Parallel Flow-Sensitive Pointer Analysis by Graph-Rewriting

Vaivaswatha Nagaraj
Indian Institute of Science, Bangalore
vaivaswatha@hpc.serc.iisc.in

R Govindarajan
Indian Institute of Science, Bangalore
govind@serc.iisc.in

***Abstract*—Precise pointer analysis is a problem of interest to both the compiler and the program verification community. Flow-sensitivity is an important dimension of pointer analysis that affects the precision of the final result computed. Scaling flow-sensitive pointer analysis to millions of lines of code is a major challenge.**

Recently, staged flow-sensitive pointer analysis has been proposed, which exploits a sparse representation of program code created by staged analysis. In this paper we formulate the staged flow-sensitive pointer analysis as a graph-rewriting problem. Graph-rewriting has already been used for flow-insensitive analysis. However, formulating flow-sensitive pointer analysis as a graph-rewriting problem adds additional challenges due to the nature of flow-sensitivity.

We implement our parallel algorithm using Intel Threading Building Blocks and demonstrate considerable scaling (upto 2.6x) for 8 threads on a set of 10 benchmarks. Compared to the sequential implementation of staged flow-sensitive analysis, a single threaded execution of our implementation performs better in 8 of the benchmarks.

***Index Terms*—Flow-sensitive Pointer Analysis, Staged Flow Sensitive Pointer Analysis, Amorphous Data-Parallelism, Graph-Rewriting.**

I. Introduction

Many compiler optimizations rely on alias information to perform a safe code transformation [1]. Pointer alias analysis is a static analysis that tries to determine if two pointer expressions refer to the same memory location. Pointer (or points-to) analysis attempts to statically determine the possible run time values of pointer variables. This information can be used to determine aliases in the program [20].

A large number of compiler optimizations, ranging from simple dead code elimination to auto parallelization [37], need alias information. For example, to infer that a variable is dead, it is necessary to know that no alias of the variable is live at that point. Without precise alias information, these optimizations will need to make conservative assumptions (for example, even though two variables do not alias, the compiler may need to assume that they are aliased). This will lead to some optimizations not being performed. Pointer analysis is also useful in program verification problems such as bug detection [12].

Static analysis, and more specifically alias analysis, is in general undecidable [34] [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. One of the least precise (and hence fast) pointer analyses is the address taken analysis. It records all variables whose addresses have been assigned to another variable. It is often used in production compilers [20]. Inclusion vs unification [36] [2] is a well studied dimension that decides the way in which pointer assignments (e.g., "$x = y$") are processed. In the inclusion based approach, processing this statement will lead to the points-to set of x being a superset of the points-to set of y. In a unification based algorithm, the points-to sets of x and y will be considered equal. The context-sensitivity [16] of an analysis decides whether the calling context of a function is considered when analysing it. A context-insensitive analysis allows values to flow from one call through the function and return to another caller. Flow-sensitivity of an analysis determines whether control-flow of the program is taken into account. A flow-insensitive [13] analysis will compute for each pointer variable, a single points-to set for the entire program. A flow-sensitive analysis [15] on the other hand respects control flow, and hence will compute points-to sets at each program point for each pointer. Field-sensitivity determines whether accesses to individual members of an aggregate (such as *struct* in C) are tracked [31]. Some pointer analysis algorithms cannot be strictly classified based on the categories mentioned above. For example, the pointer analysis proposed by Nasre et al [29] is context-sensitive but uses a probabilistic data structure to approximate the computed results. Das [10] proposed an algorithm that lies between inclusion and exclusion based algorithms.

Our focus in this paper is primarily on flow-sensitive pointer analysis. Flow-sensitive pointer analysis has been shown to be of importance to a variety of program analyses such as analysis of multi-threaded code [35] and detection of security vulnerabilities [6]. Traditional flow-sensitive pointer analysis [21] uses an iterative dataflow analysis, which is extremely inefficient for pointer analysis, mainly due to the conservative propagation of all dataflow information from each node in the control flow graph to every other reachable node, because the analysis cannot know which node actually needs what information. Since there can be hundreds of thousands of pointers and each pointer can have thousands of pointees, a huge amount of information is stored, processed and propagated.

A frequently used method to optimize flow-sensitive dataflow analyses is to perform a *sparse analysis* [7] [19].

978 -1-4799-1021-2/13/$31.00 ©2013 IEEE

These analyses directly connect variable definitions to their uses, so that, dataflow values need to be propagated only to their uses. However, pointer information is needed to compute these def-use chains [1]. This results in a cyclic dependence. Hardekopf and Lin [14] present a *semi-sparse* analysis that performs a *sparse* analysis on some variables and an iterative dataflow analysis on other variables. In [15] they propose a fully *sparse* analysis. Li et al [25] proposed a method in which the problem of flow-sensitive pointer analysis is reduced to a general graph reachability problem. A recent work by Khedker et al [22] uses the idea that points-to information of a pointer needs to be propagated only to those program points in which the pointer is live. They combine flow-sensitive pointer analysis with liveness analysis [1].

A. Contributions

This paper makes the following contributions:

- We formulate flow-sensitive pointer analysis as a graph-rewriting problem.
- We propose an efficient parallel algorithm to solve the graph-rewriting problem. To the best of our knowledge, this is the first successful attempt at parallelizing fully flow-sensitive pointer analysis.
- We show how our algorithm can be efficiently implemented using a parallel programming framework such as Intel Threading Building Blocks [32].
- We demonstrate considerable scaling (upto 2.6x) for 8 threads on a set of 10 benchmarks. Compared to the sequential implementation of staged flow-sensitive analysis, a single threaded execution of our implementation performs better in 8 of the benchmarks.

II. Background

Next, we present some necessary background information on the static single assignment form, staged flow-sensitive analysis [15], and the graph-rewriting based flow-insensitive pointer analysis [27], which are useful in the rest of the paper

The goal of any pointer analysis is to determine statically, for each pointer variable p, all elements that p may point to (called the points-to set of p) at run time [18]. A flow-insensitive analysis will compute this information without regard to the program's control flow, and hence will compute for each variable, a single points-to set for the entire program. A flow-sensitive analysis on the other hand respects control flow, and hence will compute points-to sets at each program point for each pointer.

Pointer analysis involves handling the following types of statements (called points-to constraints) in a program.

- $x = \&a$: (*Address-of*) Pointer variable x is assigned the address of variable a.
- $x = y$: (*Copy*) Pointer variable y is copied over to pointer variable x. This effectively means that, after this statement, x will point to whatever y points to.
- $x = *y$: (*Load*) For each variable a that y may point to, after this statement, x will point to whatever a points to.
- $*x = y$: (*Store*) For each variable a that x may point to, after this statement, a will point to whatever y points to.

Similar to other pointer analysis techniques, we follow a *heap model* in which we consider each static memory allocation site as a distinct abstract memory location, even though it may correspond to different concrete memory locations during program execution [18].

A. Intermediate Representation

Static single assignment (SSA) [9] is a program representation that restricts each variable to have only a single definition. If a variable has multiple definitions, it is split into different variables (sometimes called versions). Whenever more than one definition reaches a point, a ϕ node, which is a special function that indicates multiple reaching definitions, is inserted.

Indirect defs and uses through pointers complicate the process of rewriting a program into the SSA form. In an indirect def or use, the actual variable being defined or used is not known (in the absence of points-to information). To avoid this problem, modern compilers such as GCC [30] or LLVM [24] use a partial SSA representation [15]. In partial SSA form, *top-level* variables (variables whose address is never taken) are placed in the SSA form while *address-taken* variables are not placed in the SSA form. *Top-level* variables are referenced directly in the IR. *Address-taken* variables are referenced only through indirect loads or stores.

Unless otherwise stated, we will use letters at the end of the alphabet to denote *top-level* variables (e.g., x, y, w in Fig. 1) and letters at the beginning to denote *address-taken* variables (e.g., a, b, c in Fig. 1). If a variable is sub-scripted, it denotes that the variable is in SSA form.

B. Staged Flow-sensitive Analysis

Traditional flow-sensitive analysis [21] requires storing the points-to information for every pointer variable at each node in the control flow graph (CFG). After application of the transfer function at each CFG node, the information needs to be propagated. Since all uses of the points-to information at a node are not known initially (knowledge of such uses would in turn require complete points-to information), the traditional approach has been to propagate information to all successors.

Consider the example shown in Fig. 1. Suppose that after processing CFG node **1**, it is known that x_1 may point to variables a or b. Since it may not be known what z_1 or y_1 points to, the points-to sets of a and b at node **1** will need to be propagated to both node **2** and node **3**.

Since the lack of points-to information prevents selective propagation of dataflow values (in this case, the points-to information), Hardekopf et al [15] proposed the use of an auxiliary analysis (henceforth referred to as AUX) that is less precise (and hence faster) than the main flow-sensitive analysis (henceforth referred to as the primary analysis). Typically, AUX will be a flow-insensitive context-insensitive analysis. Similar to [15], we use Andersen-style [2] (inclusion based) analysis for AUX.

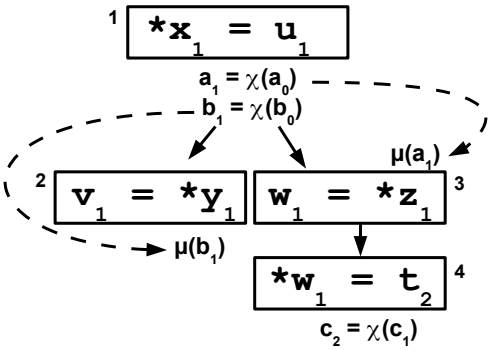

Fig. 1: Simple example to illustrate the need for AUX

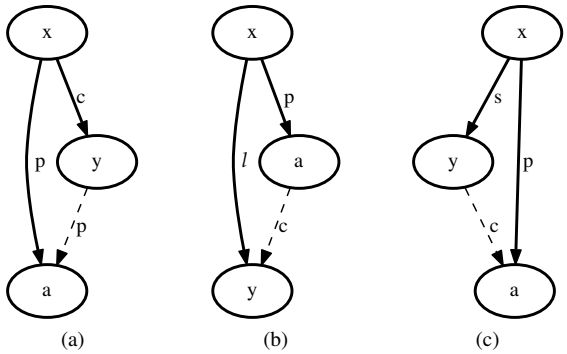

Fig. 2: Rewrite rules for Andersen's analysis. Dashed arrows indicate newly added edges. The edge types are: p(points-to), c(copy), l(load-to), s(store-from)

AUX helps in constructing conservative def-use information for *address-taken* variables. After performing AUX, flow-insensitive points-to information is available. This means that for each load or store statement, the points-to set (as computed by AUX) of the variable dereferenced at the statement forms the set of *address-taken* variables that may be indirectly referenced there. These possible indirect uses and defs are denoted by the μ and χ functions respectively [8]. Treating each χ as both a def and use of the variable, and μ as a use of the variable, the *address-taken* variables can be easily converted to SSA form using any standard SSA conversion algorithm [9]. Once the *address-taken* variables are in the SSA form, inferring def-use information is straightforward since SSA names inherently describe def-use information. With the def-use edges in place, points-to information needs to be propagated only along these def-use edges, thus saving both time and memory. The actual flow-sensitive analysis proceeds much like a traditional flow-sensitive analysis, except that information propagation happens only along def-use edges (corresponding to the variable whose dataflow information is being propagated).

Figure 1 illustrates the usage of μ and χ, assuming that the points-to sets for x_1, y_1, z_1 and w_1 as computed by AUX are $\{a, b\}$, $\{b\}$, $\{a\}$ and $\{c\}$. The *address-taken* variables are shown after conversion to SSA form. Def-use edges (built using results from AUX) are shown using dashed lines. In this example, the adoption of def-use edges lead to the points-to set of a being propagated only to node **3** and the points-to set of b being propagated only to node **2**.

A client (user) of this analysis would typically have a query as "what is the points-to set of pointer x at program point m" [29]. Since the program is assumed to be in partial SSA form, only *top-level* variables (which are in the SSA form) are referenced directly in the program. Hence, the points-to query from the client can be translated to (1) Find the SSA version (definition) of x that reaches program point m. Let this SSA version be x_1. (2) Return the points-to set of x_1 as computed by the staged flow-sensitive points-to analysis.

We use the the algorithm and implementation of Hardekopf et al [15] as our reference and will henceforth refer to it as the reference algorithm/implementation.

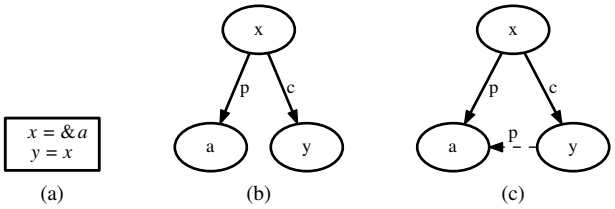

Fig. 3: (a) Example points-to constraints, (b) Initial constraint graph, (c) The constraint graph of (b) after applying the *copy* rewrite rule.

C. Graph-rewriting

Méndez-Lojo et al [27] introduced the idea of solving flow-insensitive pointer analysis via graph-rewriting. An initial constraint graph is built using the points-to constraints of the program, followed by repeated application of a set of rewrite rules on the graph, till the graph stops changing.

The constraint graph consists of a node for each pointer variable in the program. Corresponding to the four basic types of points-to constraints, there are four types of edges in the graph. There is a one-one correspondence between each edge in the initial constraint graph and each points-to constraint in the original program.

The rewrite rules for Andersen's analysis are shown in Fig. 2. Applying a rewrite rule amounts to processing a points-to constraint. After the rewriting terminates (i.e., no more rewrite rules can be applied), the outgoing "p" edges from a node form the points-to set of the variable corresponding to that node.

As an example, consider the constraints in Fig. 3(a). The initial constraint graph for this is shown in Fig. 3(b). After applying the *copy* rewrite rule (Fig. 2(a)), the graph would transform to what is shown in Fig. 3(c). The points-to sets for x and y initially are $\{a\}$ and $\{\}$. After applying the rewrite rule, the new points-to sets become $\{a\}$ and $\{a\}$ respectively.

III. Graph-rewriting formulation

The two main challenges in adapting flow-insensitive graph-rewriting rules (Section II-C) for flow-sensitive pointer analysis are (1) presence of spurious edges and (2) handling strong and weak updates. Hence we begin this section by explaining

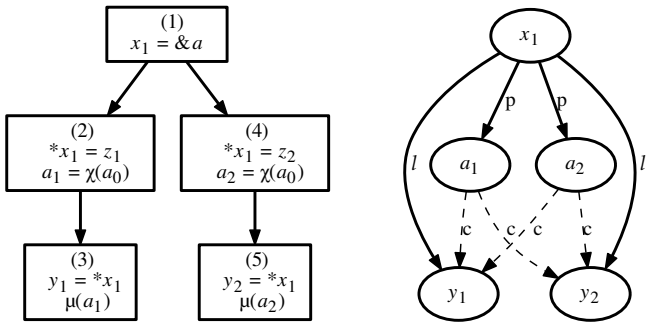

Fig. 4: (a) A simple program. (b) Flow-insensitive constraint graph corresponding to the load constraints in nodes **3** and **5** of (a).

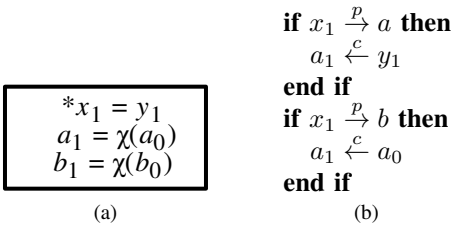

Fig. 5: The points-to set of variable a_1 in (a) is decided by the points-to set of x_1 as show in (b).

these two challenges. We then introduce a new set of rewrite rules that are required for flow-sensitive pointer analysis. Later, we identify and elucidate the sources of parallelism present in the analysis and how our formulation helps in extracting this parallelism.

A. Presence of Spurious Edges

Consider the CFG in Fig. 4(a). The CFG has been annotated with χ and μ functions using the results from AUX. The flow-insensitive constraint graph corresponding to load constraints in the CFG is shown in Fig. 4(b). Here, solid edges are part of the initial constraint graph and dashed edges are the ones added by flow-insensitive rewrite rules. The only definition of variable a reaching node **3** of the CFG is a_1 (and similarly only a_2 reaches node **5**). Hence, for a flow-sensitive pointer analysis, the points-to set of a_2 should not included in the points-to set of y_1 (and similarly, the points-to set of a_1 should not be included in the points-to set of y_2). Hence, for a flow-sensitive analysis, the edges $a_1 \xrightarrow{c} y_2$ and $a_2 \xrightarrow{c} y_1$ should not be added. These edges reduce the precision of the analysis. However, application of the flow-insensitive rewrite rules lead to the addition of these *spurious* edges (as shown in the figure).

Potential edges: For any edge type, we introduce the concept of a *potential* edge. A *potential* edge of type *t* indicates that there *could* be an actual edge of type *t* between the nodes, depending on the information computed by the flow-sensitive analysis as it progresses. Some of the rewrite rules will be modified to look for a *potential* edge before inserting an actual edge of that type. This will become clear as we explain the rewrite rules. We use a prefix "p_" on the edge type to denote *potential* edges of that type[1].

To illustrate the utility of *potential* edges, we again consider the example in Fig. 4. Our goal is now to modify the rewrite rules such that the two spurious edges $a_1 \xrightarrow{c} y_2$ and $a_2 \xrightarrow{c} y_1$ do not get added. Only $a_1 \xrightarrow{c} y_1$ and $a_2 \xrightarrow{c} y_2$ should be added as a result of applying the rewrite rules. To achieve this, we add *potential copy* edges $a_1 \xrightarrow{p\_c} y_1$ and $a_2 \xrightarrow{p\_c} y_2$ during construction of the initial constraint graph corresponding to the load constraints. The flow-insensitive load rewrite rule (Fig. 2(b)) is modified to add a new *copy* edge only if there already exists a *potential copy* edge between the two nodes. The two *copy* edges $a_1 \xrightarrow{c} y_1$ and $a_2 \xrightarrow{c} y_2$ will get added since *potential copy* edges exist between the two pairs. However, since the edges $a_1 \xrightarrow{p\_c} y_2$ and $a_2 \xrightarrow{p\_c} y_1$ are not present, the two spurious *copy* edges mentioned earlier do not get added.

B. Strong and Weak Updates

Consider the store node shown in Fig. 5(a). The flow-sensitive points-to set of a_1 depends not only on whether x_1 points to a but also on whether x_1 points to any other variable b (as computed by the flow-sensitive analysis). If x_1 points only to a, then a_1 will have a strong update, thus getting a "c" edge from y_1. If x_1 points to both a and b, then the previous points-to set of a (which is in the SSA variable a_0) also needs to be copied to a_1, making the update weak. If x_1 does not point to a, a_1 will have the same points-to set as a_0. This logic (w.r.t updating a_1) is shown in Fig. 5(b).

To handle strong and weak updates, we need a way to group all the indirect defs in a store constraint. We do this by introducing *Klique* nodes, a new node type which is used to connect all these indirect defs. Every store constraint in the original program will have an associated *Klique* node in the constraint graph. For each *address-taken* variable that may be (indirectly) defined in that store, we add a *Klique* edge (denoted by the letter "k")[2] from the corresponding SSA version of that variable to the corresponding *Klique* node. For example, for the store constraint in Fig. 5(a), a new *Klique* node is created and *Klique* edges are added from a_1 and b_1 to this *Klique* node. Thus, we have connected all indirect defs in a store constraint. The rewrite rules in the next sub-section will make it clear as to how this helps us perform a weak or a strong update.

We next present the various components of our algorithm, namely the graph structure (node and edge types) and the rewrite rules. The overall algorithm will be presented in Section III-E

C. Graph Structure

Node types: The constraint graph for flow-sensitive analysis can have the following types of nodes:

[1]The prefix "p_" used for potential edges (which can be of any type) should not be confused with the label "p" used for points-to edges

[2]Note that we use the term *Klique* for both nodes and edges. A *Klique* edge will always be directed to a *Klique* node.

- *TopLvl*: These nodes correspond to the *top-level* variables in the program. *Top-level* variables are expected to be in the SSA form [9]. Since the SSA form guarantees a single definition, the outgoing "p" edges of a *TopLvl* node will represent the flow-sensitive points-to set of the corresponding top level variables.
- *AddrTakenSSA*: There is one *AddrTakenSSA* node for each SSA version of each *address-taken* variable. Since each possible (indirect) definition of an *address-taken* variable will have its own SSA version, the outgoing "p" edges of these nodes will represent the flow-sensitive points-to sets of the corresponding *address-taken* variables.
- *NonSSA*: These nodes correspond to *address-taken* variables in the original program, before converting them to SSA form. We will use these nodes as a representative of all *AddrTakenSSA* nodes that are from the same *address-taken* variable. All "p" edges will be directed towards these nodes.
- *Klique*: These nodes are useful to handle strong and weak updates. They are artificial nodes that are introduced to recognize the set of *address-taken* variables that may (indirectly) be defined at a given store constraint.

As remarked earlier, SSA variables will be sub-scripted with a number.

Apart from the four edge types (p, c, l and s) that are used in flow-insensitive analysis, we have the following additional **edge types**:

- *Uses* (u): In a load constraint, when it is known (as the flow-sensitive analysis progresses) that the *TopLvl* node being dereferenced points to an *address-taken* variable, a *Uses* edge is added from the *TopLvl* node to the corresponding *AddrTakenSSA* node. Note that the *AddrTakenSSA* node will have a μ function in that load constraint.
- *Defines* (d): In a store constraint, when it is known (as the flow-sensitive analysis progresses) that the *TopLvl* node being dereferenced points to an *address-taken* variable, a *Defines* edge is added from the *TopLvl* node to the corresponding *AddrTakenSSA* node. Note that the *AddrTakenSSA* node will be on the LHS of a χ function in that store constraint.
- *Klique edge* (k): These edges connect *AddrTakenSSA* nodes on the LHS of χ functions in a store constraint to a *Klique* node that is unique to that store constraint.
- *NonKilledCopy* (n): These edges connect the *AddrTakenSSA* node on the RHS of a χ function to the corresponding *AddrTakenSSA* node on the LHS. They are useful to perform weak updates.
- *AlreadyDefined* (a): *AddrTakenSSA* nodes in a store constraint that are already defined (i.e., they have an incoming *Defines* edge) are marked by having an *AlreadyDefined* edge from the *Klique* node of that store constraint.
- *SSAParent* (ssa-parent) : Each *AddrTakenSSA* node will have exactly one *SSAParent* edge to identify the *NonSSA* node that corresponds to it.

Some of the edge types above can also be seen as properties of nodes. For example, the *SSAParent* edge from a node can be considered as a constant property of that node. We present them as edges for the sake of a pure graph formulation.

Initial constraint graph: The initial constraint graph is built from points-to constraints in the input program. The nodes and edges added for each type of constraint is shown in Fig. 6.

The initial constraint graph for a store constraint (Fig. 6(a)) consists of the following components:

1) The *top-level* variable (x_1 here) potentially defines ("p_d") the *address-taken* taken variables (a_1 and b_1) on the LHS of a χ function.
2) The store edge, labelled "s" (here x_1 stores from y_1).
3) Due to the store (from y_1), there is a *potential copy* ("p_c") from the RHS *top-level* variable (y_1) to each variable that may potentially be defined (a_1 and b_1).
4) *AddrTakenSSA* nodes on the RHS of χ functions (a_0 and b_0) have a *NonKilledCopy* edge (labelled "n") to the corresponding nodes (a_1 and b_1 respectively) on the LHS of the χ function. Upon a weak update, this edge leads to a *copy* edge.
5) *Klique* nodes and edges (labelled "k") are added to group together variables that may (indirectly) be defined in this constraint (here, a_1 and b_1 are connected to the *Klique* node k_1). As mentioned, this grouping helps in performing weak updates.

The initial constraint graphs for other types of constraints (Fig. 6) are similar to (but simpler than) the initial graph for store constraints. We skip explaining them in detail.

D. Rewrite Rules

Once the initial constraint graph is constructed from the input program as shown in Fig. 6, the analysis needs to be solved. The rewrite rules to solve the analysis are shown in Fig. 7. We apply these rewrite rules till no more edges are added to the graph, indicating a fixed point. At the end of the analysis, the "p" edges emanating from a node form the points-to set for the variable corresponding to that node.

Rules (a), (b) and (c) (Fig. 7) are adaptations of the three flow-insensitive rewrite rules, taking spurious edges into account. These rules now look for a potential edge before adding an actual edge. Rules (d) and (e) add *Defines* and *Uses* edges based on whether the *TopLvl* node being dereferenced points to the concerned *address-taken* variable. This requires knowing the *SSAParent* of *AddrTakenSSA* nodes that are potentially defined or used (in χ or μ functions). Rules (f) and (g) handle weak updates. Rule (f) marks a node belonging to a *Klique* as already defined (i.e., it has an incoming *Defines* edge). This is the first step in performing a weak update. For an *AddrTakenSSA* node on the LHS of a χ function, if another *AddrTakenSSA* node in the same *Klique* is already defined, then the value on the RHS of the χ function needs to be copied. This is carried out by rule (g), completing the weak update process. Thus, rules (c), (f) and (g) together handle the logic for strong and weak updates (Fig. 5).

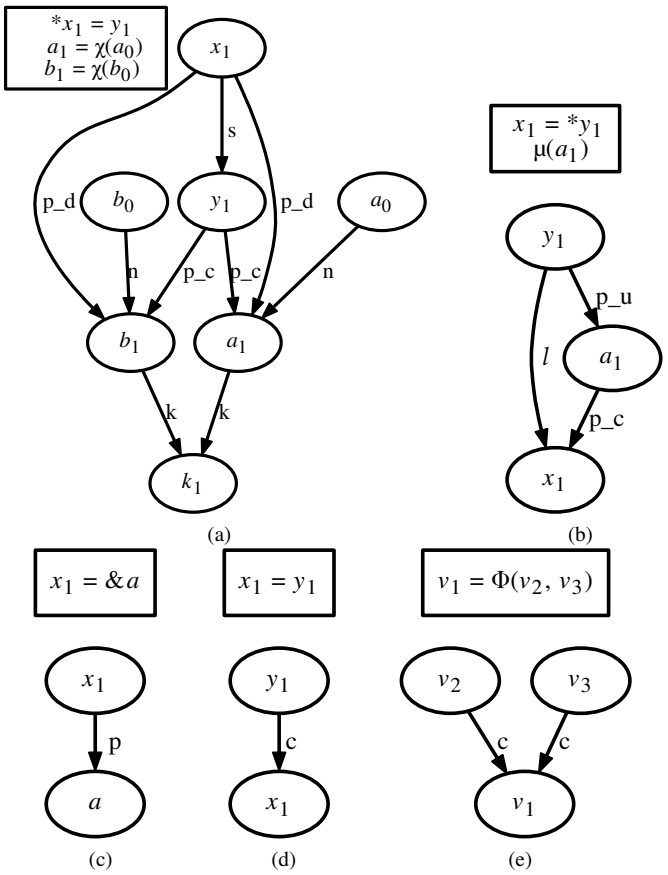

Fig. 6: Initial graph for for different types of constraints. In (a), k_1 is a *Klique* node. In (e), v_1, v_2, v_3 may be either *top-level* or *address-taken*.

Similar to the flow-insensitive rewrite rules in [27], our rewrite rules have the property that they may be applied on different nodes in parallel, as long as the underlying data structure that represents edges can handle concurrent updates.

E. Putting It All Together

A summary of the steps in our method is presented here.

1) Convert *TopLvl* variables to SSA form. Most compilers already have an SSA based representation, and hence this step may not be necessary.
2) Perform AUX. The less precise points-to information computed by AUX is used to annotate the program with χ and μ functions.
3) Convert the *address-taken* variables (which are in the χ and μ functions) to SSA form using any standard SSA conversion algorithm.
4) Build the initial constraint graph according to Fig. 6.
5) Apply the flow-sensitive graph rewrite rules (Fig. 7), in parallel, repeatedly till fixed point.
6) The points-to set for each *top-level* variable can be obtained from the corresponding *TopLvl* graph node. The points-to ("p") edges from this node form the points-to set. Since only *top-level* variables are directly referenced

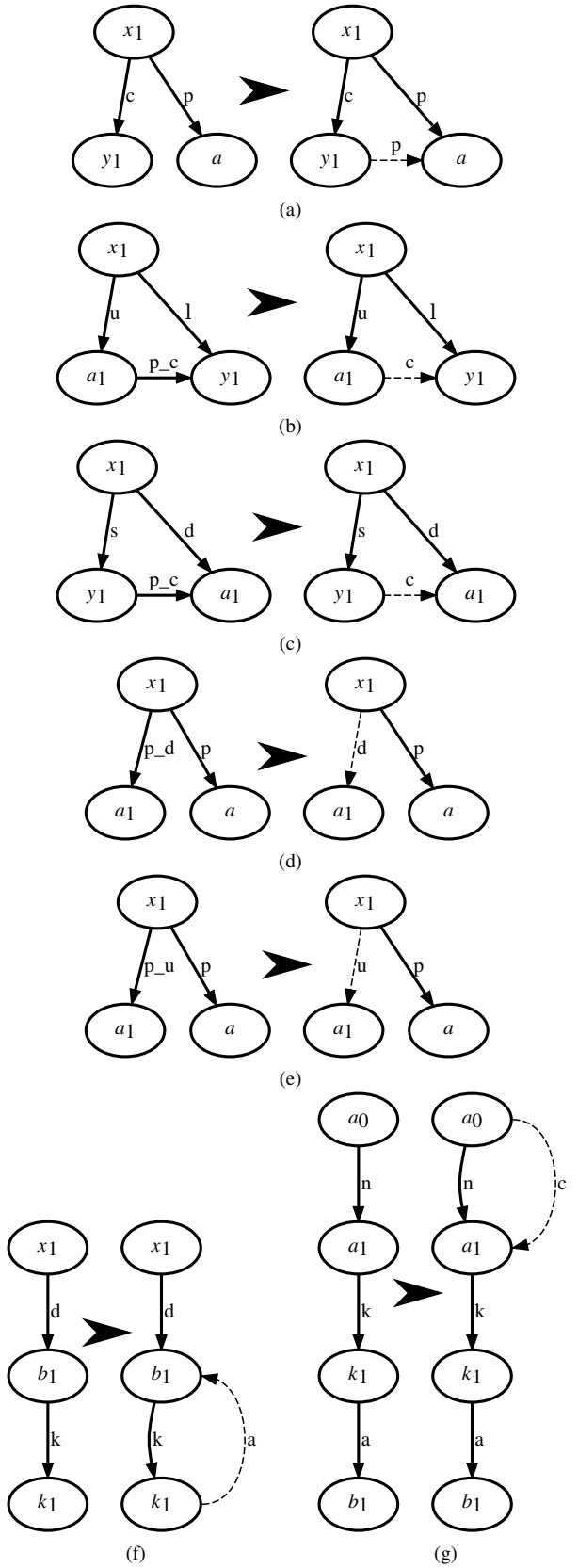

Fig. 7: Rewrite rules for flow-sensitive pointer analysis

in partial SSA form, the points-to sets of *address-taken* variables are not needed after the algorithm terminates.

F. Properties of Our Graph-rewriting System

In this section, we show that our algorithm described in Section III-E: (1) Terminates. (2) Computes a unique solution whose precision is equivalent to that of the reference algorithm. (3) Easily exploits parallelism. Due to space limitation, we provide only an informal argument and do not provide a formal proof.

Theorem 1. *The algorithm described in the previous section terminates in a finite number of steps.*

Proof: Since the total number of edges that can be added to the graph is finite and we add at least one edge to the graph in each iteration (we stop if no edges are added in an iteration), the algorithm terminates after a finite number of iterations. ■

Theorem 2. *The fixed point (final graph) obtained when the algorithm terminates is unique.*

Proof: Similar to the rewrite rules in [27], our rewrite rules are locally confluent. Combining this with the result of Theorem 1, we can conclude (by Newman's lemma [4]) that the rewriting system is globally confluent, and hence the final graph (fixed point) is unique. ■

Theorem 3. *The results computed by our algorithm is equivalent to that of the reference algorithm.*

Proof: Every *TopLvl* node in our graph corresponds to a *top-level* variable. The points-to sets for these variables are flow-sensitive since they are in the SSA form. For *address-taken* variables, the reference algorithm stores the points-to sets at each program point where the variable is in a χ or μ function. We achieve the same by having different nodes for each SSA version of the *address-taken* variable, with these SSA versions corresponding to the use/def of the variable in a χ or μ function. Actual dataflow propagation in the reference algorithm happens through def-use edges built using the SSA names of variables. However, our method does not have explicit def-use edges. The node for a variable holds the points-to information. Update of points-to set of a variable happens through incoming *copy* edges to the node (that corresponds to the variable) in the constraint graph. These *copy* edges are added wherever points-to set propagation happens in the reference algorithm. ■

Observation 1. *Our formulation of the analysis exposes parallelism.*

In the analysis performed by the reference algorithm, any points-to constraint that has new incoming (IN) information can be processed independently of other constraints. This means that there are multiple items on the worklist that can be processed in parallel. The second source of parallelism in the analysis is the update of variables in a single constraint. Since a load or store constraint can involve more than one *address-taken* variable, each of them can be updated independently of the other while processing the constraint. Our graph formulation exposes both these sources of parallelism in a natural way. Any two graph nodes can be processed in parallel. Synchronization is required only when two threads simultaneously try to update edges of the same node. Such a parallelism in which different nodes in the graph can be processed in parallel, but with certain constraints, is called *amorphous data-parallelism* [33].

IV. Implementation and Optimizations

We implemented our parallel algorithm in C++, using Intel Threading Building Blocks [32] (referred to as TBB henceforth) for parallel work management.

The initial constraint graph for all benchmarks were generated using Ben Hardekopf's LLVM implementation of the staged flow-sensitive pointer analysis [15]. Hence, our implementation has the same properties (w.r.t various dimensions of pointer analysis) as Ben Hardekopf's implementation. Our implementation is context-insensitive, field-sensitive and flow-sensitive. Although our graph formulation description in the previous section ignores pointer arithmetic statements [3] (statements of the form "$x = y + o$", where x, y are pointer variables and o is a constant offset), our implementation handles these points-to constraints in a way similar to earlier work [26].

Edge representation: In our graph-rewriting approach, although different nodes can be processed independently, synchronization is required whenever two threads try to add edges to the same node. To handle this efficiently, we use concurrent data structures that allow parallel updates by different threads. A concurrent sparse bit vector is highly suitable for this purpose. However we were unable to obtain a good implementation of concurrent sparse bit vectors in C++. So we used the *concurrent_unordered_set* data structure provided by TBB in our implementation. This is a hash table based data structure that allows concurrent addition of set elements.

Potential edges can be represented in two ways. Since they lose importance as soon as an actual edge of the same type is added between the two nodes, one way to represent them would be to have two bits per edge. 00 represents no edge, 01 represents a *potential* edge and 10 (or 11) can represent an actual edge. The other way to implement these edges is to treat them as a different edge type altogether. We use the latter approach in our implementation. In such an implementation, there is an option to delete *potential* edges once an actual edge has been added in its place. Deleting may improve performance since deleted edges will not be redundantly checked when applying a rewrite rule.

Worklist approach: Although it is possible to try and apply each rewrite rule to all nodes until no more edges are added to the graph, it is very inefficient. We use a worklist based approach to apply rewrite rules, maintaining a separate worklist for each of the rewrite rules. "p" edges in the initial

[3]Since pointer arithmetic statements do not add any new challenge to flow-sensitive analysis, we skipped describing them in our main algorithm.

constraint graph are used to build the initial worklists. Any new edge that is added during the analysis can trigger more nodes to be added to one or more worklists. For example, a new "p" edge can trigger the rules (a), (d) and (e) in Fig. 7.

We employ a dual worklist based approach (similar to double buffering [3]) to add and remove items from the worklists. Rewrite rules are applied to nodes on the *current* worklist, while we add new nodes (which need processing) to the *next* worklist. Once the *current* worklist becomes empty, the *current* and *next* worklists are swapped and the process repeats. This is done until both the worklists are empty. We implement worklists using vectors provided by the standard template library [28]. Further, the *next* worklist is maintained per thread (as thread local storage) and at the end of an iteration, before swapping *next* and *current*, we combine the per thread worklists. The *current* worklist is a global worklist. We use a per node flag to determine if a node is in a worklist or not. We rely on the *parallel_for* construct of TBB to manage work scheduling. TBB implements work-stealing [32] [5] to manage parallel workload.

Incremental updates: When new "p" edges are added to a node, the node gets added to the worklist of rewrite rule (a) (in Fig. 7). When this node is processed again in the next iteration to apply the rule, only the new "p" edges need to be propagated (through the "c" edges). Points-to edges that were added before the previous iteration can be ignored here. This optimization can be applied to other rewrite rules as well. Thus it helps to maintain newly added edges as a delta over existing edges. This idea has already been used by Méndez-Lojo et al in their GPU implementation of Andersen's analysis [26]. We adapt this idea to our approach. This optimization ensures that no redundant work is carried out when a rewrite rule is applied.

V. Results

In this section, we evaluate our algorithm and implementation by comparing it to the current state-of-art algorithm by Hardekopf et al [15]. We use the implementation available at `http://www.cs.ucsb.edu/~benh/` as our reference. We first show some properties of the benchmarks that we used followed by the performance results.

Our experiments are conducted on a 4-socket machine with a 2.0 GHz 8-core processor on each socket, with a total of 64GB memory. The machine runs Debian GNU/Linux 6.0 and has version 4.0 of Intel Threading Building Blocks [32] installed.

We use a subset of benchmarks from SPEC2006 [17] that are relatively larger (in terms of lines of code), and a few other programs that have all been used in previous studies [15] [27]. Program names with a numbered prefix are from SPEC2006. *Ex* is a text processor, *Nethack* is a text based game, *Sendmail* is an email server, *Svn* (subversion) is a revision control system and *Vim* is a text editor.

Table I shows the number of nodes (of each type) present in the initial constraint graph. Table II shows the average (across iterations) number of nodes in each worklist for each benchmark. The worklists (a) to (g) correspond to rewrites rules from (a) to (g) in Fig. 7. Worklist (h) corresponds to the rewrite rule for processing pointer arithmetic statements (as stated earlier in Section IV, we have not mentioned processing of pointer arithmetic statements in our algorithm description, though we process them in our implementation). While these columns give us some idea of the extent to which the analysis for the benchmarks can be parallelized, it is important to note that not all of it can be exploited fully. There may be contention among threads (when applying a rewrite rule) to access the same node, which may slow down the accesses.

TABLE I: Number of nodes of each type

benchmark	TopLvl	NonSSA	AddrTakenSSA	Klique
ex	9852	1229	7953	148
254.gap	45946	2393	635639	469
176.gcc	114994	5908	255774	1770
nethack	85994	11977	124448	223
197.parser	8514	1020	3631	129
253.perlbmk	50538	2829	270833	543
sendmail	45155	4136	22220	347
svn	99181	8740	6328901	2738
vim	238031	8935	1017678	724
255.vortex	17910	3304	12138	107

TABLE II: Average number of nodes in each worklist

benchmark	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)
ex	86	12	11	11	13	11	16	6
254.gap	335	23	20	20	23	20	5	27
176.gcc	220	76	27	27	76	27	23	72
nethack	1147	270	23	23	275	23	278	956
197.parser	57	8	16	16	9	16	4	13
253.perlbmk	106	15	9	9	16	9	10	14
sendmail	225	31	17	17	32	17	31	22
svn	713	180	82	82	180	82	315	26
vim	4638	81	34	34	85	34	625	66
255.vortex	260	40	10	10	40	10	151	64

A. Performance

Figure 8 shows the performance of our analysis compared to the reference implementation. For eight of the benchmarks, a single threaded execution of our algorithm outperforms the reference implementation. A maximum speedup of 14.4x is seen for *gap*.

We are able to scale upto four threads easily for most benchmarks. However, some of the benchmarks (such as *parser*) do not scale well. As expected, the number of nodes on each worklist for a benchmark limits the amount of parallelism that can be extracted. For example, *svn*, *vim* and *nethack* show good scaling as they have a larger number of nodes (on an average) on their worklists (*svn* and *vim* also scale to eight threads easily). On the other hand, *parser* and *perl* have relatively lower number of nodes (on an average) on the worklists, and hence do not show good scaling.

An important aspect of our implementation is the use of hash table based sets. We use the *concurrent_unordered_set* data structure provided by TBB to represent edges. As we remarked earlier, this data structure is not efficient for representing graph edges for the following reasons: (1) Hash based

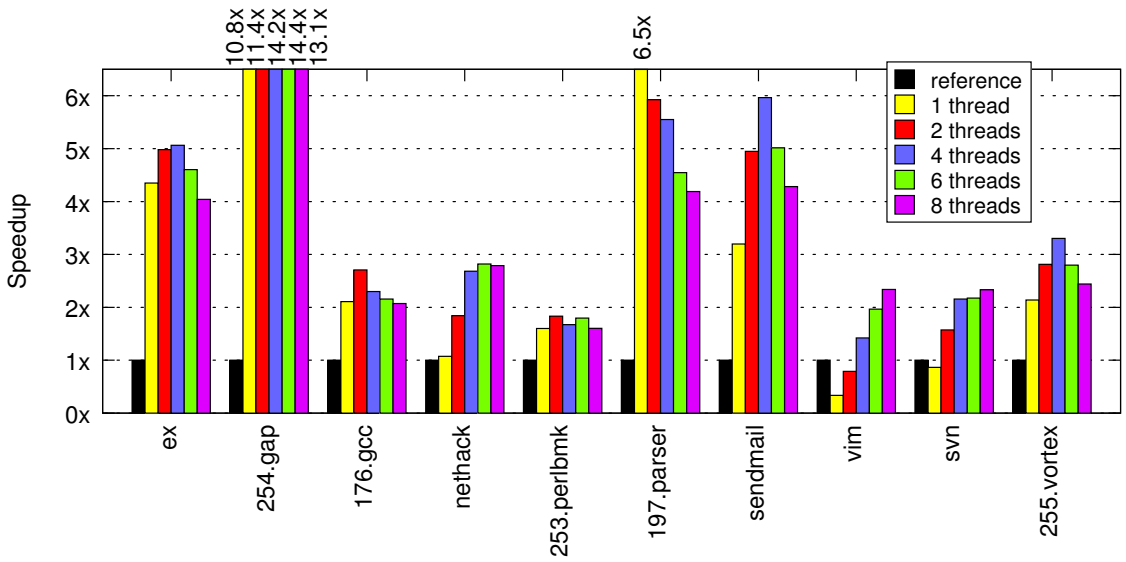

Fig. 8: Speedup compared to the reference implementation

sets require one node in the chaining linked list (of a bucket) to store a single bit. (2) The entire table is allocated even when the number of bits stored is low. A good data structure for representing edges in constraint graphs is the *sparse bit vector* [26]. A sparse bit vector can store multiple bits in a node of its linked list. Hence, it can skip over bits faster during a lookup and it also takes up lesser memory to store the same number of bits. However, we were unable to obtain a good implementation of *sparse bit vectors* that supports concurrent accesses. We plan to implement sparse bit vector based edge representation in the future. We believe that using a concurrent sparse bit vector can further improve the scalability of our algorithm.

VI. Related work

The area of pointer analysis is rich in literature. A good survey can be found in [20]. A good description of traditional flow-sensitive pointer analysis can be found in [21]. Flow-sensitive pointer analyses based on iterative dataflow analysis are in general inefficient as they redundantly propagate large points-to sets. However, a recent work by Khedker et al [22] avoids redundant propagation by taking liveness of pointer variables into account. Points-to sets are propagated only to places where they are live. This work by Khedker et al is both flow and context-sensitive. It was evaluated on benchmarks upto 30kLoC.

Hardekopf and Lin proposed a *semi-sparse* flow-sensitive analysis [14] which takes advantage of *sparse* representation for some of the variables in the program. Their staged flow-sensitive pointer analysis [15] is a fully *sparse* analysis and succeeds in scaling to millions of lines of code. Li et al [25] formulate flow-sensitive pointer analysis as a graph reachability problem using value flow graphs.

The work by Méndez-Lojo et al [27] is the first one to formulate pointer analysis as a graph-rewriting problem and parallelize it. They extend this work to perform the same analysis on GPUs [26]. However, their analysis is flow-insensitive. As we mentioned earlier in Section III, flow-sensitivity introduces additional challenges that we address in our work.

In [11], Edvinsson et al take advantage of control flow branches and polymorphic calls to find independent tasks that can be processed in parallel. Although ordering of constraints during processing provides partial flow-sensitivity, the analysis is not fully flow-sensitive since no strong updates are performed. In terms of parallelism, the granularity of parallelism in their algorithm is limited to processing multiple constraints in parallel. Our algorithm, in addition to this, enables updating multiple variables within a constraint in parallel.

To the best of our knowledge, our work is the first to parallelize fully flow-sensitive pointer analysis.

VII. Conclusion and future work

Flow-sensitive pointer analysis is an important problem for the compiler community. Improving the efficiency of this analysis can result in significant speedup of the compilation process. To this effect, our work contributes a new algorithm to perform flow-sensitive pointer analysis efficiently.

In this paper, we identified two sources of parallelism in flow-sensitive pointer analysis and introduced a graph-rewriting formulation of the analysis that can easily extract this parallelism. Our algorithm is efficient and easy to implement. It scales considerably upto 8 threads.

We currently use a hash table based set representation for graph edges. We believe that our implementation can be further improved by employing *sparse bit vectors* to represent graph edges. Our approach can also be extended to incorporate context-sensitivity. We leave this for future work.

VIII. Acknowledgments

We would like to thank Ben Hardekopf for his LLVM implementation of the staged flow-sensitive pointer analysis and for providing us the benchmarks. We would also like to thank Mario Méndez-Lojo for making available his parallel implementation of Andersen's analysis and his suggestions for our implementation. Rupesh Nasre's comments were useful in improving the paper.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, 1994.

[3] Edward Angel. *Interactive Computer Graphics: A Top-Down Approach Using OpenGL*. Addison-Wesley Publishing Company, USA, 5th edition, 2008.

[4] M. Bezem, J.W. Klop, Terese, and R. de Vrijer. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

[5] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[6] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 39–50, New York, NY, USA, 2008. ACM.

[7] 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, pages 55–66, New York, NY, USA, 1991. ACM.

[8] Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in ssa form. In *Proceedings of the 6th International Conference on Compiler Construction*, CC '96, pages 253–267, London, UK, UK, 1996. Springer-Verlag.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[10] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 35–46, New York, NY, USA, 2000. ACM.

[11] Marcus Edvinsson, Jonas Lundberg, and Welf Löwe. Parallel points-to analysis for multi-core machines. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 45–54, New York, NY, USA, 2011. ACM.

[12] Samuel Z. Guyer and Calvin Lin. Error checking with client-driven pointer analysis. In *Science of Computer Programming*, 2005.

[13] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM.

[14] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. *SIGPLAN Not.*, 44(1):226–238, January 2009.

[15] 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, pages 289–298, Washington, DC, USA, 2011. IEEE Computer Society.

[16] Laurie Hendren. Context-sensitive points-to analysis: Is it worth it. In *Compiler Construction, 15th International Conference, volume 3923 of LNCS*, pages 47–64. Springer, 2006.

[17] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[18] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[19] 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.

[20] Michael Hind and Anthony Pioli. Which pointer analysis should i use? *SIGSOFT Softw. Eng. Notes*, 25(5):113–123, August 2000.

[21] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[22] Uday P. Khedker, Alan Mycroft, and Prashant Singh Rawat. Liveness-based pointer analysis. In *Proceedings of the 19th international conference on Static Analysis*, SAS'12, pages 265–282, Berlin, Heidelberg, 2012. Springer-Verlag.

[23] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[24] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, December 2002.

[25] 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.

[26] Mario Méndez-Lojo, Martin Burtscher, and Keshav Pingali. A gpu implementation of inclusion-based points-to analysis. In *PPOPP*, pages 107–116, 2012.

[27] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. Parallel inclusion-based points-to analysis. *SIGPLAN Not.*, 45(10):428–443, October 2010.

[28] David R. Musser and Atul Saini. *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[29] Rupesh Nasre, Kaushik Rajan, R. Govindarajan, and Uday P. Khedker. Scalable context-sensitive points-to analysis using multi-dimensional bloom filters. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 47–62, Berlin, Heidelberg, 2009. Springer-Verlag.

[30] Diego Novillo. Tree ssa - a new optimization infrastructure for gcc. In *GCC Developers Summit*, 2003.

[31] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst.*, 30(1), November 2007.

[32] Chuck Pheatt. Intel threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008.

[33] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. *SIGPLAN Not.*, 46(6):12–25, June 2011.

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

[35] 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.

[36] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[37] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool.