC]⟫ do
13       if cond evaluates to true then solve(C);
14       else solve(elseC) ;
15     case FunctionCall constraint of the form ⟦cond_r, S_c⟧
         do
16       Say cond_r is of the form ⟦r ∈ A⟧;
17       Add S_c to funcDep(A);
18       foreach M ∈ classesOf(A) /*classesOf(X)
           returns the types of each object in A*/ do
           handleMethod(M, S_c) ;
19 Function handleMethod(M, S_c) : //M is the receiver type
     for the function call in S_c
20   Identify the function f that is being called at S_c, based
       on M;
21   Generate the inter-procedural transfer constraints;
22   if f has not already been processed then
       workList.add(f) ;
23 Function propagate(M, S) :
24   if M ∉ S then
25     add M to S;
26     foreach edge S ⟶ B ∈ propDep do
27       propagate(M, B);  // propagate along the
           edge
28     foreach constraint C_r ∈ condDep(S) do
29       C' = C_r[r/M]; // Replace free
           occurrences of r with M
30       solve(C');
31     foreach call-statement S_c ∈ funcDep(S) do
         handleMethod(M, S_c) ;
```

Figure 5. Solve the generated constraints. Grey text indicates the common part with traditional solving scheme [21]

Type	Statement(s)	Constraints

Member	`4: A a=new A()`
	`5: A b=new B()`
	`6: A c=new A()`
	`10: a=new A()`
	`13 a=new B()`
		⟦04 ∈ *varPts*(a)⟧
		⟦05 ∈ *varPts*(b)⟧
		⟦06 ∈ *varPts*(c)⟧
		⟦010 ∈ *varPts*(a)⟧
		⟦013 ∈ *varPts*(a)⟧

Prop.	`15: B r=(B)a`	⟦*varPts*(a) ⊆ *varPts*(r)⟧
Cond.	`7: a.f=b` `8: c.f=a`	⟪*o* ∈ *varPts*(a), *varPts*(b) ⊆ *fieldPts*(*o*.f)⟫ ⟪*o* ∈ *varPts*(c), *varPts*(a) ⊆ *fieldPts*(*o*.f)⟫
FuncCall	`16: r.foo()`	⟦*o* ∈ *varPts*(r); r.foo()⟧

Figure 6. Generated Constraints for various statements of `main` method of code snippet in Fig. 1

② Copy the return value: ⟪$s \in succ(S_c)$, ⟦$retVal(M : bar) \subseteq varPts(s, k)$⟧⟫. Assume $succ(S_c)$ gives the control-flow successors of S_c.

③ Propagate *fieldPts* from S_c to L_1 and from the return statement to the CFG successor of S_c: constraints skipped for brevity.

After generating the above constraints, we add the method to the work-list if it is not already processed. This way, a method body is processed only once, that too only when needed. When an already processed method from a different call site is encountered only the above shown constraints are generated, but the method is not added to the worklist.

2.2.4 Details of the Function *propagate*.

The *propagate* function adds the element M to the set S, if not already present and propagates it through the edges in *propDep* graph whose source is S. The function also solves the pending constraints (from conditional constraints), if any for S. Similarly, it also handles any call-statements Here, we perform type-based filtering [15]. Before adding a reference M to the points-to set S of a variable v (at Line 25 of Fig. 5), we check if the type of M is compatible with the static type of v as per the class hierarchy information [8]. If the check fails, we return from the propagate function without performing any of the computation.

3 Example Run

We use the code snippet shown in Fig. 1 to illustrate the working of *PInter* while performing a flow-insensitive points-to analysis. *PInter* first adds the `main` method to worklist (see algorithm in Fig. 2). The `main` method is picked from worklist and constraints are generated for the statements in it. For the allocation statements in Lines 4,5,6,10, and 13, member constraints will be generated. For the store statements in Lines 7,8, conditional constraints are generated with condition on membership of points-to set of base variables `a,c` respectively. For the copy statement in Line 15, propagation constraint is generated, and for the virtual method call in

Line 16, a FunctionCall constraint is generated. These constraints are all shown in Fig. 6. Now, the constraint solving starts as the worklist is empty. Constraints can be solved in any order, yielding the same final result. We illustrate the solving of four specific constraints:

Assume that initially the conditional constraint for the store statement `a.f = b` at Line 7 is solved: The dependency on the membership of `a` is stored in *condDep*(*varPts*(`a`)). Since, the *varPts*(`a`) is empty, the dependent constraint is not solved.

Next, solving member constraints for the allocation statements in Lines 4,10,13 results in *varPts*(`a`) = {`04`, `010`, `013`}. Solving of propagation constraint for the copy statement `B r = (B)a` at Line 15 includes first storing the propagation dependency *varPts*(`a`) → *varPts*(`r`) to *propDep* and then adding elements already present in *varPts*(`a`) to *varPts*(`r`) after type based filtering. This results in *varPts*(`r`) = {`013`} because `04`, `010` are not of type `B`.

Next, we will see solving of functioncall constraint for `r.foo()` at Line 16. Here, the dependency is first stored in *funcDep*(*varPts*(`r`)), and then the *handleMethod* function is called for already existing elements in *varPts*(`r`). Since *varPts*(`r`) = {`013`}, it results in the *handleMethod* function resolving `r.foo()` to `B :: foo`, and `B :: foo` getting added to worklist. This constraint generation and solving for all methods in the worklist continues until no more methods are present in the worklist.

4 Correctness

Our constraint solving algorithm is an adaption of the traditional constraint solving algorithm of Oxhøj et al. [21]. However, in contrast to Oxhøj et al., we propose and use a scheme of interleaved constraint generation and solving. We introduced Conditional and FunctionCall constraints which are parametric in nature. So it suffices to show the correctness of Conditional and FunctionCall constraints as the Member and Propagation constraints are traditional.

It is easy to see that compared to Oxhøj et al., we may generate fewer conditional constraints (as we generate parametric constraints). But during the solving phase of Oxhøj et al., if the predicate of any conditional constraint is satisfied and they process any new constraint C, then our approach would also process C. A similar argument holds for constraints resulting from virtual function calls. We present this argument as the following two theorems and a lemma to prove the correctness of cardinality based conditional constraints.

Theorem 4.1. *Consider a sequence of traditional conditional constraints of the form* $O_1 \in A \Rightarrow C_{O_1}$, $O_2 \in A \Rightarrow C_{O_2}$, $\ldots O_n \in A \Rightarrow C_{O_n}$, *and the corresponding conditional constraint generated by us of the form* $[\![r \in A, C_r]\!]$. *Note the constraint* C_{O_i} *may refer to* O_i. *If the traditional solver evaluates one of the predicates (say,* $O_x \in A$*) as true and processes* C_{O_x} *then* C_{O_x} *will also be processed by our solving algorithm.*

Proof. (*Sketch*) The proof is straightforward, as during the solving phase, when our solver finds that $O_x \in A$, then it will process C_{O_x}. We now detail the proof.

Say, we have a sequence *Seq* of membership and inclusion based conditional constraints in the context of traditional constraints. For ease of explanation, we will only focus on processing the conditional constraints. We will prove the theorem by induction on the number n of processed conditional constraints in *Seq*.

Base case. Say, we have processed zero conditional constraints ($n = 0$), then there is nothing to disagree upon.

Induction hypothesis. Assume that we have processed $n = k$ number of conditional constraints, and both the solving schemes have processed the same set of inclusion based constraints.

Further, say, a constraint C_{O_1} has been processed by the traditional solver, as part of processing the conditional constraint $O_1 \in A \Rightarrow C_{O_1}$, when O_1 was found as a member of A. Then our proposed scheme would have processed C_{O_1}, as part of solving the conditional constraint $[\![r \in A, C_r]\!]$. A consequence of the induction hypothesis is that the solution obtained so far is matching between the traditional and our proposed scheme.

Induction step. We will now show (by contradiction) that if the induction hypothesis holds, then the theorem will hold for processing $k + 1$ conditional constraints. Say, the $k + 1^{th}$ constraint is of the form $O_2 \in A \Rightarrow C_{O_2}$ and the traditional scheme evaluated the condition ($O_2 \in A$) to be true and processed the constraint C_{O_2}. Assume that our proposed system did not evaluate C_{O_2}. It implies that our system evaluated $O_2 \in A$ to be false, at the end of solving the first k conditional constraints. This is a contradiction to the induction hypothesis. Hence proved.

□

Theorem 4.2. *Consider a set* X *of conditional constraints generated by the traditional constraint generation scheme, for a call-site* S_c *of the form: x.bar(args). Each of these conditional constraints are predicated by the possible type of the receiver object. Let us consider a conditional constraint* $C_i \in X$, *of the form* $c \Rightarrow C$, *where* c *is of the form* $A \in classesOf(varPts(S_c, x))$, *and* C *represents a constraint for handling the flow between the actual/formal parameters and the return value. Assume that the FunctionCall constraint generated by our scheme is given by* $[\![o \in varPts(S_c, x); S_c]\!]$. *If the traditional solver identifies* c *to be true and processes* C, *then our solver will also process* C.

Proof. The proof is again straightforward, as during the solving phase, when our solver finds that A is a member of $classesOf(varPts(S_c, x))$ (Line 18, Fig. 5), then it will generate and process all the constraints relevant for the function

call (see the method *handleMethod*, in Fig. 5); these will include C. The proof logic for this theorem follows that of Theorem 4.1 and is skipped for brevity. □

Lemma 4.3. *Consider a cardinality based conditional constraint of the form* $(\!|cond, C, elseC|\!)$. *Say, the target set updated by* C *and* $elseC$ *is* Y. *There is a guarantee that the computed value of* Y *will be consistent with value of cond in the final solution. That is, say in the final solution the value of cond is true. Then the computed value of* Y *is same as that obtained after evaluating* C *(without having to evaluate* $elseC$*). Say, in the final solution the value of cond is false. Then the computed value of* Y *is same as that obtained after evaluating* $elseC$ *(without having to evaluate* C*).*

Proof. (*Sketch*)

Our design ensures that *cond* may only be of the form $|S == 1|$. During the solving phase, depending on the order of evaluation of constraints, the predicate value of *cond* may change, and hence both the constraints may be evaluated.

During the solving phase of *PInter* it is possible that the solver may first observe *cond* to be true, and later observe it to be false; but never the other way around. Thus, it is possible that the solver evaluates C, followed by $elseC$, or only $elseC$; but never $elseC$, followed by C. Also, note that both update the same target set as they follow the monotonic discipline(see Section 2.1.2).

We see two cases based on the initial value of *cond*:

1. *cond* first evaluates to false: Y will be computed based on $elseC$. Since *cond* can never become true, C will never be evaluated. And hence the set of elements of Y are consistent with the final value of *cond*.
2. *cond* first evaluates to true. Y will be computed based on C. Now two subcases can occur:
   a. *cond* remains true till the end: set of elements of Y are consistent with the final value of *cond*.
   b. *cond*'s values changes to false: Y will be updated based on $elseC$. But since C and $elseC$ follow monotonic discipline (see Section 2.1.2), the evaluation of $elseC$ will add all the elements added by C and may be more. Thus, the set of elements of Y are consistent with the final value of *cond*. Note that the value of *cond* cannot turn back to true.

□

5 Implementation and Evaluation

We implemented different points-to analyses using the *PInter* constraint system within the Soot framework [29]. To evaluate different facets of our approach, we developed four variants of points-to analyses: (i) *fiPoA*, a flow-insensitive points-to analysis; (ii) *nfiPoA*, which extends *fiPoA* with traditional non-interleaved constraint solving [21]; (iii) *fsPoA*, a flow-sensitive points-to analysis; and (iv) *nfsPoA*, a flow-sensitive analysis employing non-interleaved constraint solving. To obtain non-interleaved solving, we modified the *solve* algorithm (Fig. 5) to use traditional non-parametric constraints in place of parametric Conditional and FunctionCall constraints. For example, a FunctionCall constraint is added for each callable method at a call-site based on the static type information of the receiver. Additionally, since the benchmarks used exhibit parallelism, we incorporate pre-computed concurrency information to perform flow-sensitive heap updates in both *fsPoA* and *nfsPoA* analyses. In addition to the constraints shown in Section 2.1.3, we also model arrays, exceptions in our implementation.

Our implementation consists of approximately 3K lines of Java code. We conducted an evaluation on a system equipped with an Intel Xeon Gold 5218 processor running at 2.3 GHz, 100 GB of memory, and Ubuntu 22.04.1 LTS. The benchmark set includes 12 programs selected from the DaCapo benchmark suite, specifically versions 9.12 (DaCapo-bach) and 23.11 (DaCapo-chopin) [4]. These benchmarks are listed in Fig. 7, along with some static characteristics.

Consistent with prior work [12, 28], we employed TamiFlex [5] to manage reflection in the benchmarks. Since, we use Tamiflex in our implementation, although our proposed method is sound, the reported results may be considered soundy [17], consistent with common practice in the literature. For both versions of the DaCapo suites, we excluded any benchmark that failed to execute properly with TamiFlex or Soot due to errors occurring at various points. We also compare our *fiPoA* results with the equivalent flow- and context-insensitive points-to analyses from Spark [15] in the Soot framework and Doop [6]. We analyzed only application methods and ran Spark and Doop in application-only mode to ensure fair comparison.

We conduct an evaluation aimed at addressing three key research questions.

RQ1: What is the impact of the proposed interleaved generation and solving of constraints? RQ2: How does *PInter* compare against the flow, context-insensitive points-to analysis Spark in Soot framework? RQ3: How does *PInter* compare against the flow, context-insensitive points-to analysis implemented in Doop, a state-of-the-art declarative points-to analysis framework?

(RQ1) What is the impact of the proposed interleaved generation and solving of constraints? We evaluated the execution times of *fiPoA* and *fsPoA* against their traditional non-interleaved counterparts *nfiPoA* and *nfsPoA*. The comparative plots are shown in Fig. 8, 9. Each bar in the plots further illustrates the breakdown of the total time into constraint generation and constraint solving components. We found that for *fiPoA* and *nfiPoA*, constraint generation accounts for 47% and 19% (geomean) of total time, respectively. The *nfsPoA* analysis did not terminate (given a timeout of 2 hours) for all benchmarks except avrora, sunflow, graphchi, zxing, jme. As illustrated in Fig. 8 and Fig. 9, *nfiPoA* and *nfsPoA* required substantially more time, whereas *fiPoA* and

S.No.	Benchmark			No. of Constraints				Memory (MB)			
	Name	Source	#Stmts	*nfiPoA*	*fiPoA*	*nfsPoA*	*fsPoA*	*nfiPoA*	*fiPoA*	*nfsPoA*	*fsPoA*
1	avrora	chopin	21,682	2,563,609	44,292	11,493,295	1,485,072	375	142	2754	558
2	xalan	chopin	44,859	10,143,700	12,316	-	141,017	912	210	-	244
3	sunflow	chopin	10,754	403,786	45,687	1,959,927	1,384,498	174	144	599	479
4	batik	chopin	37,057	3,994,959	35,884	-	607,658	699	280	-	416
5	graphchi	chopin	7972	447,470	23,134	851,003	714,566	160	124	263	240
6	zxing	chopin	17,137	785,033	21,340	2,077,367	316,083	221	206	720	298
7	jme	chopin	6294	175,043	22,188	341,781	292,102	245	240	312	305
8	lusearch	bach	10,777	790,798	39,695	-	790,593	174	109	-	304
9	luindex	bach	15,607	2,077,854	49,670	-	3,413,560	266	121	-	602
10	pmd	bach	31,091	3,183,130	20,829	-	339,377	513	163	-	259
11	fop	bach	80,204	30,395,639	25,670	-	275,642	2458	413	-	356
12	h2	bach	38,436	6,112,416	17,546	-	209,472	677	144	-	201

Figure 7. Comparison of number of generated constraints and memory for *nfiPoA*, *fiPoA*, *nfsPoA*, *fsPoA*. The "-" indicates the analysis did not terminate (timeout of 2 hours). "bach": DaCapo-bach, "chopin": DaCapo-chopin benchmark suites. #Stmts: total points-to related statements in reachable application methods of benchmark.

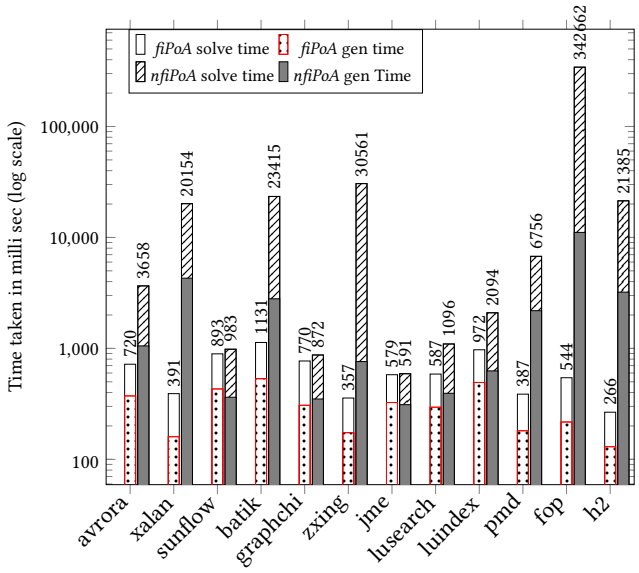

Figure 8. Time comparison of *nfiPoA* and *fiPoA*, showing the breakdown of time spent on constraint generation and constraint solving. Lower the better.

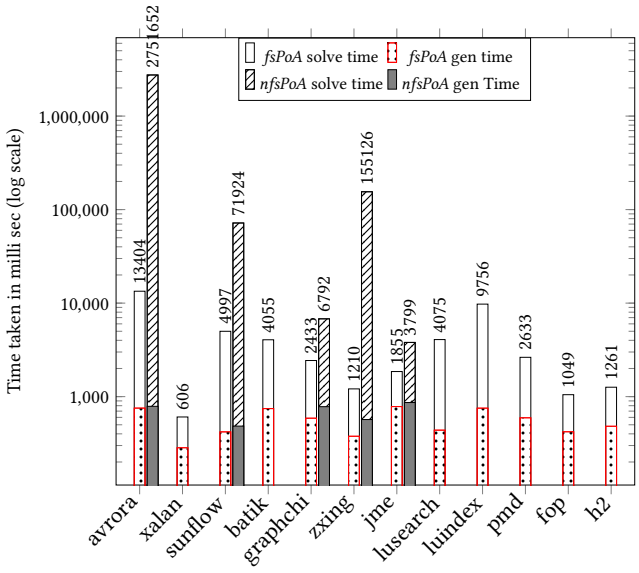

Figure 9. Time comparison of *nfsPoA* and *fsPoA*, showing the breakdown of time spent on constraint generation and constraint solving. Lower the better

fsPoA achieved notable speedups—43% and 78% (geomean) reductions in runtime, respectively. These improvements stem primarily from two factors: (i) reduced execution time due to the generation of fewer constraints and (ii) lower processing overhead from handling fewer constraints. Since the contribution of the second factor is harder to quantify, our discussion focuses on the first.

In Fig. 7, we list the number of generated constraints and the memory consumed by *nfiPoA*, *fiPoA*, *nfsPoA*, *fsPoA*. Overall, even though the number of constraints have some relation to the number of statements (column 4), the actual correspondence is difficult to establish. The number of constraints depend on the number of points-to analysis related statements, and their interactions in the input program.

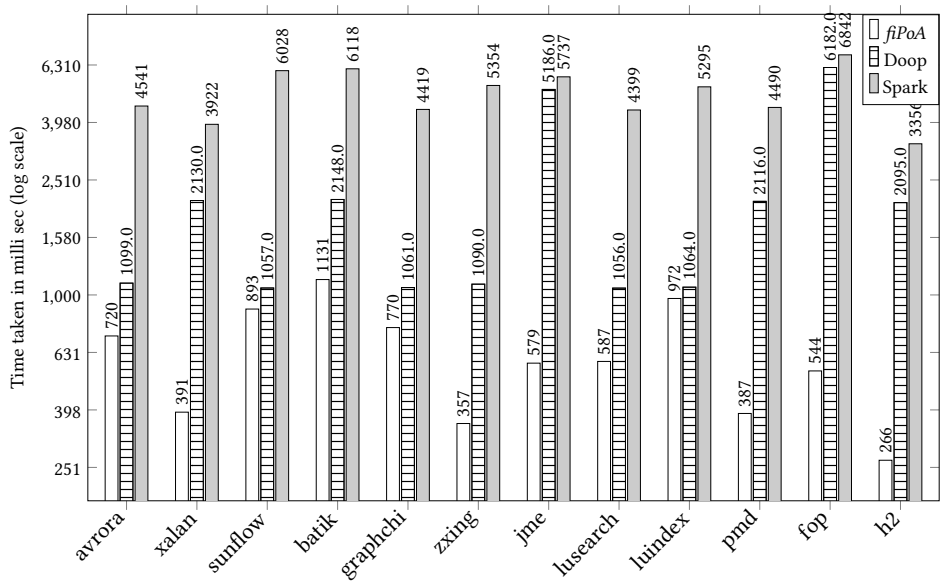

Figure 10. Time comparison of *fiPoA*, Spark and Doop; lower the better.

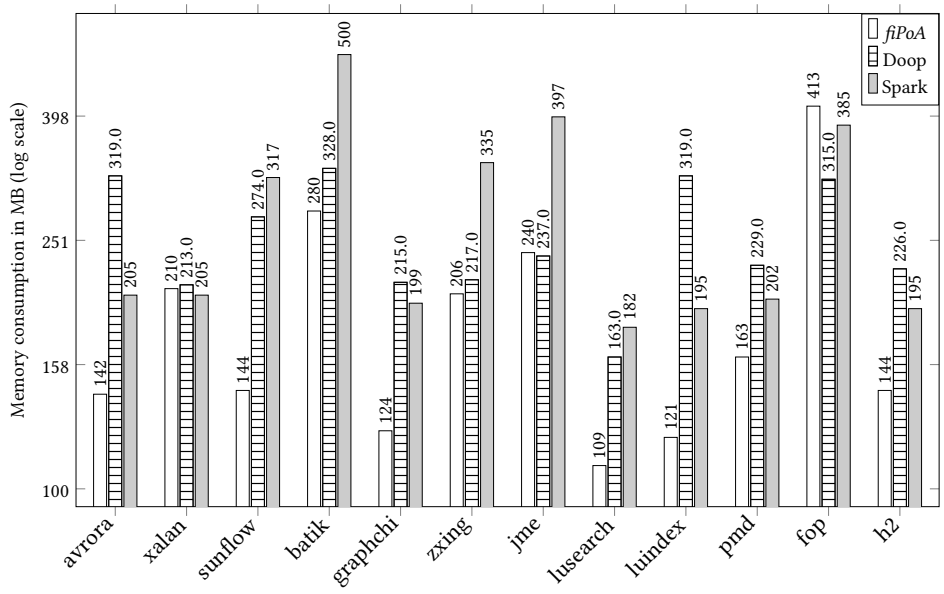

Figure 11. Memory comparison of *fiPoA*, Spark and Doop; lower the better.

Fig. 7 reveals that *fiPoA* produces significantly fewer constraints (96% less geomean) than *nfiPoA*, while *fsPoA* generates 35% (geomean) fewer constraints than *nfsPoA*. This reduction is particularly evident for the larger of the benchmarks such as fop, xalan, h2, pmd, and batik.

(RQ2) How does *PInter* compare against the Spark tool in Soot framework?

To show the impact of the proposed scheme *PInter*, in Fig. 10, we additionally provide a comparison between *fiPoA* and the flow-insensitive, context-insensitive points-to analysis Spark (with on-the-fly call graph, field-sensitive, type-based filtering enabled). As shown in Fig. 10, in terms of execution time, we find that *fiPoA* performs better than Spark for all benchmarks. Across all the benchmarks, the geomean of improvement was 87%. This attests to the benefit of our proposed interleaved constraint generation and solving scheme.

As an academic study, we used GC logs to get an approximate understanding of the memory usage and show the plot in Fig. 11. Since this measurement is not precise, we can only draw conclusions confidently where the difference

is significant. We find that *fiPoA* clearly takes less memory than Spark for all the benchmarks except fop, xalan. For these benchmarks, the memory usage is comparable.

(RQ3) How does *PInter* compare against the flow-insensitive, points-to analysis implemented in Doop? Fig. 10 also shows the comparison of execution times of *fiPoA* and Doop's flow-, and context-insensitive points-to analysis.

As observed, *fiPoA* consistently outperforms Doop in terms of execution time. Across all the benchmarks, the geomean of improvement was 47%. We believe that the performance gap arises because Doop generates many facts eagerly upfront during preprocessing, before beginning its iterative solving phase.

Similar to RQ2, here also we did a study of the memory usage. We find that while *PInter* took clearly less memory for eight out of 12 benchmarks, Doop is a clear winner for one (fop).

Note that we included this additional comparison with Doop only to indicate how our approach positions itself relative to a state-of-the-art framework. It would be an interesting future work to perform a user study that aims to implement a wide set of analyses (already available in Doop) in PInter, and assess the programmability and performance trade-offs between the two.

Summary: *PInter* significantly enhances the efficiency of points-to analysis. The interleaved analyses (*fiPoA*, *fsPoA*) achieve geometric mean speedups of 43% and 78%; generate 96% and 35% less number of constraints over their non-interleaved counterparts. The *fiPoA* analysis consistently outperforms established frameworks Spark (by 87%) and Doop (by 47%) in execution time (geometric means) and uses less memory across most benchmarks.

6 Related Work

Constraint-based analyses form the basis for many points-to and type inference analysis in languages like C, Java, and JavaScript. Andersen [1] introduced flow-insensitive, context-sensitive points-to analysis for C using inclusion constraints. Oxhøj et al. [21] proposed a type inference algorithm for untyped object-oriented programs, modeling type flows as inclusion constraints solved incrementally, inspiring our constraint solving scheme. For Java, Spark [15]—integrated into Soot [29]—constructs a Points-to Analysis Graph (PAG) with constraints for all potential method targets upfront, but its variable-based heap edges require reprocessing load/store constraints, increasing overhead. Paddle [16], another points-to analysis [29] in Soot, employs binary decision diagrams (BDDs) to represent points-to sets compactly, excelling in scalable, context-sensitive analyses for large programs. While it generates fully on-the-fly call-graph, the efficiency of BDD solving is very sensitive to optimizations like variable ordering. Rountev et al. [23] extend Andersen's analysis with annotated inclusion constraints, generating constraints for reachable methods, yet use fresh variables for heap operations that enlarge the graph and apply type-based filtering post-analysis, yielding larger points-to sets. Doop [6] a declarative Datalog-based framework, does eager fact generation during preprocessing and has heavy join-based evaluation (using optimized engines like Soufflé [2] or LogicBlox [3]) during resolution. Although the framework is built for generality and extensibility it incurs higher upfront costs and can be resource-intensive for lightweight points-to analysis tasks. For JavaScript, Hackett and Guo [10] use subset constraints with runtime type barriers to address polymorphism, achieving up to 50% performance gains in Firefox's JIT compiler, but rely on dynamic checks, unlike our static approach. Other popular optimizations in inclusion-based points-to analysis for C – by Hardekopf and Lin [11], Pereira and Berlin [22], and Heintze and Tardieu [13] – include lazy cycle detection, wave propagation, and online cycle elimination. These orthogonal techniques could further enhance the efficiency of our constraint-solving approach.

7 Conclusion

In this work, we introduced several improvements to make constraint-based analyses for performing points-to analysis of object-oriented languages more efficient and scalable. Our compact representation of Andersen-style conditional constraints helps cut down on memory usage and speeds up solving. To better handle the complexity of virtual method calls, we proposed a new "FunctionCall" constraint that avoids the usual explosion of redundant constraints caused by method overriding. Our interleaved constraint generation and solving approach, *PInter*, ensures that only essential constraints are generated and resolved, leading to more efficient analysis. We implemented two Java points-to analyses using *PInter* within the Soot framework and evaluated them on applications from the DaCapo benchmark suite. As is standard, we used Tamiflex to handle dynamic features of Java benchmarks to perform a soundy evaluation. Finally, our evaluation demonstrates that our approach not only reduces the number of constraints and analysis time significantly but also offers a viable and practical alternative to existing tools like Spark.

Acknowledgments

This work is partially supported by the SERB CRG grant (sanction number CRG/2022/006971).

8 Data Availability Statement

Data supporting the findings of this study are openly available in Figshare [14].

References

[1] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. Datalogisk Institut,

Københavns Universitet.

[2] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting doop to Soufflé: a tale of inter-engine portability for Datalog-based analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Barcelona, Spain) *(SOAP 2017)*. Association for Computing Machinery, New York, NY, USA, 25–30. doi:10.1145/3088515.3088522

[3] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1371–1382. doi:10.1145/2723372.2742796

[4] 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 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. doi:10.1145/1167473.1167488

[5] 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 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 241–250. doi:10.1145/1985793.1985827

[6] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 243–262. doi:10.1145/1640089.1640108

[7] Yingxia Cui, Longshu Li, and Sheng Yao. 2009. Inclusion-Based Multi-level Pointer Analysis. In *2009 International Conference on Artificial Intelligence and Computational Intelligence*, Vol. 2. 204–208. doi:10.1109/AICI.2009.157

[8] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, Mario Tokoro and Remo Pareschi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 77–101.

[9] Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. 2017. Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 329–340. doi:10.1145/3092703.3092729

[10] Brian Hackett and Shu yu Guo. 2012. Fast and Precise Hybrid Type Inference for JavaScript. *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 239–250. doi:10.1145/2254064.2254094

[11] Ben Hardekopf and Calvin Lin. 2007. The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 290–299. doi:10.1145/1250734.1250767

[12] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. 2023. Selecting Context-Sensitivity Modularly for Accelerating Object-Sensitive Pointer Analysis. *IEEE Transactions on Software Engineering* 49, 2 (Feb 2023), 719–742. doi:10.1109/TSE.2022.3162236

[13] Nevin Heintze and Olivier Tardieu. 2001. Ultra-fast Aliasing Analysis Using CLA: A Million Lines of Code in a Second. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 254–263. doi:10.1145/378795.378851

[14] Ramya Kasaraneni and V. Krishna Nandivada. 2026. Compact Representation and Interleaved Solving for Scalable Constraint-Based Points-to Analysis - artifact. https://doi.org/10.6084/m9.figshare.30925898.v3

[15] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using Spark. In *Compiler Construction*, Görel Hedin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 153–169.

[16] Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.* 18, 1, Article 3 (Oct. 2008), 53 pages. doi:10.1145/1391984.1391987

[17] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. doi:10.1145/2644805

[18] V. Benjamin Livshits and Monica S. Lam. 2003. Tracking pointers with path and context sensitivity for bug detection in C programs. *SIGSOFT Softw. Eng. Notes* 28, 5 (Sept. 2003), 317–326. doi:10.1145/949952.940114

[19] Ravichandhran Madhavan and Raghavan Komondoor. 2011. Null dereference verification via over-approximated weakest pre-conditions analysis. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 1033–1052. doi:10.1145/2048066.2048144

[20] Steven S. Muchnick. 1998. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[21] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. 1992. Making type inference practical. In *ECOOP '92 European Conference on Object-Oriented Programming*, Ole Lehrmann Madsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 329–349.

[22] Fernando Magno Quintao Pereira and Daniel Berlin. 2009. Wave Propagation and Deep Propagation for Pointer Analysis. *2009 International Symposium on Code Generation and Optimization*, 126–135. doi:10.1109/CGO.2009.9

[23] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. 2001. Points-to Analysis for Java Using Annotated Constraints. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Tampa Bay, FL, USA) *(OOPSLA '01)*. Association for Computing Machinery, New York, NY, USA, 43–55. doi:10.1145/504282.504286

[24] Erik Ruf. 2000. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 208–218. doi:10.1145/349299.349327

[25] Radu Rugina and Martin Rinard. 1999. Pointer Analysis for Multithreaded Programs. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '99)*. Association for Computing Machinery, New York, NY, USA, 77–90. doi:10.1145/301618.301645

[26] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (apr 2015), 1–69. doi:10.1561/2500000014

[27] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 17–30. doi:10.1145/1926385.1926390

[28] Manas Thakur and V. Krishna Nandivada. 2020. Mix Your Contexts Well: Opportunities Unleashed by Recent Advances in Scaling Context-Sensitivity. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/3377555.3377902

[29] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13.

Received 2025-11-12; accepted 2025-12-10