Efficient Pointer Analysis via Def-Use Graph Pruning

Jiaqi He
University of Alberta
jhe15@ualberta.ca

Karim Ali
NYU Abu Dhabi
karim.ali@nyu.edu

***Abstract*—Precise pointer information is essential for data-flow analysis. While flow-sensitive pointer analysis yields highly precise results, it introduces more computations. To provide a better trade-off between precision and performance, prior work has developed LEVPA, a pointer analysis that processes pointers according to their pointer level, generally accelerating the analysis. However, LEVPA still performs redundant computations for intermediate variables that are generated during Static Single Assignment (SSA) transformation. For real-world programs, we have observed that those variables may constitute up to 87% of all analyzed variables. These unnecessary computations cause LEVPA to timeout for large programs.**

To address these limitations, we present Level-based Intermediate variable Skipping Pointer Analysis (LISPA), an enhancement of LEVPA that operates on a def-use graph without computing points-to set of intermediate variables introduced during SSA transformation. The key insight behind LISPA is that intermediate variables do not introduce new pointers into the analysis, because they always alias with variables in the original program. Therefore, there is no need to compute their points-to set in a fixed-point computation. Across 27 real-world C/C++ programs, we show that LISPA is on average $2.14\times$ faster than LEVPA, while using $2.38\times$ less memory.

***Index Terms*—Flow Sensitivity, Pointer Analysis, Intermediate Variables**

I. INTRODUCTION

Pointer analysis computes the set of memory locations to which program variables may point. This information is essential to various data-flow analyses that are used to perform compiler optimizations (e.g., dead code elimination [1]), reason about semantic properties of programs (e.g., data races detection [2]), and detect security vulnerabilities (e.g., SQL injection analysis [3]). To reduce the number of resulting false positives, a precise data-flow analysis requires precise points-to information. For example, well-established compilers such as LLVM [4] use precise pointer information to perform code optimizations that ensure a program behaviour remains intact.

Traditionally, flow-sensitive pointer analysis is modelled as a fixed-point computation problem over the Control-Flow Graph (CFG) of a program. However, this model is not practical for programs with more than 100K lines of code (LOC) due to the tremendous size of the CFG. To accelerate flow-sensitive pointer analysis, LEVPA [5] groups all pointers according to their pointer level. LEVPA starts with the highest pointer level and constructs a sparse graph representing the define-and-use relationship of points-to sets between program locations. LEVPA then transform the related instructions into full SSA form [6] and computes their points-to set using an Andersen-style pointer analysis [7]. The computed points-to set is used to construct the sparse graph for the next pointer level. This level-by-level processing enables LEVPA to scale to programs with more than 100 KLOC. However, LEVPA unnecessarily computes points-to sets of intermediate variables that are introduced during SSA transformation, hindering its ability to reach its full potential performance. As we will show on our benchmark suite, intermediate variables may constitute up to 87% of all analyzed variables, rendering most LEVPA points-to set computations redundant. Our key observation here is that these intermediate variables do not introduce new pointers into the analysis because they always alias to existing ones. Therefore, an efficient pointer analysis should not process them in a similar fashion to original program variables by computing and propagating their points-to sets using an Andersen-style pointer analysis.

To address the inefficiencies of traditional level-by-level pointer analysis, we present Level-based Intermediate variable Skipping Pointer Analysis (LISPA), an enhancement of LEVPA that operates on a Def-Use Graph (DUG) without computing the points-to sets of intermediate variables introduced during SSA transformation. LISPA treats intermediate variables differently by directly passing points-to sets of the pointers that alias with these intermediate variables to the program locations that use them. Unlike LEVPA, LISPA does not compute and propagate the points-to set for intermediate variables to compute the DUG. Instead, for a specific pointer level, LISPA builds def-use edges for pointers at that level with any def-use relationship on intermediate variables being eliminated. When a def-use edge leads to an intermediate variable as its destination, LISPA automatically update the destination to the use locations of that intermediate variable. LISPA then performs its fixed-point computation directly on those edges before moving on to process the next pointer level. This approach enables LISPA to avoid redundant points-to set computations and propagation for intermediate variables.

To evaluate LISPA, we implemented it as an optimization analysis in LLVM 17 [4]. We then ran LISPA on 16 benchmarks from the SPEC CPU 2017 benchmark suite [8] and 14 benchmarks from prior work [9], ranging from 1K to 1M+ LOC. Across 27 benchmarks for which at least one analysis terminates within the time limit, LISPA is $2.14\times$ (min: $1.01\times$, max: $13.65\times$, geomean: $2.14\times$) faster than LEVPA, while consuming $2.38\times$ (min: $1.21\times$, max: $6.82\times$, geomean: $2.38\times$) less memory. To further evaluate its competitiveness, we compare LISPA with two state-of-the-art flow-sensitive pointer analyses: SFS [10] and VSFS [9]. On average, LISPA

is 2.12× (min: 0.01×, max: 23.59×, geomean: 2.12×) faster than SFS and has a slightly worse performance than VSFS (min: 0.01×, max: 15.39×, geomean: 0.88×).

II. Background

A. Flow-Sensitive Pointer Analysis

Traditional flow-sensitive pointer analysis is based on finding a fixed-point solution on a CFG $G = \langle V, E \rangle$. Each node $v \in V$ represents a *basic block*, and each edge $e = (v_1, v_2) \in E$ represents the control flow between two basic blocks. Each node $v \in V$ has two transfer functions associated with it that compute the points-to set before entering ($\texttt{in}_v$) and after exiting ($\texttt{out}_v$) node v. The functions rely on $\texttt{pred}(v)$, $\texttt{gen}_v$, and $\texttt{kill}_v$. While $\texttt{pred}(v)$ represents the set of predecessors of node v and only depend on the CFG, $\texttt{gen}_v$ and $\texttt{kill}_v$ represent the set of pointer information generated and eliminated at node v, relying on the semantics of v. The analysis iteratively computes $\texttt{in}(v)$ and $\texttt{out}(v)$ for each $v \in V$ according to Equation 1 until convergence.

$$\begin{aligned} \text{in}_v &= \bigcup_{v' \in \text{pred}(v)} \text{out}_{v'} \\ \text{out}_v &= \text{gen}_v \cup (\text{in}_v - \text{kill}_v) \end{aligned} \tag{1}$$

B. Intermediate Representations (IRs)

SSA form [11] is an IR that ensures each variable in the program is defined only once throughout a program. Converting a program into SSA form is beneficial for performing data-flow analysis [12], because SSA explicitly shows the relation between the definition and uses of a variable. However, translating programs into full SSA form requires pointer information due to potential indirect definitions/uses. To work around this dependency, modern compilers use partial SSA form, where only variables whose addresses are never taken (i.e., top-level variables) are represented in SSA form. All operations on the remaining variables (i.e., address-taken variables) are performed using `store` and `load` instructions.

LLVM IR is based on partial SSA form, where all top-level variables are put into SSA form, which means each has only one definition. However, address-taken variables are never explicitly used throughout the IR. Therefore, their points-to sets may be indirectly updated multiple times using `store` and `load` instructions. While a top-level variable may be introduced by both a `load` instruction and an `alloca` instruction, an address-taken variable is only introduced at an `alloca` instruction. For example, an `alloca` instruction `x = alloca ptr` introduces both a top-level variable `x` and an address-taken variable `atv_x` into the analysis and sets `x` points to `atv_x`. In this paper, we use lowercase letter to represent a top-level variable (e.g., `a`) and prefix a variable name with `atv_` to represent an address-taken variable (e.g., `atv_a`).

C. Def-Use Graph (DUG)

A DUG is a graph $G = \langle V, E \rangle$ where each $v \in V$ represents an instruction and each edge $e = (v_1, v_2, \textit{label}) \in E$ represents the define and use relationship among instructions. A DUG is useful in pointer analysis because it connects use locations of a points-to set directly to its corresponding defining location. Compared to operating on a CFG, computing a fixed-point solution on a DUG eliminates the propagation of unnecessary information because all modifications to a points-to set from a definition location are propagated directly to use locations.

However, computing def-use information for all variables in a program requires pointer information because pointers introduce indirect memory accesses. Due to potential aliases, it is difficult to determine whether a particular memory location is being read or written through a pointer. To build a DUG, existing techniques either perform a *staged analysis* or *level-by-level analysis*. A staged analysis bootstraps a flow-sensitive pointer analysis with the result of a flow-insensitive pointer analysis that over-approximates the precise points-to sets at each program location. Following this approach, a DUG may contain imprecise def-use edges, along which the main analysis may later propagate the points-to set of a pointer that may not be used at the use locations of those imprecise edges. In contrast, level-by-level analysis groups pointers according to their pointer levels and processes each pointer level flow-sensitively. Later, the result of the current pointer level is used to bootstrap the analysis in the next pointer level.

III. Avoiding Unnecessary Computations

To scale pointer analysis to large codebases, Yu et al. [5] introduced LevPA, an analyzer that associates each pointer with a numeric value called *pointer level* that represents the number of dereferences needed to retrieve a non-address value. LevPA then groups pointers in a program according to their pointer levels and computes a fixed-point solution for each group of pointers in each round of the analysis. To achieve that, LevPA starts by setting the current pointer level l_c to the highest pointer level among all pointers in the program. Once LevPA computes the points-to sets of all pointers at l_c, it decrements it by one to process the next pointer level with the points-to sets computed by far. This approach yields correct results because the points-to set of a pointer `p` (i.e., `pts(p)`) may only be modified by an assignment to `p` (e.g., `p = q`) or dereferencing a pointer with a higher pointer level (e.g., `store x y` and $\texttt{p} \in \texttt{pts(y)}$). Thus, if LevPA processes pointers in a descending order of pointer levels, when it reaches pointer level l, the points-to sets of all pointers at pointer levels higher than l would have been computed and will remain fixed during the remainder of the analysis. Unlike LevPA, LisPA handles intermediate variables that are introduced during translation into partial SSA form differently. Our key observation is that those intermediate variables do not introduce new pointers into the analysis, because they are introduced to represent specific versions of address-taken variables at a program location. As a result, these intermediate variables always alias with existing pointers. Thus, it is redundant to compute and propagate their points-to sets during fixed-point computation.

Figure 1 shows an example where the intermediate variable `%1` aliases with the address-taken variable `atv_a` (Line 17),

```
1  def main()                        16 branch1:
2  entry:                            17   %1 = load a
3    A = alloca i8                   18    store 'A' %1
4    B = alloca i8                   19 branch2:
5    a = alloca i8*                  20   %2 = load t1
6    b = alloca i8*                  21   %3 = load t2
7    tmp = alloca i8*                22   call swap %2 %3
8    t1 = alloca i8**                23   %4 = load a
9    t2 = alloca i8**                24   store 'B' %4
10   store a t1                      25 end:
11   store b t2                      26   %5 = load a
12   store A a                       27   store '?' %5
13   store B b                       28 def swap(i8** p, i8** q)
14   br <cond> branch1 branch2       29  omitted...
```

Fig. 1: The simplified LLVM IR of a program that swaps the points-to sets of two pointers.

because top-level variable a is set to (Line 5) point to address-taken variable atv_a throughout the program. By the semantics of load instructions, we know the points-to set of %1 is an exact copy of the points-to set of atv_a. Thus, there is no need to compute the points-to set of %1 once the points-to set of atv_a is known. To build correct def-use chains for atv_a, we can transit all def-use edges for atv_a that end in Line 17 to the use locations of %1 (Line 18). The transition of def-use edge enables LISPA to avoid computing points-to set of intermediate variables, which is required by LEVPA due to using an Andersen-style pointer analysis [7]. In this section, we present the algorithm detail of LISPA. We first explain how LISPA works intra-proceduraly, and we then show how to extend it to the inter-procedural case.

A. Computing Pointer Levels in LLVM

LISPA processes input programs in partial SSA form. In the remainder of this paper, we will use LLVM IR as the primary partial SSA form that LISPA accepts, without loss of generality to its applicability to other partial SSA forms. To decompose complex assignments and maintain SSA form, LLVM IR introduces a list of temporary variables to store the intermediate result of dereferencing a pointer. A pointer variable in the original source code is then allocated in the IR using an alloca instruction. A well-formed LLVM IR instruction implicitly imposes the pointer level relationship for its pointer operands. For example, the IR instruction "%0 = load a" implies that pointer a is one pointer level higher than %0. However, the introduction of opaque pointers in LLVM 17 omits the detailed type information of a pointer from the textual representation. This omission forces us to compute pointer levels using the same approach of LEVPA. In particular, LISPA runs a Steensgaard-style analysis [13], which runs in almost linear time, to compute imprecise points-to information. Using the results of this pre-analysis, LISPA constructs a points-to graph and detects all Strongly Connected Components (SCC) in the graph. The pointer level of a SCC s is computed recursively (i.e., $pl(s)$) using the equation: $pl(s) = \max_{k \in \texttt{pts(s)}} pl(k) + 1$. Then, the pointer level of a pointer p (i.e., $pl(p)$) has pointer level l if $p \in s$ and $pl(s) = l$.

Algorithm 1: Intra-procedural LISPA

```
Data: A program prog
Result: A CFG annotated with points-to information
for each function F in prog do
    workList, pointerLevel ← Initialize(F);
    while pointerLevel ≠ 0 do
        BuildDefUseGraph(pointerLevel, worklist);
        propagateList ←
          initializePropagateList();
        Propagate(propagateList);
        CreateDefUseLabels(pointerLevel, worklist);
        pointerLevel ← pointerLevel - 1
    end
end
return;
```

B. Intra-Procedural LISPA

Algorithm 1 shows the pseudocode of the core algorithm for intra-procedural LISPA, which consists of four steps.

1) Initialization: LISPA starts off by constructing a worklist for each analyzed function. The higher the pointer level of a pointer, the higher its priority is in a worklist. LISPA then finds the highest pointer level across all worklists and starts processing pointers in a descending order of pointer levels.

2) Building Def-Use Graph: Similar to LEVPA, LISPA only processes pointers with l_c for the current analysis round. Each analysis round starts with building def-use chains for all pointers p such that $pl(p) = l_c$, where pl maps a pointer to its pointer level. LISPA guarantees that when new def-use edges are constructed, all def labels and use labels regarding pointers in l_c are already marked. One exception to this is when LISPA is processing the highest pointer level, where no labels are marked. However, since there are no address-taken variables in that pointer level, it is safe to construct an empty DUG in this case.

To bypass intermediate variables in building def-use edges, LISPA recognizes all define and use labels introduced when an intermediate variable is created. Without special handling, such labels results in def-use edges with intermediate variables as destinations. LISPA then moves all such labels to the use locations of that intermediate variable. LISPA repeats this label updating scheme for all pointers in l_c before moving to the next step of the algorithm. The correctness of LISPA comes from the following theorem.

Theorem 1. *At a pointer level, the def-use label set created by* LISPA *is obtained from the corresponding* LEVPA *set by filtering out all use labels introduced at program locations that introduce intermediate variable, while retaining all others.*

Informally, a use label $u(a)$ at a program location that introduces an intermediate variable $\%0$ is removed, because LISPA moves all such labels to other program locations. The moving destination always contain another use label $u'(a)$ that LEVPA creates due to using $\%0$, which aliases with a.

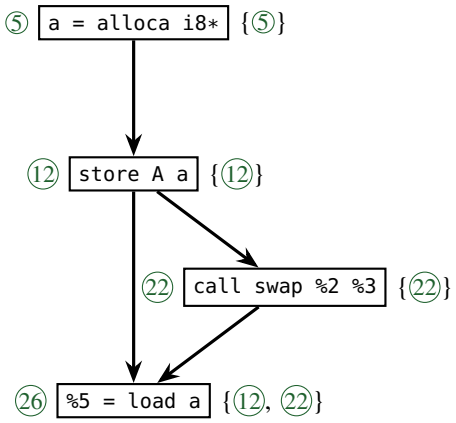

Fig. 2: Dominator graph for atv_a from Figure 1. Each node has a label, which is its line number from Figure 1, shown to its left. The figure shows the out set of each node to its right.

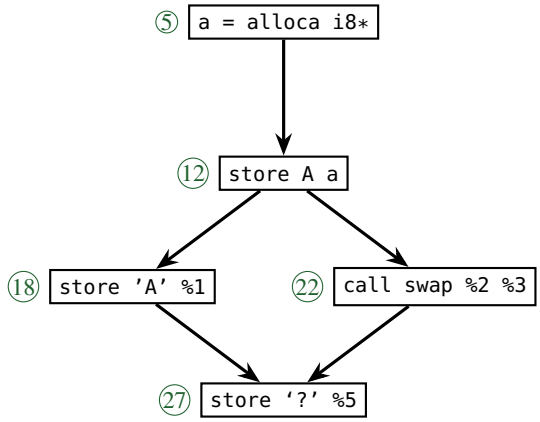

Fig. 3: The part of the DUG related to atv_a from Figure 1. Each node has a label, which is its line number from Figure 1, shown to its left.

For a define label $d(a)$ created by LEVPA, it is also created by LISPA because LISPA recognizes all uses of intermediate variables in a `store` instruction `store x %1` and creates define labels for the aliases to %1.

With all def/use labels marked for a pointer, LISPA determines where the possible locations that define the used points-to set are. Similar to LEVPA, to determine the source of a points-to set, LISPA computes the iterated dominance frontier [14] of each definition label by building a dominator graph for all definition labels. Accordingly, the source of a points-to set at a program location is its immediate dominator in the dominator graph. To compute its fixed-point solution, LISPA then records each def-use relationship as a def-use edge in the DUG. Figure 2 shows the dominator graph of atv_a from Figure 1. All nodes except ㉖ represent a definition location of atv_a. LISPA adds ㉖ to the dominator graph, because more than one control-flow reaches its basic block. In the figure, we associate each node with the set of definition locations that affect pts(atv_a). The call node ㉒ also defines atv_a, and we will explain how LISPA handles it later when we extend LISPA to the inter-procedural case. To compute the program locations that pass the effect of a points-to set definition for a node, LISPA performs the following fixed-point computation based on the following set of equations:

$$\begin{aligned} \text{in}_v &= \bigcup_{v' \in \text{pred}(v)} \text{out}_{v'} \\ \text{out}_v &= \text{gen}_v \cup (\text{in}_v - \text{kill}_v) \\ \text{gen}_v &= \{v\} \text{ if } v \text{ is a definition location else } \emptyset \\ \text{kill}_v &= \text{in}_v \text{ if strong update else } \emptyset \end{aligned} \tag{2}$$

In Figure 2, ⑤ creates pointers a and atv_a. Therefore, LISPA marks them as definition locations for a and atv_a by considering them as initializations, with random values, of the points-to set of a pointer. LISPA then updates the dominating definition locations of ⑫ and ㉒ to themselves, because they perform strong updates for the underlying points-to set and all definitions before them are irrelevant to the use location. Finally, when the two branches merge at ㉖, LISPA updates its dominating definition location to be the union of the out sets of ⑫ and ㉒, because they may affect pts(atv_a) at ㉖.

The source of a points-to set can then be determined with the above dominator graph. Figure 3 shows the def-use edges related to atv_a from Figure 1. For each program location with a use label for atv_a, LISPA finds its immediate dominator in the dominator graph and creates a def-use edge for atv_a. As we have previously discussed, LISPA transfers labels that have been generated for intermediate variables to the program locations that use them. Therefore, instead of creating a def-use edge from ⑫ to ⑰ like LEVPA, LISPA creates a def-use edge from ⑫ to ⑱.

3) Propagating Points-To Sets: Given a DUG, LISPA starts at the first definition location of a pointer p. It then passes pts(p) from that location to all uses of p in the DUG. Then, pts(p) is defined or used according to the semantics of the instruction at the use location. If pts(p) is updated at the use location, LISPA propagates it until it remains fixed at all program locations. LISPA performs another fixed-point computation similar to Equation (2) on the current DUG, but with gen_v = pts(x) if v is `store x y` else $\emptyset$.

4) Creating Def-Use Labels: When LISPA computes the fixed-point solution for all pointers at l_c, it concludes its iteration by creating all indirect def-use labels for pointers at the next pointer level. The fixed-point solution computed at Line 7 of Algorithm 1 represents the final version of the points-to sets for all pointers at l_c. Equipped with a fixed and correct points-to set, LISPA knows how pointers in the next pointer level are indirectly defined or used. For example, the instruction at Line 12 in Figure 1 defines the points-to set of atv_a, the pointer that is pointed to by a. To process a, LISPA creates a definition label for atv_a so that it knows that pts(atv_a) is defined here and that its original points-to set is required in case of a weak update. For each `load` instruction `a = load b`, LISPA creates a use label for all pointers p such that

$p \in$ `pts(b)`. For each instruction `store a b`, LISPA creates both a def label and a use label for each pointer $p \in$ `pts(b)`, where the use label supports performing a weak update.

C. Inter-Procedural LISPA

To extend LISPA to the inter-procedural case, LISPA should handle pointers used as arguments at a call site and returned from a callee, which requires computing the points-to sets for parameters of all functions. Since LISPA is flow-sensitive, it does not need to differentiate the points-to set of a parameter under a different calling context. Whenever LISPA detects that the points-to set of an argument to a function has been modified, it triggers another iteration of fixed-point computation by updating the points-to set of the corresponding parameter. Similarly, if the points-to set of the return value has been modified, LISPA passes the new points-to set back to the call site, triggering another fixed-point computation at the caller.

Inter-procedural LISPA also requires building DUG edges that represent def-use relation across functions. Since pointers may be passed across functions, the points-to set of a pointer that has been allocated in one function may be defined (or used) in other functions. To build inter-procedural def-use edges, LISPA can simply build the direct inter-procedural def-use edge by connecting program locations in different functions. However, this approach is hard for large programs, because it would require finding inter-procedural def-use edges over large inter-procedural Control Flow Graph (ICFG). Instead, LISPA splits an inter-procedural def-use edge from function f to function g into a set of intra-procedural def-use edges. To achieve that, LISPA creates two def-use edges when a points-to set of a pointer is defined or used in g. One edge is from the immediate dominating definition location to the call site in f. The other edge starts from the entry of g and ends at the program location that defines (or uses) the pointer within g. If the points-to set of a pointer is later defined or used in f after calling g, LISPA creates def-use labels at the call site to g to correctly propagate the updated points-to set.

Inter-procedural LISPA must maintain the invariance of level-by-level analysis; all pointers with a higher pointer level must be processed before analyzing pointers at a lower pointer level. Since functions may have different highest pointer levels, and they may call each other, processing all pointers in a function before moving on to process other functions breaks the level-by-level invariance. To overcome this challenge, LISPA first gets the highest pointer level across all functions. For each pointer level, LISPA processes all pointers at it for all functions before processing the next pointer level. By induction, we can then show that LISPA maintains the invariance, primarily by showing that there will be no missing def-use labels for any pointers at the current level l when LISPA starts building DUG edges for these pointers. For the base case, LISPA is analyzing the highest pointer level, where there are no missing def-use labels, because pointers at that level do not have indirect definitions or uses. For the induction case, let's assume LISPA is at pointer level l of function f, f calls another function g, and the highest pointer level of the parameters of g is m. If $l > m$, then all pointers in g with pointer level $\geq l$ cannot define or use the points-to set of a pointer allocated in f, because there is no control-flow among these variables. If $l = m$, g may define or use a pointer `p` with pointer level $l - 1$ as a result of passing arguments to the points-to sets of parameters. By induction on l, LISPA has already created all labels for pointers at level l, it can correctly compute the points-to set of these pointers. Therefore, LISPA correctly creates all def-use labels in g for pointers allocated in f if they are pointed to by a parameter of g.

IV. EVALUATION

We have implemented LISPA in LLVM 17 as a module analysis. We chose LLVM 17 as our platform to show that our implementation works on recent LLVM versions that use opaque pointers.

To evaluate LISPA, we compare its runtime performance and memory usage to LEVPA [5]. Comparing LISPA to LEVPA assesses the potential performance gains due to eliminating the redundant computations of the points-to sets of intermediate variables. Since the original implementation of LEVPA is no longer available, we had to reimplement it by carefully following the algorithm described by the authors [5]. To ensure a fair comparison with LISPA, we implemented a version of LEVPA that is flow-sensitive but context-insensitive. To evaluate LISPA against more modern pointer analyses, we compare it with two state-of-the-art flow-sensitive pointer analyses: SFS [10] and VSFS [9]. Unlike LISPA, SFS and VSFS are staged analyses, which means they first build a DUG by running a flow-insensitive pointer analysis as a pre-analysis.

A. Benchmarks

Since several programs from the original LEVPA benchmark are no longer available, we could not use them for our empirical evaluation. To evaluate the performance of LISPA across different sizes of programs, we instead chose 30 programs from two benchmark suites: 16 programs from the SPEC CPU 2017 benchmark [8] and 14 programs from the evaluation suite for VSFS [9]. Table I shows more details about our full benchmark suite. All programs are written in C/C++ and their size range from a 1 KLOC to more than 1,500 KLOC. We have compiled them using Clang 17.0.1 [15] with optimization set to `-O0`. We choose this optimization level to keep the code structure of the original benchmark as enabling other optimization levels potentially alter control flow or pointer behavior.

All analyses take a single compiled bitcode as their input. For programs in SPEC CPU 2017, we replace the linker in the compilation script with `llvm-link` to force the script to output a single bitcode file instead of a binary. For the VSFS benchmark, we use `wllvm`, which internally uses `llvm-link`, to compile each input program into a single bitcode file.

B. Experimental Setup

We have conducted all our experiments on a server with AMD EPYC 7531 16-Core processor, 488 GB RAM, running

Ubuntu 22.04. To reduce the impact of the execution environment, we ran each analysis 5 times for each program and report the average performance across all runs. To measure the runtime for LISPA and LEVPA, we use the LLVM option `-time-passes`, which reports the time used in each pass. To measure the memory usage for each analysis/pass, we use the LLVM option `-track-memory`. For the comparison against SFS and VSFS, we use the direct output of both analyses. We set the timeout for each analysis run to be 2 hours, because we noticed that if an analysis runs for longer than 2 hours, it rarely terminates even when we extend the timeout to 12 hours. We also set the memory limit to 488 GB (i.e., all available RAM).

C. Comparing LISPA to LEVPA

Table I reports the running time and memory usage of LEVPA and LISPA for all benchmark programs. For each program, we highlight the best performing analysis in bold. For LISPA, we also report the time it takes to compute pointer levels, the time to build Memory SSA, and the time to solve its fixed-point solution. This decomposition gives a better understanding of the various components that contribute to the running time of LISPA.

With respect to the overall runtime, Table I shows that LISPA has better performance than LEVPA for all 27 programs that at least one analysis can analyze. Across all programs, LISPA achieves an average speedup of $2.14\times$ (min: $1.01\times$, max: $13.65\times$, geomean: $2.14\times$) compared to LEVPA, showing that removing the unnecessary computation and propagation of the points-to sets for intermediate variables helps accelerate level-by-level flow-sensitive pointer analysis. Nevertheless, LISPA and LEVPA fail to analyze 3 programs (PERLBENCH, GCC, BLENDER) within the time limit. This is because the average points-to sets size is quite large for both analyses to handle, because these programs have relatively large program size (over 500 KLOC in LLVM IR instructions). However, we have observed that a large program size does not always imply longer analysis time. For example, NINJA has 198 KLOC LLVM IR instructions, yet LISPA takes only 0.76 seconds to analyze it. On the other hand, LISPA requires 2.31 seconds to analyze NANO, which is a program with only 87 KLOC LLVM IR instructions.

To evaluate any potential loss in precision, we have compared the resulting points-to sets at every program point. For all pointer queries, both LISPA and LEVPA compute the same points-to set, demonstrating that LISPA maintains the same precision as LEVPA despite avoiding unnecessary points-to propagation and computation for intermediate variables.

> On average, LISPA is $2.14\times$ faster than LEVPA, without any loss in precision of the computed points-to sets.

With respect to memory usage, Table I shows that LISPA uses $2.38\times$ less memory (min: $1.21\times$, max: $6.82\times$, geomean: $2.38\times$) than LEVPA, confirming our initial key observation that removing the computation and propagation for the points-to sets of intermediate variables should reduce the memory used by the analysis. Similar to the running time, we have observed that the memory usage of both analyzers does not always increase with the size of the analyzed program. For example, LEVPA and LISPA use 28.65 MB and 12.36 MB, respectively, to analyze NINJA. However, they use 186.71 MB and 66.27 MB, respectively, to analyze LEELA, which is $2.48\times$ smaller than NINJA.

We have also observed that the running time for LISPA is correlated to its memory usage. If LISPA spends more time analyzing a program, it is more likely that it uses more memory and vice versa. Running a correlation test between the memory usage and the runtime of LISPA shows that their Pearson product-moment correlation coefficient [16] is 0.86, indicating that they are indeed positively correlated.

> On average, LISPA uses $2.38\times$ less memory than LEVPA. The longer LISPA runs for, the more memory it uses.

D. Discussion

Both LISPA and LEVPA perform pointer analysis in a level-by-level fashion. Performing the analysis this way does not require a pre-analysis that computes def-use chains for address-taken variables. While LISPA removes the computation and propagation of the points-to set for intermediate variables, it also introduces more work in finding the def-use chains for address-taken variables. Therefore, we have investigated several factors of the input program that may affect the performance of LISPA.

Table II shows detailed statistics that we have collected while running LISPA. For each program, we show the number of different LLVM IR instructions that LISPA has processed. We also show the number of nodes and edges of the built DUG, as well as the average points-to set size. The number of `getelementptr` and `bitcast` instructions are shown together as `copy`, because LISPA considers them both as copy instructions. Since LEVPA and LISPA fail to analyze PERLBENCH, GCC, and BLENDER, we could not collect their DUG information.

Examining the data in Table II, we observe that the complexity of DUG construction affects the performance of LISPA. This is expected in a fixed-point computation, because LISPA must propagate the points-to set from a node to all its children in the DUG to compute its solution. We quantify the complexity of a DUG by computing the ratio between the number its edges and the number of its nodes. The higher that ratio is, the more complex the DUG is, because an update to a points-to set at a node will on average affect more nodes. The results suggest that if a DUG has more edges per node, it makes the program harder to analyze. For example, LISPA spends 203.3 seconds analyzing MUTT while spending 3,234.43 seconds analyzing MRUBY. Both programs are similar in size and number of nodes in their DUGs. However, the DUG for MRUBY contains approximately $26\times$ more edges than MUTT, which is the main reason behind its longer analysis time. Overall, LISPA spends more time when the generated DUG has a high edge-node ratio.

TABLE I: Benchmark program information, running time (in seconds) and memory usage (in MBs) for LISPA and LEVPA. For each program, we list the number of lines of the original C/C++ source code in KLOC and the number of compiled LLVM IR instructions in KLOC. The shortest time and least used memory in a row are in bold. *OOT* indicates analysis timeouts, and "-" means the data cannot be collected due to timeouts.

Program	Source	LLVM	Memory Usage (MBs)			Running Time (Seconds)					
			LEVPA	LISPA	Rate	LEVPA	LISPA				Rate
							Level	DUG	Solving	Total	
LBM	1	8	3.44	**1.88**	1.83	0.07	0.01	0.01	0.02	**0.05**	1.4
MCF	3	8	3.99	**2.97**	1.34	0.12	0.02	0.02	0.03	**0.08**	1.5
DEEPSJENG	10	31	14.83	**9.25**	1.6	0.72	0.1	0.1	0.11	**0.31**	2.32
DPKG	40	58	72.62	**27.27**	2.66	17.76	0.24	0.31	0.58	**1.31**	13.65
NAB	24	60	34.2	**24.05**	1.42	1.76	0.24	0.27	0.43	**1.05**	1.68
DU	28	60	33.24	**24.34**	1.37	1.41	0.21	0.21	0.45	**1.07**	1.32
XZ	33	60	34.05	**23.03**	1.48	1.51	0.24	0.13	0.38	**0.89**	1.7
LEELA	21	80	186.71	**66.27**	2.82	16.66	0.41	0.51	2.1	**3.85**	4.33
NANO	59	87	161.97	**71.11**	2.28	118.39	0.33	78.21	34.64	**113.62**	1.04
PSQL	25	89	80.2	**36.27**	2.21	56.78	0.29	7.15	1.97	**10.26**	5.53
JANET	23	94	155.27	**65.62**	2.37	124.2	0.42	80.91	13.79	**96.35**	1.29
I3	27	109	64.44	**45.4**	1.42	3.63	0.46	0.73	0.83	**2.26**	1.61
BAKE	25	117	170.87	**38.05**	4.49	25.72	0.23	1.68	15.86	**18.19**	1.41
TMUX	48	128	146.95	**28.24**	5.2	20.36	0.44	0.27	0.65	**1.61**	12.65
ASTYLE	14	155	276.64	**40.57**	6.82	59.18	0.62	0.33	1.21	**4.62**	12.81
X264	96	194	98.53	**73.49**	1.34	4.32	0.86	0.43	1.82	**3.48**	1.24
NINJA	17	198	28.65	**12.36**	2.32	2.31	0.16	0.2	0.39	**0.76**	3.04
MRUBY	48	218	244.29	**168.44**	1.45	3,504.56	10.4	2,062.89	1,159.54	**3,234.43**	1.08
MUTT	96	232	2,764.15	**691.76**	3.99	304.45	0.93	38.88	162.03	**203.3**	1.5
NAMD	8	271	191.09	**157.9**	1.21	8.53	1.34	0.05	2.93	**5.63**	1.52
POVRAY	170	275	1,370.44	**305.96**	4.48	571.15	1.25	334.93	94.52	**482.22**	1.18
BASH	115	304	446.93	**198.41**	2.25	524.63	1.09	397.69	160.77	**512.71**	1.02
LYNX	134	348	1,286.96	**928.65**	1.39	477.16	1.26	164.06	306.43	**472.87**	1.01
IMAGICK	259	555	5,289.76	**594.45**	8.89	3,590.6	2.68	1,465.87	2,040.05	**3,509.97**	1.02
OMNETPP	134	556	3,282.58	**492.11**	6.67	542.7	2.86	10.71	30.22	**57.82**	9.39
PERLBENCH	362	850	-	-	-	*OOT*	-	-	-	*OOT*	-
XALANCBMK	520	1341	1,787.64	**1,189.41**	1.5	638.22	7.52	275.96	208.8	**503.02**	1.27
PAREST	427	2,781	-	25,290.05	-	*OOT*	16.52	20.52	5,923.53	**5,988.25**	-
GCC	1,304	3,302	-	-	-	*OOT*	-	-	-	*OOT*	-
BLENDER	1,577	3,444	-	-	-	*OOT*	-	-	-	*OOT*	-

> The more DUG edges that each node is associated with, the longer time LISPA needs to terminate.

We also observed that the analysis time increases as the average points-to set size grows. For LISPA to solve fixed-point solution at each pointer level, more work needs to be done if the average points-to set size is large. For example, LISPA spends 18.19 seconds analyzing BAKE. However, LISPA takes only 10.26 seconds to analyze PSQL, which has a similar number of DUG nodes and edges. The main difference between these two benchmarks is that the average points-to set size of BAKE (22.29) is approximately 2.5× larger than the average points-to set size of PSQL (9.16). In addition, while more `alloca` instructions introduces more pointers into the analysis, it does not always result in a larger average points-to set. For example, IMAGICK contains 17,538 `alloca` instructions, however, the average points-to set size is 93% smaller than DU that contains only 2,756 `alloca` instructions. Therefore, the effort required to analyze a program is hard to predict from simply counting its textual representation (e.g., number of instructions). Instead, accurate insights can only be gleaned by examining data that are only obtained after running the pointer analyses (e.g., DUG nodes and edges and average points-to set size).

> The larger the average points-to set size is for a program, the more analysis time LISPA would have spent.

E. LISPA *vs Modern Flow-Sensitive Pointer Analyses*

Table III shows the running time of SFS and VSFS on our benchmark suite, as well as the speedup rate compared to LISPA. A rate larger than 1 indicates that LISPA has better runtime performance than the compared analysis. Across all programs, 27 programs can be analyzed by at least one analysis. In addition, LISPA can analyze 3 more programs (OMNETPP, XALANCBMK, and PAREST) then SFS and VSFS.

1) Comparison against SFS: Across the 24 programs that can be solved by both analyses, LISPA achieves an average speedup rate of 2.12× (min: 0.01×, max: 23.59×, geomean: 2.12×) against SFS. However, LISPA does not always out-

TABLE II: The number of LLVM IR instructions for each benchmark and the number of nodes and edges in the built DUG for LISPA. The column for `copy` shows the sum of `getelementptr` and `bitcast` instructions for each program. The last column shows the final average points-to set. We use "-" to indicate that the data cannot be collected due to timeouts.

	LLVM IR Instructions						Def-Use Graph		
Program	`alloca`	`load`	`store`	`copy`	`call`	`phi`	**# nodes**	**# edges**	avg. `pts()` size
LBM	139	2,771	419	889	71	1	48	40	0.86
MCF	348	2,578	924	1,171	180	30	320	532	1.82
DEEPSJENG	905	9,007	2,803	3,240	943	23	609	2,407	1.26
DPKG	2,912	14,402	5,081	5,920	5,813	132	8,684	9,768	8.04
NAB	2,723	18,802	6,060	6,792	3,016	112	5,460	6,089	1.17
DU	2,756	14,819	5,763	6,304	2,008	329	6,928	7,428	25.77
XZ	2,695	16,225	6,365	8,985	1,843	122	5,359	4,964	34.65
LEELA	8,626	12,993	9,853	8,641	9,665	88	14,258	10,829	10.5
NANO	3,289	22,317	7,936	8,389	5,337	429	11,331	62,846	64.36
PSQL	3,283	19,321	8,364	5,452	7,135	212	13,048	17,069	9.16
JANET	6,018	21,775	8,920	15,134	6,428	352	13,877	57,282	16.5
I3	4,352	27,111	8,868	15,313	8,427	887	13,812	18,609	14.36
BAKE	3,853	13,670	6,512	4,553	5,547	127	12,266	19,092	22.29
TMUX	789	15,897	6,509	19,384	11,910	5,893	728	1,091	12.4
ASTYLE	3,580	17,216	14,508	32,396	20,502	2,700	945	1,288	23.7
X264	6,723	52,967	15,177	38,656	5,036	573	16,731	18,412	126.3
NINJA	655	5,108	3,326	10,145	4,797	1,745	1,253	1,329	15.83
MRUBY	9,558	51,578	21,731	45,816	10,259	809	34,484	2,602,994	17.71
MUTT	8,386	57,523	18,239	25,122	14,637	1,916	35,623	106,098	5.33
NAMD	18,726	95,651	36,746	30,637	4,404	702	761	592	0.85
POVRAY	12,324	72,972	28,356	41,519	15,245	214	36,755	144,282	193.64
BASH	11,475	70,325	29,350	19,111	16,832	2,118	37,679	251,505	103.65
LYNX	9,131	80,267	25,791	27,046	22,383	3,460	54,636	150,639	253.63
IMAGICK	17,538	159,026	57,071	85,586	34,629	804	102,818	964,965	1.93
OMNETPP	45,585	94,257	60,440	62,441	57,479	1,396	77,505	68,654	52.36
PERLBENCH	21,741	220,260	58,486	132,286	27,615	8,200	-	-	-
XALANCBMK	122,545	234,904	158,646	143,937	131,551	1,718	238,559	203,321	98.36
PAREST	267,742	444,110	311,547	300,330	322,417	23,501	401,074	302,237	614.5
GCC	113,305	785,686	248,311	491,234	183,620	23,137	-	-	-
BLENDER	220,616	894,017	369,324	532,919	164,520	12,460	-	-	-

perform SFS, as SFS outperforms it on 5 programs. This is due to the different methods that LISPA and SFS construct a DUG. SFS is a staged analysis that runs an Andersen-style pointer analysis as a pre-analysis to bootstrap the construction of its DUG. While the generated DUG may contain def-use edges that lead to redundant points-to sets propagation, it is less computational intensive due to using a flow-insensitive analysis. On the other hand, LISPA builds its DUG with level-by-level analysis, leading to a more precise DUG than that built by LEVPA. While a more precise DUG benefits the analysis by removing unnecessary points-to sets propagation, it is also more computational intensive. For some programs, the benefit of performing DUG efficiently surpasses the benefit of avoiding redundant points-to sets propagation, while for others, the latter is more impactful.

2) Comparison against VSFS: VSFS improves upon SFS by assigning version identifiers to points-to sets so that different program locations have the same version identifier refer to a shared points-to set. Naturally, VSFS outperforms SFS on all 24 benchmarks that can be solved by both analyses. Across these programs, LISPA achieves an average speedup rate of $0.88\times$ (min: $0.01\times$, max: $15.39\times$, geomean: $0.88\times$) against VSFS. While VSFS outperforms LISPA on 13 out of 24 programs, LISPA outperforms VSFS on the remaining 11 programs. Similar to SFS, VSFS relies on an Andersen-style pointer analysis to build itsDUG, which is the main reason for this speedup discrepancy.

> LISPA is competitive with modern flow-sensitive pointer analysis algorithms. On average, LISPA is $2\times$ faster than SFS while achieving a comparable performance to VSFS.

V. RELATED WORK

There has been several proposed approaches for flow-sensitive pointer analysis in the past two decades. We can generally divide these approaches into two categories: (1) performing fixed-point computation of points-to set on a graph representation of a program and (2) reducing the computation to solving a Context-Free-Language Reachability problem on a points-to graph.

TABLE III: Runtime performance (in seconds) for SFS and VSFS and the speedup rate of LISPA against them. A speedup rate larger than 1 means LISPA is faster than the underlying analysis. Each entry is a mean of 5 separate runs. *OOT* indicates analysis timeouts, and "-" means the data cannot be collected due to timeouts.

Program	SFS		VSFS	
	Runtime (s)	Rate	Runtime (s)	Rate
LBM	0.06	1.2	0.03	0.6
MCF	0.17	2.13	0.1	1.25
DEEPSJENG	0.33	1.06	0.16	0.52
DPKG	18.85	14.39	6.37	4.86
NAB	1.66	1.58	0.96	0.91
DU	10.35	9.67	3.13	2.93
XZ	5.04	5.66	2.79	3.13
LEELA	11.99	3.11	5.81	1.51
NANO	36.1	0.32	13.65	0.12
PSQL	7.1	0.69	4.37	0.43
JANET	30.8	0.32	11.3	0.12
I3	21.1	9.34	8.28	3.66
BAKE	38.61	2.12	8.88	0.49
TMUX	25.4	15.78	10.61	6.59
ASTYLE	97.75	21.16	67.06	14.52
X264	82.08	23.59	53.57	15.39
NINJA	10.38	13.66	3.49	4.59
MRUBY	39.18	0.01	15.46	0.01
MUTT	466.48	2.29	129.21	0.64
NAMD	6.24	1.11	4.65	0.83
POVRAY	605.48	1.26	173.21	0.36
BASH	1,556.54	3.04	215.84	0.42
LYNX	5,296.33	11.2	856.75	1.81
IMAGICK	131.53	0.04	80.07	0.02
OMNETPP	*OOT*	-	*OOT*	-
PERLBENCH	*OOT*	-	*OOT*	-
XALANCBMK	*OOT*	-	*OOT*	-
PAREST	*OOT*	-	*OOT*	-
GCC	*OOT*	-	*OOT*	-
BLENDER	*OOT*	-	*OOT*	-

For the first approach, most prior work [9], [10], [17] either computes the fixed-point solution on the CFG of a program or another graph that encodes the semantics of the program data flow (e.g., DUG). This approach propagates points-to sets along the edges of the underlying graph until all points-to sets are fixed. However, most fixed-point computations on a CFG do not scale with the size of CFG. To improve scalability, some implementations reduce unnecessary propagations of points-to sets along the CFG edges. For example, Hardekopf et al. [10] designed a staged algorithm (SFS) that performs fixed-point computation on DUG of a program instead of its CFG. To build that DUG, SFS first runs an Andersen-style pointer analysis to collect the necessary information.

Although Hardekopf [10] claim that SFS can analyze programs of millions of LOC, a later implementation of SFS [18] shows that it does not scale to programs more than 100 KLOC. Barber et al. [9] have later designed an algorithm that improves SFS by adding a graph pre-labelling extension to reduce the unnecessary propagation of points-to sets for the same pointer. According to their evaluation, the improved algorithm achieves an average speed-up rate of $5.31\times$ against SFS. Unlike these algorithms, LISPA does not require any pre-analysis to build the DUG of a program.

Similar to LISPA, Yu et al. [5] introduces LEVPA, an approach that uses the idea of splitting pointers into different groups by their pointer level. Unlike LISPA, LEVPA performs both bottom-up and top-down analyses to summarize the effect of a function. LEVPA do not differentiate intermediate variables with other variables in the program like LISPA does, and it relies on computing memory SSA form and uses Andersen pointer analysis to analyze pointers flow-sensitively at a pointer level. Since LEVPA also computes points-to sets for intermediate variables, the performance is slowed down due to duplicate computation of points-to set.

For the second approach, an algorithm first builds a points-to graph that represents the relationship between memory objects. The algorithm then computes the points-to set of a given pointer by finding all nodes that are reachable from it. Li et al. [19] have proposed an algorithm that builds a graph to track how a memory object flows into a load or a store instruction. A pointer `p` then points to a memory object `m` if `p` is reachable from `m`. In contrast to this algorithm, LISPA does not track how memory objects flow throughout a program but instead propagates both pointer and alias information along a DUG.

Gharat et al. [20] have implemented a flow-sensitive, context-sensitive, and field-sensitive pointer analysis by constructing a points-to graph from an input program. The analysis starts with a precise but possibly non-scalable summary of a function and repeatedly applies optimizations including the elimination of data dependence, control flow minimization, and call inlining to the summary. During the optimization, the scalability of the summary is improved by reducing the indirection levels of pointers and simplifying the CFG while maintaining precision. The authors have shown that their approach analyzes C programs of 128 KLOC. Similarly, Lei et al. [21] have modelled the problem of pointer analysis as a Context-Free Language (CFL) reachability problem, performing the analysis on a points-to graph. Rather than only checking graph reachability, the CFL approach also checks the sentence implied by the path between two nodes in the graph of the underlying CFL. In other words, a pointer `p` points to a memory object `m` if there is a path between `p` and `m`, and the sentence generated from the path belongs to the defined CFL. Lei et al. [21] have shown that their algorithm analyzes programs up to 259 KLOC.

Schubert et al. [12] view a program as a collection of compilation units. Their method aims to efficiently summarize a module with respect to call graphs, points-to sets, and data-flow information. The summary of related modules is merged later when the whole program analysis is performed. This approach scales to programs of 100 KLOC.

Phasar [22] is an open source static analysis framework that enables user-defined data-flow analysis, and the points-to information is automatically computed by the framework. Since phasar is using Andersen pointer analysis, which is flow-insensitive, to compute the points-to information, we do not include them in our evaluation experiment.

VI. Conclusion

In this paper, we present a flow-sensitive pointer analysis algorithm that processes pointers according to their pointer level. Our algorithm (LisPA) avoids the computation of intermediate variables when building a def-use graph while performing the flow-sensitive pointer analysis. The key idea behind LisPA is the intermediate variables always alias to existing variables, and computing points-to set for these variables is redundant.

To evaluate LisPA, we compared its runtime performance and memory usage against LevPA. We used 16 benchmarks from SPEC CPU 2017 benchmark suite and 14 benchmarks from the VSFS benchmark suite [9] to evaluate both analyses. The results show that LisPA greatly improves LevPA regarding runtime, achieving an average speedup of $2.14\times$ (min: $1.01\times$, max: $13.65\times$, geomean: $2.14\times$), while improving memory usage of $2.38\times$ (min: $1.21\times$, max: $6.82\times$, geomean: $2.38\times$). In addition, we compare the runtime of LisPA with two state-of-the-art flow sensitive pointer analyses SFS and VSFS. LisPA achieves an average speedup of $2.12\times$ (min: $0.01\times$, max: $23.59\times$, geomean: $2.12\times$) against SFS and matches the performance with VSFS (min: $0.01\times$, max: $15.39\times$, geomean: $0.88\times$).

References

[1] T. Theodoridis, M. Rigger, and Z. Su, "Finding missed optimizations through the lens of dead code elimination," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 697–709. [Online]. Available: https://doi.org/10.1145/3503222.3507764

[2] B. Fischer, G. Garbi, S. La Torre, G. Parlato, and P. Schrammel, "Static data race detection via lazy sequentialization," in *Networked Systems - 12th International Conference, NETYS 2024, Rabat, Morocco, May 29-31, 2024, Proceedings*, ser. Lecture Notes in Computer Science, A. Castañeda, C. Enea, and N. Gupta, Eds., vol. 14783. Springer, 2024, pp. 124–141. [Online]. Available: https://doi.org/10.1007/978-3-031-67321-4_8

[3] Y. Yuan, Y. Lu, K. Zhu, H. Huang, L. Yu, and J. Zhao, "A static detection method for sql injection vulnerability based on program transformation," *Applied Sciences*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:264580534

[4] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88. [Online]. Available: https://doi.org/10.1109/CGO.2004.1281665

[5] 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 CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*, A. Moshovos, J. G. Steffan, K. M. Hazelwood, and D. R. Kaeli, Eds. ACM, 2010, pp. 218–229. [Online]. Available: https://doi.org/10.1145/1772954.1772985

[6] D. Novillo, "Design and implementation of tree ssa," in *In 2004 GCC Developers' Summit*, 2004, pp. 119–130.

[7] L. O. Andersen and P. Lee, "Program analysis and specialization for the c programming language," 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:20876553

[8] J. Bucek, K. Lange, and J. von Kistowski, "SPEC CPU2017: next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018*, K. Wolter, W. J. Knottenbelt, A. van Hoorn, and M. Nambiar, Eds. ACM, 2018, pp. 41–42. [Online]. Available: https://doi.org/10.1145/3185768.3185771

[9] M. Barbar, Y. Sui, and S. Chen, "Object versioning for flow-sensitive pointer analysis," in *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, J. W. Lee, M. L. Soffa, and A. Zaks, Eds. IEEE, 2021, pp. 222–235. [Online]. Available: https://doi.org/10.1109/CGO51591.2021.9370334

[10] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. IEEE Computer Society, 2011, pp. 289–298. [Online]. Available: https://doi.org/10.1109/CGO.2011.5764696

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

[12] P. D. Schubert, B. Hermann, and E. Bodden, "Lossless, persisted summarization of static callgraph, points-to and data-flow analysis," in *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, ser. LIPIcs, A. Møller and M. Sridharan, Eds., vol. 194. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 2:1–2:31. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2021.2

[13] B. Steensgaard, "Points-to analysis in almost linear time," in *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, H. Boehm and G. L. S. Jr., Eds. ACM Press, 1996, pp. 32–41. [Online]. Available: https://doi.org/10.1145/237721.237727

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

[15] Clang, 2023. [Online]. Available: https://github.com/llvm/llvm-project/releases/tag/llvmorg-17.0.1

[16] K. Pearson and F. Galton, "Vii. note on regression and inheritance in the case of two parents," *Proceedings of the Royal Society of London*, vol. 58, no. 347-352, pp. 240–242, 1895. [Online]. Available: https://royalsocietypublishing.org/doi/abs/10.1098/rspl.1895.0041

[17] J. Zhao, M. G. Burke, and V. Sarkar, "Parallel sparse flow-sensitive points-to analysis," in *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, C. Dubach and J. Xue, Eds. ACM, 2018, pp. 59–70. [Online]. Available: https://doi.org/10.1145/3178372.3179517

[18] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, A. Zaks and M. V. Hermenegildo, Eds. ACM, 2016, pp. 265–266. [Online]. Available: https://doi.org/10.1145/2892208.2892235

[19] L. Li, C. G. Cifuentes, and N. Keynes, "Boosting the performance of flow-sensitive points-to analysis using value flow," in *ESEC/FSE '11*, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:14978113

[20] P. M. Gharat, U. P. Khedker, and A. Mycroft, "Generalized points-to graphs: A precise and scalable abstraction for points-to analysis," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 2, pp. 8:1–8:78, 2020. [Online]. Available: https://doi.org/10.1145/3382092

[21] Y. Lei, C. Bossut, Y. Sui, and Q. Zhang, "Context-free language reachability via skewed tabulation," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, jun 2024. [Online]. Available: https://doi.org/10.1145/3656451

[22] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An inter-procedural static analysis framework for C/C++," in *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*, ser. Lecture Notes in Computer Science, T. Vojnar and L. Zhang, Eds., vol. 11428. Springer, 2019, pp. 393–410. [Online]. Available: https://doi.org/10.1007/978-3-030-17465-1_22