PIP: Making Andersen's Points-to Analysis Sound and Practical for Incomplete C Programs

Håvard Rognebakke Krogstie
Norwegian University of Science and Technology
Trondheim, Norway
havard.r.krogstie@ntnu.no

Helge Bahmann
Independent Researcher
Zürich, Switzerland
hcb@chaoticmind.net

Magnus Själander
Norwegian University of Science and Technology
Trondheim, Norway
magnus.sjalander@ntnu.no

Nico Reissmann
Independent Researcher
Trondheim, Norway
nico.reissmann@gmail.com

***Abstract*—Compiling files individually lends itself well to parallelization, but forces the compiler to operate on incomplete programs. State-of-the-art points-to analyses guarantee sound solutions only for complete programs, requiring summary functions to describe any missing program parts. Summary functions are rarely available in production compilers, however, where soundness and efficiency are non-negotiable. This paper presents an Andersen-style points-to analysis that efficiently produces sound solutions for incomplete C programs. The analysis accomplishes soundness by tracking memory locations and pointers that are accessible from external modules, and efficiency by performing this tracking implicitly in the constraint graph. We show that implicit pointee tracking makes the constraint solver 15× faster than any combination of five different state-of-the-art techniques using explicit pointee tracking. We also present the Prefer Implicit Pointees (PIP) technique that further reduces the use of explicit pointees. PIP gives an additional speedup of 1.9×, compared to the fastest solver configuration not benefiting from PIP. The precision of the analysis is evaluated in terms of an alias-analysis client, where it reduces the number of `MayAlias`-responses by 40% compared to LLVM's BasicAA pass alone. Finally, we show that the analysis is scalable in terms of memory, making it suitable for optimizing compilers in practice.**

***Index Terms*—Static analysis, points-to analysis, partial analysis.**

I. Introduction

Alias information is a prerequisite for program analyses and transformations that are essential in optimizing compilers, program verification, and program comprehension tools. Important compiler transformations, such as loop invariant code motion, dead load and store elimination [1], loop versioning [2], vectorization [3], etc., rely on alias information to be effective. A points-to analysis provides sets of possible targets for pointers, and can form the basis for an alias analysis, in addition to other analyses like call graph and mod/ref summary creation [4]. Ideally, a points-to analysis should *efficiently* provide *precise* information, but a balance must be found in practice, as precise points-to analysis is undecidable [5]. Practical and scalable implementations therefore approximate the exact solution, and a plethora of trade-offs between precision and performance exist [4, 6–12].

```
1  static int x, y;
2  int z;
3  extern int* getPtr();
4  int* p = &x;
5
6  void callMe(int* q) {
7      int w;
8      int* r = getPtr();
9      if (r == NULL)
10         r = &w;
11 }
```

Fig. 1. Example of an incomplete program with pointers `p`, `q`, and `r`, all of which may point to unknown targets from external modules.

These trade-offs receive a lot of attention in the literature, where their scalability is shown by performing whole-program analysis. However, even if all optimizations are deferred to link time, which is the exception rather than the norm, very few programs outside the domain of embedded systems are complete. On the contrary, the majority of programs are incomplete, i.e., programs with unknown external functions, as they depend on external libraries and/or system calls [4, 13].

The problem is illustrated in Figure 1. In this incomplete program, the pointers `q` and `r` originate from outside the current module, making it impossible for an analysis to determine the origins of the pointer values. Nevertheless, it is possible to infer some facts about their targets. Our analysis correctly infers that `p`, `q` and `r` may target `x`, `z`, or any memory object defined in external modules, but never `y`. Only `r` may target `w`.

In contrast, other analyses require the presence of summary functions to render the program in Figure 1 complete [14–16], or otherwise produce an unsound solution [6, 17, 18]. While unsound analyses might be acceptable for some tools or clients, such as in bug detection or refactoring assistance, optimizing compilers can not tolerate unsoundness [19]. Summary functions are not always available, such as when using external libraries, and can at most be an optional tool for improving the precision of common library functions.

979-8-3315-9288-2/26 © 2026 IEEE

In this paper, we present an Andersen-style points-to analysis [20] that produces sound solutions for all well-formed incomplete C programs adhering to the standard provenance-aware memory model [21]. Our key insight is that a sound solution can be accomplished by tracking which memory locations and pointers are accessible to external modules. Pointers with external origins may only target these externally accessible memory locations, avoiding the need for an overly conservative "Unknown"-flag seen in other analyses [4, 20]. Importantly, a pointer value of unknown origin may not target any memory locations from the current module that have not escaped. Our efficient implementation relies on six new constraint types, which enable us to implicitly represent externally accessible memory locations in the constraint graph, significantly reducing solver runtime. We also present the *Prefer Implicit Pointees* (PIP) technique that further reduces the use of explicit pointees by detecting situations where explicit pointees do not affect the final solution.

The results from running our analysis on a set of 3 659 C files from nine SPEC CPU2017 [22] benchmarks and four open-source programs show that the implicit representation of pointees is the most important factor for efficient solving. We achieve a total speedup of 15× over an oracle that always chooses the fastest configuration with explicit representation. Enabling PIP in addition to representing pointees implicitly gives an additional 1.9× speedup. The evaluation includes five additional solver techniques from the literature [14, 16, 23]. In the context of individual file analysis, none of them improve upon PIP on average. Finally, the precision of the analysis results is evaluated in terms of an alias-analysis client. Compared to using a local IR-traversing alias analysis, LLVM's BasicAA, incorporating information from the points-to graph reduces the number of `MayAlias`-responses by 40% on average. The results show that our sound points-to analysis can provide additional alias information at scale, making it practical for production compilers.

II. Background

A points-to analysis takes all pointers and memory objects in a program, and produces a points-to set $\mathrm{Sol}(p)$ for each pointer p. The solution is *sound* if $\mathrm{Sol}(p)$ contains all memory objects p may ever target at runtime in all possible executions of the program. $\mathrm{Sol}(p)$ may also contain *spurious* memory objects that p never targets, which make the solution less *precise* [24]. An efficient analysis typically requires trading precision for scalability, and multiple trade-offs exist. A *flow-insensitive* analysis ignores control flow, and treats the program as an unordered set of statements [14]. A *context-insensitive* analysis unifies arguments and return values between all calls to a function [13]. A *field-insensitive* analysis represents all fields of an aggregate memory object using a single points-to set [25].

Points-to analyses can typically be categorized as either *unification-based* or *inclusion-based* [13], where the latter models the flow of points-to sets directionally, giving more precise solutions. Analyses that are inclusion-based and flow-insensitive are commonly referred to as *Andersen-style* [13], and are the topic of this paper. We consider a field- and context-insensitive implementation.

TABLE I
Constraint types in an Andersen-style constraint language, and examples of C statements they represent.

Name	Constraint	Corresponding C code
Base	$p \supseteq \{a\}$	`p = &a;`
Simple	$p \supseteq q$	`p = q;`
Load	$p \supseteq *q$	`p = *q;`
Store	$*p \supseteq q$	`*p = q;`
Function	$\mathrm{Func}(f, r, a_1, ..., a_n)$	`void* f(a1, ..., an) { ...; return r; }`
Function call	$\mathrm{Call}(f, r, a_1, ..., a_n)$	`r = (*f)(a1, ..., an)`

A. Building Constraint Sets

In the first phase of an Andersen-style analysis, the program is converted into a finite set of abstract memory locations M, pointers P, and constraints C. The constraints model all possible ways in which pointers may be given pointees, such as taking the address of memory objects, copying pointer values, loading and storing pointers, and passing pointers into and out of function calls. We define a constraint language to represent these constraints, shown in Table I. The first four constraints are identical to the ones used by Pearce [14]. Function calls are handled similarly to Foster's lam construct [26], and support indirect function calls.

The exact process of converting the source program into constraints depends on the program representation. Modern intermediate representations, like LLVM IR, make a distinction between variables stored in memory and temporary values stored in *virtual registers* [27]. Registers can not be pointed to, so the analysis only needs to consider them if their type is *pointer compatible*. Types that are not pointer compatible do not have points-to sets, and can be ignored by the analysis [4]. In this work, we consider a type to be pointer-compatible if it is a pointer or an aggregate type containing a pointer. The set of pointers P includes all pointer-compatible virtual registers.

During runtime, the program may allocate arbitrarily many memory objects, which must be represented by a finite set of abstract memory locations M. Named memory objects, such as local and global variables, functions, and imported symbols, are represented by distinct abstract memory locations. Heap allocations are named after their allocation site, using distinct abstract memory locations to represent the memory objects allocated at each site. Thus, each memory object in the program is represented by one, not necessarily unique, abstract memory location. If an abstract memory location may represent values of pointer-compatible types, it is included in both P and M. The universe of constraint variables is denoted $V = P \cup M$. The $\mathrm{Func}(\ldots)$ and $\mathrm{Call}(\ldots)$ constraints ignore return values and arguments that are not pointer compatible.

TRANS	LOAD	STORE	CALL
			$h \supseteq \{f\}$
$q \supseteq \{x\}$	$q \supseteq \{x\}$	$p \supseteq \{x\}$	$\text{Func}(f, r_\bullet, a_{1\bullet}, \cdots, a_{n\bullet})$
$p \supseteq q$	$p \supseteq *q$	$*p \supseteq q$	$\text{Call}(h, r, a_1, \cdots, a_n)$
$p \supseteq \{x\}$	$p \supseteq x$	$x \supseteq q$	$r \supseteq r_\bullet$
			$a_{i\bullet} \supseteq a_i, \forall i \in \{1, \ldots, n\}$

Fig. 2. Rules of inference for sound points-to set tracking

B. Solving Constraints

The second analysis phase takes the sets M, P, and C and solves the constraints to produce the final solution $\text{Sol} : P \to \mathcal{P}(M)$. To formalize how the constraints propagate points-to sets, a system of inference rules is used. The rules are shown in Figure 2, and mostly correspond to those of Pearce et al. [25]. The rules are designed to have the following property: "If it is possible for a pointer p to point to a memory location x at runtime, it is also possible to infer the constraint $p \supseteq \{x\}$" [25]. The analysis solution can thus be defined as:

$$\text{Sol}(p) \coloneqq \{x \in M \mid p \supseteq \{x\} \text{ can be inferred from } C\}.$$

The set of constraints can be formulated as a *constraint graph*, as presented by Heintze and Tardieu [28]. The technique has been refined and visualized in several works [11, 14, 16, 25]. In the constraint graph, the variables in V are drawn as nodes, while constraints are drawn as edges and labels. Nodes representing virtual registers are drawn as circles, while abstract memory locations are squares. The node representing a pointer $p \in P$ includes a braced list containing the targets of all its base constraints ($p \supseteq \{x\}$). This list shows the current approximation of $\text{Sol}(p)$. Simple constraints ($p \supseteq q$) are drawn as simple edges $q \longrightarrow p$. Load and store constraints are drawn as the complex edges $\circ\!\!\dashrightarrow$ and $\dashrightarrow\!\!\circ$, respectively, where the circle indicates the side being dereferenced. Figure 3 shows an example of constraint variables and constraints, and the corresponding constraint graph.

The inference rules from Figure 2 can all be represented as operations on the constraint graph, using existing constraints to infer new ones. The TRANS rule corresponds to propagating the contents of Sol sets along simple edges. The LOAD and STORE rules correspond to converting complex edges into simple edges by dereferencing the end with the circle. Figure 4 shows inferring all possible constraints starting with the set in Figure 3, by using the constraint graph.

The constraint graph can be extended to include function calls by assigning each Func and Call constraint a unique number, and adding labels to nodes representing functions, call targets, arguments, and return values. The CALL inference rule can be applied when a node labeled Call_i has a node labeled Func_j in its Sol set. Applying the rule adds simple edges representing the flow of possible pointees through arguments $\text{Call}_i\, a_n \to \text{Func}_j\, a_n$ and through the return value $\text{Func}_j\, r \to \text{Call}_i\, r$. Figure 5 shows a C program with two functions and the corresponding constraint graph.

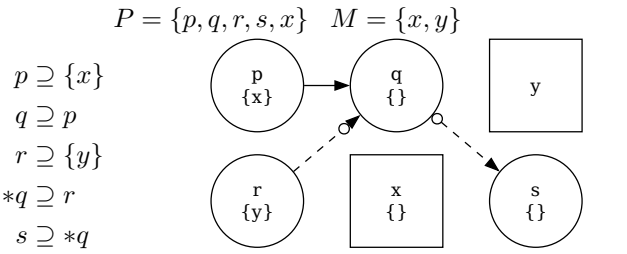

Fig. 3. A set of example constraints, and the corresponding constraint graph. Note that $y \notin P$, so there is no $\text{Sol}(y)$.

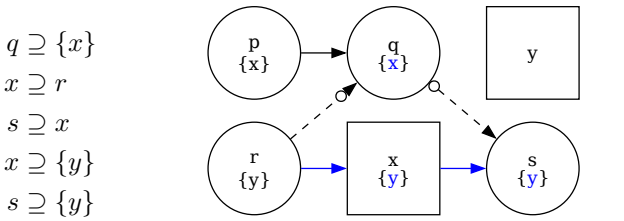

Fig. 4. The solved version of the constraint graph from Figure 3. The inferred constraints are listed on the left, and colored blue in the graph.

C. Worklist Solver

Multiple variations of worklist algorithms have been used to solve constraint graphs efficiently [14, 16]. The worklist is a list of nodes that need to be visited before solving is finished. Visiting a node checks the constraints currently defined on the node to see if any of the inference rules from Figure 2 can be used to infer new constraints. When a node receives new constraints, it is added to the worklist to ensure the new constraints are processed. Once the worklist is empty, all constraints have been processed, and a fixed point has been reached. The order in which nodes from the worklist are processed is known as its *iteration order*, and can have a drastic effect on solving performance [16].

```
1  int* f(int** x) {
2    int* y = *x;
3    return y;
4  }
5  int*(*h)(int**) = &f;
6
7  int a;
8  int* g() {
9    int* p = &a;
10   int** r = &p;
11   int* s = (*h)(r);
12   return s;
13 }
```

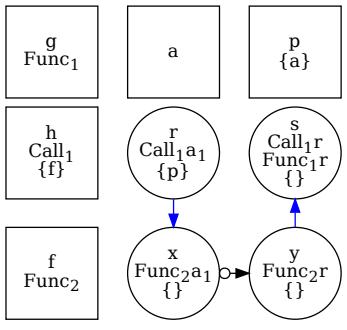

Fig. 5. A sample C program containing two functions and an indirect function call. The corresponding constraint graph is drawn in black. The result of applying the CALL inference rule to Call_1 and Func_2 is drawn in blue. Local variables that never have their address taken are represented by virtual registers, drawn as circles. The remaining constraint variables are abstract memory locations, drawn as squares.

D. Cycle Detection

An important observation made by Fähndrich et al. is that cycles in the constraint graph can be eliminated [29]. When a cycle of simple edges $a \to b \to c \to \ldots \to a$ is formed, any member of $\mathrm{Sol}(a)$ also becomes a member of $\mathrm{Sol}(b)$, $\mathrm{Sol}(c)$, etc. Instead of propagating pointees through the cycle, the pointers in the cycle can share a single Sol set, saving time and memory. Several techniques exist for discovering such cycles, both before [23, 30] and during [11, 14, 16] solving.

E. Pointer Provenance

The C standard places restrictions on the use of pointers, prohibiting out-of-bounds pointer arithmetic or using pointers to freed memory [31]. This can lead to situations where two pointers have the same address, yet are non-interchangeable, as demonstrated in Defect Report DR260. The resolution from the C language working group WG14 confirmed: "[Implementations may] treat pointers based on different origins as distinct even though they are bitwise identical" [32]. The origin of a pointer is known as its *provenance*, and compilers routinely exploit this fact when performing optimizations [33].

Some points-to analyses handle pointer-integer casting by tracking the pointees of integers [4, 15]. This may have unintended side effects, however, as it adds the concept of provenance to integers. Integers with identical representation may no longer be interchangeable, which breaks common assumptions, and has led to miscompilations [34]. When provenance was standardized in Technical Specification 6010 [21], the committee therefore went with a provenance-aware memory object model called *PNVI-ae-udi* [35]. In this model, provenance is not carried via integers (*PNVI*). Instead, casting an integer to a pointer recreates the provenance of the memory object it targets. The target must, however, have previously had its address exposed (*ae*) as an integer.

III. Handling Incomplete Programs

The analysis from Section II only produces sound solutions when analyzing whole programs. This prevents its use in situations where only parts of the source code are available, such as when compiling a single file in a larger program or when using external libraries. In these situations, the analysis is said to operate on *incomplete programs* [4, 13].

In this section, we show how to extend the analysis to make it sound for incomplete programs. The key insight is that this can be accomplished by tracking which memory locations escape the module, and which pointers originate from external modules. Production C compilers also need to handle pointer-integer conversions, which are covered in Section III-C. The resulting analysis is sound for all well-formed incomplete C programs, but its naive implementation is slow, as shown in Section VI. A more efficient implementation is presented in Section III-D.

A. Tracking Externally Accessible Locations and Pointers

An incomplete program is always executed as part of a larger complete program, which may include arbitrary statements and variables from other modules. We define the soundness of points-to analysis solutions for incomplete programs as follows: *For an incomplete program A with pointers P and abstract memory locations M, a points-to analysis solution $\mathrm{Sol} : P \to \mathcal{P}(M)$ is sound if and only if $\mathrm{Sol}(p)$ contains all pointees p may ever target at runtime, during any execution of any complete program A is linked into.*

We call the incomplete program being analyzed the *current module*, and all other modules *external modules*. To conservatively model all possible statements in external modules, the analysis keeps track of which abstract memory locations in M *escape* the current module. A memory location $x \in M$ may escape when:

- x is exported from the current module as a named symbol.
- A pointer to x is an argument to an external function call.
- A pointer to x is the return value in a function f, where f itself has escaped and may be called by external modules.
- $x \in \mathrm{Sol}(y)$, and y has escaped.

Memory locations that escape from the current module, and memory locations imported from other modules, make up the set of *externally accessible* memory locations. Tracking which memory locations are in this set lets the analysis retain some precision when handling pointer values of unknown origin. The situations in which a pointer p has an unknown origin are:

- p is the return value of an external function call.
- p is an argument in an escaped function f that may be called from an external module.
- p is a memory location that is externally accessible. An external module may thus store a new pointer into p.

These cases correspond to the pointers `r`, `q`, and `p` from Figure 1, respectively. In all these cases, the pointer values originate from external modules. This means they may only target externally accessible memory locations. Importantly, a pointer value of unknown origin may not target any memory locations from the current module that have not escaped. Even if an external module is able to correctly guess the address of a non-escaped memory object, pointers created this way would not have the correct provenance, and would be invalid [33].

B. The Ω Node

A new constraint variable is introduced, called Ω, to keep track of externally accessible memory locations. It represents all memory locations defined in external modules that are not represented by any other abstract memory location. This memory may contain pointers, so $\Omega \in P$. Effectively, $\mathrm{Sol}(\Omega)$ contains all memory locations that are externally accessible. It can also be pointed to, so $\Omega \in M$. Along with the Ω node, the following constraints are added:

$$\textcircled{1}\ \ \Omega \supseteq \{\Omega\} \qquad \textcircled{2}\ \ \Omega \supseteq *\Omega \qquad \textcircled{3}\ *\Omega \supseteq \Omega$$

$$\textcircled{4}\ \ \mathrm{Call}(\Omega, \Omega, \cdots, \Omega) \qquad \textcircled{5}\ \ \mathrm{Func}(\Omega, \Omega, \cdots, \Omega)$$

Constraint $\textcircled{1}$ represents the fact that any pointer represented by Ω may target memory represented by Ω. This also gives the desired property that loading from an unknown pointer results in another unknown pointer. Any pointer p with an unknown

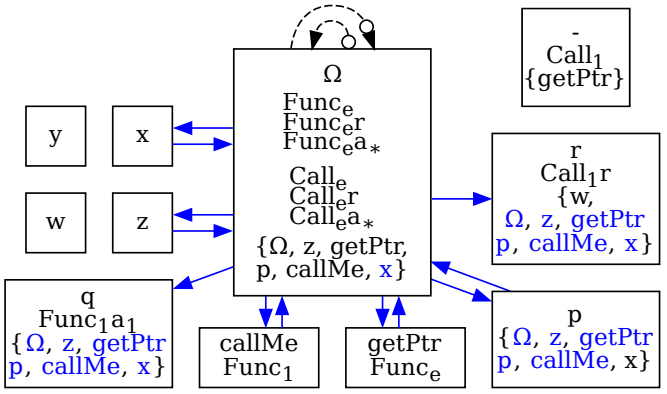

Fig. 6. The constraint graph corresponding to the program in Figure 1. Black constraints are added during phase 1, and blue constraints are added during solving. The direct function call on line 10 has been given a dummy pointer to `getPtr` to be the target of the Call_1 constraint.

origin is given the constraint $p \supseteq \Omega$, which makes p target Ω itself, as well as all externally accessible memory locations.

The remaining constraints are used to conservatively represent all statements that may ever occur in any external module. Constraint ② represents external modules loading from any pointer they may hold, causing the pointees of escaped pointers to also escape. Constraint ③ represents external modules storing unknown pointer values into any memory that has escaped, giving all escaped pointers unknown pointees.

Constraint ④ handles functions that escape from the current module being called from external modules. Pointer-compatible return values and function arguments are propagated to and from Ω, respectively. Constraint ⑤ handles cases where indirect function calls target an unknown pointer, which may target functions in external modules.

Finally, copies of constraint ⑤ also need to be added to all nodes representing functions imported from other modules. Recall that the Ω node only represents external memory that is not otherwise represented by any other abstract memory location, so imported functions need their own $\text{Func}(\cdot)$ constraint. If the imported function is a common library function, it is also possible to use a handwritten summary function instead of the overly conservative constraint ⑤.

The constraints on the Ω node effectively model all statements that may be executed in external modules. The constraints are expressed using the regular constraint language, which means existing constraint graph solvers can support the Ω node with only minor modifications. The main modifications required to handle incomplete programs are the following additions to the first phase of the analysis:

- The Ω node and its constraints are added.
- Any exported global variable or function e is marked as externally accessible ($\Omega \supseteq \{e\}$).
- Any imported global variable or function i is marked as externally accessible ($\Omega \supseteq \{i\}$).
- Any imported function f must either be represented by a generic constraint $\text{Func}(f, \Omega, \cdots, \Omega)$, or be mapped to a custom summary function for f.

The additions to the constraint graph only consider top-level definitions and declarations from the analyzed program. This is sufficient, as named symbols form the basis of all cross-module interactions. The constraints on the Ω node ensure that all statements that may be executed in external modules are reflected in the current module's analysis solution. Figure 6 shows the constraint graph for the program in Figure 1.

In the example program, the exported symbols are `z`, `p`, and `callMe`, while the only imported symbol is `getPtr`. These are all externally accessible as symbols, and are thus included in the initial $\text{Sol}(\Omega)$ set. Since `getPtr` is an externally defined function, it is also given the Func_e constraint, whose return value and arguments are all represented by Ω.

The constraints defined on the external node ensure that all statements that may be executed in external modules are handled when the graph is solved. The Ω load and store self-edges add simple edges to and from all memory locations in $\text{Sol}(\Omega)$. One of these edges, $p \longrightarrow \Omega$, causes x to be propagated from $\text{Sol}(p)$ to $\text{Sol}(\Omega)$, representing x's escape from the current module. The edge $\Omega \longrightarrow p$ adds all externally accessible memory locations to $\text{Sol}(p)$, representing the fact that external modules may store pointer values of unknown origin into p.

The Call_e constraint on Ω represents external modules calling all externally accessible functions, such as `callMe`. Applying the CALL inference rule adds the simple edge $\Omega \longrightarrow q$. Every externally accessible memory location is propagated along this edge, representing that q may hold a pointer value of unknown origin. Likewise, the function call on line 10 targets `getPtr`, which has the Func_e constraint, adding the simple edge $\Omega \longrightarrow r$ to represent pointer values of unknown origin. r itself does not escape, so it can also point to w without making w externally accessible.

C. Pointer-Integer Conversions and Pointer Smuggling

To make the points-to analysis sound under the PNVI-ae-udi provenance model, integers can not be pointer compatible. Instead, when a pointer p is cast to an integer, all pointees of p are marked as externally accessible by adding the constraint $\Omega \supseteq p$. When an integer is cast to a pointer q, it soundly handles having an unknown origin by adding the constraint $q \supseteq \Omega$. The flow of pointer values via integers is thus soundly represented via the Ω node. This also handles cases where pointers are cast to integers in the current module, and cast back to pointers in an external module, or vice versa.

Since integers are not pointer compatible, a memory location x representing an integer variable is not included in the set of pointers P. The analysis does not track the pointees of x, and there is no $\text{Sol}(x)$ set. It is, however, still possible for constraints of the form $p \supseteq x$ or $x \supseteq p$ to appear, where $p \in P$. These constraints effectively represent conversions between pointers and integers, so a sound analysis must instead treat these constraints as $p \supseteq \Omega$ or $\Omega \supseteq p$, respectively.

The last piece necessary to make the analysis sound is to handle indirect casting of pointer types through type punning. By casting an `int**` to a `char*`, and reading eight

$$\textsc{In}\Omega\ \frac{\Omega \sqsupseteq \{p\}}{\begin{array}{c}\Omega \sqsupseteq p \\ p \sqsupseteq \Omega\end{array}} \qquad \textsc{Trans}\Omega\ \frac{p \sqsupseteq \Omega \quad q \supseteq p}{q \sqsupseteq \Omega} \qquad \textsc{To}\Omega\ \frac{\Omega \sqsupseteq p \quad p \supseteq \{x\}}{\Omega \sqsupseteq \{x\}} \qquad \textsc{StoreTo}\Omega\ \frac{*p \supseteq q \quad p \sqsupseteq \Omega}{\Omega \sqsupseteq q}$$

$$\textsc{LoadFrom}\Omega\ \frac{q \supseteq *p \quad p \sqsupseteq \Omega}{q \sqsupseteq \Omega} \qquad \textsc{StoreScalar}\ \frac{*p \sqsupseteq \Omega \quad p \supseteq \{x\}}{x \sqsupseteq \Omega} \qquad \textsc{LoadScalar}\ \frac{\Omega \sqsupseteq *p \quad p \supseteq \{x\}}{\Omega \sqsupseteq x}$$

$$\textsc{CallImp}\ \frac{\begin{array}{c}\mathrm{Call}(f, r, a_1, \cdots, a_k) \\ f \supseteq \{g\} \\ \mathrm{ImpFunc}(g)\end{array}}{\begin{array}{c}r \sqsupseteq \Omega \\ \Omega \sqsupseteq a_i,\ \forall i \in \{1, \cdots, k\}\end{array}} \qquad \textsc{Call}\Omega\ \frac{\begin{array}{c}\mathrm{Call}(f, r, a_1, \cdots, a_k) \\ f \sqsupseteq \Omega\end{array}}{\begin{array}{c}r \sqsupseteq \Omega \\ \Omega \sqsupseteq a_i,\ \forall i \in \{1, \cdots, k\}\end{array}} \qquad \textsc{CalledBy}\Omega\ \frac{\begin{array}{c}\mathrm{Func}(f, r, a_1, \cdots, a_k) \\ \Omega \sqsupseteq \{f\}\end{array}}{\begin{array}{c}\Omega \sqsupseteq r \\ a_i \sqsupseteq \Omega,\ \forall i \in \{1, \cdots, k\}\end{array}}$$

Fig. 7. Additional rules of inference added to represent the Ω node implicitly.

consecutive `chars`, an `int*` has effectively been converted from a pointer to a scalar. Reversing this process converts the scalar back into a pointer. We call this indirect casting via memory *pointer smuggling*.

Pointer smuggling can be handled soundly by adding constraints on loads and stores of pointer-incompatible types. Given a `char*` p, loading a `char` from p adds the constraint $\Omega \supseteq *p$, and storing a `char` to p adds the constraint $*p \supseteq \Omega$. If p happens to have a pointer-compatible target $q \in \mathrm{Sol}(p)$, it will correctly be marked as $\Omega \supseteq q$ and $q \supseteq \Omega$, respectively.

D. Representing Ω implicitly (IP)

The introduction of the Ω node enables the sound points-to analysis of incomplete programs, expressed within the conventional language of constraints for Andersen-style points-to analysis. The Ω node is, however, likely to be a hot spot during solving. Every externally accessible memory location is included in $\mathrm{Sol}(\Omega)$, and all pointers of unknown origin have Sol sets that are supersets of $\mathrm{Sol}(\Omega)$. This Cartesian product of pointer-pointee relations scales poorly, both in memory usage and solver runtime. Cycle detection is of limited help, as can be seen in the solved constraint graph in Figure 6. Only p is in a cycle with Ω, while q and r are not.

The key observation to make the analysis scalable is that the Ω node can be represented entirely implicitly. The Ω node is removed, and six new types of constraints are added to the constraint language in its place, replacing constraints previously defined using the Ω node. The new constraint types, shown in Table II, use the symbol $\sqsupseteq$ to clearly distinguish them from the original constraint language. In the new language, Ω is strictly a language construct, not a constraint variable.

Turning Ω into a part of the language allows us to implement the behavior described in Section III-B using an alternative set of inference rules, shown in Figure 7. The new rule $\textsc{Trans}\Omega$ avoids propagating copies of $\mathrm{Sol}(\Omega)$ around the constraint graph, by instead propagating the constraint $p \sqsupseteq \Omega$ itself along simple edges $p \longrightarrow q$. The rule $\textsc{To}\Omega$ marks targets sent to Ω as externally accessible, while $\textsc{In}\Omega$ emulates the store and load self-edges of the Ω node. The $\textsc{StoreTo}\Omega$ and $\textsc{LoadFrom}\Omega$ rules handle storing and loading of pointers that may have an unknown origin, while the $\textsc{Call}$* rules emulate the Func and Call constraints on Ω. In total, the new rules replicate all behavior that was previously modeled by the Ω node, without inferring any constraints from the original language.

TABLE II
CONSTRAINT TYPES ADDED IN THE EXTENDED LANGUAGE TO REPRESENT THE Ω NODE IMPLICITLY.

Old	New	Comment
$\Omega \supseteq \{x\}$	$\Omega \sqsupseteq \{x\}$	x is externally accessible
$p \supseteq \Omega$	$p \sqsupseteq \Omega$	p targets all externally accessible memory
$\Omega \supseteq p$	$\Omega \sqsupseteq p$	Pointees of p are externally accessible
$*p \supseteq \Omega$	$*p \sqsupseteq \Omega$	A scalar is stored at $*p$
$\Omega \supseteq *p$	$\Omega \sqsupseteq *p$	$*p$ is loaded as a scalar
$\mathrm{Func}(f, \Omega, \cdots, \Omega)$	$\mathrm{ImpFunc}(f)$	f is an imported external function

There are now two ways of encoding that a pointer p may point to a target memory location x. If the solver infers the base constraint $p \supseteq \{x\}$, we say that x is an *explicit* pointee of p. If the solver infers both constraints $p \sqsupseteq \Omega$ and $\Omega \sqsupseteq \{x\}$, we say that x is an *implicit* pointee of p. While it would be possible to add a rule that infers explicit pointees from implicit pointees, this rule has been omitted on purpose. Instead, the definition of Sol is changed to include both explicitly and implicitly encoded pointers, defined as Sol_e and Sol_i, respectively:

$$\mathrm{Sol}_e(p) \coloneqq \{x \in M \mid p \supseteq \{x\} \text{ has been inferred}\}$$

$$\mathrm{Sol}_i(p) \coloneqq \begin{cases} E, & p \sqsupseteq \Omega \text{ has been inferred} \\ \{\}, & \text{otherwise} \end{cases}$$

$$\mathrm{Sol}(p) \coloneqq \mathrm{Sol}_e(p) \cup \mathrm{Sol}_i(p)$$

$$\text{where } E = \{x \in M \mid \Omega \sqsupseteq \{x\} \text{ has been inferred}\}$$

The final Sol sets end up being identical to the solution produced using the explicit Ω node. The benefit of introducing the implicit pointee representation is that many pointer-pointee relations only exist because all pointers of unknown origin may target all externally accessible memory locations. Representing these possible pointer-pointee pairs explicitly would require one base constraint per pair, which scales poorly both in memory usage and solver runtime. The implicit representation keeps the $\Omega \sqsupseteq \{x\}$ and $p \sqsupseteq \Omega$ constraints separate, and leaves the Cartesian product of pointer-pointee relations implicit. Only pointer-pointee relations that go via Ω can be represented implicitly, so explicit pointees are still used to represent all other

kinds of pointer-pointee relations. The conventional worklist solver can also be modified to include the new inference rules shown in Figure 7. The new constraint types can all be implemented as 1-bit flags on constraint variables. Pseudocode for the modified worklist algorithm is given in Algorithm 1.

IV. Prefer Implicit Pointees (PIP)

The introduction of the implicit pointee representation gives the analysis an efficient way to encode the pointer-pointee relations between all pointers of unknown origin and all externally accessible memory locations. It is, however, possible for a pointee to be represented both implicitly and explicitly, i.e., $x \in \mathrm{Sol}_e(p) \wedge x \in \mathrm{Sol}_i(p)$. This happens when the solver infers all three of the following constraints:

$$p \supseteq \{x\} \qquad p \sqsupseteq \Omega \qquad \Omega \sqsupseteq \{x\}$$

In these cases, we say that x is a *doubled-up* pointee of p. These doubled-up pointees do not affect the final solution, but are undesired, as they have a higher representational overhead. Explicit pointees may also be propagated along simple edges or be used to infer additional simple edges, which can only serve to create more doubled-up pointees. This extra work increases runtime, without ever affecting the analysis solution.

Prefer Implicit Pointees (PIP) is an online solver technique that attempts to reduce the number of doubled-up pointees that occur during solving. It is based on the observation that for any node p with the constraints $\Omega \sqsupseteq p$ and $p \sqsupseteq \Omega$, the final solution will always satisfy $\mathrm{Sol}(p) = \mathrm{Sol}_i(p)$. Thus, any explicit pointee of p is guaranteed to end up as a doubled-up pointee, and can safely be removed, or never added in the first place. In practice, the technique works by adding the following checks to the worklist algorithm presented in Algorithm 1:

1) When the algorithm visits a node p, and p is not marked $\Omega \sqsupseteq p$, it starts by checking if p can gain the flag through "backpropagation". This is possible if there exists an outgoing simple edge $p \to q$, where q is marked $\Omega \sqsupseteq q$. In that case, any member $x \in \mathrm{Sol}_e(p)$ will be propagated to $\mathrm{Sol}_e(q)$, where it will be marked $\Omega \sqsupseteq \{x\}$. Adding $\Omega \sqsupseteq p$ thus does not affect the final solution.
2) If a node p is marked both $p \sqsupseteq \Omega$ and $\Omega \sqsupseteq p$, the set $\mathrm{Sol}_e(p)$ can be cleared, after all members $x \in \mathrm{Sol}_e(p)$ have been marked $\Omega \sqsupseteq \{x\}$.
3) When attempting to add an edge $p \to q$, first check if the node q has the constraint $\Omega \sqsupseteq q$. If so, backpropagate it to $\Omega \sqsupseteq p$. If $\Omega \sqsupseteq p$, and $q \sqsupseteq \Omega$, the addition of the simple edge $p \to q$ can be skipped entirely, as $\mathrm{Sol}(q)$ becomes a superset of $\mathrm{Sol}(p)$ via Ω.
4) Before propagating explicit pointees along a simple edge $p \to q$, check if the nodes are marked $q \sqsupseteq \Omega$ and $\Omega \sqsupseteq p$. If so, the simple edge can be removed, using the same reasoning as in the previous point.

By adding PIP, the worklist algorithm gains some important properties: Any node x that is marked both $x \sqsupseteq \Omega$ and $\Omega \sqsupseteq x$ will be visited by the worklist at most once. $\mathrm{Sol}_e(x)$ will also be empty in the final solution. Nodes like x are common, as they include all externally accessible memory locations.

Algorithm 1 Worklist algorithm for inferring all constraints from Figure 2 and Figure 7. The blue comments show where to add extra logic for the PIP technique.

Input: Sets of pointers P, memory locations M and constraints C

$\mathrm{Sol}_e(p) \leftarrow \{x \in M \mid (p \supseteq \{x\}) \in C\},\ \forall p \in P$ ▷ *Initialize* Sol_e
procedure PropagatePointees(f, t) ▷ *Propagates from f to t*
  $\mathrm{Sol}_e(t) \leftarrow \mathrm{Sol}_e(t) \cup \mathrm{Sol}_e(f)$
  if f **is marked** $f \sqsupseteq \Omega$ **then mark** t **as** $t \sqsupseteq \Omega$
  if $\mathrm{Sol}_e(t)$ **changed or** t **was marked then** $W \leftarrow W \cup \{t\}$
procedure CallToImported($r, a_1, \cdots, a_k$)
  mark r **as** $r \sqsupseteq \Omega$
  mark a_i **as** $\Omega \sqsupseteq a_i, \quad \forall i \in \{1, \cdots, k\}$
  $W \leftarrow W \cup \{v \in (P \cup M) \mid v$ gained a flag in this procedure$\}$
procedure MarkExternallyAccessible(x)
  mark x **as** $\Omega \sqsupseteq \{x\}$ **and** $x \sqsupseteq \Omega$ **and** $\Omega \sqsupseteq x$
  for all $\mathrm{Func}(x, r, a_1, \cdots, a_k) \in C$ **do**
    mark r **as** $\Omega \sqsupseteq r$
    mark a_i **as** $a_i \sqsupseteq \Omega, \quad \forall i \in \{1, \cdots, k\}$
  $W \leftarrow W \cup \{v \in (P \cup M) \mid v$ gained a flag in this procedure$\}$
▷ *Handle nodes that are externally accessible from the start* ◁
for all x **marked** $\Omega \sqsupseteq \{x\}$ **do**
  MarkExternallyAccessible(x)
▷ *Initialize Worklist with every node* ◁
$W \leftarrow P \cup M$
while $W \neq \emptyset$ **do**
  $n \leftarrow$ PopWorklist(W)
  $\Delta E \leftarrow \{\}$ ▷ *Simple edges to add*
  ▷ *PIP addition 1: Backpropagate to make $\Omega \sqsupseteq n$ if possible* ◁
  if n **is marked** $\Omega \sqsupseteq n$ **then**
    for all $x \in Sol_e(n)$ **do**
      if x **not marked** $\Omega \sqsupseteq \{x\}$ **then**
        MarkExternallyAccessible(x)
  ▷ *PIP addition 2: If $\Omega \sqsupseteq n$ and $n \sqsupseteq \Omega$, clear $\mathrm{Sol}_e(n)$* ◁
  for all $p \supseteq n$ **in** C **do** ▷ *Simple edges*
    ▷ *PIP addition 4: If $p \sqsupseteq \Omega$ and $\Omega \sqsupseteq n$, remove the edge* ◁
    PropagatePointees(n, p)
  for all $*n \supseteq p$ **in** C **do** ▷ *Store edges*
    for all $x \in \mathrm{Sol}_e(n)$ **do**
      $\Delta E \leftarrow \Delta E \cup \{x \supseteq p\}$
    if n **is marked** $n \sqsupseteq \Omega$ **then**
      mark p **as** $\Omega \sqsupseteq p$, **add** p **to** W **if mark is new**
  if n **is marked** $*n \sqsupseteq \Omega$ **then** ▷ *Storing scalar*
    for all $x \in \mathrm{Sol}_e(n)$ **do**
      mark x **as** $x \sqsupseteq \Omega$, **add** x **to** W **if mark is new**
  for all $p \supseteq *n$ **in** C **do** ▷ *Load edges*
    for all $x \in \mathrm{Sol}_e(n)$ **do**
      $\Delta E \leftarrow \Delta E \cup \{p \supseteq x\}$
    if n **is marked** $n \sqsupseteq \Omega$ **then**
      mark p **as** $p \sqsupseteq \Omega$, **add** p **to** W **if mark is new**
  if n **is marked** $\Omega \sqsupseteq *n$ **then** ▷ *Loading scalar*
    for all $x \in \mathrm{Sol}_e(n)$ **do**
      mark x **as** $\Omega \sqsupseteq x$, **add** x **to** W **if mark is new**
  for all $\mathrm{Call}(n, r, a_1, \cdots, a_k)$ **in** C **do** ▷ *Calls*
    for all $x \in \mathrm{Sol}_e(n)$ **do**
      for all $\mathrm{Func}(x, r_\bullet, a_{1\bullet}, \cdots, a_{k\bullet})$ **in** C **do**
        $\Delta E \leftarrow \Delta E \cup \{r \supseteq r_\bullet, a_{1\bullet} \supseteq a_1, \cdots, a_{k\bullet} \supseteq a_k\}$
      if x **is marked** $\mathrm{ImpFunc}(x)$ **then**
        CallToImported($r, a_1, \cdots, a_k$)
    if n **is marked** $n \sqsupseteq \Omega$ **then**
      CallToImported($r, a_1, \cdots, a_k$)
  for all $(p \supseteq q) \in \Delta E \backslash C$ **do** ▷ *Add new simple edges*
    ▷ *PIP addition 3: Skip adding edge if $p \sqsupseteq \Omega$ and $\Omega \sqsupseteq q$* ◁
    $C \leftarrow C \cup \{p \supseteq q\}$
    PropagatePointees(q, p)

TABLE III

PROGRAMS USED TO BENCHMARK POINTS-TO ANALYSIS RUNTIME AND PRECISION. V IS THE SET OF CONSTRAINT VARIABLES, AND C IS THE SET OF CONSTRAINTS, PER ANALYZED FILE.

		Summary of non-empty C files						
			IR instructions		$\|V\|$		$\|C\|$	
Name	KLOC†	#Files	Mean	Max	Mean	Max	Mean	Max
500.perlbench	362	68	22 725	165 497	4 226	28 236	6 686	47 046
502.gcc	902	372	16 244	535 524	3 434	64 847	5 148	101 572
505.mcf	2	12	1 228	4 778	304	1 197	530	2 149
507.cactuBSSN	102	345	5 691	123 596	1 251	24 770	2 665	60 339
525.x264	24	35	10 963	87 991	1 582	11 922	2 961	22 438
526.blender	981	996	8 600	443 034	1 917	99 558	2 969	142 950
538.imagick	155	97	11 195	154 125	2 425	29 864	4 336	53 287
544.nab	12	20	5 741	22 276	1 509	5 593	2 712	11 299
557.xz	15	89	1 448	18 935	220	1 470	390	3 074
emacs-29.4	253	143	14 085	260 284	3 377	48 127	5 367	77 533
gdb-15.2	172	251	5 508	101 443	1 179	36 708	1 887	51 718
ghostscript-10.04	797	1 116	7 042	441 161	1 243	16 665	2 078	30 929
sendmail-8.18.1	89	115	3 752	39 205	799	7 675	1 453	14 410

†Total lines of code, excluding whitespace and comments, divided by 1000.

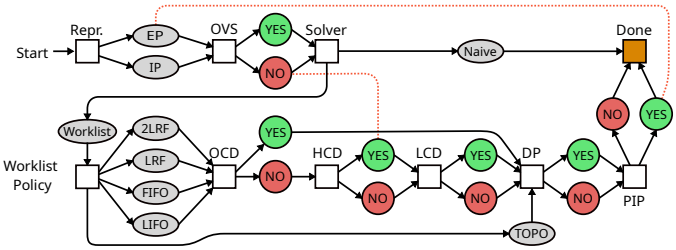

Fig. 8. All paths through the flowchart represent valid combinations, except for choices connected by dashed red edges, which are incompatible.

V. METHODOLOGY

The described analysis has been implemented in the `jlm` research compiler [36], which converts LLVM IR [27] into an intermediate representation based on the Regionalized Value State Dependence Graph (RVSDG) [37]. All instructions that are relevant to points-to analysis in the original LLVM IR have a one-to-one representation in the RVSDG, making the results applicable to LLVM IR or other SSA-based IRs. The analysis is performed on all nine C benchmarks from SPEC2017 [22], and four C programs often used in the literature [9, 11, 25]. The benchmarks are summarized in Table III, totaling 3 659 non-empty C files. Evaluation is performed in the context of a conventional compilation model, where each C file is analyzed separately. The files are converted to LLVM IR by `clang 18.1.8` using optimization level `O0`, and analyzed by `jlm` on an Intel Xeon Gold 6348 CPU running at 3.0 GHz.

A. Analysis Configurations

The analysis implementation has several options that enable and disable different techniques. Five of these techniques are from the literature, where they have been shown to improve solver performance in the context of whole-program analysis. This paper evaluates all techniques in the context of analyzing individual files. The full set of options is given in Table IV, and Figure 8 shows a flowchart depicting all valid combinations.

TABLE IV

THE TECHNIQUES THAT CAN BE ENABLED OR DISABLED, AND THE CHOICES OF SOLVER WHEN CONFIGURING THE POINTS-TO ANALYSIS. SQUARES ARE BINARY CHOICES, AND DASHES ARE EXCLUSIVE CHOICES.

Technique	Abbrev.	Described in
Pointer representation		
– Only Explicit Pointees	EP	Section III-B
– Implicit Pointees	IP	Section III-D
Offline constraint processing		
☐ Offline Variable Substitution	OVS	Rountev et al. [23]
Solver		
– Naive Solver	Naive	Andersen's thesis [20]
– Worklist Solver	WL	Section II-C
Worklist Iteration Order		
– First In First Out	FIFO	Pearce et al. [14]
– Last In First Out	LIFO	Pearce et al. [14]
– Least Recently Fired	LRF	Pearce et al. [14]
– 2-Phase Least Recently Fired	2LRF	Hardekopf and Lin [16]
– Topological	TOPO	Pearce et al. [25]
Worklist Online Techniques		
☐ Prefer Implicit Pointees	PIP	Section IV
☐ Online Cycle Detection	OCD	Pearce et al. [16]
☐ Hybrid Cycle Detection	HCD	Hardekopf and Lin [16]
☐ Lazy Cycle Detection	LCD	Hardekopf and Lin [16]
☐ Difference Propagation	DP	Pearce et al. [14]

A configuration is described by the techniques it uses. For example, the configuration called `IP+WL(LRF)+OCD+PIP` uses the implicit pointer (IP) representation, the worklist (WL) solver with the Least Recently Fired (LRF) iteration order, Online Cycle Detection (OCD), and the Prefer Implicit Pointees (PIP) techniques. Not all combinations of choices are valid, e.g., OCD detects all cycles as soon as they appear, so there is no point in combining it with any of the opportunistic cycle detection techniques (HCD or LCD). In total, there are 208 valid configurations, and each configuration is benchmarked solving the constraint graph for each C file 50 times. The solution is validated to ensure that all configurations produce the exact same solution.

B. Implementation Details

Constraint variables are indexed using 32-bit integers, and their Sol_e sets are implemented as hash sets. Base constraints are placed directly into Sol_e. The six constraints added in the extended constraint language (see Table II) are implemented as 1-bit flags on each constraint variable. The only external library functions with special handling are `malloc`, `free`, and `memcpy`. Cycle unification uses union-find with path compression and union-by-rank [38], and a single Sol_e set is shared between the members of a union. Constraints are indexed by type and node to make iterating fast. Hash sets are used to avoid duplicated constraints. Simple edges involving one pointer incompatible variable, e.g., $x \in M \wedge x \notin P$, must be treated as pointer-integer casts. This is achieved in practice by making $\mathrm{Sol}(x) \coloneqq \mathrm{Sol}(\Omega)$. When the explicit Ω node is used, x is unified with Ω. When the implicit Ω node is used, x is marked both $x \sqsupseteq \Omega$ and $\Omega \sqsupseteq x$.

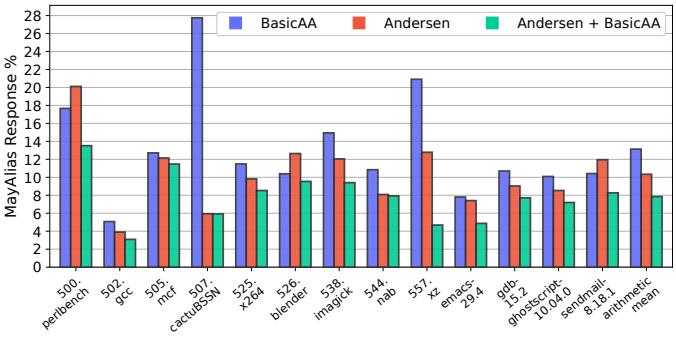

Fig. 9. Percentage of intra-procedural alias analysis queries that return "May Alias", when querying all load/store and store/store pairs. Lower is better.

VI. Results

After performing the points-to analysis on all C files, 51% of all pointers end up pointing to external memory (i.e., $p \sqsupseteq \Omega$). Confirming that the analysis still provides useful precision is covered in Section VI-A. While all 208 solver configurations produce the same solution, they have significant runtime variation, which is presented in Section VI-B. Finally, Section VI-C covers solver scalability.

A. Precision

We evaluate the precision of a points-to-analysis solution in terms of a pairwise alias-analysis client, by evaluating the load/store conflict rate, as described by Nagaraj and Govindarajan [39]. For each store instruction, the analysis is queried about possible aliasing with every other load and store instruction in the same function. The analysis returns *NoAlias* if the instructions have distinct points-to sets. Otherwise, *MayAlias* is returned.

For comparison, LLVM's BasicAA is used, which performs ad-hoc IR traversals to find the origin(s) of pointers. It does not handle function calls or nested pointers, but knows that local variables that never have their address taken never alias with anything. It also tracks pointer offsets when possible. Both analyses return *MustAlias* when the pointers are identical.

The results are shown in Figure 9. The analyses have different strengths, so the *Andersen + BasicAA*-bar shows the precision achieved by combining them, as is often done in practice. While the benefit of adding the Andersen-style points-to analysis varies by benchmark, the reduction in *MayAlias*-responses is on average 40%, indicating that the analysis adds valuable information for compiler transformations to exploit.

B. Solver Runtime

The choice of solver configuration has a large impact on the runtime of the constraint-solving phase, the most important choice being that of pointer representation. Table V shows the distribution of solver runtimes for the benchmarked C files, for selected configurations. It first compares the fastest configuration using explicit pointees (`EP`) against the fastest configuration using implicit pointees (`IP`) without PIP. On average, the `EP+OVS+WL(LRF)+OCD` configuration takes 36.322 ms per file to solve the constraint graph, while `IP+WL(FIFO)+LCD+DP` takes 2.133 ms, around 17× faster.

TABLE V
Constraint graph solver runtime for selected configurations.

Configuration	Solver Runtime [µs] p10	p25	p50	p90	p99	Max	Mean
`EP+OVS+WL(LRF)+OCD`	40	168	1 060	29 480	414 729	43 437 029	36 322
`EP Oracle`	21	118	886	25 191	347 921	39 594 566	32 376
`IP+WL(FIFO)+LCD+DP`	19	62	249	2 562	20 015	1 112 770	2 133
`IP+WL(FIFO)`	15	51	219	2 337	19 526	40 869 977	15 370
`IP+WL(FIFO)+PIP`	16	52	222	2 260	14 220	203 850	1 105

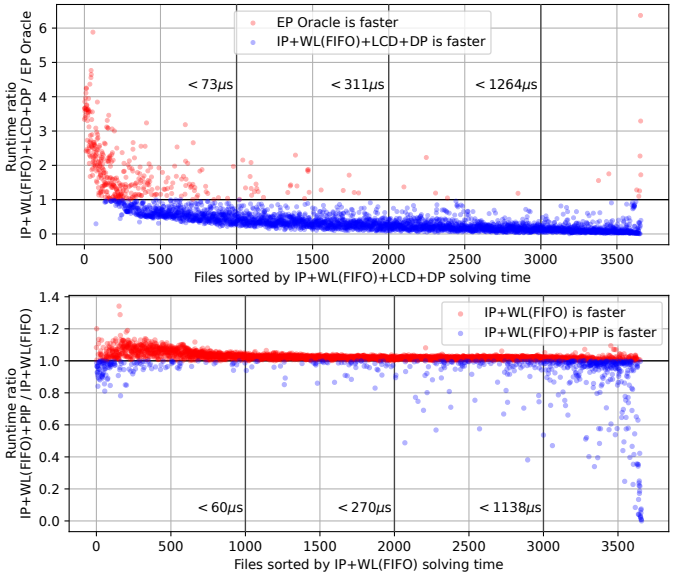

Fig. 10. Solver runtime ratio of IP vs. EP (top), and PIP vs. no PIP (bottom).

Next, we consider an `EP Oracle`, which always picks the fastest `EP` configuration for each file. The oracle saves time on most files, but ends up being just 8% faster than `EP+OVS+WL(LRF)+OCD` in total, as it is unable to significantly improve the runtime of the slowest outliers. The `IP` configuration is thus still 15× faster than the `EP Oracle`.

While `IP+WL(FIFO)+LCD+DP` is the fastest configuration without PIP, the fastest configuration overall is `IP+WL(FIFO)+PIP`, shown in the bottom half of Table V. With an average solver runtime of 1.105 ms per file, it is 1.9× faster than the best configuration without PIP. Adding any of the techniques from the literature to this configuration only increases the average solver runtime. Taking advantage of these techniques would thus require heuristics to determine for which files each technique should be used.

Lastly, we consider the effect of `PIP` alone by removing it from the fastest configuration, leading to `IP+WL(FIFO)`. As seen in Table V, enabling PIP decreases the average solver runtime by 14×, demonstrating that it is essential for reducing runtime. While the combination `LCD+DP` decreases runtime as well, it only reduces the average by 7× as depicted in Table V.

A detailed per-file comparison is shown in Figure 10. The upper graph depicts the relative solving time between the `EP Oracle` and the fastest configuration without PIP. Among the files where the `EP Oracle` is better, 98% are solved using the `Naive` solver, which happens to be efficient on some files regardless of pointer representation. For the seven rightmost red dots, the `EP Oracle` uses `OVS`, as that technique proves beneficial on some large files, but not enough to be included in the on average fastest `IP` configuration.

The relative effect of enabling PIP is shown in the lower graph of Figure 10. The results are not as conclusive as for `IP` vs. `EP`, but they clearly show a reduction in solving time for the longest running cases. For 83% of the files, enabling PIP makes the solving time slower, by 2.9% on average. For the files that take more than 1 138 µs to solve, which represent 81% of total solver runtime, 52% of them are slower with `PIP`, with the largest slowdown being 9%. In contrast, for the 48% of the files where `PIP` is faster, the average solving time is reduced by 16×. In the most extreme case, the slowest file with `IP+WL(FIFO)`, `base/gdevp14.c` from *ghostscript*, goes from 41 s to 9.5 ms, making an otherwise pathological file unnoteworthy. This has a drastic effect on the Max column in Table V. The fastest configuration without PIP, `IP+WL(FIFO)+LCD+DP`, uses 939 ms on the same file, which is much better than 41 s, but still 4.6× slower than the slowest file solved with `IP+WL(FIFO)+PIP`.

C. Solver Scalability

The worklist algorithm spends most of its time iterating through and propagating Sol_e sets between nodes, which makes the total number of explicit pointees a relevant metric for understanding solver runtime. They also account for the majority of memory usage. All configurations produce identical solutions, but may use different amounts of explicit pointees. Statistics about the number of explicit pointees produced by the different configurations are shown in Table VI.

The topmost configuration detects all cycles, which helps to reduce the number of explicit pointees by making nodes in cycles share Sol_e sets. The next configuration uses the implicit pointee representation to represent that pointers of unknown origin may target any externally accessible memory locations. Then comes the fastest configuration without PIP, which adds some cycle detection. Lastly, the fastest configuration overall, which uses PIP to avoid doubled-up pointees. The results make it clear that cycle elimination is in no way a substitution for using the implicit pointee representation. It also shows that some files end up with solutions containing almost exclusively doubled-up pointees, which PIP is able to skip, reducing memory usage drastically.

TABLE VI
NUMBER OF EXPLICIT POINTEES IN THE SOLUTIONS.

Configuration	Number of explicit pointees						
	p10	p25	p50	p90	p99	Max	Mean
`EP+OVS+WL(LRF)+OCD`	38	305	3 169	106 575	1 599 946	154 866 262	147 841
`IP+WL(FIFO)`	18	63	276	2 357	30 162	2 145 215	3 188
`IP+WL(FIFO)+LCD+DP`	18	63	274	2 323	29 486	1 897 247	2 816
`IP+WL(FIFO)+PIP`	17	59	258	1 977	11 700	95 195	922

VII. RELATED WORK

The handling of incomplete C programs has not received much attention in the points-to analysis literature, even though the problem was already pointed out by Hind [13] more than 20 years ago. Most of the previous works on points-to analysis sidestep the problem by assuming whole-program analysis is performed [6–12, 16–18], where external functions, such as standard library functions or system calls, are summarized to complete the missing program parts [6, 14–16].

Another approach is the computation of module-based analysis summaries, that are embedded along with the compilate. The analysis consumer stitches these summaries together to form a complete picture of the analysis results [40–43]. This approach is particularly popular for JIT compiled languages, where the summaries are produced statically and then consumed by the JIT compiler, but has also been proposed for conventionally compiled languages [44, 45]. However, the problem with this approach is that it defers the analysis to link time, preventing its use by per-module optimizations. Even at link time, a program might still not be complete, as some libraries may not have analysis results available, or libraries may be dynamically loaded at runtime.

There are only a handful of works that address the handling of incomplete C programs. Andersen [20] addresses the issue by introducing the abstract memory location *Unknown*. The location represents all accessible memory locations at runtime, and any pointer pointing to it may alias with any other pointer in the program. This approach has the downside that a complete loss of precision arises when an unknown pointer is dereferenced on the left-hand side of an assignment.

Smaragdakis and Kastrinis [46] present a method for sound points-to analysis in Java. Similarly to PIP, they avoid materializing points-to sets of pointers that can be shown to originate from what they call opaque code. However, they model pointers originating from or escaping to opaque code as being able to point to everything, including locations that provably did not escape. This works for Java, where all pointer accesses are based on named fields on a typed heap, but does not work in C, where pointers can be dereferenced directly.

The closest to our work is Lattner et al. [4]. They present a context-sensitive and field-sensitive unification-based analysis with full heap cloning that supports incomplete programs. Their algorithm has a complete flag, denoting that all operations on objects at a node have been processed. Nodes that are reachable from unavailable external functions or global variables are not marked as complete, indicating that the information represented by this node must be treated conservatively. While their work yields sound solutions for incomplete programs, it does not generalize to inclusion-based analyses, which typically give more precise solutions.

VIII. Conclusion

This paper presents an Andersen-style points-to analysis that efficiently produces sound solutions for incomplete programs. We introduce the implicit pointee representation to efficiently represent unknown pointers that possibly target every externally accessible memory location, and show in our evaluation against several state-of-the-art techniques that it is by far the most important factor for scalability. It achieves a total speedup of 15× over always picking the fastest configuration using an explicit pointee representation. We also introduce the Prefer Implicit Pointees (PIP) technique that further reduces the number of explicit pointees by avoiding doubled-up pointees in the solution, making the analysis an additional 1.9× faster than the fastest configuration without it. It also renders the other evaluated speedup techniques superfluous for incomplete programs, as none of them were able to outperform or aid PIP in terms of average solver runtime across all benchmarks. Most importantly, the average solver runtime of 1.1 ms per file makes our sound analysis practical for production compilers.

IX. Data-Availability Statement

The implemented analysis and experimental setup, and instructions for reproducing the results in the paper, are available as an artifact on Zenodo (DOI: 10.5281/zenodo.16900791) [47]. The latest version of the `jlm` compiler is found on GitHub [36].

Appendix A
Artifact Appendix

A. Abstract

Our artifact provides the source code for the `jlm` compiler, including an implementation of the Andersen-style analysis presented in the paper.

The artifact also includes scripts for performing the benchmarks described in the paper, and producing the figures and tables from the paper using the benchmark results.

The artifact contains the four free open-source benchmarks from Table III. It also contains redistributable versions of the SPEC2017 benchmarks, with the exception of `505.mcf`. If you provide your own copy of SPEC2017, the full set of benchmarks from the paper will be used.

B. Artifact check-list (meta-information)

- **Algorithm:** Points-to analysis / Alias analysis.
- **Program:** Four open-source benchmarks (source included). Nine benchmarks from SPEC2017 (8 of which have redistributable sources included).
- **Compilation:** `clang` 18 or above, or `g++` 12 or above.
- **Transformations:** Compiling C to LLVM IR with `clang` 18.
- **Run-time environment:** We recommend running in a container based on the provided `Dockerfile`. It can also run directly on Linux, such as Ubuntu 24.04, if the dependencies listed in the `Dockerfile` are installed.
- **Hardware:** We recommend a CPU with at least eight cores, and at least 32 GB of RAM.
- **Execution:** Should be the sole user on CPU, with fixed frequency. Avoid running other applications while executing the artifact to limit interference.
- **Metrics:** Analysis execution time, analysis precision.
- **Output:** Graphs and tables included in this paper. Numbers referenced in the text of the paper. Expected results included.
- **Experiments:** Create the Docker image and run the provided script in it.
- **How much disk space is required (approximately)?:** 40 GB.
- **How much time is needed to prepare workflow (approximately)?:** 5 minutes.
- **How much time is needed to complete experiments (approximately)?:** 9 hours.
- **Publicly available?:** Yes.
- **Code licenses:** LGPL 2.1
- **Data licenses:** See `sources/README.md`
- **Archived?:** https://doi.org/10.5281/zenodo.16900791

C. Description

1) *How delivered:* Our source code, benchmarking scripts, and the four open-source benchmarks + eight redistributable SPEC benchmarks, are available at the above DOI.
2) *Hardware dependencies:* The benchmarks should run on a CPU with at least eight physical cores and 32 GB of RAM.
3) *Software dependencies:* We recommend using the provided `Dockerfile` to build a Docker image that contains all necessary dependencies.
4) *Data sets:* If you own a copy of SPEC2017, then you can provide it. Otherwise the included redistributable sources will be used, with the following differences: The `505.mcf` benchmark is skipped. A subset of the C files in `500.perlbench` are skipped. `538.imagick` uses the original ImageMagick source code without SPEC's modifications.

D. Installation

Extract the file `pip-2026-artifact.tar.gz` in a suitable location.

```
$ tar xzf pip-2026-artifact.tar.gz
$ cd pip-2026-artifact
```

If you own SPEC2017, place `cpu2017.tar.xz` in the folder `sources/programs/`. Avoid symlinking as it may not work inside the Docker container.

If your machine has more than 32GB of RAM and more than eight physical cores, you can update the `PARALLEL_INVOCATIONS` variable in `run.sh` to make the evaluation run faster. (Default is `8`).

E. Experiment workflow

Build the Docker image:

```
$ docker build -t pip-2026-image .
```

Configure your CPU to run at a stable frequency, e.g., using:

```
$ cpupower frequency-set
↪ --min 3GHz --max 3GHz --governor performance
```

The evaluation for the paper was performed at 3 GHz, but pick a frequency that is low enough to prevent frequency boosting or throttling on your own system.

Execute the `run.sh` script inside the Docker image, with the current directory mounted:

```
$ docker run -it
↪ --mount type=bind,source="$(pwd)",target=/artifact
↪ pip-2026-image ./run.sh
```

For details about what the script does, see the `README.md` file. If the command is aborted, it can be restarted, and it will continue where it left off. To fully reset the evaluation workflow, append `clean` to the end of the command.

Running without Docker: If you wish to run on a different Linux system, dependencies might be located at different paths. Compilation commands may thus need to be changed, to reference the correct include paths. See `README.md` for re-creating the list of traced compiler invocations for your own system.

F. Evaluation and expected result

After running the experiment, results can be found in `results/`:

- Table III: `file-sizes-table.txt`
- Figure 9: `precision.pdf`
- Table V: `configuration-runtimes-table.txt`
- Figure 10: `ip_sans_pip_vs_ep_oracle_ratio.pdf` and `pip_vs_best_just_without_pip_ratio.pdf`
- Table VI: `configuration-memory-usage-table.txt`

The folder also contains `.log` files where numbers mentioned in the text of the paper are calculated.

Results based on measured runtime will vary based on the machine, but the overall ratios between configurations should be roughly the same. The quantiles given in the tables should also be similarly distributed, even if they are overall faster or slower.

Precision numbers and the number of explicit pointees can have tiny variations due to some of the open-source benchmarks configuring themselves slightly differently on different systems.

G. Experiment customization

Custom experiments can be performed by, for example:

- Creating a custom `sources.json` file containing compilation commands for any C program. This file can then be passed to the `benchmark.py` script.
- Modifying the `benchmark.py` script to add extra flags to `clang`, `opt`, and/or `jlm-opt`.
- Using the `jlm-opt` binary directly on any LLVM IR file made with LLVM 18, and dumping analysis runtime and/or precision metrics.

Details can be found in the `README.md` file.

ACKNOWLEDGMENTS

PIP has partly been developed and evaluated on the IDUN/EPIC [48] computing cluster at the Norwegian University of Science and Technology.

REFERENCES

[1] R. Surendran, R. Barik, J. Zhao, and V. Sarkar, "Inter-iteration scalar replacement using array SSA form," in *Proceedings of the International Conference on Compiler Construction*, A. Cohen, Ed., 2014, pp. 40–60. [Online]. Available: https://doi.org/10.1007/978-3-642-54807-9_3

[2] K. Chitre, P. Kedia, and R. Purandare, "The road not taken: exploring alias analysis based optimizations missed by the compiler," *Proceedings of the ACM on Programming Languages*, vol. 6, p. 153:786–153:810, Oct. 2022. [Online]. Available: https://doi.org/10.1145/3563316

[3] R. Karrenberg and S. Hack, "Whole-function vectorization," in *Proceedings of the International Symposium on Code Generation and Optimization*, Apr. 2011, p. 141–150. [Online]. Available: https://doi.org/10.5555/2190025.2190061

[4] C. Lattner, A. Lenarth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2007, pp. 278–289. [Online]. Available: https://doi.org/10.1145/1250734.1250766

[5] W. Landi, "Undecidability of static analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, p. 323–337, Dec. 1992. [Online]. Available: https://doi.org/10.1145/161494.161501

[6] Y. Lei and Y. Sui, "Fast and precise handling of positive weight cycles for field-sensitive pointer analysis," in *SAS*, Oct. 2019, p. 27–47. [Online]. Available: https://doi.org/10.1007/978-3-030-32304-2_3

[7] S. Ye, Y. Sui, and J. Xue, "Region-based selective flow-sensitive pointer analysis," in *SAS*, M. Müller-Olm and H. Seidl, Eds., 2014, pp. 319–336. [Online]. Available: https://doi.org/10.1007/978-3-319-10936-7_20

[8] Y. Sui, Y. Li, and J. Xue, "Query-directed adaptive heap cloning for optimizing compilers," in *Proceedings of the International Symposium on Code Generation and Optimization*, Feb. 2013, pp. 1–11. [Online]. Available: https://doi.org/10.1109/CGO.2013.6494978

[9] R. Nasre and R. Govindarajan, "Prioritizing constraint evaluation for efficient points-to analysis," in *Proceedings of the International Symposium on Code Generation and Optimization*, Apr. 2011, pp. 267–276. [Online]. Available: https://doi.org/10.1109/CGO.2011.5764694

[10] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang, "Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code," in *Proceedings of the International Symposium on Code Generation and Optimization*, Apr. 2010, p. 218–229. [Online]. Available: https://doi.org/10.1145/1772954.1772985

[11] F. M. Q. Pereira and D. Berlin, "Wave propagation and deep propagation for pointer analysis," in *Proceedings of the International Symposium on Code Generation and Optimization*, Mar. 2009, pp. 126–135. [Online]. Available: https://doi.org/10.1109/CGO.2009.9

[12] J. Zhu, "Towards scalable flow and context sensitive pointer analysis," in *Proceedings of the ACM/IEEE Design Automation Conference*, Jun. 2005, pp. 831–836. [Online]. Available: https://doi.org/10.1109/DAC.2005.193930

[13] M. Hind, "Pointer analysis: haven't we solved this problem yet?" in *Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, Jun. 2001, p. 54–61. [Online]. Available: https://doi.org/10.1145/379605.379665

[14] D. Pearce, P. Kelly, and C. Hankin, "Online cycle detection and difference propagation for pointer analysis," in *Proceedings of the IEEE International Conference on Source Code Analysis and Manipulation*, Sep. 2003, pp. 3–12. [Online]. Available: https://doi.org/10.1109/SCAM.2003.1238026

[15] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, Jan. 1996, p. 32–41. [Online]. Available: https://doi.org/10.1145/237721.237727

[16] B. Hardekopf and C. Lin, "The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2007, pp. 290–299. [Online]. Available: https://doi.org/10.1145/1250734.1250767

[17] M. Barbar, Y. Sui, and S. Chen, "Object versioning for flow-sensitive pointer analysis," in *Proceedings of the International Symposium on Code Generation and Optimization*, Feb. 2021, pp. 222–235. [Online]. Available: https://doi.org/10.1109/CGO51591.2021.9370334

[18] M. Barbar and Y. Sui, "Compacting points-to sets through object clustering," *Proceedings of the ACM on Programming Languages*, vol. 5, p. 159:1–159:27, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3485547

[19] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Communications of the ACM*, vol. 58, p. 44–46, Jan. 2015. [Online]. Available: https://doi.org/10.1145/2644805

[20] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, DIKU, University of Copenhagen, 1994.

[21] ISO/IEC JTC1/SC22, "Programming languages - C - a provenance-aware memory object model for C," ISO/IEC, Technical Specification 6010:2025, May 2025. [Online]. Available: https://webstore.iec.ch/en/publication/107524

[22] Standard Performance Evaluation Corporation, "SPEC CPU2017 benchmark suite," 2017. [Online]. Available: http://www.specbench.org/cpu2017/

[23] A. Rountev and S. Chandra, "Off-line variable substitution for scaling points-to analysis," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 2000, p. 47–56. [Online]. Available: https://doi.org/10.1145/349299.349310

[24] D. J. Pearce, P. H. Kelly, and C. Hankin, "Online cycle detection and difference propagation: Applications to pointer analysis," *Software Quality Journal*, vol. 12, pp. 311–337, Dec. 2004. [Online]. Available: https://doi.org/10.1023/B:SQJO.0000039791.93071.a2

[25] ——, "Efficient field-sensitive pointer analysis of C," *ACM Transactions on Programming Languages and Systems*, vol. 30, p. 4–es, Nov. 2007. [Online]. Available: https://doi.org/10.1145/1290520.1290524

[26] J. S. Foster, M. Fahndrich, and A. Aiken, "Flow-insensitive points-to analysis with term and set constraints," Tech. Rep., Jul. 1997.

[27] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International*

Symposium on Code Generation and Optimization, Mar. 2004, pp. 75–86. [Online]. Available: https://doi.org/10.1109/CGO.2004.1281665

[28] N. Heintze and O. Tardieu, "Ultra-fast aliasing analysis using CLA: a million lines of C code in a second," *ACM SIGPLAN Notices*, vol. 36, p. 254–263, May 2001. [Online]. Available: https://doi.org/10.1145/381694.378855

[29] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken, "Partial online cycle elimination in inclusion constraint graphs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 1998, p. 85–96. [Online]. Available: https://doi.org/10.1145/277650.277667

[30] B. Hardekopf and C. Lin, "Exploiting pointer and location equivalence to optimize pointer analysis," in *Static Analysis*, H. R. Nielson and G. Filé, Eds., 2007, pp. 265–280. [Online]. Available: https://doi.org/10.1007/978-3-540-74061-2_17

[31] ISO/IEC JTC1/SC22, "Information technology – programming languages – C," ISO/IEC, International Standard 9899:2024, 2024.

[32] C. D. Feather, "Defect report #260," Sep. 2004. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm

[33] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell, "Exploring C semantics and pointer provenance," *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, vol. 3, p. 67:1–67:32, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290380

[34] J. Lee, C.-K. Hur, R. Jung, Z. Liu, J. Regehr, and N. P. Lopes, "Reconciling high-level optimizations and low-level code in LLVM," *Proceedings of the ACM on Programming Languages*, vol. 2, p. 125:1–125:28, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276495

[35] J. Gustedt, P. Sewell, K. Memarian, V. B. F. Gomes, and M. Uecker, "A provenance-aware memory object model for C," ISO/IEC, Draft Technical Specification N2676, Mar. 2021.

[36] "JLM: A research compiler based on the RVSDG IR," Apr. 2024. [Online]. Available: https://github.com/phate/jlm

[37] N. Reissmann, J. C. Meyer, H. Bahmann, and M. Själander, "RVSDG: An intermediate representation for optimizing compilers," *ACM Transactions on Embedded Computing Systems*, vol. 19, pp. 49:1–49:28, Dec. 2020. [Online]. Available: https://doi.org/10.1145/3391902

[38] R. E. Tarjan, *Disjoint Sets*. Society for Industrial and Applied Mathematics, Jan. 1983. [Online]. Available: https://doi.org/10.1137/1.9781611970265.ch2

[39] V. Nagaraj and R. Govindarajan, "Approximating flow-sensitive pointer analysis using frequent itemset mining," in *Proceedings of the International Symposium on Code Generation and Optimization*, Feb. 2015, p. 225–234. [Online]. Available: https://doi.org/10.1109/CGO.2015.7054202

[40] M. Thakur and V. K. Nandivada, "PYE: A framework for precise-yet-efficient just-in-time analyses for Java programs," *ACM Transactions on Programming Languages and Systems*, vol. 41, p. 16:1–16:37, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3337794

[41] A. Le, O. Lhoták, and L. Hendren, "Using inter-procedural side-effect information in JIT optimizations," in *Proceedings of the International Conference on Compiler Construction*, R. Bodik, Ed., 2005, pp. 287–304. [Online]. Available: https://doi.org/10.1007/11406921_22

[42] S. Halalingaiah, V. Sundaresan, D. Maier, and V. K. Nandivada, "The ART of sharing points-to analysis: Reusing points-to analysis results safely and efficiently," *Proceedings of the ACM on Programming Languages*, vol. 8, p. 363:2606–363:2632, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3689803

[43] A. Anand, S. Adithya, S. Rustagi, P. Seth, V. Sundaresan, D. Maier, V. K. Nandivada, and M. Thakur, "Optimistic stack allocation and dynamic heapification for managed runtimes," *Proceedings of the ACM on Programming Languages*, vol. 8, p. 159:296–159:319, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3656389

[44] A. Rountev and B. G. Ryder, "Points-to and side-effect analyses for programs built with precompiled libraries," in *Proceedings of the International Conference on Compiler Construction*, Apr. 2001, pp. 20–36. [Online]. Available: https://doi.org/10.1007/3-540-45306-7_3

[45] A. Rountev, S. Kagan, and T. Marlowe, "Interprocedural dataflow analysis in the presence of large libraries," in *Proceedings of the International Conference on Compiler Construction*, A. Mycroft and A. Zeller, Eds., 2006, pp. 2–16. [Online]. Available: https://doi.org/10.1007/11688839_2

[46] Y. Smaragdakis and G. Kastrinis, "Defensive points-to analysis: Effective soundness via laziness," in *Proceedings of the European Conference on Object-Oriented Programming*, T. Millstein, Ed., 2018, p. 23:1–23:28. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2018.23

[47] H. R. Krogstie, "PIP: Making andersen's points-to analysis sound and practical for incomplete c programs (artifact)," Aug. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.16900791

[48] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, "EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure," Feb. 2022. [Online]. Available: http://arxiv.org/abs/1912.05848