Parallel Sparse Flow-Sensitive Points-to Analysis

Jisheng Zhao
Rice University
Houston, Texas, USA
jisheng.zhao@rice.edu

Michael G. Burke
Rice University
Houston, Texas, USA
mgb2@rice.edu

Vivek Sarkar
Georgia Institute of Technology
Atlanta, Georgia, USA
vsarkar@gatech.edu

Abstract

This paper aims to contribute to further advances in pointer (or points-to) analysis algorithms along the combined dimensions of precision, scalability, and performance. For precision, we aim to support interprocedural flow-sensitive analysis. For scalability, we aim to show that our approach scales to large applications with reasonable memory requirements. For performance, we aim to design a points-to analysis algorithm that is amenable to parallel execution. The algorithm introduced in this paper achieves all these goals. As an example, our experimental results show that our algorithm can analyze the 2.2MLOC Tizen OS framework with < 16GB of memory while delivering an average analysis rate of > 10KLOC/second.

Our points-to analysis algorithm, **PSEGPT**, is based on the **Pointer Sparse Evaluation Graph (PSEG)** form, a new analysis representation that combines both points-to and heap def-use information. **PSEGPT** is a scalable interprocedural flow-sensitive context-insensitive points-to analysis that is amenable to efficient task-parallel implementations, even though points-to analysis is usually viewed as a challenge problem for parallelization. Our experimental results with 6 real-world applications on a 12-core machine show an average parallel speedup of 4.45× and maximum speedup of 7.35×. The evaluation also includes precision results by demonstrating that our algorithm identifies significantly more inlinable indirect calls (IICs) than SUPT [15] and SS [9], two state of the art SSA-based points-to analyses implemented in LLVM.

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

Keywords Static Analysis, Pointer Analysis, Parallelism

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CC'18, February 24–25, 2018, Vienna, Austria
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5644-2/18/02...$15.00
https://doi.org/10.1145/3178372.3179517

ACM Reference Format:
Jisheng Zhao, Michael G. Burke, and Vivek Sarkar. 2018. Parallel Sparse Flow-Sensitive Points-to Analysis. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3178372.3179517

1 Introduction

Points-to analysis is a fundamental requirement for many program analyses, optimizations, and debugging/verification tools. It is used to determine if two pointer expressions may refer to the same memory location. Static analysis and more specifically, alias analysis, is in general undecidable [21]. Hence, a large number of approximation algorithms have been published that balance the precision and the efficiency of pointer analysis. These algorithms explore various dimensions to achieve this balance. However, finding an effective balance across precision, scalability, and performance in points-to analysis remains a major challenge. Many flow-sensitive algorithms achieve a desirable level of precision but are impractical for use on large software. Likewise, many flow-insensitive algorithms scale to large software, but do so with major limitations in precision. Further, in light of the recent multicore hardware trends, more attention needs to be paid to the use of parallelism for improved performance. Our focus in this paper is primarily on flow-sensitive points-to analysis, which has been shown to be important for a growing list of program analyses [7], including those that check for security vulnerabilities [5, 8], and that analyze multi-threaded codes. A further goal of this paper is to leverage sparseness and parallelism to achieve scalability, as discussed below.

The traditional flow-sensitive approach [4, 14, 27] uses a dense iterative dataflow analysis, which does not scale to large programs. A frequently used method for optimizing a flow-sensitive dataflow analysis is to perform a sparse analysis, such as in the flow-sensitive points-to analysis of [2, 12], which uses the Sparse Evaluation Graph (SEG) [3] to directly connect variable definitions (defs) with their uses, allowing data flow facts to be propagated only to those program locations that need the values. In general, sparse points-to analysis can be challenging because of an inherent circularity — pointer information is required to compute the def-use information needed to enable a sparse points-to analysis.

Hardekopf and Lin [9] present a semi-sparse (*SS*) flow-sensitive points-to analysis which exploits partial SSA form to perform a sparse analysis on "top-level" (scalar) variables

that do not have their address taken but may point to memory locations, while using dense dataflow analysis on address-taken memory locations. SS advanced the state of the art by introducing a scalable context-sensitive, flow-sensitive interprocedural points-to analysis that uses a combination of semi-sparse analysis and binary decision diagrams (BDDs). It also avoids the circularity faced by sparse algorithms, since it only performs sparse analysis on top-level variables, which can be converted to SSA def-use edges without using pointer information.

Lhotak and Chung [15] perform a SSA-based flow-sensitive points-to analysis that leverages strong updates (*SUPT*). For a more compact representation that contributes to increased scalability, they maintain a single points-to-graph for the whole program instead of one per program point. This results in a loss of flow-sensitive precision. In cases where this representation identifies a singleton points-to set for the variable of a store operation, they perform a strong update. For input programs with frequent occurrences of strong updates, their analysis provides an effective balance between precision and speed. However, the partial flow-insensitive nature of their algorithm makes extensive use of shared points-to sets, which in turn impedes parallelization.

Given the rapid development of multi-core systems in the last decade, leveraging parallel computation for points-to analysis can improve the scalability of compilation, verification, and other software engineering tasks. As a result, there have already been a few efforts to develop parallel points-to analyses. Mendez-Lojo et al. [18] introduced a parallel points-to analysis algorithm based on graph rewriting. Nagaraj and Govindarajan [19] extended the graph rewriting approach by adding rewrite rules to support flow sensitivity.

In this paper, we introduce a flow-sensitive points-to analysis algorithm based on the new **Pointer SEG** (**PSEG**) form, which extends the SEG and treats memory store and load operations as defs and uses of an aggregate "heap" that includes all address-taken variables. **PSEG** chains heap defs in the same manner as SSA form's def variables, and connects each heap use with its immediately dominating heap def. The sparse points-to analysis, **PSEGPT**, is performed with respect to both scalar and address-taken variables. It maintains a global points-to graph for scalar variables in the same manner as SUPT and SS. For address-taken variables, **PSEGPT** only maintains points-to information for the address-taken variables that are in the alias set associated with a given program point. This results in significant memory savings in practice, when compared to maintaining a whole program points-to graph at every point (as with SS), although it does not reduce the worst case space complexity in the absence of strong updates. In the presence of strong updates, the worst case complexity of our analysis is reduced, as discussed in Section 3.6.

The **PSEGPT** algorithm addresses the circularity of sparse points-to analysis and def-use analysis by integrating them into a single analysis. It builds precise def-use chains on demand when performing the flow-sensitive points-to analysis. The algorithm as presented is context insensitive, but it could be extended to add context sensitivity using one of several available techniques.

More specifically, the contributions of this paper are as follows:

- A novel **Pointer SEG** (**PSEG**) program representation, designed for scalable flow-sensitive points-to analysis.
- A novel interprocedural flow-sensitive, field-sensitive and context-insensitive points-to analysis (**PSEGPT**) based on **PSEG** form with 1) an integrated sparse points-to and def-use analysis for all memory locations; 2) support for parallelization; and 3) support for weakly typed languages.
- Sequential and parallel versions of the **PSEGPT** algorithm.
- An evaluation of an implementation of the sequential and the parallel versions of **PSEGPT** using LLVM and a lightweight task parallelism library. The parallel version achieves a parallel speedup of up to 7.35× on 12 processor cores, relative to the sequential version. Further, the results show that our algorithm can analyze a 2.2MLOC application (Tizen) while using < 16GB of memory and delivering an average analysis rate of > 10KLOC/second.

The rest of this paper is organized as follows. Section 2 provides background on LLVM IR, flow-sensitive points-to analysis, and Array SSA form. Section 3 introduces the **PSEG** form and presents the **PSEG**-based points-to analysis (**PSEGPT**). Section 4 describes a parallelization of **PSEGPT**, that we call **PPSEGPT**, based on the use of fine-grain task parallelism in processing worklist elements. Section 5 presents an evaluation of an LLVM-based implementation of the algorithm, both from the viewpoint of its scalability and from a study of its effectiveness when applied to an analysis of inlinable indirect calls. Section 6 compares related work to our approach. Section 7 concludes the paper and discusses future work.

2 Background

In this section, we briefly summarize terminology and concepts from past work that will be used as building blocks in the rest of the paper.

2.1 Memory Model and LLVM IR

For simplicity of presentation, we follow the LLVM [17] convention of separating variables into two disjoint sets of top-level and address-taken variables. The address of a top-level variable is not exposed.

A variable whose address may be taken can be a stack variable, a global variable, or a dynamically allocated memory object, and is referred to as as an "address-taken variable"

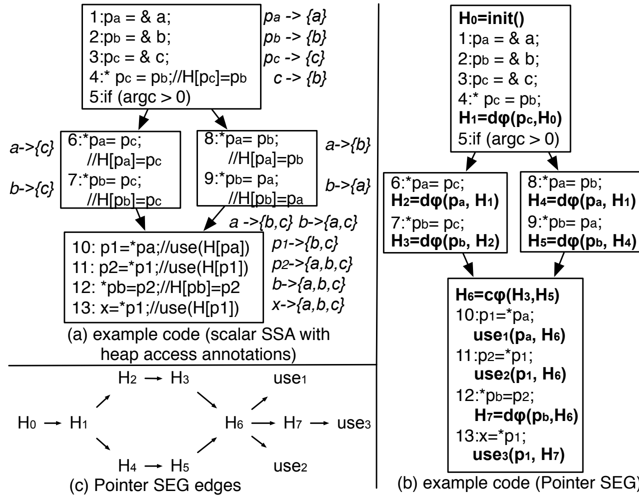


Figure 1. Code example for C language and **PSEG** form.

in this paper. Thus, an address-taken variable represents a memory location that can be the target of a load or a store instruction in a program. We also refer to the collection of all address-taken variables as a "heap", and each renamed instance of the heap is called a "heap variable" in our Pointer SEG (PSEG) form.

For simplicity, Figure 1(a) shows an example control flow graph with C statements rather than LLVM IR instructions. Variables $\mathbf{p}_a$, $\mathbf{p}_b$, $\mathbf{p}_c$, $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{x}$ are top-level variables. Variables **a**, **b** and **c** are address-taken variables (e.g., they may be global variables).

The LLVM IR is based on SSA form, which provides def-use chains for top-level variables. For address-taken variables, there is no explicit def-use information, since the read/write operations of address-taken variables are based on load/store instructions that take top-level pointer variables as arguments. This paper will assume the use of the LLVM IR, which implies that any named variable is a top-level variable and not an address-taken variable. We also use the terms *scalar SSA* for the LLVM IR and *scalar variable* for a top-level variable.

2.2 Semi-Sparse Flow-sensitive Pointer Analysis

Flow-sensitive pointer analysis is based on a program's control flow and can compute distinct points-to solutions at different program points. A standard way of representing the output of flow-sensitive pointer analysis is via separate points-to graphs at each such program point, an approach that is expensive with respect to memory.

SS is a semi-sparse flow-sensitive pointer analysis which exploits a partial SSA form that provides def-use information for top-level variables but not for address-taken variables. SS performs a sparse analysis on scalar variables, using a single global points-to graph for all top-level variables. An iterative dataflow analysis is required to propagate information for address-taken variables along paths in a SEG. The SEG used by SS is an optimized version of the control-flow graph that elides nodes that neither define nor use pointer information.

Since address-taken variables can be modified at any program point without renaming, some flow-sensitive pointer analyses also introduce a *labeling* flag [9] l_i at each program point i. In the example shown in Figure 1(a), a has distinct points-to address-taken variables c and b at Line 6 and 8, where the points-to information is represented as $\mathbf{a}[l_6] \rightarrow \mathbf{c}$ (for line 6) and $\mathbf{a}[l_8] \rightarrow \mathbf{b}$ (for line 8). Since the top-level variables are in SSA form, each top-level variable necessarily has the same pointer information over the entire program. The analysis can then avoid storing and propagating pointer information for top-level variables among CFG nodes. Local points-to graphs, i.e., separate IN and OUT graphs for each CFG node, are still needed for LOAD and STORE statements, and these graphs only hold pointer information for address-taken variables.

2.3 Array SSA and Heap SSA Forms

Array SSA form [13] extends scalar SSA form by capturing precise element-level data flow information for arrays. A $d\phi$ operator is inserted immediately after each preserving definition of an array variable, i.e., a definition that does not kill all the values in the array. The $d\phi$ merges the values of the element(s) modified in the definition with the values available immediately prior to the definition. As with scalar SSA, Array SSA form also includes a merge ϕ function that merges values computed along distinct control paths. Further, a requirement on the placement of both $d\phi$ and ϕ operations is that they enable each use to refer to a single def that dominates the use. Due to its more precise handling of array variables compared to scalar SSA form, Array SSA form was shown to contribute to enhanced parallelization [13] and improved precision for program analyses such as constant propagation [23].

Subsequently, a Heap SSA form was introduced as an extension of Array SSA form to enable analysis of object references in strongly typed languages like Java [6] by modeling each field as a distinct logical "heap array". It adds use-ϕ ($u\phi$) operators to create a new name whenever a statement reads a heap array element. The $u\phi$ operator links together load instructions for the same heap array in control flow order. Like the other ϕ operators, the $u\phi$ operator creates a new SSA variable name, with which a sparse data flow analysis can associate a lattice value. However, neither Array SSA form nor Heap SSA form can be applied to pointer analysis of weakly typed languages like C.

In this paper, we introduce a **Pointer SEG** representation that overcomes the limitations of Array SSA form and Heap SSA form for pointer analysis of C programs. A notable feature of the **Pointer SEG** representation is that it enables a

Backtrace function, which implements an on-demand construction of def-use edges, thereby integrating def-use analysis with points-to analysis in a scalable and efficient manner. As we will see, the **Pointer SEG** algorithm constructs def-use edges by leveraging pointer alias analysis information, unlike Array SSA form and Heap SSA form, for which the def-use edges are fixed from the start and are conservative with respect to pointer aliasing.

3 PSEG-based Pointer Analysis

This section introduces a sequential **PSEG**-based points-to analysis algorithm (**PSEGPT**), which takes scalar SSA and the **PSEG** form as input and computes interprocedural flow-sensitive points-to information and def-use connections between heap variables in a single integrated analysis.

Section 3.1 introduces **Pointer SEG** form, which is the input form for our pointer analysis algorithm. Section 3.2 introduces heap uses, **Value Flow Edges** (which serve as def-use edges for heap variables), and SSA merge nodes for heap definitions. Section 3.3 formally presents the constraints that must be satisfied by the **PSEGPT** analysis algorithm. Section 3.4 describes the intraprocedural component of **PSEGPT**, where a single function is input to it in **PSEG** form. Section 3.5 describes the full interprocedural algorithm. Section 3.6 discusses the monotonicity and fixpoint convergence properties of the **PSEGPT** algorithm, and its worst-case time and space complexity.

3.1 Pointer SEG Form

Pointer SEG (**PSEG**) form introduces def-use information for heap variables. Due to pointer aliasing in C/C++, def-use connections have to be built conservatively. Since **PSEG** needs to assume that all operations on heap variables may potentially be aliased, memory store instructions are modeled in **PSEG** as definitions of a single "heap" that contains all address-taken variables chained with renaming (as with SSA) to obtain distinct heap variables named $\mathbf{H}_0$, $\mathbf{H}_1$, etc. To build the connections between store instructions and renamed heap variables, **PSEG** employs a $d\phi$ function that returns a heap H_i to represent the renamed heap variable at each store instruction. A $d\phi$ takes two arguments: the store address and the previous heap variable. Figure 1(b) gives the **PSEG** version of the example program shown in (a). Similar to SSA form, **PSEG** employs $c\phi$ nodes to handle the merge of control flow edges (e.g., $\mathbf{H}_6$).

For read operations on heap variables, **PSEG** employs a **use** operator that has two parameters: the load address and the reaching def heap variable. **Use** operators are also renamed to represent read operations at different program points. **PSEG** chains heap defs in the same manner as SSA form's def variables, and connects each heap use (whether in a $d\phi$, $c\phi$, or **use** operator) with its immediately dominating heap def.

Based on the definition given above, a **PSEG** edge represents a directed connection between a definition (single assignment) of a heap variable and a reference to that heap variable. Figure 1(c) shows the **PSEG** edges for the example program in 1(a). Further, we refer to the immediate dominator and immediate postdominator of **PSEG** node x as $ImmDom(x)$ and $ImmPDom(x)$ respectively.

The **PSEGPT** analysis is run on the dataflow graph that is composed of scalar SSA and the **PSEG** (details in Section 3.3).

3.2 Value Flow Edges for Address-taken Variables

As discussed in Section 3.1, **PSEG** edges connect each def and use of a heap variable with its input heap variable operand(s) in the corresponding $d\phi$ and $c\phi$ operations. (There may be more than one heap variable operand in the case of a $c\phi$ operator.) Due to renaming, the set of defs that reach a use U along a path in the **PSEG** form a conservative approximation (i.e. a superset) of the defs D for which there is a def-use edge (D, U). The **PSEGPT** flow-sensitive points-to algorithm uses **PSEG** edges as input to efficiently and precisely compute def-use edges for address-taken variables, which we refer to as **Value Flow Edges**. The analysis associates each renamed heap variable H_i with a points-to set (PTS) and an alias set (AS) of heap memory locations. For the example shown in Figure 1(b) and (c), $\mathsf{AS}(\mathsf{H}_1)$ is {c} and $\mathsf{AS}(\mathsf{H}_2)$ is {a}. As discussed later, these alias sets are computed during the points-to analysis. The computation of the points-to set of H_i is represented as the union of the points-to sets for each element in H_i's alias set.

A $c\phi$ node in the **PSEG**, in a similar fashion to a ϕ node in SSA, is used to merge distinct heap definitions (i.e., $d\phi$ nodes) from multiple incoming control flow edges. Since each $d\phi$ represents the potential updates of a set of memory addresses, a $c\phi$ node is associated with all such memory addresses and their points-to sets from all $d\phi$s on the incoming control flow edges. The points-to information in $c\phi$ is represented as a points-to graph (PTG($c\phi$)), where each element of its key set is a member of the alias set for a reaching $d\phi$ node.

3.3 PSEGPT Algorithm

We use a sparse dataflow analysis approach, i.e., propagating points-to information via def-use connections among scalar and heap variables. The sparse points-to analysis is challenging because points-to information is required to compute the def-use information based on indirect writes to variables through pointers. The **PSEGPT** algorithm addresses this problem by simultaneously computing flow-sensitive points-to information and value flow edges in a single integrated analysis.

The following equations define the constraints for points-to analysis of both scalar and heap variables with respect to scalar SSA and **PSEG** nodes. The algorithm must process scalar SSA ϕ functions for merging scalar values, as well as $c\phi$ functions for merging heap definitions, computed along

distinct control paths. A ϕ node and a $c\phi$ node at the same program point have identical incoming and outgoing control flow edges.

Table 1. Constraints for Points-to Analysis

Eq.1	ADDROF: p = &a;	$\text{PTS}(p) = \{a\}$
Eq.2	COPY: $p = q$;	$\text{PTS}(p) = \text{PTS}(q)$
Eq.3	LOAD: $p = *q$; $use_i(q, H_j)$	$\forall\, a \in \text{AS}(use_i), \text{PTS}(a) \subseteq \text{PTS}(p)$ here $\text{AS}(use_i)$ is equivalent to $\text{PTS}(q)$
Eq.4	STORE: $*p = q$; $H_i = d\phi(p, H_j)$	$\forall\, a \in \text{AS}(H_i), \text{PTS}(q) \subseteq \text{PTS}(a)$ here $\text{AS}(H_i)$ is equivalent to $\text{PTS}(p)$
Eq.5	SCALAR MERGE ϕ: $p = \phi(p_1, p_2, ...)$	$\forall\, p_i \in \text{ARGS}(\phi),\ \text{PTS}(p_i) \subseteq \text{PTS}(p)$
Eq.6	HEAP MERGE $c\phi$: $H_i = c\phi(H_1, H_2, \ldots)$	$\forall\, H_r \in \text{REACH}(H_i)$ $\{\text{AS}(H_r), \text{PTS}(H_r)\} \subseteq \text{PTG}(H_i)$

In **Eq.5**, $ARGS(\phi)$ is the set of scalar definitions reaching ϕ along all incoming edges. In **Eq.6**, $REACH(H_i)$ is the set of heap definitions reaching $c\phi$ along all incoming edges. For $c\phi$, the points-to analysis is dependent on reaching heap definition information. The reaching heap definition analysis is a component of the flow-sensitive points-to analysis, and is based on the edges that connect each heap variable use and def to its heap operand(s) in **Pointer SEG** form.

Strong Updates: For flow-sensitive precision, the points-to analysis needs to identify strong updates. Our algorithm identities strong updates in two cases.

1. $d\phi$ D strongly updates address-taken variable a: i.e., D's alias set is a, which is a **singleton** set; [1]
2. $c\phi$ C strongly updates address-taken variable a: for each of C's incoming edges e_i, there is a $d\phi$ D_i or $c\phi$ C_i that strongly updates a.

Value Flow Edges: For a value flow edge to be added between a heap variable's definition $H_i = d\phi(v_a)$ and use $use_k(v_b)$, it is necessary but not sufficient that $H_i[v_a]$ reach a node in which $use_k[v_b]$ is used. A value flow edge is added from H_i to $use_k(v_b)$ only if $H_i[v_a]$ may-equal $use_k[v_b]$ (i.e., $\text{AS}(H_i[v_a]) \cap \text{AS}(use_k[v_b]) \neq \emptyset$). As with a $d\phi$, a value flow edge is added from a $c\phi$ C to $use_k(v_b)$ only if $\text{KEYSET}(\text{PTG}(C)) \cap \text{AS}(use_k[v_b]) \neq \emptyset$. Here $\text{KEYSET}(G)$ represents the key set for G. There are two special cases in which a $c\phi$ C acts as a use: when C merges points-to information from $d\phi$s or C belongs to the dominance frontier of a $c\phi$.

The four equations listed below cover the four cases for adding value flow edges. We use D, U, and C to represent a $d\phi$, use, and $c\phi$ respectively. Here $\text{VALUE}(G, key)$ is the function that gets the corresponding values for the key in points-to graph G.

[1] The definition of **singleton** is implementation dependent and explained further in Section 5

Table 2. Rules for Adding Value Flow Edges

Eq.7	an edge from a dϕ D to a use U,	$\text{PTS}(D) \subseteq \text{PTS}(U)$
Eq.8	an edge from a $d\phi$ D to a cϕ C	$\{\text{AS}(D), \text{PTS}(D)\} \subseteq \text{PTG}(C)$
Eq.9	an edge from a $c\phi$ C to an use U	$\forall\, key \in \text{PTG}(C)$. if $key \in \text{AS}(U)$ then $\text{VALUE}(\text{PTG}(C), key) \subseteq \text{PTS}(U)$
Eq.10	an edge from a $c\phi$ C_i to a use $c\phi$ C_j	$\forall\, key \in \text{PTG}(C_i)$, $\{key, \text{VALUE}(\text{PTG}(C_i), \text{key})\} \subseteq \text{PTG}(C_j)$

3.4 Intraprocedural PSEGPT

The full **PSEGPT** algorithm is interprocedural, but for simplicity of exposition we first consider its intraprocedural component shown in Algorithm PSEGPT on pages 6 and 7, where a single function is input to **PSEGPT** in **PSEG** form. It is a worklist-based algorithm. The worklist WL, the points-to sets and alias sets of scalar and heap variables, and the set of heap variable value flow edges are all initialized to empty.

The **InitWorkList** function of **PSEGPT** (Line 2) adds all known allocation sites to the worklist **WL**, e.g., **alloca** instructions, global variables, and known functions that can be modeled as allocation sites. In addition, all defs are initialized to **undef** as explained below. Here we use v_i to denote top-level variables, and a_i for heap variables. Global variables are included as allocation sites due to their representation in LLVM as heap variables pointed to by constant pointers. We model allocation sites as ADDROF instructions; i.e., instructions of the form $v_i = \&a_j$. **InitWorkList** adds each such scalar variable v_i to WL, and adds to its alias set and points-to set in accord with **Eq.1**.

Defs that currently in an analysis have not been set to point to any address-taken variable are set to **undef**, with the following interpretation:

1. A $d\phi$ D is **undef** $\iff$ D's alias set is $\emptyset$.
2. A $c\phi$ C is **undef** $\iff$ $\exists$ $d\phi$ D whose dominance frontier is C and D is **undef**. In **PSEG** form, C consists of a single $c\phi$ node or is empty.

Analysis Operations: After the execution of **InitWorkList**, **PSEGPT** iteratively selects an item from WL (**PSEGPT** Line 3) and performs an operation based on the kind of item that is selected. Each of the four kinds of items corresponds to a separate operation. These four analysis operations propagate points-to information and build value flow edges. The processing of worklist items continues until the worklist is empty, indicating that the algorithm has converged.

We now describe the four analysis operations of **PSEGPT**.

1. sv_prop: If the current item popped from the worklist is a definition of a scalar variable v (**PSEGPT** Lines 5,6), then function **PropToUses** is invoked with input operand v. **PropToUses** propagates v's points-to set via scalar SSA def-use chains. The set $Use(v)$ contains the instructions that use v. There is a scalar def-use edge from v to each instruction vi $\in Use(v)$ (**PropToUses** Line 16).

Algorithm: Intraprocedural PSEGPT (Part 1 of 2)

```
1  function PSEGPT ()
     Input : F : input function in PSEG form
     Output: Points-to sets and value flow edges
2  InitWorkList (F);
3  while WL ≠ ∅ do
4      v := PopFront (WL);
5      if v is scalar then
6          WL ∪ =PropToUses (v, GlobalWL);
7      else if v is dphi or cphi then
8          if IsModified (AS(v)) then
9              WL ∪ =CollectUses (v, AS(v));
10         else
11             WL ∪ =PropDPhi (v); //Or invoke PropCPhi(v, GlobalWL)
12     else if v is heap use then
13         WL ∪ = BackTrace (v);
14 function PropToUses ()
     Input : v : variable or instruction, GlobalWL : global worklist
     Output: wl : worklist
15 wl := ∅;
16 foreach vi in Use (v) do
17     if IsCopy (vi) then
18         PTS (vi) ∪ = PTS (v); //vi = v
19         wl ∪ = IsModified (PTS (vi)) ? PropToUses (vi) : ∅;
20     else if IsScalarPhi (vi) then
21         PTS (vi) ∪ = PTS (v); //vi = ϕ(v, ...)
22         wl ∪ = IsModified (PTS (vi)) ? {vi} : ∅;
23     else if IsDPhiValueOp (vi, v) then
24         hv := AddrTakenVar (vi); PTS (hv) ∪ = PTS (v); //vi: *p=v
25         wl ∪ = IsModified (PTS (hv)) ? PropDPhi (hv) : ∅;
26     else if IsPointerOp (vi, v) then
27         hv := AddrTakenVar (vi); AS (hv) ∪ = PTS (v); //vi: *v=... or
             vi: ...=*v
28         wl ∪ = IsModified (PTS (hv)) ? {hv} : ∅;
29 return wl;
30
31 function CollectUses ()
     Input : D: dϕ or cϕ, as : the input alias set
     Output: wl : worklist
32 wl := ∅; as := AS (D); C := DomFrontier (D);
33 if C then
34     n := ImmPDom (D);
35     while n ≠ C do
36         if AS (n) = undef then
37             wl ∪ = D; return wl;
38         if IsSUSet (n, as) then
39             as := as \ AS(n);
40         if as = ∅ then
41             return wl;
42         n := ImmPDom (n);
43     changed := UpdatePTG (as, PTS (D), PTG (C));
44     if changed then
45         CollectUses (C, as);
46 foreach U in DomBy (D) do
47     if AS (U) ≠ ∅ then
48         wl ∪ = U;
49 return wl;
```

Algorithm: Intraprocedural PSEGPT (Part 2 of 2)

```
1  function BackTrace ()
     Input : U: heap use
     Output: wl : worklist
2  as := AS (U); wl := ∅; n := U; mod := false;
3  while n ≠ INIT_NODE && AS(n) ≠ undef && as ≠ ∅ do
4      D := ImmDom (n); Δ := as ⋂ AS (D);
5      if Δ ≠ ∅ then
6          PTS (U) ∪ = PTS (D); AddEdge (D, U);
7          mod |= IsModified (PTS (U));
8          if IsSUSet (D, Δ) then
9              as := as \ Δ;
10     n := D;
11 if mod then
12     v := LHS (U); PTS (v) ∪ = PTS (U); wl ∪ = v;
13 return wl;
14
15 function PropDPhi ()
     Input : D: dϕ
     Output: wl : worklist
16 wl := ∅;
17 foreach edge e from D do
       //U is a heap use or cϕ
18     U := dest (e);
19     if U is heap use then
20         PTS (U) ∪ = PTS (D);
21         wl ∪ = IsModified (PTS (U))? U : ∅;
22     else if U is cϕ then
23         changed := UpdatePTG (AS (D), PTS (D), PTG (U));
24         if changed then
25             PropCPhi (U, wl);
26 return wl;
27
28 function PropCPhi ()
     Input : C: cϕ, wl : worklist
     Output: wl : worklist
29 mods := GetModifiedKey (PTG (C));
30 foreach edge e from C do
31     U := dest (e); Δ := mods ⋂ AS (U);
32     if U is heap use then
33         foreach k in Δ do
34             PTS (U) := GetPTS (PTG (C), k);
35             wl ∪ = IsModified (PTS (U))? U : ∅;
36     else if U is cϕ then
37         hasChanged := false;
38         foreach k in Δ do
39             pts := GetPTS (PTG (C), k);
40             hasChanged |= UpdatePTG (k, pts, PTG (U));
41         if hasChanged then
42             PropCPhi (U, wl);
```

2. hvd_conn: If the current item is a heap variable $d\phi$ or $c\phi$ operator, D, and its alias set AS has changed, function **CollectUses** is invoked at **PSEGPT** Line 9 to propagate points-to information to $c\phi$ nodes that may be impacted by D's alias set. During this search, **CollectUses** generates a set of local worklist items that it returns and adds to WL. The items added to the worklist are heap uses with non-empty alias sets that are immediately dominated by D and heap defs and $c\phi$s that need to be processed later if D is **undef**.

3. hv_prop: If the current item is a heap variable $d\phi$ or $c\phi$ operator, and its alias set has not changed, function **PropDPhi** or **PropCPhi** is invoked to propagate its points-to sets (PTS) to heap uses and to the points-to sets of $c\phi$s via value flow edges (**PSEGPT** Line 11). For a def-use edge from a heap def D to a heap use U, there are four cases to handle: a) D is a $d\phi$ and U is a use of a heap variable: see **PropDPhi** Lines 19~21 and **Eq.7**. b) D is a $d\phi$ and U is a $c\phi$: see **PropDPhi** Lines

22~25 and **Eq.8**. Line 23 is in accord with **Eq.6**. c) D is a $c\phi$ and U is a use of a heap variable: see **PropCPhi** Lines 32~35 and **Eq.9**. d) D is a $c\phi$ and U is a $c\phi$: see **PropCPhi** Lines 36~42 and **Eq.10**. Line 39 is in accord with **Eq.6**. Also, the altered points-to sets of $c\phi$ uses are propagated to succeeding $c\phi$ uses (Line 25 of **PropDPhi** and Line 42 of **PropCPhi**).

4. `hvu_conn`: If the current item is a heap use `U` and its alias set has changed, the **Backtrace** function is invoked (**PSEGPT** Line 13) to build value flow edges for `U`. **Backtrace** builds value flow edges(D, U) for `U` by traversing immediately dominating heap def edges in reverse, starting from `U` (**BackTrace** Line 4).

When a value flow edge has been added during the **Backtrace** traversal, **PropDPhi** is invoked to handle the four value flow edge cases described above.

3.5 Interprocedural PSEGPT Algorithm

Algorithm: Interprocedural PSEGPT

```
1  function MainProc ()
     Input  : F : root function
     Output:
2  GlobalWL := ∅; GlobalWL ∪ = {F};
3  while GlobalWL ≠ ∅ do
4  |   func := PopFront (GlobalWL);
5  |   PSEGPT (func, GlobalWL);
6  function PSEGPT ()
     Input  : F : input function, GlobalWL : global worklist
     Output: boolean flag for side-effect
7  InitWorkList (F)
8  while WL ≠ ∅ do
9  |   v := PopFront (WL);
10 |   if IsCallSite (v) then
11 |   |   GlobalWL ∪ = GetCallTarget (v);
12 |   else if v is scalar then
13 |   |   WL ∪ =PropToUses (v, GlobalWL);
14 |   else if v is dphi or cphi then
15 |   |   if IsModified (AS(v)) then
16 |   |   |   WL ∪ =CollectUses (v, AS(v));
17 |   |   else
18 |   |   |   WL ∪ =PropDPhi (v);
19 |   else if v is heap use then
20 |   |   WL ∪ = BackTrace (v);
21 callers := UpdateCallers (F);
22 GlobalWL ∪ = callers;
23 return callers ≠ ∅;
```

To extend **PSEGPT** to an interprocedural analysis, we adapt **PSEG** to add four kinds of heap nodes:

1. an *entry* $d\phi$ node associated with the entry of each function `F`;
2. an *exit* heap use node is associated with the set of returns from each function `F`;
3. a *call* heap use node associated with the program point immediately preceding a call site;
4. a *call return* $d\phi$ node associated with each program point immediately following a call site.

It is straightforward to convert the **PSEGPT** for a function so that it has only one return node: add a control flow edge from the other return nodes to a single return node, which we now call an *exit* node. Merging call returns into a single return node does not result in a loss of precision, as the interprocedural analysis is context insensitive. Callahan's program summary graph for flow-sensitive interprocedural analysis is made up of the same four kinds of nodes [1]. The points-to information at each of these heap nodes is represented as a points-to-graph, where its alias set is the key set.

A call heap use is mapped to the entry $d\phi$ of the invoked function, as this store information needs to be propagated to all possible uses. Similarly, the exit heap use node is mapped to the return call $d\phi$ node, since it collects all possible modifications of heap variables that are shared between caller and callee.

A program is input in **PSEG** form to the interprocedural **PSEGPT** algorithm. The interprocedural analysis is context insensitive. The main entry of the algorithm is **MainProc** (Lines 1~5). The global worklist `GlobalWL` now also contains functions as elements. **MainProc** initializes `GlobalWL` to the root function and invokes function **PSEGPT**.

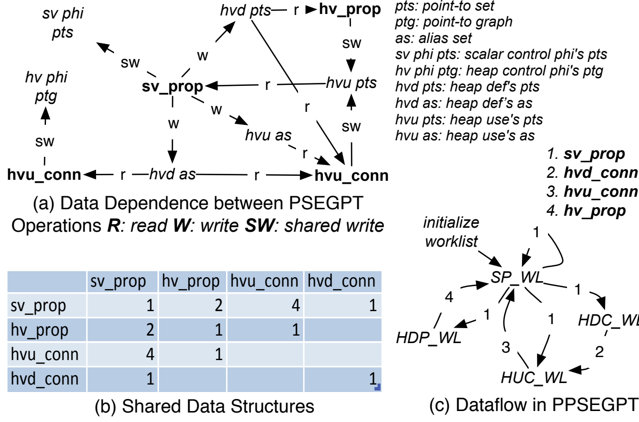
Function **PSEGPT** is the main driver for the points-to analysis. It takes a function `F` as an argument and applies the four analysis operations introduced in Section 3.3 to `WL` for the given input function.

PSEGPT handles strong updates at call sites. A function `F` strongly updates address-taken variable `a` ⇔ `a` is strongly updated on all paths that dominate **F**'s exit heap use node. In the function **BackTrace**, a call return dϕ node strongly updates heap variable `a` ⇔ `a` is strongly updated by all of call site `c`'s target functions. Thus, **PSEGPT** handles all three cases of strong updates (see Section 3.3) in the **BackTrace** step that connects heap uses `U` to all heap defs that may alias `U` by tracing backward through **PSEGPT** def-use chains.

3.6 Discussion

Monotonicity and Fixpoint Convergence: As discussed above, **PSEGPT** iteratively applies the four steps that propagate points-to information. Both the transfer function and the join operation are monotonic with respect to the partial orders for both scalar and heap variable's def-use chains, thereby ensuring **PSEGPT**'s fixpoint convergence. For the interprocedural case, **PSEGPT** maintains the per-function memo (i.e., the extra $d\phi$s and heap uses) that record the changes made by the last visit, and only the modified parts can be re-analyzed. Thus the algorithm will also reach a fixpoint in the interprocedural case.

Space Complexity: We discuss **PSEGPT**'s space complexity by using the following terms: $\mathcal{A}$, the number of address-taken variables; $\mathcal{P}$, the number of program points. The space complexity for the SS algorithm is $O(\mathcal{A}^2 \times \mathcal{P})$ [9]. In contrast, the space complexity for **PSEGPT** is $O((\mathcal{A}-\mathcal{S})^2 \times \mathcal{P} + \mathcal{S} \times \mathcal{P})$, where $\mathcal{S}$ is the number of address-taken variables with strong updates. In the extreme case when all address-taken variables



(a) Data Dependence between PSEGPT Operations *R*: *read* *W*: *write* *SW*: *shared write*

	sv_prop	hv_prop	hvu_conn	hvd_conn
sv_prop	1	2	4	1
hv_prop	2	1	1	
hvu_conn	4	1		
hvd_conn	1			1

(b) Shared Data Structures

(c) Dataflow in PPSEGPT

Figure 2. Data Dependence in PSEGPT and Dataflow in PPSEGPT

participate in strong updates ($\mathcal{S} = \mathcal{A}$), the space complexity for **PSEGPT** reduces to $O(\mathcal{S} \times \mathcal{P}) = O(\mathcal{A} \times \mathcal{P})$, which is quadratic instead of the cubic space complexity of the SS algorithm.

Time Complexity: We discuss the time complexity for **PSEGPT** using the following additional terms: $\mathcal{H}_{\mathcal{D}}$, the number of $d\phi$ nodes; $\mathcal{H}_{\mathcal{U}}$, the number of heap uses; $\mathcal{H}_{C}$, the number of $c\phi$ nodes; $\mathcal{H}_{\mathcal{F}}$, the number of extended $d\phi$s and heap uses for function invocations; and $\mathcal{V}$, the number of scalar variables.

We use the number of visited scalar/address-taken variables as the unit for time complexity. **PSEGPT**'s time complexity is composed of the four analysis operations. O_{sv_prop} is $O(\mathcal{V} \times \mathcal{A})$, i.e., the number of scalar variables propagated through scalar SSA def-use edges. O_{hv_prop} depends on the worst case for value flow edges, which is $O(\mathcal{H}_{\mathcal{D}} \times \mathcal{H}_{\mathcal{U}} \times \mathcal{A})$. hvu_conn's complexity also depends on value flow edges, so O_{hvu_conn} is the same as O_{hv_prop}. hvd_conn depends on the number of accessed $d\phi$ and $c\phi$ nodes, thus O_{hvd_conn} is $O((\mathcal{H}_{\mathcal{D}} + \mathcal{H}_{C}) \times \mathcal{A})$. Based on these, **PSEGPT**'s worst-case complexity is cubic in time, the same as with SS and SUPT.

4 Parallel PSEG Points-To Algorithm

The previous section introduced a **PSEG**-based analysis algorithm that produces alias and points-to information for each heap variable and value flow edge between heap variables. In this section, we discuss how to parallelize the analysis presented in the previous section, and introduce **PPSEGPT**, a parallel version of **PSEGPT**. Section 4.1 discusses how the parallelized algorithm will make use of asynchronous task parallelism by processing worklist elements in parallel as tasks. Section 4.2 discusses its complexity.

4.1 Parallelizing Points-To Analysis

The **PSEGPT** algorithm has four analysis operations that propagate points-to information and build value flow edges on demand. All of these operations are sparse and functional in nature, thereby creating an opportunity for executing them in parallel.

Parallel execution is constrained by data dependences among operations. Figure 2(a) represents the four operations (annotated in bold font) described in Section 3.4: sv_prop, hvd_conn, hvu_conn, and hv_prop. It also represents the data structures (annotated in italic font) that hold points-to and alias sets for heap defs and uses, points-to sets for scalar ϕ nodes, and points-to graphs for heap $c\phi$ nodes. Dependences are enforced between pairs of operations whenever necessary to ensure that read/write or write/write operations are not performed concurrently on the same datum. Figure 2(b) presents, for each pair of operations, the number of data structures from Figure 2(a) that they share. To maximize parallelism, we separate worklist processing into four stages, each corresponding to a separate operation.

Structure of Worklists: Based on the four execution stages corresponding to the four types of operations, we now describe the restructuring of the single global worklist described in Section 3.4 into four worklists. The interaction among these worklists is shown in Figure 2 (c). The worklists contain the following elements:

- **SP_WL**: scalar variables and heap uses that have modified points-to sets (processed by sv_prop);
- **HDC_WL**: $d\phi$ and $c\phi$ nodes whose alias sets have been modified and need to be connected to additional heap uses (processed by hvd_conn);
- **HUC_WL**: heap uses and dominating heap defs whose alias sets have been modified, and need to be connected to reaching definitions (processed by hvu_conn);
- **HDP_WL**: $d\phi$ and $c\phi$ nodes whose points-to sets have been modified and need to be propagated to their heap uses via value flow edges (processed by hv_prop).

Initially, **PPSEGPT**, like **PSEGPT**, collects the allocation sites and accessed global variables for a given function. But here it adds these elements to SP_WL. In Stage 1 (sv_prop), **PPSEGPT** processes elements in SP_WL and adds each element to one of the four worklists. In Stage 2 (hvd_conn), the elements of HDC_WL are processed and the collected heap uses are added to HUC_WL. Stage 3 (hvu_conn) connects heap uses whose alias sets have changed with their possible defs and puts those heap uses whose points-to sets are modified into SP_WL. Stage 4 (hv_prop) propagates the points-to information from $d\phi$s and $c\phi$s to heap uses and adds the heap uses with modified points-to-sets to SP_WL, as with Stage 3.

Figure 2(c) shows the inputs/outputs corresponding to each stage of **PPSEGPT**, which are based on heap variables' alias sets, points-to sets and the four worklists discussed above.

The reason for dividing **PSEGPT** into four stages is to enable the creation of lightweight tasks with maximum parallelism. In this way, the synchronization required for shared data between elements in the same stage/worklist is minimized. The processing of a worklist element is implemented as a task that can run in parallel with other such tasks, as in other parallel worklist algorithms [20]. At every point in the execution of the algorithm, one of the worklists can process multiple elements in parallel. `sv_prop` is data independent (excluding the $c\phi$ processing); the collecting of heap uses (i.e. `hvd_conn`) needs to be synchronized with updating $c\phi$'s points-to graph; `hvu_conn` is fully data independent; `hv_prop` needs to be synchronized on shared uses.

4.2 Complexity Analysis

PPSEGPT does not change the space complexity relative to **PSEGPT**, so we only focus on the time complexity in this section assuming a parallel machine with $\mathcal{N}$ processors (cores). As discussed in Section 4.1, **PPSEGPT** runs the four operations (introduced in Section 3.4) in parallel in four different stages, thus **PPSEGPT**'s time complexity is the sum of those four stages. For `sv_prop`, the synchronization happens on concurrently accessed scalar $c\phi$ nodes, so O_{sv_prop} is: $O((\frac{\mathcal{V}-\mathcal{V}_C}{\mathcal{N}} + \mathcal{V}_C) \times \mathcal{A})$. Here $\mathcal{V}_C$ stands for the number of scalar merge ϕs. O_{hv_prop} depends on the number of concurrently updated heap uses. In the worst case, it is the same as **PSEGPT**. `hvu_conn` can be fully parallelized, so we have $O(\frac{\mathcal{H}_\mathcal{D} \times \mathcal{H}_\mathcal{U} \times \mathcal{A}}{\mathcal{N}})$. O_{hvd_conn} depends on the concurrently accessed $\mathcal{H}_C$, thus it is: $O((\frac{\mathcal{H}_\mathcal{D}}{\mathcal{N}} + \mathcal{H}_C) \times \mathcal{A})$.

5 Evaluation

5.1 Implementation

The **PSEGPT** and **PPSEGPT** versions of the algorithm were implemented as analysis passes in the LLVM [17] Version 3.6.2 compiler framework. They take LLVM bitcode as input and generate in-memory IR, invoking the SSA builder to produce the **PSEG** form. **PSEGPT** or **PPSEGPT** can then be performed on the **PSEG** form. For parallelism, we used the Habanero C/C++ library [11], a lightweight task parallelism library that supports nested task creation and termination APIs (**async**, **finish**) with work-stealing schedulers. We also used the compare-and-swap API to implement lightweight spin locks to support mutual exclusion with guaranteed deadlock and livelock avoidance.

Since our pointer analysis is flow-sensitive, **singleton** points-to sets are used to identify strong updates for a given heap variable. In our implementation, a singleton points-to set is a set containing a single element that is one of following: a global variable; a local variable that is not allocated or used in a loop or recursive call path; or a dynamically allocated variable that is not allocated in a loop or recursive call path. Similar to [15], our pointer analysis starts from an estimated call graph, i.e., a call graph in which an invoked function pointer is conservatively assumed to point to any function whose address has been taken. The call graph is then updated as the algorithm progresses.

As mentioned in Section 1, our analysis is *field sensitive*. It analyzes the offset calculation (via LLVM's GEP instruction) for struct, class and array pointers when it processes load and store instructions. If the offset can be identified as a constant integer value, then it is encoded as a unique address for load and store operations. If the offset is unknown, then it is encoded as an address that can point to any offset of the given struct, class or array.

5.2 Evaluation

Experimental Setup: Our experimental evaluation was conducted on the six benchmarks listed in Table 3, which provides their number of lines of source code, the number of functions and LLVM instructions, and the number of scalar variables (**scalar vars**) in the first six columns. Three of the benchmarks (GhostScript, Vim, GCC) have been used in past work on pointer analysis [9, 15, 19], of which GCC is the largest benchmark in lines of code. We added three more benchmarks (V8, Ruby, Tizen) to obtain larger and more representative examples from real-world C/C++ applications.

Columns 7~11 in Table 3 show the **PSEG**-related statistics, including the number of address-taken variable uses, $d\phi$s, $c\phi$s, call sites, and allocation sites (**allocs**), which include LLVM **alloca** instructions, global variables, memory allocation intrinsics and memory-related APIs.

All C/C++ source code was compiled into LLVM bitcode via the Clang front end. Before performing pointer analysis, we applied the LLVM optimization **mem2reg** (i.e., scalar replacement and ϕ creation for scalar variables) and scalar optimizations to the bitcode. All results were obtained on an Intel Westmere node, which has two 6-core Intel Xeon X5660 CPUs at 2.83GHz with 48GB of memory running Red Hat Linux (RHEL 5).

Evaluation: We evaluated the **PPSEGPT** analysis with respect to performance, scalability, memory usage and precision. For precision, we compared its effectiveness with respect to identifying strong updates in store operations with SUPT and compared its number of discovered inlinable indirect call sites with respect to SUPT and SS. To obtain as close to an apples-to-apples comparison as possible, we ported the implementations of SUPT ([15]) and SS ([9]) from LLVM 2.6 to 3.6.2, since **PPSEGPT** was implemented in LLVM 3.6.2.

The memory usage of our algorithm is shown in the last two columns in Table 3, including running the **PPSEGPT** analysis and its corresponding memory usage for LLVM initialization (which includes parsing bitcode, **mem2reg**, scalar optimizations, and building the **PSEG**).

For the performance comparison, Table 4 gives the execution time for SS [9], SUPT [15], **PSEGPT** and **PPSEGPT** (12 threads). Due to its semi-sparse nature (i.e., running densely on load/store operations), SS is the slowest. For example, SS

Table 3. Statistics for benchmark characteristics, **PSEG** characteristics, **PPSEGPT** memory usage, and LLVM memory usage

Benchmark	Desc	LOC	# of Funcs	# of insts	# of scalar vars	# of uses	# of $d\phi$s	# of allocs	# of $c\phi$s	# of calls	PPSEGPT (MB)	LLVM (MB)
GhostScript	PS&PDF Intepreter	282.7k	6,408	433.4k	164,543	56,438	22,293	3,902	23,397	22,780	548	116
Vim	Vim editor	310.8k	3,793	350.3k	95,271	57,800	16,917	4,241	30,269	27,583	499	94
V8	Google V8 JS engine	332.3k	19,683	583.9k	291,964	59,138	34,946	38,204	25,285	99,241	1,250	167.6
GCC	GNU GCC Compiler	382.9k	5,577	587.6k	208,656	125,595	20,158	1,599	31,765	52,249	671	148
Ruby	Ruby RT& Compiler	638.8k	5,366	87.1k	301,732	40,365	10,395	15,609	19,402	28,339	467	82
Tizen	TizenOS	2,205k	91,839	3,771.9k	1,404,564	461,844	214,156	698,008	254,666	109,483	10,824	6,439

Table 4. Analysis execution times in seconds for SS [9], SUPT [15], **PSEGPT** (1 thread) and **PPSEGPT** (12 threads). *Time-out* refers to an execution that had to be terminated after running for more than 5,400 seconds.

Benchmark	SS	SUPT	PSEGPT	PPSEGPT
GhostScript	42.0	1.7	2.0	0.7
Vim	429.5	5.9	6.0	1.0
V8	1,139.5	418.7	402.6	118.4
GCC	83.5	32.0	60.0	14.6
Ruby	37.8	2.4	2.5	1.0
Tizen	*Time-out*	601.5	642.4	193.4

Table 5. A comparison of the precision of SUPT [15], SS [9] and **PPSEGPT** algorithms, obtained by counting the number of strong updates (SUDs) and inlinable indirect calls (IICs). For both metrics, larger values represent higher levels of precision.

	# of SUDs		# of IICs		
Benchmark	SUPT	PPSEGPT	SUPT	SS	PPSEGPT
GhostScript	5,254	6,390	12	57	57
Vim	4,391	5,298	32	53	53
V8	17,794	26,616	31	45	47
GCC	2,507	5,617	18	45	45
Ruby	2,297	5,492	21	43	43
Tizen	8,820	108,696	314	—	533

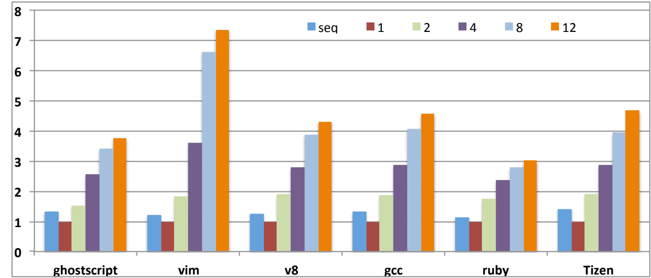


Figure 3. Speedup of sequential **PSEGPT**, and parallel implementations of **PPSEGPT**, relative to a single-thread implementation of **PPSEGPT**.

was unable to complete the analysis of the Tizen workload in 90 minutes (5400 seconds). Due to its flow-sensitive nature, the sequential version of **PSEGPT** ran a bit slower than SUPT on most of the benchmarks. However, by leveraging parallelism, **PPSEGPT** (running with 12 threads) resulted in faster analysis times than SUPT, while also delivering increased precision as indicated below.

For the precision comparison, the 2nd and 3rd columns in Table 5 show the number of strong updates (SUDs) obtained when using the SUPT and **PPSEGPT** algorithms. Large SUD values indicate a more precise points-to analysis. **PPSEGPT** identified more strong updates than SUPT due to its flow sensitivity and its field sensitivity, which can identify struct/class fields and array elements with constant integer indices.

We also developed a client application for points-to analysis, an inlinable indirect calls (IIC) analyzer which determines whether an indirect call site (i.e., function pointer invocation) has a single target thereby enabling it to be expanded inline without a conditional guard. The last three columns in Table 5 show how many inlinable call sites (IICs) were identified by the SUPT, SS and **PPSEGPT** algorithms. **PPSEGPT** identified more IICs than SS and SUPT. Since **PPSEGPT** analysis can identify array elements with constant indices, vtable elements that have a single target can easily be identified. SS is field sensitive as well, but it still missed some array element cases.

For scalability, we evaluated **PPSEGPT** on a 12-core Xeon SMP system. Figure 3 shows the speedups of sequential **PSEGPT** (denoted by `seq`) and of parallel implementations of **PPSEGPT**, relative to a single-thread implementation of PPSEGPT. The best case for scalability is **Vim**, which gives an improvement of *7.35*× running on 12 threads. The average speedup for the 12 threads case is *4.45*× and the geometric mean is *4.62*×. Our experimental results show that the

PPSEGPT algorithm can fully analyze a 2.2MLOC application (Tizen) while using < 16GB of memory and delivering an average analysis rate of > 10KLOC/second.

6 Related Work

LLVM-based Pointer Analysis: Pointer analysis based on SSA form has been studied intensively in past work. Here we confine our discussion to the most closely related SSA-based work, which is based on LLVM.

Hardekopf and Lin present SS [9]: a flow-sensitive points-to analysis based on LLVM's partial SSA. Their analysis is sparse for top-level variables, for which it follows scalar SSA def-use chains. It uses iterative dataflow analysis for address-taken variables and maintains a points-to graph per program point for all such variables. Operations on address-taken variables are labeled and connected by a SEG representation [22]. For memory efficiency, they use binary decision diagrams for maintaining points-to graphs. In later work [10], they stage pointer analysis with a pre-analysis, an auxiliary flow-insensitive pointer analysis that computes conservative def-use information for address-taken variables. This information is used by the primary flow-sensitive analysis, which is sparse and uses SSA form for all variables. The resulting analysis is an order of magnitude more scalable than their SS analysis. Su et al. [28] present a similar multi-stage technique that employs flow-insensitive pointer analysis as the auxiliary analysis. Their technique supports both flow and context sensitivity. However, it is unclear if any of these algorithms is amenable to parallelization.

Lhotak and Chung's SUPT [15] algorithm performs an SSA-based points-to analysis and maintains a global points-to graph for scalar variables. For address-taken variables, they maintain a single points-to-graph for the whole program instead of per program point. This results in a loss of flow-sensitive precision. However, where this representation identifies a singleton points-to set for the variable of a store operation, they perform a strong update. Based on an experimental evaluation that shows that strong update stores occur frequently in many applications, they find an effective balance between precision and speed/memory efficiency. SUPT employs a global worklist, whose elements contain constraints that are applied to points-to sets. This worklist-based algorithm could be staged like ours, making it amenable to task parallelism. But because their algorithm is partly flow-insensitive, it has larger points-to sets, which would result in more data sharing than in our case.

Li et al. [16] introduce a value-flow based points-to analysis that iteratively examines the set of pointers that point to allocation sites by checking graph reachability, and dynamically adds value-flow graph edges to connect scalar variables and memory objects (indirect flows). Their algorithm is also worklist-based and produces the points-to set for each variable. Since their indirect flow calculation needs to resolve dataflow equations sequentially, this algorithm is also difficult to be parallelized due to data sharing. In [25], Sui et al. present a sequential flow-sensitive pointer analysis for multi-threaded programs. Their technique performs may-happen-in-parallel and value-flow analysis to identify the happen-before pairs that assist lock analysis to establish thread-aware def-use information for pointer analysis.

PPSEGPT balances precision and performance in a novel way, compared to past work. In **PPSEGPT**, the dataflow traversal process is divided into multiple subtasks, and thus can be easily parallelized by using lightweight task parallelism with proper synchronization. **PPSEGPT** maintains a global points-to graph for scalar variables in the same manner as SUPT and SS. For address-taken variables, it maintains points-to information for each heap variable. This saves memory in comparison to maintaining a points-to graph at every point, as with SS.

Sparse Pointer Analysis:

The algorithm by Tok et al [26] is similar to our algorithm in that it integrates points-to analysis with an on-demand construction of heap def-use chains. It differs from ours in the use of interprocedural def-use chains and in maintaining a worklist of basic blocks per procedure. Also, they employ the technique of dynamically computing SSA form, which limits the scalability of their analysis.

Parallel Pointer Analysis: Mendez-Lojo et al. introduce a parallel inclusion-based flow-insensitive points-to analysis [18]. This algorithm is based on graph rewriting, defining a set of rules for constraint solving. In [24], Su et al. present a parallel points-to analysis for Java programs. Their technique runs CFL-reachability operations in parallel by identifying data sharing and adding locks for protection.

Nagaraj and Govindarajan [19] furthered the development of the graph rewriting approach by extending the rewriting rules from [18] to support flow sensitivity, thereby obtaining a parallel flow-sensitive pointer analysis. They use an auxiliary flow-insensitive analysis (as in [10]) to conservatively build def-use chains for address-taken variables. The flow-insensitive aspect of their work leads to more data dependences and adds synchronization overhead for protecting the shared data structures. Their experimental results showed many cases of negative scalability, whereas **PPSEGPT** always showed improved performance improvements with an increased number of threads. We did not find a publicly available implementation of their algorithm for an experimental comparison.

7 Conclusions and Future Work

This paper introduced a novel algorithm for points-to analysis, based on **Pointer SEG (PSEG)** form, an adaptation of SEG form. In contrast to classic constraint-solving based approaches, our algorithm can decompose the analysis into units of granularity that are well suited for fine-grained

task parallelism. The analysis also produces precise def-use connections (value flow edges) between memory stores and loads. We implemented our **Parallel Pointer SEG-based Pointer Analysis** in the LLVM compilation framework by leveraging the fine-grain parallelism capabilities of HCLib. We evaluated this analysis with six real-world applications on a 12-core Intel Xeon SMP, and obtained an average speedup of *4.45*× with 12 threads and maximum speedup of *7.35*×, compared to the sequential runs.

In this paper, we evaluated our algorithm on a weakly-typed language, in which case **PSEG** form has to conservatively assume that each node may be aliased to all others in same heap space. There are many opportunities for future work. The analysis can be extended to apply to binary programs, which are inherently weakly typed. One can also explore the use of multiple heap spaces in strongly-typed programs, and the addition of context sensitivity to our interprocedural flow-sensitive pointer analysis algorithm. Finally, one can extend the implementation of our algorithm to leverage many-core/accelerator parallelism and distributed-memory parallelism for further scalability.

Acknowledgments

We are grateful to Jong-Deok Choi and the anonymous reviewers for their constructive suggestions and comments, which helped improve the presentation of the paper.

References

[1] David Callahan. 1988. The Program Summary Graph and Flow-sensitive Interprocedural Analysis. In *PLDI '88*. ACM, New York, NY. https://doi.org/10.1145/960116.53995

[2] Jong-Deok Choi, Michael G. Burke, and Paul R. Carini. 1993. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. In *POPL'93*. https://doi.org/10.1145/158511.158639

[3] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. 1991. Automatic Construction of Sparse Data Flow Evaluation Graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '91)*. ACM, New York, NY, USA, 12. https://doi.org/10.1145/99583.99594

[4] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. 1994. Context-sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *PLDI '94*. ACM, New York, NY, USA, 15. https://doi.org/10.1145/178243.178264

[5] Stephen Fink et al. 2006. Effective Typestate Verification in the Presence of Aliasing. In *ISSTA '06*. ACM, New York, NY, USA, 12. https://doi.org/10.1145/1348250.1348255

[6] Stephen J. Fink, Kathleen Knobe, and Vivek Sarkar. 2000. Unified Analysis of Array and Object References in Strongly Typed Languages. In *SAS '00*. http://dl.acm.org/citation.cfm?id=647169.718147

[7] Rakesh Ghiya and Laurie J. Hendren. 1998. Putting Pointer Analysis to Work. In *POPL '98*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/268946.268957

[8] Samuel Z. Guyer and Calvin Lin. 2005. Error Checking with Client-driven Pointer Analysis. *Sci. Comput. Program.* 58, 1-2 (Oct. 2005), 32. https://doi.org/10.1016/j.scico.2005.02.005

[9] Ben Hardekopf and Calvin Lin. 2009. Semi-sparse flow-sensitive pointer analysis. In *POPL '09*. ACM, New York, NY, USA, 13. https://doi.org/10.1145/1480881.1480911

[10] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive Pointer Analysis for Millions of Lines of Code. In *CGO '11*. Washington, DC, USA, 10. https://doi.org/10.1109/CGO.2011.5764696

[11] Akihiro Hayashi, Sri Raj Paul, Max Grossman, Jun Shirako, and Vivek Sarkar. 2017. Chapel-on-X: Exploring Tasking Runtimes for PGAS Languages. In *ESPM2'17*. ACM, New York, NY, USA, Article 5, 8 pages. https://doi.org/10.1145/3152041.3152086

[12] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. 1999. Interprocedural Pointer Alias Analysis. *ACM Trans. Program. Lang. Syst.* 21, 4 (July 1999), 848–894. https://doi.org/10.1145/325478.325519

[13] Kathleen Knobe and Vivek Sarkar. 1998. Array SSA form and its use in parallelization. In *POPL '98*. ACM, New York, NY, USA, 107–120. https://doi.org/10.1145/268946.268956

[14] William Landi and Barbara G. Ryder. 1992. A Safe Approximate Algorithm for Interprocedural Aliasing. In *PLDI '92*. ACM, New York, NY, USA, 235–248. https://doi.org/10.1145/143095.143137

[15] Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to analysis with efficient strong updates. In *POPL '11*. ACM, New York, NY, USA, 14. https://doi.org/10.1145/1926385.1926389

[16] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the Performance of Flow-sensitive Points-to Analysis Using Value Flow. In *ESEC/FSE '11*. ACM, New York, NY, USA, 11. https://doi.org/10.1145/2025113.2025160

[17] LLVM 2018. LLVM Compiler Infrastructure. http://llvm.org.

[18] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel Inclusion-based Points-to Analysis. In *OOPSLA '10*. ACM, NY, USA, 16. https://doi.org/10.1145/1869459.1869495

[19] Vaivaswatha Nagaraj and R. Govindarajan. 2013. Parallel Flow-sensitive Pointer Analysis by Graph-rewriting. In *PACT '13*. IEEE Press, Piscataway, NJ, USA, 10. https://doi.org/10.1109/PACT.2013.6618800

[20] Keshav Pingali et al. 2011. The Tao of Parallelism in Algorithms. In *PLDI '11*. ACM, New York, NY, USA, 12–25. https://doi.org/10.1145/1993316.1993501

[21] G. Ramalingam. 1994. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.* 16, 5 (Sept. 1994), 1467–1471. https://doi.org/10.1145/186025.186041

[22] G. Ramalingam. 2002. On sparse evaluation representations. *Theor. Comput. Sci.* 277, 1-2 (April 2002), 119–147. https://doi.org/10.1016/S0304-3975(00)00315-7

[23] Vivek Sarkar and Kathleen Knobe. 1998. Enabling Sparse Constant Propagation of Array Elements via Array SSA Form. In *SAS'98*. http://dx.doi.org/10.1007/3-540-49727-7_3

[24] Yu Su, Ding Ye, and Jingling Xue. 2014. Parallel Pointer Analysis with CFL-Reachability. In *ICPP'14*. IEEE Computer Society, Washington, DC, USA. https://doi.org/10.1109/ICPP.2014.54

[25] Yulei Sui, Peng Di, and Jingling Xue. 2016. Sparse Flow-sensitive Pointer Analysis for Multithreaded Programs. In *CGO '16*. ACM, New York, NY, USA, 160–170. https://doi.org/10.1145/2854038.2854043

[26] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. 2006. Efficient Flow-Sensitive Interprocedural Data-Flow Analysis in the Presence of Pointers. In *CC'06,*. 17–31. https://doi.org/10.1007/11688839_3

[27] Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-sensitive Pointer Analysis for C Programs. In *PLDI '95*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/223428.207111

[28] Hongtao Yu et al. 2010. Level by Level: Making Flow- and Context-sensitive Pointer Analysis Scalable for Millions of Lines of Code. In *CGO '10*. ACM, New York, NY, USA, 218–229. https://doi.org/10.1145/1772954.1772985