# Parallel Flow-Sensitive Points-to Analysis

Jisheng Zhao
Rice University
jisheng.zhao@rice.edu

Micheal Burke
Rice University
mgb@rice.edu

Vivek Sarkar
Rice University
vsarkar@rice.edu

## Abstract

Points-to analysis is a fundamental requirement for many program analyses, optimizations, and debugging/verification tools. However, finding an effective balance between performance, scalability and precision 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 on precision. Further, given the recent multicore hardware trends, more attention needs to be paid to the use of parallelism for improved performance.

In this paper, we introduce a new pointer analysis based on Pointer SSA form (an extension of Array SSA form,) which is flow-sensitive, memory efficient, and can readily be parallelized. It decomposes the points-to analysis into fine-grained units of work that can be easily implemented in an asynchronous task-parallel programming model. More specifically, our contributions are as follows: 1. A Pointer SSA (**PSSA**)-based scalable interprocedural flow-sensitive context-insensitive pointer analysis (**PSSAPT**) that produces both points-to and heap def-use information, and supports the task parallel programming model; 2. a preliminary evaluation, including scalability and precision, of the implementation of parallel **PSSAPT** using a lightweight task-parallel library. Our experimental results with 6 real world applications (including the 2.2MLOC Tizen OS framework) on a 12-core machine show an average speedup of *4.45*× and maximum speedup of *7.35*×. Our evaluation also includes precision results for an inlinable indirect call analysis.

## 1. Introduction

Many compiler analyses and optimizations rely on pointer alias information. Pointer (or points-to) analysis determines if two pointer expressions may refer to the same memory location. A large number of compiler optimizations, ranging from simple dead code elimination to automatic parallelization, need pointer alias information. Static analysis, and more specifically alias analysis, is in general undecidable [23]. Hence, a large number of approximation algorithms have been published that balance the precision of the results and the efficiency of the analysis. These algorithms explore various dimensions to achieve this balance. Our focus in this paper is primarily on flow-sensitive pointer analysis, which has been shown to be important for a growing list of program analyses [1, 8], including those that check for security vulnerabilities [5, 9], that synthesize hardware [30] and that analyze multi-threaded codes [26].

The traditional flow-sensitive approach [2, 4, 14, 28] uses an 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 [3, 12], which directly connects variable definitions (defs) with their uses, allowing data flow facts to be propagated only to those program locations that need the values. Hardekopf and Lin [10] present a semi-sparse flow-sensitive pointer analysis which exploits partial SSA form to perform a sparse analysis on scalar variables, while using iterative dataflow analysis on other variables.

Sparse pointer analysis is problematic because pointer information is required to compute the def-use information that would enable a sparse analysis. Hardekopf and Lin [11] address this problem with a staged analysis which includes a flow-insensitive pointer analysis that generates conservative def-use information as a pre-analysis, followed by a sparse primary analysis.

Lhotak and Chung [15] perform an SSA-based points-to analysis. For a more compact representation for heap variables, 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. When strong update stores are the most common case in applications, their analysis provides an effective balance between precision and speed. However, the flow-insensitive aspect of their algorithm increases the size of their points-to sets, which is an impediment to effective parallelization.

Given the rapid development of multi-core systems in the last decade, leveraging parallel computation infrastructure to pointer analysis can accelerate the performance of compilation, verification, and other software tooling. There are now efforts to develop parallel pointer analyses. Mendez-Lojo et al. [19] introduce a parallel points-to analysis algorithm based on graph rewriting. Nagaraj and Govindarajan [20] further the development of the graph rewriting approach to parallel pointer analysis by extending the rewriting rules to support flow sensitivity.

In this paper, we introduce a new pointer analysis, based on Pointer SSA form (an extension of Array SSA form [13], details described in Section 3.1), which focuses on analysis of weakly-typed languages. It is flow sensitive and can readily be parallelized. More specifically, our contributions are as follows:

- A new interprocedural flow-sensitive, field-sensitive and context-insensitive pointer analysis (PSSAPT) with the following characteristics:
  - The analysis is built on a *Pointer SSA* (PSSA) form that, for supporting weakly-typed languages, is based on treating the



, .
Copyright © ACM [to be supplied]. . . $15.00.
http://dx.doi.org/10.1145/

entire heap as a single array in Array SSA form extended with *pointer flow* edges.

  - The analysis produces precise points-to information and precise heap-def-use information in a strongly coupled way. Both forms of information can be used to enable a wide range of sparse analyses including devirtualization analysis, escape analysis, etc.

- Creation of sequential and parallel versions of the PSSAPT algorithm.
- An evaluation of an implementation of the sequential and the parallel versions of PSSAPT using LLVM and a lightweight task parallelism library. The parallel version achieves an average parallel speedup of *4.45*×on 12 processor cores. Further, the algorithm shows improved precision compared to well-known past work [15].

The rest of this paper is organized as follows. Section 2 provides background on the LLVM infrastructure, flow-sensitive pointer analysis, and Array SSA form. Section 3 introduces the Pointer SSA form (PSSA) and presents the PSSA-based pointer analysis (PSSAPT). Section 4 describes a parallelization of PSSAPT (PPSSAPT) based on the use of fine-grain task parallelism in processing worklist elements. Section 5 presents details of the implementation in LLVM, and an evaluation of the algorithm based on the implementation that includes its effectiveness with respect to inlinable indirect call analysis. 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 our paper.

### 2.1 LLVM and Static Single Assignment Form

The pointer analysis introduced in this paper is implemented in LLVM [17], and follows the LLVM convention of separating variables into two disjoint sets: top-level variables whose address is never taken, and address-taken variables that can be indirectly referenced via a pointer. The address-taken variable can be a stack or global variable or a dynamically allocated heap object. Figure 1(a) gives an example in C. Variables $\mathbf{p}_a$, $\mathbf{p}_b$, $\mathbf{p}_c$, $\mathbf{p}_1$, $\mathbf{p}_2$ and **x** are top-level variables, since their addresses are not exposed. Variables **a**, **b** and **c** are address-taken variables.

The LLVM IR is based on SSA form, which provides def-use chains for scalar variables (i.e. top-level variables). But 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 operations. In this paper, we use the terms *scalar SSA* for the LLVM IR, *scalar variable* for a top-level variable, and *Pointer SSA Form* (**PSSA**) that will be introduced in Section 3.1 to indicate the extension of scalar SSA form to support pointer accesses.

### 2.2 Flow-sensitive Pointer Analysis

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

Since address-taken variables can be modified at any program point without renaming, many flow-sensitive pointer analyses also introduce a *labeling* flag [10] $l_i$ at each program point. Here we use $v_i$ to denote scalar variables, $a_i$ for address-taken variables.

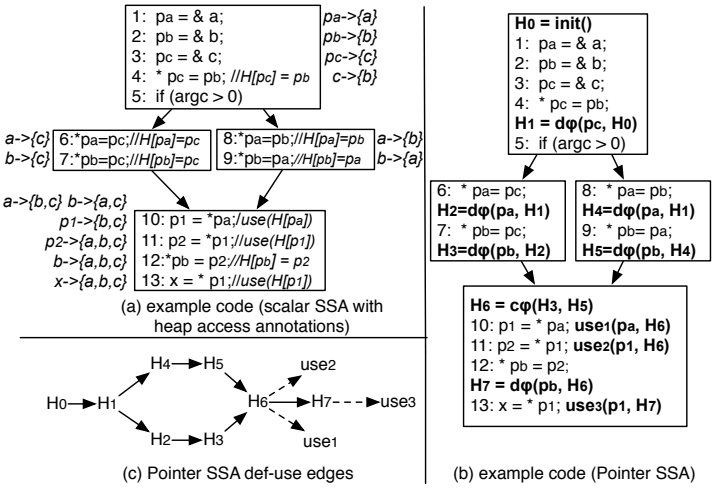



**Figure 1.** Code Example for **PSSA** form.

The representation of points-to graph edges in scalar SSA form is: **flow-insensitive**: $v_i \rightarrow a_j$ or $a_i \rightarrow a_j$; **flow-sensitive**: $v_i \rightarrow a_j$ or $a_i[l_j] \rightarrow a_k$. 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 $\mathbf{a}[l_6] \rightarrow \mathbf{c}$ and $\mathbf{a}[l_8] \rightarrow \mathbf{b}$, respectively, in a flow-sensitive pointer analysis.

### 2.3 Array SSA and Heap SSA Forms

Unlike scalar SSA form, Array SSA form [13] captures precise element-level data flow information for arrays, with the introduction of definition-$\phi$ ($d\phi$) operators. Past work has shown how program analyses such as sparse conditional constant propagation of array elements can be performed efficiently using Array SSA form. Subsequently, a Heap SSA form was introduced as an extension of Array SSA form to enable analysis of object references in strongly typed languages [6] by modeling each field as a distinct logical "heap array", and adding use-$\phi$ ($u\phi$) operators to chain together uses with the same def. [6] also introduced two analysis algorithms based on Heap SSA form, one to identify redundant loads and another to identify dead stores. In this paper, we show how Array SSA form can be extended to model pointer accesses in weakly-typed languages like C, by using a single heap array (Section 3.1).

## 3. PSSA based Pointer Analysis

This section introduces a new points-to analysis algorithm (**PSSAPT**) based on Pointer SSA Form (Section 3.1). This algorithm builds interprocedural flow-sensitive points-to information, and def-use connections between heap memory locations. The two analyses are tightly coupled. Section 3.2 describes how precise def-use connections for heap variables are created. Section 3.3 describes the basic mechanism of the PSSAPT algorithm, and Section 3.4 describes the details of PSSAPT. Section 3.5 discusses the monotonicity and fixpoint convergence properties, the precision and the worst-case time and space complexity.

### 3.1 Pointer SSA Form and Heap Variable

Pointer SSA (**PSSA**) form is based on Array SSA form [13], which chains array element defs to capture precise element-level dataflow information. Here we extend Array SSA to connect heap uses to their immediately dominating heap def or merge $\phi$ node, and perform renaming on both heap defs and uses. Figure 1 (b) shows the **PSSA** nodes added to the scalar SSA example shown in Figure 1

(a). To accommodate weakly-typed languages like C, we use a single heap array, **H**, to model all memory locations that can be accessed by pointers. Thus, all accesses to address-taken variables are modeled as elements of heap array **H**. In this paper, we define heap array elements as **heap variable**s that present the heap memory location and the relevant values. Line 4 illustrates the use of a $d\phi$ to model a heap def in a *store* instruction. Heap variable $\mathbf{H}_1$ represents the result of the STORE — it is identical to $\mathbf{H}_0$ except that **c** now contains the value $\mathbf{p}_b$. From a static analysis viewpoint, a heap variable can be abstracted as an <**address**, **value**> pair. For example, this abstraction of $\mathbf{H}_1$ will include the one pair $<\mathbf{p}_c, \mathbf{p}_b>$. The control merge of heap defs is presented as $c\phi$ in this paper. For example, the $c\phi$ before Line 10 represents the control merge for heap defs $\mathbf{H}_3$ and $\mathbf{H}_5$,

The heap use functions in Lines 10, 11 and 13 are used to model LOAD instructions (see the dashed lines in Figure 1 (c)). Unlike Heap SSA [6], **PSSA** does not have $u\phi$ nodes. However, each heap use still gets an unique name for the state of the heap after each LOAD instruction. The motivation for this feature is to allow the efficient identification of a given heap use's potential heap defs. For example in Figure 1 (c), the heap use $\mathbf{use}_3$'s potential reaching defs are identified as $\mathbf{H}_7$, $\mathbf{H}_6$ and $\mathbf{H}_1$ by tracking the immediately dominating connection $\mathbf{H}_7$ backward.

Based on this representation, a dataflow analysis can associate lattice variables with all $d\phi$, heap use, and $c\phi$ nodes. This is a natural representation of flow-sensitivity; i.e., each heap variable $\mathbf{H}_i$ represents a label $\mathbf{l}_i$ (see Section 2.2) since it has an unique name by **PSSA** renaming, and $\mathbf{H}_i$'s address operand is associated with a set of address-taken variables. The value of $\mathbf{H}_i$ can be another set of address-taken variables. This establishes flow-sensitive points-to information for each program point with each address-taken variable, and is a motivation for the use of **PSSA** as the input to our pointer analysis.

### 3.2 Pointer-Flow Edges for Heap Variables

As discussed in Section 3.1, PSSA chains heap defs and uses; i.e., its def-use edges connect heap uses with their immediately dominating $d\phi$ or $c\phi$. This is not precise def-use information for heap variables, but serves as input to the PSSAPT algorithm. PSSAPT takes PSSA form as input to perform flow-sensitive points-to analysis. Precise def-use connections, which we call **pointer flow edges**, are computed on demand as part of this analysis. We associate each heap variable $\mathtt{H}_i$ with a points-to set (PTS) and an alias set (AS) of heap locations, which represents the alias set for the pointer operands (i.e. address-taken variables) involved in its $d\phi$s, $c\phi$ nodes and heap uses. For the example shown in Figure 1 (b) (c), $\mathtt{AS}(\mathtt{H}_1)$ is $\{\mathtt{c}\}$, $\mathtt{AS}(\mathtt{H}_2)$ and $\mathtt{AS}(\mathtt{use}_1)$ are $\{\mathtt{a}\}$. As discussed later, these alias sets are computed on the fly using points-to information.

A pointer-flow edge edge is added between heap variables $\mathtt{H}_i$ and $\mathtt{H}_j$, only if $\mathtt{H}_i$ may-equal $\mathtt{H}_j$ (i.e. $\mathtt{AS}(\mathtt{H}_i) \cap \mathtt{AS}(\mathtt{H}_j) \neq \emptyset$) or $\mathtt{H}_i$ must-equal $\mathtt{H}_j$ (i.e. $\mathtt{AS}(\mathtt{H}_i) = \mathtt{AS}(\mathtt{H}_j)$, both $\mathtt{AS}(\mathtt{H}_i)$ and AS ($\mathtt{H}_j$) are **singleton** [1]).

Similar to scalar SSA form, a $c\phi$ node is also used to merge distinct heap defs(i.e. $d\phi$) from multiple control flow edges in PSSA. Since each $d\phi$ represents the potential updates of a set of address-taken variables, a $c\phi$ represents all such address-taken variables 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, where the alias set of $d\phi$ is its key set. For function invocations, we use a pair of heap variables (one heap use and one heap def) to represent a potential side effect. There can be multiple address-taken variables that are potentially read/written in the invoked function. The points-to information for each of these two heap variables is represented as a points-to graph, where its alias set is its key set.

### 3.3 Basic Mechanism and Structure of Worklists

**Basic Operations**: A legacy flow-sensitive points-to analysis [10] employs input/output points-to graph information at each program point, which leads to a dense analysis. This paper motivates a sparse analysis algorithm, which uses a dataflow traversal approach, i.e. propagating pointer information via def-use connections for scalar and heap variables. Since the def-use connections for heap variables are built on-the-fly starting from the empty set, the algorithm has to connect the heap variables when their alias sets are changed. Here are the 4 basic operations in the sparse analysis.

1. pointer information propagation via scalar variables' def-use connection (**sv_prop**): propagate scalar variables' points-to sets (PTS) via scalar SSA def-use edges (see Algorithm **PropToUses**);
2. connect heap variables when heap defs' alias sets AS are changed (**hvd_conn**): collect all heap uses that may be impacted by this def and propagate the points-to information to $c\phi$ nodes (see Algorithm **CollectUses**);
3. connect heap variables when heap uses' alias sets (AS) are changed (**hvu_conn**): connect this use with all possible heap defs(i.e. $d\phi$ or $c\phi$)(see Algorithm **Backtrace**) by adding pointer-flow edges, and update heap uses' PTS from defs;
4. pointer information propagation via pointer-flow edges when heap defs' PTS are changed (**hv_prop**): propagate heap variables' points-to sets (PTS) via pointer-flow edges, which are built on-the-fly (see next item) (see Algorithm **PropDPhi**).

**Basic Mechanism**: The PSSAPT algorithm is a worklist based algorithm, which keeps processing worklist elements until the worklist is empty. The elements in the worklist are either scalar variables or heap variables (i.e. heap use, def and $c\phi$).

PSSAPT starts from the instantiation of address-taken variables, including the allocation sites: e.g., global variables, **alloca** instructions, and known functions that can be modeled as allocation sites. The initialization of the worklist **WL** for a given function captures its allocation sites and accessed global variables. PSSAPT uses these elements as seeds for propagating the points-to information via scalar SSA def-use chains and the pointer-flow edges built on-the-fly.

The 4 operations that handle points-to information propagation and the building of heap variable def-use connections have been described in the previous section. PSSAPT iteratively repeats these 4 types of operations for the worklist until there are no more changes. In the initial step, PSSAPT collects all allocation sites, global variables and modifications from callers, and puts them into worklist (WL). For each element popped from WL, **PSSAPT** processes it using the corresponding operation.

**Illustrative Example**: To illustrate the PSSAPT algorithm, recall the example shown in Figure 1 (b),(c). Figure 2 demonstrates the progress of this algorithm in a sequence of steps, where each step corresponds to an operation and its result represents a "snapshot" of the computation, including the status of WL, the heap variables' alias/points-to sets, and pointer-flow edges. For d$\phi$s and heap uses, we use the format: $\mathtt{H}_i$:< *alias set*, *points-to set*>. For c$\phi$s we provide its points-to graph.

WL is initialized to the scalar variables that point to address-taken variables a, b and c (see *Step 1*). In *Step 2*, sv_prop is applied to each of the three scalar variables in WL. Through their scalar SSA def-use chains, heap variables are added to WL, and PTS and AS sets are updated. For example, a is used at Lines 6, 8, and

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

10, so $H_2$, $H_4$, and $use_1$ are added to WL; c is added to PTS($H_2$) and b is added to PTS($H_4$); a is added to AS($use_1$). The uses of $p_b$ and $p_c$ are processed in the same manner. The three scalar variables are removed from the worklist.

In *Step 3*, hvd_conn is applied to the $d\phi$s $H_2$, $H_4$ and $H_5$ Their points-to information is propagated to $c\phi$ $H_6$; their impacted heap uses, $use_2$ and $use_3$, are added to WL. Note that $H_3$ has still not been processed, so c has not yet been added to PTS(b). In *Step 4*, the hvu_conn operation is applied to heap use $use_1$. Elements of the points-to set of heap def $H_6$ are added to the points-to set of $use_1$. The pointer-flow edge $e_1$ is added to connect $H_6$ to $use_1$. The lhs scalar variable $p_1$ for the heap use is added to WL and $p_1$'s points-to set {b, c} is resolved.

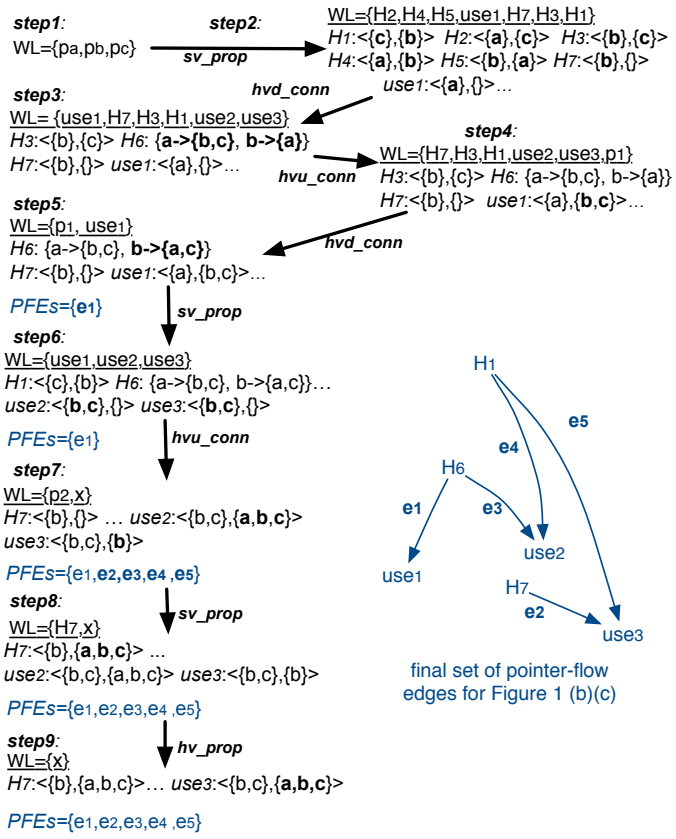



**Figure 2. PSSAPT** running example

In *Step 5*, hvd_conn is applied to the $d\phi$s $H_7$, $H_3$ and $H_1$. Their impacted heap use, $use_1$, is added to WL. $use_2$ and $use_3$'s alias sets are still empty since $p_1$ has not been processed yet, thus those two heap uses are not added to WL; $H_3$'s points-to information is propagated to $c\phi$ $H_6$. In *Step 6*, sv_prop is applied to $p_1$. Its points-to set is propagated to $use_2$ and $use_3$'s alias sets, and $use_2$ and $use_3$ are added to WL. In *Step 7*, hvu_conn is applied to heap uses $use_2$ and $use_3$. The points-to sets of $d\phi$ $H_1$ and b in $H_6$ are added to the points-to set of $use_2$. The points-to set of $d\phi$ $H_1$ is added to the points-to set of $use_3$. $use_2$ is connected to heap defs via pointer-flow edges $e_3$ and $e_4$. $use_3$ is connected to heap defs via pointer-flow edges $e_2$ and $e_5$. The lhs scalar variables $p_2$ and x for these heap uses are added to WL. $p_2$'s points-to set $\{a, b, c\}$ is resolved when hvu_conn is applied to $use_2$. In *Step 8*, sv_prop is applied to $p_2$, propagating PTS($p_2$) to $H_7$.

In *Step 9*, hv_prop is applied to $H_7$. Its points-to set is propagated to $use_3$ via pointer-flow edge $e_2$. The processing of $p_2$ produces no further changes.

### 3.4 Algorithm Details

Here we present the details of the PSSAPT algorithm. The main entry of the algorithm is **MainProc** (Lines 1∼5), which starts at the root function and processes the elements of a global worklist GlobalWL until the worklist is empty. The interprocedural analysis is context-insensitive.

Function **PSSAPT** is the main driver for the points-to analysis (i.e. applying the 4 operations introduced in Section 3.3) to the worklist elements for the given input function. First **PSSAPT** invokes **InitWorkLists** (Line 8) to initialize the worklists, then it iteratively perform the 4 points-to analysis operations (Lines 10∼20).

The **InitWorkLists** function performs context mapping from all callers to the current function (i.e., mapping the points-to information that needs to be input to the current function) and adds all address-taken variables and $d\phi$ s to the worklist. The allocation function's local sites only need to be processed once – the re-analysis of the current function will not handle them again. When the worklist is empty, **PSSAPT** updates the call site by invoking the **UpdateCallers** function and returns a set of all caller functions that need to be re-analyzed due to side effects (Line 21). The callers are put into GlobalWL for re-analysis (Line 22).

For Operation 1 (sv_prop, invoked at Line 13), **PropToUses** is the major function, which takes input operand v and propagates its points-to information via scalar SSA def-use chains. It creates a worklist (Line 26) that is returned and merged with the worklist in **PSSAPT** (Line 13). If the input operand v is a function call target's identifier or a function pointer, then the set of callees are added to GlobalWL for re-analysis (Line 27∼29) and **PropToUse** returns. Otherwise, **PropToUse** propagates v's points-to set via scalar SSA def-use chains. Here vi either stands for a scalar variable that is the assignee of a copy instruction or a heap variable which is accessed (i.e. load/store). For the copy operation, v's points-to set is merged with vi's points-to set. If there is a modification, then vi is passed to **PropToUses** (copy case, Line 34) or added to wl (merge $\phi$ case,Line 37). If v is a store instruction's value operand (Lines 38∼40) (i.e. v's point-to set represents vi's corresponding heap variable hv's point-to set), then merge its points-to set with the hv's points-to set and add hv to wl if there is a modification. The function **HeapVar** here is used to extract the heap variable from the given load or store instructions. If v is the pointer operand for either a load or store instruction, i.e. v's points-to set represents the alias set of a heap variable (Lines 41∼43), then merge v's points-to set with vi's heap variable hv's alias set and add hv to wl if there is a modification.

The function **CollectUses**, for Operation 2 (hvd_conn )of PSSAPT algorithm, is invoked at Line 16 if the alias set of v (i.e. a $d\phi$ ) has been modified. For the given input $d\phi$ **D**, it first checks and registers strong update (see Line 48, **CheckAndRegisterSU**). The functionality of **CheckAndRegisterSU** is: if **D** is strong update then register **D** with its dominance froniter **C**, otherwise unregister with **C**; if **D** is strong update and dominates current function **F**'s return path then register **D** with **F**, otherwise unregisters **D** with **F**. The second step is to propagate **D**'s alias set changes to its dominance frontier **C** using **UpdatePTG**(Line 59), and invokes **CollectUses** on **C**(Line 61).If there is a $d\phi$ **n** that post dominates **D** and strongly updates variable in **as**, then eliminates the strongly updated variable from **as** (see Line 54,55). Finally, all heap uses immediately dominated by **D** are collected and returned(Line 62∼65).

The **BackTrace** function handles Operation 3 (hvu_conn)'s connecting of a heap use to defs that reach it (**PSSAPT** Lines 19,20). This function takes a copy of a heap use's alias set as as a check set and traces backward through the PSSA def-use chains until there is no more heap defs (Line 3). For each heap def ($d\phi$ or $c\phi$), if its alias set has a non-empty intersection with the check set, then add a pointer-flow edge from the def node to the use and

```
1  function MainProc ()
   Input  : F : root function
   Output:
2  GlobalWL := ∅; GlobalWL ∪ = {F};
3  while GlobalWL ≠ ∅ do
4      func := PopFront (GlobalWL);
5      PSSAPT (func, GlobalWL);
6
7  function PSSAPT ()
   Input  : F : input function, GlobalWL : global worklist
   Output: boolean flag for side-effect
8  if ! InitWorkList (F) then
9      return FALSE;
10 while WL ≠ ∅ do
11     v := PopFront (WL);
12     if v is scalar then
13         WL ∪ =PropToUses (v, GlobalWL);
14     else if v is dphi 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 ≠ ∅;
24
25 function PropToUses ()
   Input  : v : variable or instruction, GlobalWL : global worklist
   Output: wl : worklist
26 wl := ∅;
27 if IsCallSite (v) then
28     callees := GetCallTarget (v);
29     GlobalWL ∪ = callees;
30 else
31     foreach vi in Use (v) do
32         if IsCopy (vi) then
33             PTS (vi) ∪ = PTS (v); //vi = v
34             wl ∪ = IsModified (PTS (vi)) ? PropToUses (vi) : ∅;
35         else if IsScalarCPhi (vi) then
36             PTS (vi) ∪ = PTS (v); //vi = φ (v, ...)
37             wl ∪ = IsModified (PTS (vi)) ? {vi} : ∅;
38         else if IsDPhiValueOp (vi, v) then
39             hv := HeapVar (vi); PTS (hv) ∪ = PTS (v); //vi: *p=v
40             wl ∪ = IsModified (PTS (hv)) ? PropDPhi (hv) : ∅;
41         else if IsPointerOp (vi, v) then
42             hv := HeapVar (vi); AS (hv) ∪ = PTS (v); //vi:
                  *v=... or vi: ...=*v
43             wl ∪ = IsModified (PTS (hv)) ? {hv} : ∅;
44 return wl;
45
46 function CollectUses ()
   Input  : D: dφ or cφ, as : the input alias set
   Output: wl : worklist
47 wl := ∅; as := AS (D); C := DomFrontier (D);
48 CheckRegisterSU (D, C, as);
49 if C then
50     n := ImmPostDom (D);
51     while n ≠ C do
52         if AS (n) = undef then
53             wl ∪ = D; return wl;
54         if IsSU (n, as) then
55             as := as \ AS(n);
56         if as = ∅ then
57             return wl;
58         n := ImmPostDom (n);
59     changed := UpdatePTG (as, PTS (D), PTG (C));
60     if changed then
61         CollectUses (C, as);
62 foreach U in DomBy (D) do
63     if AS (U) ≠ ∅ then
64         wl ∪ = U;
65 return wl;
```

add the def's points-to set to the use's points-to set (Line 6). If the heap def strongly updates the variables in $\Delta$ (function **IsSU** checks strong update), then eliminate the strongly updated address-taken variable from the check set (Line 8,9). Iterate in this fashion until the check set `as` is empty or there is no more heap def (Line 3). If there is update in the heap use's points-to set (the `mod` flag set at Line 7), then update its lhs variable `v`'s points-to set and put `v` back on the worklist (Line 12). This trace will also stop if the heap defs' are `undef` (see Definition A.3 A.4 A.5 in Appendix), which means that the alias sets were not initialized and PSSAPT can not check if the heap def is a strong update or not. Once the heap def's alias set is modified, the relevant heap uses' trace process will be triggered again.

```
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 IsSU (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 in 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 in 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);
```

After connecting heap variables' defs by pointer-flow edges, the points-to information from heap variables' defs is propagated to

uses via Operation 4 (hv_prop, **PSSAPT** Line 18). **PropDPhi** propagates $d\phi$s' points-to sets via pointer-flow edges. For a $c\phi$, the algorithm updates its points-to graph by **UpdatePTG**, and invokes **PropCPhi** to further propagate points-to information.

**Interprocedural Analysis**: As stated in Section 3.2, we extended **PSSA** to add a $d\phi$ and heap use pair for each function call site, where the heap use represents the potential uses of the address-taken variables in its alias set and $d\phi$ represents the potential defs of the address-taken variables in its alias set. Within the function, the call site's heap uses are mapped to $d\phi$(i.e. the functionality of **InitWorkList** in Section 3.4), as the store information needs to be propagated to all possible uses. In similar fashion, the return site is represented as a heap use that is mapped to the call site's $d\phi$ since it collects all possible modifications of those address-taken variables that are shared between caller and callee, producing side effects (i.e. the functionality of **UpdateCallers** in Section 3.4).

### 3.5 Discussion

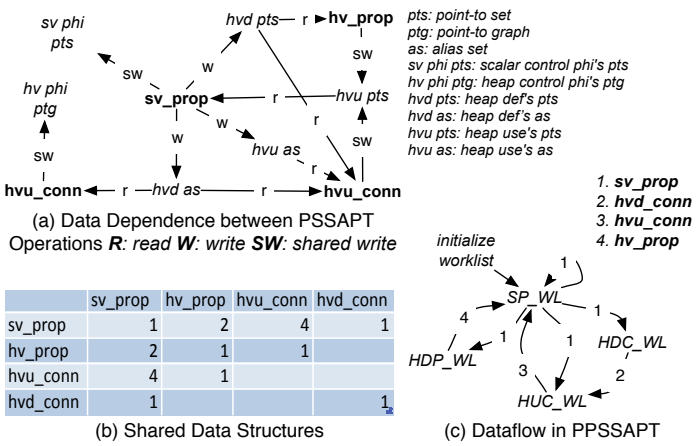
**Monotonicity and Fixpoint Convergence**: As shown above, PSSAPT iteratively applies the 4 steps that propagate points-to information. Both the transfer function and the join operation are monotonic with respect to the partial orders for both scalar and heap variables' def-use chains, thereby ensuring PSSAPT's fixpoint convergence. For the interprocedural case, PSSAPT 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.

**Precision** The main issues for the flow-sensitive pointer analysis is to identify strong updates. PSSA form covers strong updates in 3 cases: 1. $d\phi$ D strongly update variable a (i.e. D's alias set is {a} which is singleton); 2. $c\phi$ C strongly update variable a (for each C's in edge $e_i$, there is a $d\phi$ $D_i$ that strongly updates a)); 3. call site c that presented as a $d\phi$ strongly update variable a (a was strongly updated on all c's target functions see definitions A.1 A.2 in Appendix). PSSAPT handles the strong updates in hvu_conn operation (i.e. **BackTrace** algorithm) that connects heap use U to all of heap defs that may or must alias U by tracing backward through the PSSA def-use chains. If an address-taken variable a is strongly updated at heap def D, a is removed from the U's check set, thus D's dominators that may or must alias U on a will not be connected to U due to the **KILL** on D. Since each visited heap def immediately dominates the heap def visited in previous step, then PSSAPT does not lost any strong updtes and is as precision as other classic flow-sensitive and context-insensitive algorithms.

**Worst-case Complexity**: We discuss both space and time complexity for PSSAPT using the following terms: $\mathcal{H}_\mathcal{D}$: the numbers of $d\phi$s; $\mathcal{H}_\mathcal{U}$: the number of heap uses; $\mathcal{H}_\mathcal{C}$: the number of $c\phi$nodes; $\mathcal{H}_\mathcal{F}$: the number of extended $d\phi$s and heap uses for function invocations; $\mathcal{A}$: the number of address-taken variables; $\mathcal{V}$: the number of scalar variables.

Space complexity is composed of the points-to information and the pointer-flow edges. The worst case memory usage for scalar variables' points-to sets is: $O(|\mathcal{V}||\mathcal{A}|)$. For heap def/uses that are represented by alias sets and points-to sets, only points-to sets need to be considered (their alias sets are represented by the points-to sets of the load/store instructions' address operands ), thus its memory usage is: $O(|\mathcal{H}_\mathcal{D} + \mathcal{H}_\mathcal{U}||\mathcal{A}|)$. Since $c\phi$ and function level heap variables' points-to information is represented as a points-to graph, their worst-case memory usage is: $O(|\mathcal{H}_\mathcal{C} + \mathcal{H}_\mathcal{F}|\mathcal{A}^2)$. The pointer-flow edges memory usage is: $O(|\mathcal{H}_\mathcal{D}||\mathcal{H}_\mathcal{U}|)$. The space complexity is cubic regarding $\mathcal{H}_\mathcal{C}$ and $\mathcal{H}_\mathcal{F}$, and otherwise quadratic.

We use the number of visited scalar/heap variables as the unit for time complexity. PSSAPT's time complexity is composed of the 4 analysis operations. $O_{sv\_prop}$ is: $O(|\mathcal{V}||\mathcal{A}|)$; i.e., the number of scalar variables propagated through scalar SSA def-use edges. $O_{hv\_prop}$ depends on the worst case for pointer-flow edges, which is: $O(|\mathcal{H}_\mathcal{D}||\mathcal{H}_\mathcal{U}||\mathcal{A}|)$. hvu_conn's complexity also depends on pointer-flow edges, so $O_{hvu\_conn}$ is the same as $O_{hv\_prop}$. hvd_conn depends on the number of accessed $d\phi$s and $c\phi$nodes, thus $O_{hvd\_conn}$ is: $O(|\mathcal{H}_\mathcal{D} + \mathcal{H}_\mathcal{C}||\mathcal{A}|)$. Based on these, PSSAPT's complexity is cubic in time.




(a) Data Dependence between PSSAPT 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 PPSSAPT

**Figure 3.**

## 4. Parallel PSSA Points-To Algorithm

The previous section presented a PSSA-based analysis algorithm that produces alias and points-to information for each heap variable and def-use connections between heap variables. In this section, we discuss how to parallelize the analysis presented in the previous section; i.e., we present the parallel version of PSSAPT (**PPSSAPT**). Section 4.1 discusses how the parallelized algorithm will make use of asynchronous task parallelism by processing each worklist element as a parallel task and gives a running example. Section 4.2 provides a detailed description of the **PPSSAPT** algorithm, and Section 4.3 discusses its complexity.

### 4.1 Parallelizing Pointer Analysis

Given our goal of processing worklist elements in parallel, Figure 3 (a) presents the dependences between the shared data structures (annotated as italic font). A dependence holds between a pair of operations (annotated as bold font) if there is a need to read/write or write/write the same data structure concurrently. Figure 3 (b) presents, for each pair of operations, the number of data structure types involved in their dependence. To minimize data dependences, we run the 4 operations separately by separating worklist processing into 4 stages, each corresponding to a type of operation.

**Structure of Worklists**: Based on the 4 execution stages corresponding to the 4 types of operations, here we describe the restructuring of the single worklist into 4 worklists. The interaction between these worklists is shown in Figure 3 (c). The worklists contain the following elements: **SP_WL**: scalar variables and heap uses that have modified points-to sets and need to be propagated to uses; **HUC_WL**: heap uses whose alias sets have been modified and need to be connected to reaching definitions; **HDP_WL**: $d\phi$ and $c\phi$ nodes whose points-to sets have been modified and need to be propagated to heap uses; **HDC_WL**: $d\phi$ and $c\phi$ nodes whose alias sets have been modified and need to be connected to additional heap uses.

In the initial stage PPSSAPT, like PSSAPT, 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), PPSSAPT processes elements in SP_WL and adds each element to one of the 4 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 with their possible defs and puts those heap uses whose points-to sets were modified into SP_WL. Stage 4 (hv_prop) propagates the points-to information from the $d\phi$s to heap uses and adds the heap uses with modified points-to-sets to HDP_WL, as with Stage 3. Figure 3(c) shows the inputs/outputs corresponding to each stage of PPSSAPT (more detail in Section 4.2), which are based on heap variables' alias sets, points-to sets and the 4 worklists discussed above.

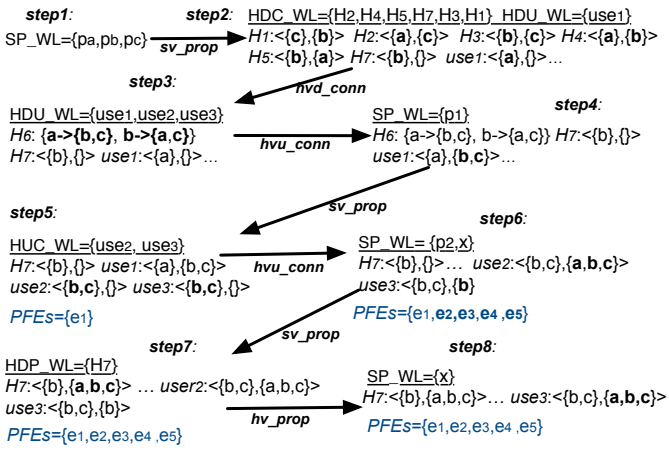



**Figure 4. PPSSAPT** running example

The motivation for dividing **PSSAPT** into 4 stages is to enable task parallelism with minimal dependences. In this way the shared data between elements in the same worklist is minimized. Processing a worklist element is a task that can run in parallel with other such tasks [21]. At every point in the execution of the algorithm, one of the worklists can process multiple elements in parallel. The synchronization between tasks executing in parallel within the same stage is minimized based on this multi-stage worklist process. 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 the shared uses.

**Illustrative Example**: To illustrate the effectiveness of the PPSSAPT algorithm, here we run the example shown in Figure 1 (b)(c) again. Figure 4 demonstrates the progress of the **PPSSAPT** algorithm step by step. In each step, PPSSAPT handles one worklist by processing its elements in parallel. In the first step, the scalar variables $p_a$, $p_b$ and $p_c$ are added to SP_WL since they point to address-taken variables a, b and c. In *Step 2*, their points-to sets are propagated through scalar SSA def-use chains, and heap variables $H_2$, $H_4$, $H_5$, $H_3$, $H_7$, $H_1$ are added to HDC_WL, since they are $d\phi$ s and their alias sets were modified. $use_1$ is added to HUC_WL, since it is a heap use whose alias set was modified. In *Step 3*, PPSSAPT performs hvd_conn and collects all possible heap uses for those $d\phi$s and adds them to HUC_WL, thus $use_2$ and $use_3$ are added to HUC_WL as well. $H_2$, $H_3$, $H_4$ and $H_5$'s pointer information are propagated to $c\phi$ $H_6$.

In *Step 4*, PPSSAPT performs hvu_conn for each heap use to update its points-to sets from its reaching defs and create the pointer-flow edges to connect to its defs. $use_1$'s points-to set gets updated and its lhs $p_1$ is added to SP_WL. The edge $e_1$ is added to connect $H_6$ and $use_1$. $use_2$ and $use_3$ are not processed since their alias sets are still empty. $p_1$'s point-to set is propagated to $use_2$ and $use_3$'s alias set in *Step 5*. As with PSSAPT, PPSSAPT applies hvu_conn to those heap uses in *Step 6*, updating their points-to sets and adding new pointer-flow edges. In *Step 7*, $p_2$'s points-to set is resolved and propagated to $H_7$ and $H_7$ is added to HDP_WL. In *Step 8*, $H_7$'s points-to set is propagated to $use_3$ via pointer-flow edge $e_2$ and $use_3$'s lhs x is added to SP_WL.

Prior to presenting the algorithm in detail, we introduce the parallel constructs and tool functions that are used to implement the parallelism in the PPSSAPT. These constructs are derived from the X10 [29] and include: **async**: spawn a parallel task; **finish**: synchronize all of the tasks spawned within its scope; **isolated**: isolate the access to the objects enclosed in the parameter set. These constructs are also similar to those used in the Cilk++ [7] programming model. We also use **TryLock** for non-blocking lock operations. Some tool functions and operators are used to implement concurrent accesses with lock protection, including functions with the **Safe** prefix and the operator $\uplus =$, which concurrently adds elements to a worklist.

### 4.2 Algorithm Details

Here we present the details of the **PPSSAPT** algorithm by comparing it to **PSSAPT**. As with PSSAPT , the main entry of the algorithm is **MainProc** (Lines 1∼7). Now it spawns tasks for processing the functions taken from the global worklist GlobalWL, until there are no more functions. When a task has been spawned to process a function, the function may already be getting processed by another task, in which case the current task has to postpone. To avoid blocking, the PPSSAPT algorithm returns such a temporarily unavailable function to GlobalWL, and **MainProc** will spawn a new task for processing it later. This delayed process mechanism avoids the potential deadlocks resulting from recursive calls or the strongly connected components of the call graph.

Function **PPSSAPT**(Lines 9∼35) is the main driver for the points-to analysis (i.e. applying the 4-stage analysis introduced in Section 4.1) for a given input function. First PPSSAPT checks if it can hold ownership of the input function with **TryLock** (Line 10). If not, it adds the function to GlobalWL. If it can hold ownership, it invokes **InitWorkList** (Line 13) to initialize the worklists and iteratively performs the 4-stage points-to analysis (Lines 15∼31). The **InitWorkList** function adds all address-taken variables to SP_WL worklist, and the $d\phi$s to the HDP_WL worklist. If some worklist is not empty then PPSSAPT's main loop iteratively processes the 4 worklists as described earlier. When all 4 worklists are empty (Line 15), PPSSAPT updates callers (Lines 32,33) if there is a side effect (i.e. some address-taken variables' points-to information was changed) in the current function. Then it releases ownership by **unlock** (Line 34). This approach avoids the potential infinite loop resulting from recursive calls or strongly connected components in the call graph. In each stage, **PPSSAPT** spawns parallel tasks to process the worklist elements in parallel (i.e. **async**) and all tasks spawned from the same worklist will be synchronized (i.e. **finish**) before starting the next stage.

For Stage 1 (Line 16∼19), **PropToUse** handles element v from SP_WL. It creates 4 temporary worklists (Line 38) that are used to collect new elements and merge with the 4 worklists in **PSSAPT** (Line 19). As with the sequential version introduced in Section 3.4, **PropToUse** processes the function invocation case (Lines 39∼41). For the scalar variable case, the only difference is in considering which worklists are updated. The copy case needs to consider all 4 worklists since it invokes **PropToUse** for v's uses (Line 46). The merge $\phi$'s copy case adds the assignee vi to sp_wl (Line 49,50) if its points-to set were changed, and the merge of points-to sets needs to be protected by mutual exclusion (Line 48). The case for v is a

```
1  function MainProc ()
   Input  : F : root function
   Output:
2  GlobalWL := ∅; GlobalWL ∪ = {F};
3  while GlobalWL ≠ ∅ do
4      finish
5          while GlobalWL ≠ ∅ do
6              func := SafePopFront (GlobalWL);
7              async PPSSAPT (func, GlobalWL);
8
9  function PPSSAPT ()
   Input  : F : input function, GlobalWL : global worklist
   Output: boolean flag for side-effect
10 if TryLock (F) ≠ LOCKED then
11     GlobalWL ⊎ = {F};
12     return FALSE;
13 if ! InitWorkList (F) then
14     return FALSE;
15 while SP_WL ≠ ∅ and HUC_WL ≠ ∅ and HDP_WL ≠ ∅ and HDC_WL ≠ ∅
   do
16     finish
17         while SP_WL ≠ ∅ do
18             v := SafePopFront (SP_WL);
19             async {SP_WL, HDP_WL, HDC_WL, HUC_WL}
               ⊎ =PropToUse (v, GlobalWL);
20     finish
21         while HDC_WL ≠ ∅ do
22             v := SafePopFront (HDC_WL); as := AS (v);
23             async HUC_WL ⊎ = CollectUses (v, as);
24     finish
25         while HUC_WL ≠ ∅ do
26             v := SafePopFront (HUC_WL);
27             async SP_WL ⊎ = BackTrace (v);
28     finish
29         while HDP_WL ≠ ∅ do
30             v := SafePopFront (HDP_WL);
31             async SP_WL ⊎ = PropDPhi (v);
32 callers := UpdateCallers (F);
33 GlobalWL ⊎ = callers;
34 unlock (F);
35 return callers ≠ ∅;
36
37 function PropToUse ()
   Input  : v : variable or instruction, GlobalWL : global worklist
   Output: sp_wl, hdp_wl, hdc_wl, huc_wl : worklist
38 sp_wl := ∅; hdp_wl := ∅; hdc_wl := ∅; huc_wl := ∅;
39 if IsCallSite (v) then
40     callees := GetCallTargets (v);
41     GlobalWL ⊎ = callees;
42 else
43     foreach vi in Use (v) do
44         if IsCopy (vi) then
45             PTS (vi) := PTS (v); //vi = v
46             {sp_wl, hdp_wl, hdc_wl, huc_wl} ⊎ = IsModified (PTS
               (vi))? PropToUse (vi) : ∅;
47         else if IsScalarCPhi (vi) then
48             isolated PTS (vi)
49                 PTS (vi) ∪ = PTS (v); //vi = φ(v, ...)
50             sp_wl ∪ = IsModified (PTS (vi))? {vi} : ∅;
51         else if IsDPhiValueOp (vi, v) then
52             hv := HeapVar (vi); PTS (hv) := PTS (v); //* p = vi
53             hdp_wl ∪ = IsModified (PTS (hv))? {hv} : ∅;
54         else if IsDPhiPointerOp (vi, v) then
55             hv := HeapVar (vi); AS (hv) := PTS (v); //vi: *v=...
56             hdc_wl ∪ = IsModified (PTS (vi))? {hv} : ∅;
57         else if IsHeapUsePointerOp (vi, v) then
58             hv := HeapVar (vi); AS (hv) := PTS (v); //vi: ...=*v;
59             huc_wl ∪ = IsModified (PTS (vi))? {hv} : ∅;
60 return sp_wl, hdp_wl, hdc_wl, huc_wl;
```

store instruction's value operand adds the result to `hdp_wl`(Lines 51∼53). For the case where `v` impacts the heap variable's alias set (i.e. `v` is the pointer operation for store or load instruction), `vi` (see the description in Section 3.4) is added to `hdc_wl` for the $d\phi$ case(Lines 54∼56) and `huc_wl` for the heap use case (Lines 57∼59).

For Stage 2 (**PPSSAPT**, Lines 20∼23), function **CollectUses** collects all heap uses that may be defined by the $d\phi$s in `HDC_WL`. Compared with **PSSAPT**, the only modification here is to protect the concurrently updated points-to graph by using **SafeUpdagtePTG** to replace **UpdatePTG** (see Line 59 in Section 3.4).

The **BackTrace** function handles heap uses from `HUC_WL` in Stage 3 (**PPSSAPT** Lines 24∼27). As discussed in previous sections, there is no data dependence in this process, so this function is same as the version used in **PSSAPT**.

```
1  function PropDPhi ()
   Input  : D: dφ
   Output: sp_wl : worklist
2  sp_wl := ∅;
3  foreach edge e in D do
4      U := dest (e); //U is a heap use or cφ
5      if U is heap use then
6          isolated PTS (U)
7              PTS (U) ∪ = PTS (D);
8          sp_wl ∪ = IsModified (PTS (U))? {U} : ∅;
9      else if U is cφ then
10         changed := SafeUpdatePTG (AS (D), PTS (D), PTG (U));
11         if changed then
12             PropCPhi (U, sp_wl);
13 return sp_wl;
14
15 function PropCPhi ()
   Input  : C: cφ, sp_wl : worklist
   Output:
16 mods := GetModifiedKey (PTG (C));
17 foreach edge e in C do
       //U is a heap use or cφ
18     U := dest (e); Δ := mods ⋂ AS (U);
19     if U is heap use then
20         foreach k in Δ do
21             isolated PTG (C)
22                 PTS (U) := GetPTS (PTG (C), k);
23             sp_wl ∪ = IsModified (PTS (U))? {U} : ∅;
24     else if U is cφ then
25         hasChanged := false;
26         foreach k in Δ do
27             isolated PTG (C)
28                 pts := GetPTS (PTG (C), k);
29                 hasChanged |= UpdatePTG (k, pts, PTG (U));
30         if hasChanged then
31             PropCPhi (U);
```

After connecting heap variables' defs and uses by pointer-flow edges, the **PropDPhi** function propagates heap defs' points-to information to uses in Stage 4 (**PPSSAPT** Lines 28∼31). The **isolated** constructs are employed to protect the points-to sets from concurrent updates (Line 6), since the heap uses' points-to sets can be accessed with multiple defs. The updating of $c\phi$'s local points-to graph is protected by **SafeUpdatePTG** (Line 8 in **PropDPhi**)), which avoids concurrent updates with **PropCPhi**.

### 4.3 Discussion

PPSSAPT dose not change the space complexity, here we discuss the time complexity for the given parallel process $\mathcal{N}$. As

| Benchmark | Desription | LOC | # of Funcs | # of insts | # of scalar vars |
|---|---|---|---|---|---|
| GhostScript | GNU Interpertor for PS&PDF | 282.7k | 6,408 | 433,427 | 164,543 |
| Vim | Vim text editor | 310.8k | 3,793 | 350,326 | 95,271 |
| V8 | Google V8 JS engine | 332.3k | 19683 | 583914 | 291964 |
| GCC | GNU GCC compiler | 382.9k | 5,577 | 587,672 | 208,656 |
| Ruby | Ruby compiler & runtime | 638.8k | 5,366 | 87,146 | 301,732 |
| Tizen | TizenOS | 2,205k | 91,839 | 3,771,928 | 1,404,564 |

(a) Benchmarks' Features and Basic Statistics

| Benchmark | # of uses | # of dϕs | # of allocs | # of cϕs | # of calls | PPSSAPT (MB) | LLVM (MB) |
|---|---|---|---|---|---|---|---|
| GhostScript | 56,438 | 22,293 | 3,902 | 23,397 | 22,780 | 548 | 116 |
| Vim | 57,800 | 16,917 | 4,241 | 30,269 | 27,583 | 499 | 94 |
| V8 | 59138 | 34946 | 38204 | 25285 | 99241 | 1249.8 | 167.6 |
| GCC | 125,595 | 20,158 | 1,599 | 31,765 | 52,249 | 671 | 148 |
| Ruby | 40,365 | 10,395 | 15,609 | 19,402 | 28,339 | 467 | 82 |
| Tizen | 461,844 | 214,156 | 698,008 | 254,666 | 109,483 | 10,823,784 | 6,438,592 |

(b) Benchmarks' PSSA Statistics and Memory Usages

| | PPSSAPT # of SUDs | SUPT # of SUDs | PSSAPT (secs) | PPSSAPT 12 threads (secs) | SUPT (secs) | PPSSAPT # of IICs | SUPT # of IICs |
|---|---|---|---|---|---|---|---|
| GhostScript | 6,390 | 5,254 | 1.98 | 0.69667 | 1.73 | 57 | 12 |
| Vim | 5,298 | 4,391 | 6.01 | 1.005 | 5.96 | 53 | 32 |
| V8 | 26,616 | 17,794 | 402.59 | 118.354 | 418.7 | 47 | 31 |
| gcc | 5,617 | 2,507 | 50.99 | 14.5527 | 32.03 | 45 | 18 |
| Ruby | 5,492 | 2,297 | 2.52 | 0.962791 | 2.44 | 43 | 21 |
| Tizen | 108,696 | 8,820 | 642.4 | 193.358 | 601.5 | 533 | 314 |

(c) The comparison with SUPT [Lhotak and Chung 2011]

**Figure 5.**

discussed in Section 4.1, PPSSAPT runs the 4 operations (introduced in Section 3.3) in parallel in 4 different stages, thus PPSSAPT's time complexity is the summary of those 4 stages. For `sv_prop`, the synchronization happens on concurrently accessed scalar $c\phi$ nodes, so $O_{sv\_prop}$ is: $O(|\frac{\mathcal{V}-\mathcal{V}_\mathcal{C}}{\mathcal{N}} + \mathcal{V}_\mathcal{C}||\mathcal{A}|)$. Here $\mathcal{V}_\mathcal{C}$ stands for the number of scalar merge $\phi$s. $O_{hv\_prop}$ depends on the number of concurrently updated heap uses. In the worst case, it is same as **PSSAPT**'s. `hvu_conn` can be fully parallelized, so we get $O(\frac{|\mathcal{H}_\mathcal{D}||\mathcal{H}_\mathcal{U}||\mathcal{A}|}{\mathcal{N}})$. $O_{hvd\_conn}$ depends on the concurently accessed $\mathcal{H}_\mathcal{C}$, thus it is: $O(|\frac{\mathcal{H}_\mathcal{D}}{\mathcal{N}} + \mathcal{H}_\mathcal{C}||\mathcal{A}|)$.

# 5. Evaluation

## 5.1 Implementation

The **PSSAPT** and **PPSSAPT** versions of the algorithm were both implemented as an analysis pass in the LLVM [17] version 3.6.2 compiler framework. They take LLVM bitcode as input and generate in-memory IR, invoking the PSSA builder to produce the PSSA form. **PSSAPT** or **PPSSAPT** can then be applied. For parallelism, we used the Habanero-C library(HCLib) [25], which is a lightweight task parallelism library. HCLib supports task spawning/synchronization APIs(i.e. **async**, **finish**) and work stealing. We used the compare-and-swap API to implement lightweight spin locks and **isolated** support.

Since our pointer analysis focus on flow-sensitivity, the **singleton** is used to identify the strong update for a given heap variable. Regarding the infrastructure(i.e. LLVM compiler and its intermediate representation) we used, the definition of singleton is a single element set whose element is: a global variable; a local variable that is not allocated in the loop or recursive call path, and used outside of loop or recursive call path; or a dynamically allocated variable that is not allocated in the loop and recursive call path. Similar to [15], our pointer analysis starts from an estimated call graph, that is, the invoked function pointer (i.e. function pointer invocation in LLVM) is conservatively assumed that it could point to any procedure whose address has been taken.

As clarified in Section 1, our analysis is field-sensitivity. It handles the offset calculation (i.e. LLVMs GEP instruction) for struct, class and array pointers when it handles load and store instructions. If the offset can be identified as a constant integer value, then it is encoded as an unique address for load and store operations. If the offset is still a variable, then it is encoded as the 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 six benchmarks from real world C/C++ applications. Figure 5(a) provides details for the benchmarks, including the number of lines of source code, the number of functions and LLVM instructions, and the number of scalar variables (**scalar vars**). Figure 5(b) shows the PSSA related statistics, including the number of heap uses (**uses**), $d\phi$s, $c\phi$s, call sites, and the count of allocation sites (**allocs**) includes LLVM **alloca** instructions, global variables, memory allocation intrinsics and memory-related APIs.

To evaluate the pointer analysis, all C/C++ source code was compiled into LLVM bitcode via the Clang frontend. We apply the LLVM optimization **mem2reg** (performs scalar replacement and $\phi$ creation for scalar variables) and scalar optimizations to bitcode. We also transformed *do-while* loops to *while* loops to avoid incorrect strong update checking for allocation sites located in *do-while* loops. 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 efficiency of the **PPSSAPT** analysis with respect to scalability and memory usage. We also evaluated our algorithm's precision by comparing its effectiveness with respect to identifying strong updates in store operations and inlinable indirect call sites with SUPT (the algorithm from [15] [2]).

To evaluate the memory usage, the two rightmost columns in Figure 5(b) (last two columns) show the memory usage for running the **PPSSAPT** analysis and its corresponding memory usage for LLVM initialization(i.e. parsing bitcode, applying **mem2reg**, scalar optimizations and building PSSA).

The 2nd and 3rd columns in Figure 5 (c) show, in comparison to SUPT, how many more strong updates in store operations (SUDs) **PPSSAPT** identifies. **PPSSAPT** identified more strong updates than SUPT due to the fully flow-sensitivity and field-sensitivity that can identify the strct/class fields and array elements with constant integer indices. Figure 5(c) also compares the execution times of PSSAPT, 12-threads PPSSAPT and SUPT (the 4rd, 5th and 6th columns). Regarding the fully flow-sensitivity, the sequential version of PSSAPT ran slower than SUPT on most of benchmarks. By leveraging the benefit of parallelism, PPSSAPT (running in 12-threads) won SUPT.

For scalability, we evaluated PPSSAPT on the 12-core Xeon SMP system. Figure 6 shows the relevant speedups from one thread to 12 threads and PSSAPT (denoted by `seq`) among those six benchmarks, where running PPSSAPT on a single thread provides the baseline. 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*×.

To demonstrate and evaluate PPSSAPT, we also developed an application, an inlinable indirect calls (IIC) analyzer, which determines whether an indirect call site (i.e. function pointer invocation) has only a single target and so can be inlined. The 7th and 8th columns in Figure 5 (c) show, in comparison to SUPT, how many more inlinable call sites (IICs) **PPSSAPT** identifies. There are mainly from the virtual function calls, which is, the function

[2] To make the comparison, we ported the [15] implementation from LLVM 2.6 to 3.6.2.

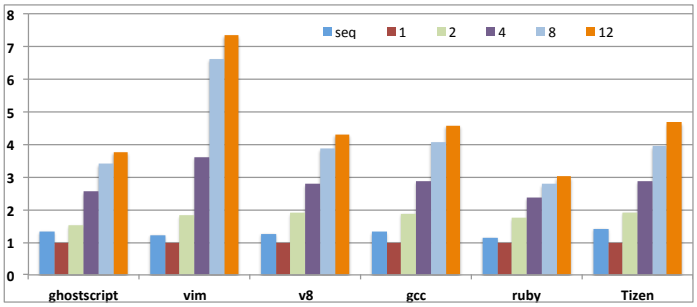



**Figure 6.** The scalability of **PPSSAPT**

pointer is loaded from vtable (presented as an array in LLVM IR). Since PPSSAPT analysis can identify the array elements with constant indices, the vtable elements that has single target can be identified.

## 6. Related Work

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

Hardekopf and Lin present Semi-sparse [10]: 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 per program point points-to graph for all address-taken variables. Operations on address-taken variables are labeled and connected by the SEG graph representation [24]. For memory efficiency, they use binary decision diagrams for maintaining points-to graphs. Because the analysis is not sparse for address-taken variables, it is not well-suited for parallelization. In later work [11], 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 Semi-sparse analysis.

Lhotak and Chung's SUPT [15] performs a SSA-based points-to analysis and also 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 show that strong update stores are the most frequent case in 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 in a manner similar to ours, making it amenable to task parallelism. But because their algorithm is partly flow-insensitive, it has larger points-to sets, which brings more data sharing than our case.

Li et.al [16] introduced a value-flow based point-to analysis that iteratively examines the set of pointers that point to the allocation sites by checking graph reachibility, 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 point-to set for each variables. Since their indirect flow calculation needs to resolve dataflow equation sequentially, this algorithm is also difficult to be parallelized due to the data sharing.

PPSSAPT balances precision and performance in a novel way. In PSSAPT, the dataflow traversal process is divided into multiple subtasks (i.e. per operand based), and thus can be easily parallelized by using lightweight task parallelism with proper synchronization. PPSSAPT maintains a global points-to graph for scalar variables in the same manner as SUPT and Semi-sparse. 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 Semi-sparse. Compared with [11], PPSSAPT brings more precision since the auxiliary phase of that approach is flow-insensitive, producing spurious def-use chains for address-taken variables.

**Sparse Pointer Analysis:** Tok et al. present a SSA-based sparse interprocedural flow-sensitive pointer analysis [27]. They too integrate points-to analysis with an on the fly construction of heap def-use chains that minimizes re-analysis. They differ from us in the use of interprocedural def-use chains and in maintaining a worklist of basic blocks per procedure. They identify potentially impacted basic blocks based on the updating of points-to information.

**Heap SSA-based Pointer Analysis:** Prabhu and Shankar [22] present a Heap SSA-based pointer analysis which targets Java. Their algorithm heavily leverages strongly-typed semantics to identify may/must equality relations and enable field sensitivity.

**Parallel Pointer Analysis:** Mendez-Lojo et al. introduce a parallel inclusion-based points-to analysis [19]. This algorithm is based on graph rewriting, defining a set of rules for constraint solving. It protects the variables that are accessed concurrently. Unlike our work, their algorithm is flow-insensitive, which enables them to represent the variables' points-to relations as a matrix. Their later work [18] parallelized this algorithm and applied it to SIMD architectures.

Nagaraj and Govindarajan furthered the development of the graph rewriting approach by extending the rewriting rules in [19] to support flow sensitivity, thereby obtaining a parallel flow-sensitive pointer analysis. They follow [11]'s approach in using a multi-stage pre-analysis, based on an auxiliary flow-insensitive analysis, that conservatively builds def-use chains for address-taken variables. The primary differences between our work and theirs is the flow-insensitive aspect of their work leads to less precision than PPSSAPT when adding the def-use connnection between address-taken variables. The redundant def-use added by the flow-insensitive pointer analysis brings more synchronization overhead for protecting the shared data structures (i.e. the alias sets and point-to sets for heap variables). Their experimental results showed many cases of negative scalability, i.e., of the execution time becoming larger as more threads are used, whereas PPSSAPT 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 approach to points-to analysis based on pointer SSA form **PSSA**, an extension of Array SSA form. In contrast to classic constraint-solving based approaches, our approach can decompose the analysis into fine units of granularity for fine-grained task parallelism. Our analysis also produces precise def-use connections (pointer-flow edges) between memory stores and loads. We implemented this Parallel Pointer SSA-based Pointer Analysis in the LLVM compilation framework by leveraging the fine-grain parallelism, capacity from HCLib. We evaluated this analysis with 6 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 PSSA form only uses a single heap array. There are multiple directions for future work. First, our analysis

should be extendable to analysis of binary programs. Second, we can explore the use of multiple heap arrays to leverage type information when available, including for strongly-typed programs. Finally, we can extend the implementation of our algorithms to leverage many-core/accelerator parallelism and distributed-memory parallelism for further scalability.

## References


[1] Peng-Sheng Chen et al. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, NY, USA, 2003. ACM.

[2] Jong-Deok Choi, Michael G. Burke, and Paul R. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *The 29 Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,*, pages 232–245, 1993.

[3] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, New York, NY, USA, 1991. ACM.

[4] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM.

[5] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, New York, NY, USA, 2006. ACM.

[6] Stephen J. Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *Proceedings of the 7th International Symposium on Static Analysis*, SAS '00, pages 155–174, London, UK, UK, 2000. Springer-Verlag.

[7] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.

[8] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 121–133, New York, NY, USA, 1998. ACM.

[9] Samuel Z. Guyer and Calvin Lin. Error checking with client-driven pointer analysis. *Sci. Comput. Program.*, 58(1-2), October 2005.

[10] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 226–238, New York, NY, USA, 2009. ACM.

[11] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, Washington, DC, USA, 2011.

[12] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, July 1999.

[13] Kathleen Knobe and Vivek Sarkar. Array ssa form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 107–120, New York, NY, USA, 1998. ACM.

[14] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM.

[15] Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 3–16, New York, NY, USA, 2011. ACM.

[16] Lian Li, Cristina Cifuentes, and Nathan Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 343–353, New York, NY, USA, 2011. ACM.

[17] Llvm compiler infrastructure. http://llvm.org.

[18] Mario Méndez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 107–116, 2012.

[19] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, NY, USA, 2010. ACM.

[20] Vaivaswatha Nagaraj and R. Govindarajan. Parallel flow-sensitive pointer analysis by graph-rewriting. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, Piscataway, NJ, USA, 2013. IEEE Press.

[21] Keshav Pingali et al. The tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[22] Prakash Prabhu and Priti Shankar. Field flow sensitive pointer and escape analysis for java using heap array ssa. In *Proceedings of the 15th International Symposium on Static Analysis*, SAS '08, pages 110–127, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[24] G. Ramalingam. On sparse evaluation representations. *Theor. Comput. Sci.*, 277(1-2):119–147, April 2002.

[25] Rice hc library. https://github.com/habanero-rice/hclib.

[26] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 12–23, New York, NY, USA, 2001. ACM.

[27] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *The 15th International Conference on Compiler Construction,*, pages 17–31, 2006.

[28] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 1–12, New York, NY, USA, 1995. ACM.

[29] X10 programming language web site. http://x10.codehaus.org/, January 2010.

[30] Jianwen Zhu. Towards scalable flow and context sensitive pointer analysis. In *Proceedings of the 42Nd Annual Design Automation Conference*, DAC '05, New York, NY, USA, 2005. ACM.


## A. Appendix

**Definition A.1.** A functions **F** strongly updates address-taken variable **a** $\iff$ **a** was strongly updated on all pathes that dominate **F**'s returns.

**Definition A.2.** A call site **c** strongly updates heap variable **a** $\iff$ **a** was strongly updated on all **c**'s target functions.

**Definition A.3.** A $d\phi$ **D** is **undef** $\iff$ **D**'s alias set is $\emptyset$.

**Definition A.4.** A $c\phi$ **C** is **undef** $\iff$ $\exists$ $d\phi$ **D** whose dominate froniter is **C** and **D** is **undef**.

**Definition A.5.** A function **F** is **undef** $\iff$ $\exists$ $d\phi$ **D** that locates in **F**.