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 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 inlined 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
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 (PSSAPT) 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 \( p_a, p_b, p_c, p_1, p_2, \) 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.

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
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 def’s 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 cφ 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. do or cφ) (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 def’s 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 PropDFphi).

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 cφ).

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 defs and heap uses, we use the format: Hi:< alias set, points-to set>. For cφs 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
10, so H2, H4, and use1 are added to WL; c is added to PTS(H2) and b is added to PTS(H4); a is added to AS(use1). The uses of p1 and p2 are processed in the same manner. The three scalar variables are removed from the worklist.

In Step 3, hvd_conn is applied to the dφs Hr, H3 and H1. Their points-to information is propagated to φ Hφ; their impacted heap uses, use2 and use3, are added to WL. Note that H3 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 use1. Elements of the points-to set of heap def H6 are added to the points-to set of use1. The pointer-flow edge e1 is added to connect H6 to use1. The lhs scalar variable p1 for the heap use is added to WL and p1’s points-to set {b, c} is resolved.

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 performs 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 dφ 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 v 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 v's points-to set. If there is a modification, then v is passed to PropToUses (Line 13, Case 34) or added to wal (merge φ case, Line 37). If v is a store instruction’s value operand (Lines 38~40)(i.e. v’s point-to set represents v’s corresponding heap variable hv’s point-to set), then merge its points-to set with hv’s points-to set and add hv to wal 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 point-to set represents the alias set of a heap variable (Lines 41~43), then merge v’s points-to set with v’s heap variable’s alias set and add hv to wal if there is a modification.

The function CollectUses, for Operation 2 (hvprop), of PSSAPT algorithm, is invoked at Line 16 if the alias set of v (i.e. a dφ ) has been modified. For the given input dφ 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 frontier C, otherwise un-register with C; if D is strong update and dominates current function F’s return path then register D with F, otherwise un-registers 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 dφ 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 (hv_conn)’s connecting of a heap use to def that reach it (PSSAPT Lines 19,20). This function takes a copy of a heap use’s alias set as a check set and traces backward through the PSSA def-use chains until there is no more heap def (Line 3). For each heap def (dφ or φ), 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
function MainTrace() 
Input: F: root function 
Output: 
while GlobalWL ≠ ∅ do 
    Input := PropToUses (v, GlobalWL); 
    GlobalWL := CollectUses (v, GlobalWL); 
    if InitializeWorklist (F) then return FALSE; 
while WL ≠ ∅ do 
v := PopFront (WL); 
if v is scalar then WL := PropToUses (v, GlobalWL); 
else if v is dphi then if IsModified (AS (v)) then WL := CollectUses (v, AS (v)); 
else WL := PropDphi (v); 
else if v is heap use then WL := BackTrace (v); 
callees := UpdateCallers (F); 
GlobalWL := callees; 
return callees ≠ ∅; 
function PropToUses () 
Input: v: variable or instruction, GlobalWL: global worklist 
Output: wl: worklist 
w := ∅; 
if IsCallSite (v) then callees := GetCallTarget (v); 
GlobalWL := callees; 
else foreach vi in Use (v) do if IsCopy (vi) then 
                    PTS (v) := PTS (vi); // vi := v 
                    wl := IsModified (PTS (vi)) ? PropToUses (vi); ∅; 
else if IsScalarCphi (vi) then 
                    PTS (v) := PTS (vi); // vi := φ (v, ...) 
                    if IsModified (PTS (vi)) ? (vi); ∅; 
else if IsDphiValueOp (vi) then 
                    hv := HeapVar (vi); PTS (hv) := PTS (vi); // vi := *p=v 
                    if IsModified (PTS (hv)) ? PropDphi (hv); ∅; 
else if IsPointerOp (vi) then 
                    hv := HeapVar (vi); AS (hv) := PTS (vi); // vi := v... or vi := ...=v 
                    if IsModified (PTS (hv)) ? {hv}; ∅; 
return wl; 
function CollectUses () 
Input: D: def or cdf, as: as the input alias set 
Output: wl: worklist 
w := ∅; as := AS (D); C := DomFrontier (D); 
CheckRegisterSU (D, C, as); 
if C then n := ImmPostDom (D); 
while n ≠ C do if AS (n) ≠ undef then 
                    wl := D; return wl; 
if IsSU (n, as) then 
                    as := as \ AS (n); 
if as = ∅ then return wl; 
n := ImmPostDom (n); 
changed := UpdateTGT (as, PTS (D), PTG (C)); 
if changed then 
                    CollectUses (C, as); 
foreach U in DomBy (D) do 
    if AS (U) ≠ ∅ then 
                    wl ∪ U; 
return wl; 
function BackTrace () 
Input: U: heap use 
Output: wl: worklist 
w := ∅; 
if IsCallSite (v) then callees := GetCallTarget (v); 
GlobalWL := callees; 
else foreach e in D do if U is a heap use or cφ then 
                    if U is heap use then 
                        PTS (U) := PTS (D); 
                        wl := IsModified (PTS (U)); ∅; 
else if U is cdf then 
                        changed := UpdateTGT (AS (D), PTS (D), PTG (U)); 
                        if changed then 
                            PropCphi (U, wl); 
                    return wl; 
function PropCphi () 
Input: C: cdf, as: worklist 
Output: wl: worklist 
mods := GetModifiedKey (PTG (C)); 
foreach edge e in C do if U := dest (e); ∆ := mods \ AS (U); 
if U is heap use then 
                    PTS (U) := GetPTS (PTG (C), k); 
                    if IsSU (D, ∆) then 
                        as := as \ ∆; 
n := D; 
if modal then 
                    v := LHS (U); PTS (v) := PTS (U); 
                    wl := v; 
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 ∆ (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 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 def’s alias set is modified, the relevant heap uses’ trace process will be triggered again.

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 doφ's points-to sets via pointer-flow edges. For a φ, 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 dφ 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 dφ 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 dφ(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 dφ 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, PSSA-APT 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 dφs 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. dφ D strongly update variable a (i.e. D’s alias set is {a} which is singleton); 2. cφ C strongly update variable a (for each C’s in edge et, there is a dφ D that strongly updates a); 3. call site c that presented as a dφ strongly update variable a (a was strongly updated on all c’s target functions see definitions A1.1A2 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 updates 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: HD_c: the numbers of dφ’s; H_c: the number of heap uses; H_c: the number of cφ nodes; H_c: the number of extended dφs and heap uses for function invocations; A: the number of address-taken variables; 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(|V||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(H_c + H_d(|A|)). Since cφ and function level heap variables’ points-to information is represented as a points-to graph, their worst-case memory usage is: O(H_c + H_f)|A|^2). The pointer-flow edges memory usage is: O(H_d + H_d)|A|^2). The space complexity is cubic regarding H_c and H_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_hv_prop is: O(|V||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(|H_c| + H_d(|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 dφs and cφ nodes, thus O_hvd_conn: O(|H_d + H_d|(|A|)). Based on these, PSSAPT’s complexity is cubic in time.

### 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: SW: 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; HDC_WL: dφ and cφ nodes whose points-to sets have been modified and need to be propagated to heap uses; HDP_WL: dφ and cφ 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 \( \text{SP}_WL \). In Stage 1 (\( \text{sv}_{\text{prop}} \)), PSSAPT processes elements in \( \text{SP}_WL \) and adds each element to one of the 4 worklists. In Stage 2 (\( \text{hv}_{\text{conn}} \)), the elements of \( \text{HC}_WL \) are processed and the collected heap uses are added to \( \text{HUC}_WL \). Stage 3 (\( \text{hvu}_{\text{conn}} \)) connects heap uses with their possible defs and puts those heap uses whose points-to sets were modified into \( \text{SP}_WL \). Stage 4 (\( \text{hv}_{\text{prop}} \)) propagates the points-to information from the \( \text{dofs} \) to heap uses and adds the heap uses with modified points-to-sets to \( \text{HDP}_WL \), 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. \( \text{sv}_{\text{prop}} \) is data independent (excluding the \( \text{HDP}_WL \) of the previous stage), and collects all possible heap uses for those \( \text{hvu}_{\text{conn}} \). In the first step, the scalar \( \text{SP}_WL \)'s points-to graph, their points-to sets are updated and its lhs \( p_1 \) is added to \( \text{SP}_WL \). The edge \( e_1 \) is added to connect \( H_0 \) and \( \text{use}_1 \). \( \text{use}_2 \) and \( \text{use}_3 \) are not processed since their alias sets are still empty. \( p_1 \)'s point-to set is propagated to \( \text{use}_2 \) and \( \text{use}_1 \)'s alias set in Step 5. As with PSSAPT, PPSSAPT applies \( \text{hv}_{\text{conn}} \) 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 \( \text{HDP}_WL \). In Step 8, \( H_7 \)'s points-to set is propagated to \( \text{use}_3 \) via pointer-flow edge \( e_2 \) and \( \text{use}_3 \)'s lhs \( x \) is added to \( \text{SP}_WL \).

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 \( \parallel \), 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 15~41). The InitWorkList function adds all address-taken variables to the SP_WL worklist, and then adds 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 \( \text{SP}_WL \). 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 \( \phi \) to \( \text{SP}_WL \) (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
function MainProc()  
Input : F: root function
Output:
GlobalWL := ∅; GlobalWL ⊇ {F};
while GlobalWL ⊇ {} do
  while GlobalWL ⊇ {} do
    func := SafePopFront (GlobalWL);
    async FFSSAPT (func, GlobalWL);
  end
  return FALSE;
end

function FFSSAPT()  
Input : F: input function, GlobalWL: global worklist
Output: boolean flag for side-effect
if TryLock (F) ≠ LOCKED then
  GlobalWL ⊇ {F};
  return FALSE;
end
if InitWorkList (F) then
  return FALSE;
end
while SP_WL ⊇ {} and HUC_WL ⊇ {} and HDP_WL ⊇ {} and HDC_WL ⊇ {} do
  while SP_WL ⊇ {} do
    v := SafePopFront (SP_WL);
    async {SP_WL, HDP_WL, HDC_WL, HUC_WL} AS := PropToUse (v, GlobalWL);
  end
  while HUC_WL ⊇ {} do
    v := SafePopFront (HUC_WL);
    async SP_WL := BackTrace (v);
  end
  while HDP_WL ⊇ {} do
    v := SafePopFront (HDP_WL);
    async SP_WL := PropDPhi (v);
  end
  callers := UpdateCallers (F);
  GlobalWL := callers;
  unlock (F);
  return callers ≠ ∅;
end

function PropToUse()  
Input : v: variable or instruction, GlobalWL : global worklist
Output: sp_WL: worklist
if sp_WL = {} then
  return sp_WL;
else
  foreach edge e in D do
    if U := dest (e); /U is a heap use or c̃φ
      PSSAPT (U) := GetPTS (C); k := GetPTG (C);
      foreach k in ∆ do
        if k is heap use then
          U := dest (e); ∆ := mods ∩ AS (U);
          SP_WL := GetSP (U);
        else
          hasChanged := false;
          foreach k in ∆ do
            isolated PSSAPT (C)
            sp_WL := IsModified (PTS (U))? {U} : ∅;
            if U is c̃φ then
              changed := SafeUpdatePTG (AS (D), PTS (D), PTG (U));
              if changed then
                PropCPhi (U, sp_WL);
              return sp_wl,
            function PropCPhi ()
Input : C: c̃φ, sp_wl: worklist
Output:
mods := GetModifiedKey (PTG (C));
foreach edge e in C do
  if U := dest (e); AS := mods ∩ AS (U);
  sp_wl := GetSP (U);
  if U is heap use then
    foreach k in ∆ do
      isolated PSSAPT (C)
      sp_wl := GetPTS (PTG (C), k);
      if hasChanged then
        hasChanged := UpdatePTG (k, pts, PTG (U));
      return PropCPhi (U);
end

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 c̃φ’s local points-to graph is protected by SafeUpdatePTG (Line 8 in PropDPhi), which avoids concurrent updates with PropCPhi.

4.3 Discussion
PPSSAPT does not change the space complexity, here we discuss the time complexity for the given parallel process X. As
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 *cφ* nodes, so \( O_{sv.prop} = \Theta(\frac{1}{N} |V| + |C| \cdot |A|) \). Here |V| stands for the number of scalar merge *cφs*. \( O_{hv_conn} \) depends on the number of concurrently updated heap nodes. In the worst case, it is same as PPSSAPT's *hv.conn* can be fully parallelized, so we get \( \Theta(\frac{1}{N} |H| + |C| \cdot |A|) \). \( O_{hv_conn} \) depends on the concurrently accessed *HC*, thus it is: \( \Theta(\frac{1}{N} |H| + |H| \cdot |C| \cdot |A|) \).

5. Evaluation

5.1 Implementation

The PPSSAPT 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. PPSSAPT 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), *dφs*, *cφs*, 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 front-end. We apply the LLVM optimization *mem2reg* (performs scalar replacement and *φ* 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 26-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])

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 struct/class fields and array elements with constant integer indices. Figure 5(c) also compares the execution times of PPSSAPT, 12-threads PPSSAPT and SUPT (the 4th, 5th and 6th columns). Regarding the fully flow-sensitivity, the sequential version of PPSSAPT 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 PPSSAPT (denoted by *seg*) 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 \( 7.35 \times \) running on 12 threads. The average speedup for the 12 threads case is \( 4.45 \times \) and the geometric mean is \( 4.62 \times \).

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.
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 of precision and performance in a novel way. In PPSSAPT, 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 connection 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


A. Appendix

Definition A.1. A functions F strongly updates address-taken variable a ⇐⇒ a was strongly updated on all paths that dominate F’s returns.

Definition A.2. A call site e strongly updates heap variable a ⇐⇒ a was strongly updated on all e’s target functions.

Definition A.3. A do D is undef ⇐⇒ D’s alias set is ∅.

Definition A.4. A co C is undef ⇐⇒ ∃ do D whose dominate frontier is C and D is undef.

Definition A.5. A function F is undef ⇐⇒ ∃ do D that locates in F.