Exploiting the Structure of the Constraint Graph for Efficient Points-to Analysis

Rupesh Nasre.

Indian Institute of Science, Bangalore, India

The University of Texas, Austin, USA

nasre@acm.org

Abstract

Points-to analysis is a key compiler analysis. Several memory related optimizations use points-to information to improve their effectiveness. Points-to analysis is performed by building a constraint graph of pointer variables and dynamically updating it to propagate more and more points-to information across its subset edges. So far, the structure of the constraint graph has been only trivially exploited for efficient propagation of information, e.g., in identifying cyclic components or to propagate information in topological order. We perform a careful study of its structure and propose a new inclusion-based flow-insensitive context-sensitive points-to analysis algorithm based on the notion of dominant pointers. We also propose a new kind of pointer-equivalence based on dominant pointers which provides significantly more opportunities for reducing the number of pointers tracked during the analysis. Based on this hitherto unexplored form of pointer-equivalence, we develop a new context-sensitive flow-insensitive points-to analysis algorithm which uses incremental dominator update to efficiently compute points-to information. Using a large suite of programs consisting of SPEC 2000 benchmarks and five large open source programs we show that our points-to analysis is 88% faster than BDD-based Lazy Cycle Detection and 2× faster than Deep Propagation. We argue that our approach of detecting dominator-based pointer-equivalence is a key to improve points-to analysis efficiency.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors-Optimization

General Terms Algorithms, Languages

Keywords constraint graph, dominators, points-to analysis, context-sensitivity

1. Introduction

Points-to analysis is a method of statically determining whether two pointers in a program may point to the same location at runtime. The two pointers are then said to be aliases of each other.

Points-to analysis enables several compiler optimizations and remains an important static analysis technique. With the advent of multi-core hardware and parallel computing, points-to analysis enjoys enormous importance as a key technique in code parallelization. Enormous growth of the code bases in proprietary and open source software systems demands scalability of heap analyses over millions of lines of code. A large number of points-to analysis algorithms have been proposed in literature that make this research area rich in content [2, 3, 6, 15, 31, 33].

For analyzing a general purpose C program in a flow-insensitive manner, it is sufficient to consider all pointer statements of the following forms: address-of assignment (p = &q), copy assignment (p = q), load assignment (p = *q) and store assignment (*p = q) [25]. Load and store assignments are also referred to as complex assignments. A heap allocation is represented using an address-of assignment. We deal with context-sensitive (which takes into account the calling context of a function), flow-insensitive (which ignores control-flow), field-insensitive (which assumes that access to a field of an aggregate is to the whole aggregate) inclusion-based (Andersen-style) points-to analysis in this work.

A flow-insensitive analysis iterates over a set of points-to constraints until a fixed-point is obtained. Typically, the flow of points-to information is represented using a constraint graph G, in which a node denotes a pointer variable and a directed edge from node n1 to node n2 represents propagation of points-to information from n1 to n2. Each node is initialized with the points-to information computed by evaluating the address-of constraints. Edges are added to G initially by copy constraints and then by complex (load and store) constraints as the analysis progresses. This is because the edges introduced by complex constraints depend upon the availability of points-to information at nodes which, in turn, depends upon the propagation. Thus, as the analysis performs an iterative progression of the points-to information propagation, new edges get introduced in G due to the evaluation of the complex constraints, resulting in the computation of more and more points-to information at its nodes. When no more edges can be added and no more points-to information can be computed, G gets stabilized and a fixed-point (points-to information at the nodes) is reached. The information can then be used by various clients (e.g., slicing, array-bounds checking, etc.). An outline of this analysis is given in Algorithm 1.

Techniques have been developed for efficient propagation of the points-to information across the edges of a constraint graph. Online cycle elimination [8] detects cycles in G on-the-fly and collapses all the nodes in a cycle into a representative node. Cycle collapsing is possible because all the nodes in a cycle eventually contain the same points-to information. This significantly reduces the number of pointers tracked and speeds up the overall analysis. Wave and Deep Propagation [25] techniques perform a topological ordering of the edges and propagate only the difference in the points-to information in breadth-first or depth-first manner respec-
Algorithm 1 Points-to Analysis using Constraint Graph

Require: set C of points-to constraints
1: Process address-of constraints
2: Add edges to constraint graph G using copy constraints
3: repeat
4: Propagate points-to information in G
5: Add edges to G using load and store constraints
6: until fixed-point

Algorithm 1 takes three iterations to compute fixed-point for this example. In each iteration, new points-to information is computed at the nodes and new edges are added to the constraint graph. In a directed graph, a node d dominates node n if all the paths from a start node a to n go through d. The node d is called a dominator and the node n is called a dominated. At the end of the analysis of our example in Figure 1, we observe that node a dominates nodes b, c, d when a is the start node. Note that the points-to information of nodes a, b, c, d is the same {x, y}. Thus, the analysis can make all these nodes as pointer-equivalent, so that, only one of them can be tracked during the analysis. Identifying pointer-equivalent variables is a key optimization technique for scaling points-to analysis. It avoids propagation of points-to information across the edges between pointer-equivalent variables, improving analysis efficiency. Thus, identifying dominator information in the constraint graph helps identify more pointer-equivalent variables resulting in faster analysis.

Note that at the end of Iteration 1, node a dominates node x when a is the start node. However, addition of the edge from node q to node x in Iteration 2 breaks this dominator-dominance relationship. Therefore, the dominator information is dynamic and it should be computed on-the-fly to achieve analysis soundness. However, exhaustively re-computing dominator information can be quite time-consuming. Therefore, we use incremental dominance computation, along with a few novel heuristics, to improve the analysis efficiency. For instance, note that although the dominator-dominance relationship between nodes a and x is broken, at the end of the analysis, points-to information of x is a superset of that of a, since no edges are removed from the graph. This helps us keep only a difference in the information between x and a.

We also observe in Figure 1 that node a dominates node e, but the two nodes have different points-to information. This happens because of the address-of constraint e = &z, which adds a different points-to information to node e. Thus, a naive way of computing dominators in the constraint graph does not yield a sound result.
We modify the notion of constraint graph in the next subsection to seamlessly deal with such special cases.

2.2 Modified Constraint Graph

In our formulation, for ease of understanding and implementation, we slightly modify the constraint graph without affecting its characteristics. In the following discussion, we refer to a constraint graph by $G$ and the modified constraint graph by $G'$. $G = (V, E)$ where a vertex $v \in V$ represents a pointer node and a directed edge $(u, v) \in E$ represents a subset relationship between the points-to sets of pointers represented using nodes $u$ and $v$, i.e., $\text{pointsto}(u) \subseteq \text{pointsto}(v)$. We call $u$ as the source node and $v$ as the target node. We are now ready to modify the constraint graph $G$ to $G'$, which we use throughout our analysis.

First, $G' = (V', E')$ contains a single unique node for each address-taken variable. Thus, $V' = V \cup \{ &v | v \text{ is an address-taken variable} \}$. In the above example, the address-taken variables $&d, &c, &x, &y, &z$ are represented using additional nodes as shown at the top in Figure 2. Note that in the original constraint graph $G$, the address-taken variables, which occur due to address-of constraints ($p = &q$), are directly added to the points-to set of the left-hand side pointer. In effect, $G$ does not contain any node corresponding to the address-taken variables, and, in fact, it contains nodes corresponding to only the pointer variables. It should be noted that $G'$ would contain one address-taken node and a separate pointer node for each address-taken variable (e.g., nodes $x$ and $&x$).

Second, a directed edge is added for an address-of constraint $p = &q$ from the node corresponding to $&q$ to pointer node $p$. Thus, $E' = E \cup \{ (k, u) | v = &k \text{ is an input constraint} \}$.

In effect, while any node may act as a start node (node without any incoming edges) in $G$, in $G'$, only those nodes that correspond to the address-taken variables can act as start nodes. If a pointer node $n$ in the modified constraint graph $G'$ does not contain any incoming edge, it can be proven that $n$ has an empty points-to set.

This formulation allows us to cast points-to analysis problem as a reachability problem in the constraint graph.

**Theorem 1.** The points-to set of a pointer $p$ is the set of start nodes from which the node $p$ is reachable.

The above claim can be easily verified from Figure 2. For instance, node $x$ is reachable from start nodes $&c$, $&x$ and $&y$ and its points-to information is also $\{ c, x, y \}$.

**Corollary 2.** Two pointers reachable from the same set of start nodes are pointer equivalent.

The above claim can be easily verified from Figure 2. For instance, node $x$ is reachable from start nodes $&c$, $&x$ and $&y$ and its points-to information is also $\{ c, x, y \}$.

One may get tempted to conclude that a reachability formulation would allow us to discard any points-to information explicitly stored at the nodes, because of the availability of the same information in the form of paths (from address-of nodes to pointer nodes). In theory, this is true. However, it should be remembered that the constraint graph is dynamic, i.e., edges get added to $G'$ in each iteration of the analysis. More succinctly, points-to analysis
is essentially a dynamic reachability formulation over the modified constraint graph. The set of newly added edges depends upon the current points-to sets of pointers. Without storing the points-to sets explicitly at the nodes, the analysis would be compelled to re-compute the reachability in each iteration for adding edges. To avoid this inefficiency, points-to sets for pointers are maintained at the pointer nodes throughout the analysis.

2.3 Dominators

In traditional data-flow analysis, dominators are defined as below. A node \( d \) dominates node \( n \) if all the paths from a start node \( s \) to \( n \) go through \( d \). When multiple start nodes \( s_1, s_2, ..., s_i, ... \) exist, a usual trick is to create an extra start node \( s \) and add edges from \( s \) to all \( s_i \). This trick allows us to take into account only a single start node without affecting the existing dominance relations. By definition, the dominance relation is reflexive, i.e., each node dominates itself.

In our analysis, we use a variant of the above definition which gets rid of the reflectivity property of the dominance relation. The modified definition not only makes our algorithm simpler, but is also more natural to understand and crucial to avoid special cases.

**Definition 3 (Strict Dominator).** A node \( d \) strictly dominates another node \( n \) if all paths from the start nodes \( s_i \) to \( n \) go through \( d \). A node does not strictly dominate itself.

A strict dominator is also called as proper dominator in literature [27]. A node may have zero, one or more strict dominators. Strict dominance is an irreflexive, asymmetric, but transitive relation. Efficient algorithms exist to compute dominators in a directed graph [5, 18].

Now onwards, unless mentioned otherwise, whenever we use the term dominator, it means a strict dominator.

2.4 Pointer Equivalence via Dominators

Computing dominators in a constraint graph is useful to identify pointer-equivalent variables. Two variables are pointer equivalent if they have the same points-to information. The above definition implicitly assumes that the points-to information of the two variables is considered at the fixed-point. However, in our case, since the dominator relationship across variables changes dynamically, pointer-equivalence is a dynamic relationship between pointer variables. In effect, two pointer-equivalent variables in an iteration may eventually cease to be pointer-equivalent. To the extent we know, ours is the first work that accounts for dynamic pointer equivalence and thus explores more opportunities to merge variables, resulting in a significantly improved analysis time.

We prove the following important claim which is a key to the efficiency of our points-to analysis.

**Theorem 4.** A dominator and its dominee exhibit the same points-to information.

**Proof.** The proof relies on the observation that \( \text{pointsto}(\text{dominator}) \subseteq \text{pointsto}(\text{dominee}) \) and that by definition of dominance, no other points-to information flows into the dominee. \( \Box \)

Theorem 4 enables our analysis to collapse dominator and dominee into a single node, reducing the number of variables tracked during the analysis, greatly improving the analysis efficiency.

We would like to emphasize that most of the earlier work has focused on must pointer equivalence, i.e., once two pointers are identified as pointer equivalent, they continue to be so throughout the analysis. In a sense, the must pointer equivalence is a static property of two pointers. In our analysis, since the dominator information dynamically changes, the pointer equivalence between two pointers also changes as the analysis progresses. As we illustrate using an example below and experimentally in Section 5, dynamic pointer equivalence provides us with more opportunities to identify pointer equivalent variables.

2.5 Dominator Chain

The (strict) dominance relation can be pictorially depicted by a directed dominance edge from dominator \( d \) to node \( n \). Since dominance is a transitive relation, we can easily build a chain of dominators for each node \( (d_0 \rightarrow d_1 \rightarrow d_2 \rightarrow \ldots \rightarrow d_l \rightarrow n) \). However, since two nodes may have the same dominator, the dominance relation exhibits a dominance tree. In general, the (strict) dominance graph is a forest.

Theorem 4 deals with a node and its dominator node. However, the theorem can also be applied to the dominator. Thus, Theorem 4, combined with the transitivity of dominance relation, provides us with the following important result.

**Theorem 5.** All the pointers in a dominator tree are pointer-equivalent.

Thus, an algorithm can represent a complete dominator tree using a single node and still compute the same fixed-point. Thus, the number of pointers tracked at any point during the points-to analysis is equal to the number of connected components (trees) in the dominance forest.

It is natural to ask whether the dominance relation covers all the pointer equivalence in the program, i.e., whether different connected components in the dominance forest can be pointer equivalent. It can be easily seen that two nodes in different dominance trees may have the same points-to information. An example is shown in Figure 3, wherein nodes \( p \) and \( q \) do not have a common dominator but they are pointer-equivalent.

**Definition 6 (Immediate Dominator).** A node \( d \) is the immediate dominator of node \( n \) if there exists no node \( d_2 \) such that \( d_2 \) dominates \( d \) and \( d_2 \) dominates \( n \).

The immediate dominator, when exists, is unique for a node. But several nodes may share the same immediate dominator. When a node has indegree \( = 1 \), i.e., it has a single incoming edge, then its parent is its immediate dominator.

**Definition 7 (Farthest Dominator).** A dominator \( d \) is the farthest dominator of node \( n \) if there exists no node \( d_2 \) such that \( d_2 \) dominates \( d \).

The farthest dominator, when exists, is unique for a node. But several nodes may share the same farthest dominator. The immediate and the farthest dominators for a node need not be distinct.

Dominator computation is a backward analysis, which starts from a node and traverses the graph backwards (from target to source of a directed edge) to reach its dominator. Such an analysis would first encounter the immediate dominator of a node and the farthest dominator in the end. Typically, one maintains only the immediate dominator information with each node which can then be transitively used to reach the farthest dominator. However, it
is easy to see that maintaining the farthest dominator information with each node would allow us more opportunities for identifying pointer equivalent variables. An example is shown in Figure 4.

Maintaining the farthest dominator with each node, although more opportunistic, also poses a challenge. Recall that the constraint graph is dynamic and the dominance relations change in each iteration. We would like these relations to change as infrequently as possible, to reduce the cost of (incrementally) recomputing the new dominators. If each node maintains its farthest dominator, then addition of a random edge to the graph is more likely to change the farthest dominator, compared to the case when each node maintains its immediate dominator. Thus, there exists a tension between the cost of updating dominance relations and the opportunities for identifying pointer equivalent variables.

We address this dilemma using a mixed approach. In the initial iterations, when the constraint graph changes rapidly, our analysis maintains only the immediate dominator with each node. After a certain threshold number of iterations (calculated based on the size of the input program), the analysis starts moving the dominators up the dominator chain. Thus, each node’s dominator information is updated as follows:

\[ \text{dominator}(n) := \text{dominator}(\text{dominator}(n)), \text{if exists}. \]

Further, we also prioritize the constraint evaluation [22] so that edges are added near to the start nodes in the initial iterations, in order to reduce the number of changes to the dominator information in the later iterations.

### 3. Points-to Analysis using Dominators

In this section we present our points-to analysis algorithm using (strict) dominance relation. Since dominator information is dynamic (changes once per analysis iteration), incrementally recomputing dominators is essential for an efficient points-to analysis. We first explain incremental update of dominators and then present our points-to analysis algorithm. We prove that our approach is sound and the algorithm computes the same information as an inclusion-based points-to analysis.

#### 3.1 Incremental Update of Dominators

Our constraint graph is essentially a directed acyclic graph; the cycles are collapsed by online cycle detection [24] in each iteration, before the dominators are incrementally updated. This enables us to use the incremental dominator update algorithm by Ramalingam and Reps [28] for reducible graphs. Since edges only get added and are never deleted from the constraint graph, we only need the relevant algorithm (Figure 3.4 from [28]). We only briefly mention the important steps of the algorithm below. The algorithm incrementally updates the dominator tree of the constraint graph for addition of an edge \((u, v)\).

1. Compute the cut between two subgraphs of the constraint graph, one which contains node \(u\) and is reachable from the start nodes, and another which contains node \(v\) and is reachable from \(v\). When \(v\) is already reachable from \(u\), then it becomes a special case and can be solved efficiently.

2. Compute the possibly affected set of nodes for each edge in the cut. The possibly-affected set is guaranteed to contain all the nodes for which a dominator update is required.

3. Find the least common ancestor of all the predecessors of the nodes in the subgraph induced by forward edges, i.e., edges whose target nodes do not dominate their source nodes.

4. Link the new dominator in the dominator tree.

We maintain the dominator tree using the same set of nodes in the constraint graph, but with an additional field pointing to its parent (when it exists) in the dominator tree. In addition, we maintain the following information:

- Whether an edge \((u, v)\) is a back edge, i.e., whether \(v\) dominates \(u\).
- An equivalence-class representative for each node, required for identifying pointer equivalent variables.
- Reachability information of each node from each of the start vertices. Note that this need not be separately maintained, as this is the same as the points-to information.
- A mapping from a dominator tree to its nodes, for easy enumeration of all the nodes of the tree when one is given as input. This mapping helps in updating the representatives of nodes when they are no longer in the same dominator tree.

The original algorithm for incremental update of dominator tree requires priorities assigned to the nodes for topological ordering. We do not explicitly maintain them since we maintain the incoming and outgoing edges with each node for traversing forward and backward in the constraint graph.

#### 3.2 Points-to Analysis Algorithm

In this section we present our dominator-based points-to analysis. For better understanding, we first provide its outline in Algorithm 2 and then present the detailed steps in Algorithm 3.

Algorithm 2 takes a set of points-to constraints and a set of pointer variables and for each pointer variable, computes a set of values indicating its points-to information. The algorithm first processes the address-of constraints (Line 1) and creates a modified constraint graph \(G'\) as discussed in the previous section. It then processes the copy constraints and adds directed edges corresponding to them in \(G'\) (Line 2). At this stage, at Line 3 the algorithm computes an immediate dominator for each node in \(G'\). This, in effect, creates a dominator tree (actually, a forest) over the nodes. Using the newly added edges in \(G'\), points-to information (generated due to the address-of constraints) is propagated to all the reachable nodes (Line 5). Various techniques like Wave/Deep Propagation [25] or Least Recently Fired [16] can be used to optimize the propagation. After the graph saturates, i.e., no more points-to information can be propagated, the algorithm adds more edges to \(G'\) using load/store constraints (Line 6). In Line 7, immediate dominators of the affected nodes are incrementally computed to obtain the modified dominator tree. Finally, in Line 8, the representatives of the dominator tree nodes are identified, which represent a complete dominator tree of pointer-equivalent variables. Steps in Lines 5–8 are repeated until there is no change in the points-to information of or until no more edges are added to the constraint graph, which suggests the algorithm has reached a fixed-point.
We now describe our points-to analysis algorithm in detail, as presented in Algorithm 3. Our algorithm is worklist-based, and in each iteration of the algorithm, the worklist keeps track of the nodes from which the points-to information is to be propagated to its reachable neighbors, i.e., to the target nodes of its outgoing edges.

In Lines 3–8, address-of constraints are processed and all the start nodes are added to the worklist. The for loop at Lines 9–12 processes the copy constraints and updates the worklist. Lines 13–20 compute the dominator trees and their representatives. The representative for a dominator tree is an arbitrary but fixed node in the tree. In our analysis we set it to the root of the tree. The repeat-until loop at Lines 21–56 is executed until the fixed-point of the points-to information and the number of edges in the constraint graph \( G' \). The while loop at Lines 22–32 iterates over each node \( u \) of the worklist to propagate information. The information, instead of propagating from \( u \) to its neighbors, is propagated from \( u \)'s representative to the representative of its neighbor. The correctness of this way of propagating information can be easily verified. Propagating information across representatives significantly reduces the amount of points-to information propagated, since otherwise, multiple nodes may try to propagate the same information to a node (or multiple nodes). It also makes sure that the representative of each tree is kept up-to-date with the information. Keeping the representative up-to-date is critical, since later, if a node in a tree ceases to be pointer equivalent with other nodes, the representative's points-to information is used to update the points-to information of the node’s new representative (Lines 50–54).

Specifically, when the dominator-dominee relationship between two nodes is broken, following steps are performed: (i) dominee’s representative is identified, (ii) dominee’s new dominator is computed by the incremental dominator update procedure, and (iii) points-to information is scheduled to be propagated from the original representative to the dominee, using difference propagation. The above steps ensure that the dominee retains all of its points-to information from the earlier representative. The for loop running across Lines 34–46 processes load and store constraints in a standard way and adds new edges to \( G' \). The set of newly added edges is kept track of in variable newedges. Note that edges are added between actual nodes and not between their representatives. This is because, later, when the dominator tree changes, we would want the edges to be present between nodes, rather than their representatives. Thus, our analysis adds the same set of edges as the traditional analysis, but uses a different propagation (via dominator tree representatives). The algorithm then calls a subroutine that detects and collapses cycles in the constraint graph. The subroutine does not actually delete any nodes and edges, but simply marks each node with its cycle representative. Note that a cycle representative is different from a dominator tree representative. The nested

<table>
<thead>
<tr>
<th>Iteration 1</th>
<th>Iteration 2</th>
<th>Iteration 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>Dominator</td>
<td>Node</td>
</tr>
<tr>
<td>p</td>
<td>&amp;d</td>
<td>p</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>d</td>
<td>b</td>
<td>d</td>
</tr>
<tr>
<td>z</td>
<td>q</td>
<td>z</td>
</tr>
</tbody>
</table>

Table 1. Immediate dominators in various iterations for the running example.

for loops at Lines 48–55 incrementally update the dominator trees for each newly added edge in newedges. The new dominator tree is computed by calculating the new immediate dominator of each affected node by the addition of an edge.

### 3.3 Example

When Algorithm 3 is executed on our running example from Section 2, the immediate dominators in various iterations are given in Table 1. The example reaches a fixed-point in Iteration 4 (which contains the same state of the dominator tree as in Iteration 3).

Note that the immediate dominator of \( d \) changes from \( b \) in Iteration 1 to \( a \) in Iteration 2, which illustrates the effect of immediate dominator moving up the dominator chain. Also note that the dominator-dominee relationship between \( (p, k) \) and \( (b, a) \) remains fixed throughout the analysis and can be used to reduce the number of variables tracked. Further, the newly formed dominator-dominee relationships between \( (z, q) \) and \( (y, q) \) continue to hold until fixed-point and can also be used to reduce the number of variables tracked. Thus, for instance, all the instances of \( y \) can be replaced by \( q \) or \( q \) versa, without affecting the final fixed-point. We emphasize that such dynamic relationships cannot be detected by any of the existing techniques and ours is the first approach which exploits the constraint graph structure to identify dominator-based pointer equivalence.

### 3.4 Soundness and Precision

We first prove that our dominator-based analysis is sound, i.e., it computes an over-approximation of the points-to information computed by an inclusion-based points-to analysis. We then prove that it is precise, i.e., it does not compute any additional information than that computed by an inclusion-based points-to analysis. In the proofs, we assume that both the analyses use the modified constraint graph \( G \) w.r.t. the computation of points-to information.

**Theorem 8.** Algorithm 3 is sound.

**Proof.** We prove the claim by contradiction. Let there exist a points-to fact \( f \) which is computed by an inclusion-based analysis \( I \) but not by our dominator-based analysis \( D \). Let \( i \) be the iteration in which \( f \) was first computed by \( I \), \( i > 0 \), because \( 0^{th} \) iteration corresponds to processing address-of constraints, and the processing of address-of constraints is handled in the same manner in both \( D \) and \( I \). If the fact \( f \) was not computed by \( D \) in iteration \( i \), then \( f:src \to dst \) was not propagated from \( src \) to \( dst \). However, since Lines 22–32 ensure that points-to information is propagated in the current constraint graph upto a fixed-point, \( I \) and \( D \) differ in iteration \( i \) with respect to the node \( src \) being in the worklist in \( I \) and not being in the worklist in \( D \). This suggests that the node \( src \) did not receive fact \( f \) as a new fact in iteration \( i \) of \( D \). In other words, \( src \) already had fact \( f \) in its points-to set from the previous iteration. Therefore, \( D \) and \( I \) must differ in iteration \( i – 1 \). Arguing the same way for iteration \( i – 1 \) and using the fact that \( i > 0 \), we prove
Algorithm 3: Dominator-based points-to analysis.

Require: set $C$ of points-to constraints, set $V$ of variables
Ensure: each variable in $V$ has its points-to set computed

1: worklist = \{\}
2: Initialize constraint graph $G' = (V, E)$ with $E = \phi$
3: for each address-of constraint $p = \&q \in C$ do
4:   $V = V \cup \{\&q\}$
5:   $E = E \cup (\{\&q\}, p)$
6: end for
7: pointsto($q\&q\&q$) = \{\{q\}\}
8: worklist.add($q\&q\&q$)
9: end for
10: for each copy constraint $p = q \in C$ do
11:   $E = E \cup (q, p)$
12:   worklist.add(q)
13: end for
14: for each node $v \in V$ do
15:   compute immediate dominator of $v$
16:   if $v$ is a start node then
17:     assign $v$ as its own representative
18:   else
19:     assign a randomly chosen but fixed representative within $v$'s dominator tree
20:     end if
21: end for
22: repeat
23:   while worklist is not empty do
24:     $u = \text{worklist.remove}()$
25:     $\text{representative of } u$
26:     for each $v \in \text{outgoing}(u)$ do
27:       $v = \text{representative of } v$
28:       pointsto($v\&q\&q\&q$) = pointsto($v\&q\&q\&q$) $\cup$ pointsto($q\&q\&q\&q$)
29:       if pointsto($v\&q\&q\&q$) changed then
30:         worklist.add($v\&q\&q\&q$)
31:       end if
32:     end for
33:   end while
34: end for
35: for each load or store constraint $c \in C$ do
36:   if $c$ is a load constraint $p = \ast q$ then
37:     for each $v \in \text{pointsto}(q)$ do
38:       $E = E \cup (v, p)$
39:       newedges = newedges $\cup$ $(v, p)$, if $(v, p)$ was newly added to $E$
40:     end for
41:   else if $c$ is a store constraint $p = q$ then
42:     for each $v \in \text{pointsto}(p)$ do
43:       $E = E \cup (q, v)$
44:       newedges = newedges $\cup$ $(q, v)$, if $(q, v)$ was newly added to $E$
45:     end for
46:   end if
47: end for
48: detect and eliminate cycles in $G'$ \{from [24]\}
49: for each $(u, v) \in \text{newedges}$ do
50:   incrementally update immediate dominators of $v$'s dominator tree \{Figure 3.4 from [28]\}
51: for each node $w$ in $v$'s (original) dominator tree do
52:   $\text{representative of } w$
53:   compute the new representative $r\text{new}_{\&q\&q\&q\&q}$ of $w$
54:   pointsto($r\text{new}_{\&q\&q\&q\&q}$) = pointsto($r\text{new}_{\&q\&q\&q\&q}$) $\cup$
55:   pointsto($\text{old}_{\&q\&q\&q\&q}$)
56: end for
57: worklist.add(u)
58: end for
59: until newedges is empty

That if the fact $f$ was computed by $I$, it must also be computed by $D$. This completes the proof. \qed

Theorem 9. Algorithm 3 is precise.

Proof. The proof is similar to the one for Theorem 8. Let there exist a points-to fact $f$ which is computed by our dominator-based analysis $D$ but not by an inclusion-based analysis $I$. Let $it$ be the iteration in which $f$ was first computed by $D$. Since address-of constraints are processed by both $D$ and $I$ in the same manner, $it > 0$. If the fact $f$ was additionally computed by $D$ in iteration $it$, then $f$.src $\rightarrow$ dst was additionally propagated from src to dst. Thus, $I$ and $D$ differ in iteration $it$ with respect to the node $src$ being in the worklist (Lines 22–32) in $D$ and not being in the worklist in $I$. This suggests that the node src did not receive fact $f$ as a new fact in iteration $it$ of $I$. In other words, src already had fact $f$ in its points-to set from the previous iteration. Therefore, $D$ and $I$ must differ in iteration $it − 1$. Arguing the same way for iteration $it − 1$ and using the fact that $it > 0$, we prove that if the fact $f$ was computed by $D$, it must also be computed by $I$. This completes the proof. \qed

3.5 Balancing Analysis Cost

Since the dominator information changes with each addition of an edge, it is essential to keep the cost of updating dominators to a minimum. We employ the following optimizations to reduce it.

- Dominators are updated for several edges in a batch, rather than individually for each edge. This helps in reducing the number of changes to the new dominator of a node.
- Points-to information is propagated across dominator tree representatives rather than individual nodes. This essentially propagates information across trees rather than individual edges.
- We prioritize the constraint evaluation [22] so that edges are added near to the start nodes in the initial iterations, in order to reduce the number of changes to the dominator information in the later iterations.
- When the dominator of a node $n$ changes from $d_1$ to $d_2$, instead of copying all the points-to information from $d_1$ to $d_2$, we simply maintain an additional pointer with $n$ suggesting that all of the points-to information of $d_1$ is contained in that of $n$.

3.6 Avoiding Dominator Update

It should be noted that in certain cases, dominator information of a node need not be updated. Some of these cases are costly to handle as they involve traversing a large part of the constraint graph. We list below some situations which can be quickly checked.

- If $v$ is not address-taken, and $v$ and the nodes on the path between itself and its immediate dominator do not appear as a destination of a load/store constraint.
- If $v$ is not address-taken, $v$ appears as a destination of a load $v = \ast q$ but $q$’s points to information did not change in this iteration.
- If $v$ is address-taken, but no edge is added to a node on the path between $v$ and its immediate dominator in this iteration.
- If $v$ is address-taken, but edges are added only between the nodes on the path between $v$ and its immediate dominator in this iteration.

4. Context-Sensitive Analysis

We extend the context-insensitive analysis in Algorithm 3 for context-sensitivity using an invocation graph based approach [7].
The approach readily disallows non-realizable interprocedural execution paths. The context-sensitive algorithm starts from function main and maintains a stack of function invocations, similar to the runtime. Thus, a return from a function always matches the function invocation at the top of the stack. Recursion is detected by examining the current call-chain at each function-invocation and checking if the function already exists in the call-chain. We handle recursion, which can introduce potentially unbounded number of contexts, by iterating over the cyclic call-chain and computing a fixed-point of the points-to information. Although this reduces analysis precision compared to a fixed-point of the points-to tuples. Yet, this reduces analysis time. The context-sensitive algorithm starts from function main and maintains a stack of function invocations, similar to the runtime. Thus, a return from a function always matches the function invocation at the top of the stack. Recursion is detected by examining the current call-chain at each function-invocation and checking if the function already exists in the call-chain. We handle recursion, which can introduce potentially unbounded number of contexts, by iterating over the cyclic call-chain and computing a fixed-point of the points-to information. Although this reduces analysis precision compared to a fixed-point of the points-to tuples. Yet, this reduces analysis time. The context-sensitive algorithm starts from function main and maintains a stack of function invocations, similar to the runtime. Thus, a return from a function always matches the function invocation at the top of the stack. Recursion is detected by examining the current call-chain at each function-invocation and checking if the function already exists in the call-chain. We handle recursion, which can introduce potentially unbounded number of contexts, by iterating over the cyclic call-chain and computing a fixed-point of the points-to information. Yet, this reduces analysis time. The context-sensitive algorithm starts from function main and maintains a stack of function invocations, similar to the runtime. Thus, a return from a function always matches the function invocation at the top of the stack. Recursion is detected by examining the current call-chain at each function-invocation and checking if the function already exists in the call-chain. We handle recursion, which can introduce potentially unbounded number of contexts, by iterating over the cyclic call-chain and computing a fixed-point of the points-to information. Yet, this reduces analysis time. The context-sensitive algorithm starts from function main and maintains a stack of function invocations, similar to the runtime. Thus, a return from a function always matches the function invocation at the top of the stack. Recursion is detected by examining the current call-chain at each function-invocation and checking if the function already exists in the call-chain. We handle recursion, which can introduce potentially unbounded number of contexts, by iterating over the cyclic call-chain and computing a fixed-point of the points-to information. Yet, this reduces analysis time. The context-sensitive algorithm starts from function main and maintains a stack of function invocations, similar to the runtime. Thus, a return from a function always matches the function invocation at the top of the stack. Recursion is detected by examining the current call-chain at each function-invocation and checking if the function already exists in the call-chain. We handle recursion, which can introduce potentially unbounded number of contexts, by iterating over the cyclic call-chain and computing a fixed-point of the points-to information. Yet, this reduces analysis time.

The algorithm takes four parameters: the function f to be processed, its calling context cc, the set of constraints C to be generated and the set of variables V to be created. The analysis first adds \( g, \{ \} \) to V for each global variable g where \( \{ \} \) denotes an empty context (not shown in the algorithm). It then makes the first call to the algorithm with parameters \( \{ \text{main}, \{ \text{main} \}, \text{C} = \{ \} \), \( V \)\). The procedure processes all the statements in the function and generates context-sensitive points-to constraints in C. C is later evaluated using Algorithm 3. Lines 2–5 in Algorithm 4 process memory allocation and create a new variable on encountering an alloc statement outside recursion. Lines 6–11 handle a non-recursive call. It first adds the callee to the callchain and then maps the actual arguments to the formal arguments. The algorithm recursively calls itself in Line 9 to process the invocation graph of the callee. The callee is analyzed the same way and the set of constraints \( \text{C} \) keeps getting updated. On the callee function’s return, its return value is mapped to the \( \ell \)-value in the call statement. Finally, the calling context is updated by removing the callee. A recursive call is handled in Lines 12–21 by iterating over the cyclic call chain and computing a fixed-point of the points-to information by the constraints in C-cyce. Note that the recursive call to Algorithm 4 in Line 17 uses the same callchain. The fixed-point over the constraints C-cyce generated in the cyclic call graph is then merged with C in Line 21. The corresponding context-sensitive constraints for address-of, copy, load and store statements are added in Lines 22–25. A context-sensitive constraint contains variables in a particular context. For instance, a copy constraint is of the form \( a_{1} = b_{2} \) where a and b are variables and \( c_{1} \) and \( c_{2} \) are contexts. The two sets, C and V are finally passed on to Algorithm 3 for solving. The reason for designing the analysis as a two step process (generating constraints and solving them) is to have a common constraint solving phase (with minor modifications). Thus, Algorithm 3 is used for both context-insensitive and context-sensitive analysis.

5. Experimental Evaluation

We evaluate the effectiveness of our approach using 16 SPEC C/C++ benchmarks and five large open source programs, namely httpd, sendmail, ghostscript, gdb and wine-server. The benchmark characteristics are given in Table 2. KLOC is the number of kilo lines of unprocessed source code, Total Inst is the total number of static LLVM instructions after optimizing at -O2 level, Pointer Inst is the number of static pointer-type LLVM instructions processed by the analysis and Func is the number of functions in the benchmark. The LLVM intermediate representations of SPEC 2000 benchmarks and open source programs were run using the opt tool of LLVM on an Intel Xeon machine with 2 GHz clock and 16 GB RAM running Debian GNU/Linux 5.0.

We have two implementations of our dominator-based points-to analysis: context-insensitive referred to as doms-ci and context-sensitive referred to as doms-cs. We compare doms with the following highly optimized implementations.

- anders: This is the base Andersen’s algorithm [2] which is field-insensitive, flow-insensitive and context-insensitive (anders-ci). We extend it for context-sensitivity using the same approach as for doms-cs (see Section 4) and the context-sensitive version is referred to as anders-cs. It uses sparse bitmaps to store points-to information.
- bddlcd: This is the Lazy Cycle Detection (LCD) algorithm implemented using Binary Decision Diagrams (BDD) from Hardekopf and Lin [12]. The base implementation (as downloaded from [10]) is context-insensitive (bddlcd-ci). We extend it for context-sensitivity using the same approach as for doms-cs (Section 4) and we refer to it as bddlcd-ci.
- deep: This is the context-insensitive Deep Propagation method from [25] (downloaded from [26]). This method propagates points-to information in the constraint graph to all the reachable nodes in a depth-first manner along a path, before the other paths are considered. It uses a sparse bitmap representation to store points-to sets has been shown to scale well.

We separate the following discussion for context-sensitive (cs) analyses and context-insensitive (ci) analyses.

5.1 Context-Sensitive Analysis

Table 2 shows the analysis time and memory requirement for various context-sensitive algorithms. We observe that doms-cs is the fastest of the three algorithms. Specifically, it is 39% faster than anders-cs and 88% faster than bddlcd-cs. Our result adds one more

<table>
<thead>
<tr>
<th>Algorithm 4 Context-sensitive analysis.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Require:</strong> Function f, callchain cc, constraints C, variable set V</td>
</tr>
<tr>
<td><strong>1:</strong> for all statements s ( \in { f ) do</td>
</tr>
<tr>
<td><strong>2:</strong> if a is of the form ( p = \text{alloc}() ) then</td>
</tr>
<tr>
<td><strong>3:</strong> if ( \text{inrecursion} == \text{false} )</td>
</tr>
<tr>
<td><strong>4:</strong> ( V = V \cup { p, cc } )</td>
</tr>
<tr>
<td><strong>5:</strong> end if</td>
</tr>
<tr>
<td><strong>6:</strong> else if a is of the form non-recursive call fnr then</td>
</tr>
<tr>
<td><strong>7:</strong> cc.add(fnr)</td>
</tr>
<tr>
<td><strong>8:</strong> add copy constraints to C for actual and formal arguments</td>
</tr>
<tr>
<td><strong>9:</strong> call Algorithm 4 with parameters fnr, cc, C</td>
</tr>
<tr>
<td><strong>10:</strong> add copy constraints to C for return value of fnr and ( \ell )-value in a</td>
</tr>
<tr>
<td><strong>11:</strong> cc.remove()</td>
</tr>
<tr>
<td><strong>12:</strong> else if a is of the form recursive call fnr then</td>
</tr>
<tr>
<td><strong>13:</strong> inrecursion = true</td>
</tr>
<tr>
<td><strong>14:</strong> C-cycle = {}</td>
</tr>
<tr>
<td><strong>15:</strong> repeat</td>
</tr>
<tr>
<td><strong>16:</strong> for all functions fc ( \in ) cyclic callchain do</td>
</tr>
<tr>
<td><strong>17:</strong> call Algorithm 4 with parameters fc, cc, C-cycle</td>
</tr>
<tr>
<td><strong>18:</strong> end for</td>
</tr>
<tr>
<td><strong>19:</strong> until no new constraints are added to C-cycle</td>
</tr>
<tr>
<td><strong>20:</strong> inrecursion = false</td>
</tr>
<tr>
<td><strong>21:</strong> C = C ( \cup ) C-cycle</td>
</tr>
<tr>
<td><strong>22:</strong> else if a is an address-of, copy, load, store statement then</td>
</tr>
<tr>
<td><strong>23:</strong> c = constraint(a, cc)</td>
</tr>
<tr>
<td><strong>24:</strong> C = C ( \cup ) c</td>
</tr>
<tr>
<td><strong>25:</strong> end if</td>
</tr>
<tr>
<td><strong>26:</strong> end for</td>
</tr>
</tbody>
</table>
Table 2. Benchmark characteristics and comparison with context-sensitive algorithms.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>KLOC</th>
<th># Total Inst</th>
<th># Pointer Inst</th>
<th># Func</th>
<th>Time (seconds)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>anders-cs</td>
<td>bddlcd-cs</td>
</tr>
<tr>
<td>176.gcc</td>
<td>222.185</td>
<td>328,425</td>
<td>119,384</td>
<td>1,829</td>
<td>330</td>
<td>17411</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>81.442</td>
<td>143,848</td>
<td>52,924</td>
<td>1,067</td>
<td>143</td>
<td>5880</td>
</tr>
<tr>
<td>254.gap</td>
<td>71.367</td>
<td>118,715</td>
<td>39,484</td>
<td>877</td>
<td>91</td>
<td>4726</td>
</tr>
<tr>
<td>255.vortex</td>
<td>67.216</td>
<td>75,458</td>
<td>16,114</td>
<td>963</td>
<td>94</td>
<td>2392</td>
</tr>
<tr>
<td>177.mesa</td>
<td>59.255</td>
<td>69,919</td>
<td>26,076</td>
<td>1,040</td>
<td>35</td>
<td>6355</td>
</tr>
<tr>
<td>186.crafty</td>
<td>20.657</td>
<td>28,743</td>
<td>3,467</td>
<td>136</td>
<td>129</td>
<td>3303</td>
</tr>
<tr>
<td>300.twolf</td>
<td>20.461</td>
<td>49,507</td>
<td>15,820</td>
<td>215</td>
<td>30</td>
<td>200</td>
</tr>
<tr>
<td>175.vpr</td>
<td>17.731</td>
<td>25,851</td>
<td>6,575</td>
<td>228</td>
<td>29</td>
<td>155</td>
</tr>
<tr>
<td>252.eon</td>
<td>17.679</td>
<td>126,866</td>
<td>43,617</td>
<td>1,723</td>
<td>89</td>
<td>22</td>
</tr>
<tr>
<td>188.ammp</td>
<td>13.486</td>
<td>26,199</td>
<td>6,516</td>
<td>211</td>
<td>34</td>
<td>55</td>
</tr>
<tr>
<td>197.parser</td>
<td>11.394</td>
<td>35,814</td>
<td>11,872</td>
<td>356</td>
<td>42</td>
<td>27</td>
</tr>
<tr>
<td>164.gzip</td>
<td>8.618</td>
<td>8,434</td>
<td>991</td>
<td>90</td>
<td>25</td>
<td>7</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>4.650</td>
<td>4,832</td>
<td>759</td>
<td>90</td>
<td>23</td>
<td>5</td>
</tr>
<tr>
<td>181.mcf</td>
<td>2.414</td>
<td>2,969</td>
<td>1,080</td>
<td>42</td>
<td>22</td>
<td>32</td>
</tr>
<tr>
<td>183.equate</td>
<td>1.515</td>
<td>3,029</td>
<td>985</td>
<td>40</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td>179.art</td>
<td>1.272</td>
<td>1,977</td>
<td>386</td>
<td>43</td>
<td>27</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>KLOC</th>
<th># Total Inst</th>
<th># Pointer Inst</th>
<th># Func</th>
<th>Time (seconds)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>anders-cs</td>
<td>bddlcd-cs</td>
</tr>
<tr>
<td>httpd</td>
<td>125.877</td>
<td>220,552</td>
<td>104,962</td>
<td>2,339</td>
<td>225</td>
<td>47</td>
</tr>
<tr>
<td>sendmail</td>
<td>113.264</td>
<td>171,413</td>
<td>57,424</td>
<td>1,005</td>
<td>173</td>
<td>118</td>
</tr>
<tr>
<td>ghostscript</td>
<td>438.204</td>
<td>906,398</td>
<td>488,998</td>
<td>6,991</td>
<td>4384</td>
<td>20613</td>
</tr>
<tr>
<td>gdb</td>
<td>474.591</td>
<td>576,624</td>
<td>362,171</td>
<td>7,127</td>
<td>9338</td>
<td>24872</td>
</tr>
<tr>
<td>wine-server</td>
<td>178.592</td>
<td>110,785</td>
<td>66,501</td>
<td>2,105</td>
<td>201</td>
<td>37</td>
</tr>
</tbody>
</table>

| average  | 92.946 | 145,874     | 67,910         | 1,358  | 738           | 3693        | 451        | 1035      | 918       | 862     |

data point to the analysis time efficiency of bitmaps versus BDDs: using sparse bitmaps is much faster than accessing BDDs. In our experience, BDDs are well suited for reducing the storage requirement, but the complex logic in enumerating and updating points-to information of pointers is significantly costly in terms of analysis time.

In terms of memory, the BDD-based implementation bddlcd-cs beats our highly optimized anders-cs. However, as a pleasant surprise, doms-cs consumes lesser memory than bddlcd-cs by a non-trivial margin (918 MB versus 862 MB). The larger saving in memory occurs due to detection of more dynamic pointer-equivalences compared to other two methods (see Section 5.3). Although doms-cs maintains additional information (like immediate dominators and reachability information), the memory benefits significantly outweigh the costs. This helps our analysis reduce not only the propagation time, but also the number of copies of the points-to sets across variables, since pointer-equivalent variables share only a single copy of points-to information.

5.2 Context-insensitive Analysis

We compare the performance of our context-insensitive dominator-based points-to analysis against Andersen’s Analysis anders-cs [2], BDD-based Lazy Cycle Detection bddlc-cs [12] and Deep Propagation deep-cs [25], in Table 3. From the results, it is clear that doms-cs is almost 4×, 3×, and more than 2× faster than anders-cs, bddlc-cs, and deep-cs, respectively. This happens mainly due to the reduction in the number of propagations of points-to sets in the constraint graph by the detection of dynamic pointer-equivalent variables. For the same reason, the memory requirement of doms-cs is also relatively smaller than that of anders-cs and deep-cs. However, in terms of memory requirement, bddlc-cs performs the best for context-insensitive analysis. Our analysis doms-cs requires 63% more memory than bddlc-cs. For smaller programs, the memory reduction by identifying pointer equivalence is offset by the additional memory requirement for storing auxiliary information like immediate dominators, the reachability information and the bookkeeping for incremental dominator update. However, it is interesting to see that the additional memory required by doms-cs goes on reducing with the increasing program size. Specifically, for the two largest benchmarks in our suite, ghostscript and gdb, the memory requirements of both bddlc-cs and doms-cs are almost the same. This suggests that the benefit of identifying pointer-equivalence outweighs the cost of additional bookkeeping especially for larger programs. This is evident from the (lesser) memory requirement of doms-cs in case of context-sensitive analysis (see Section 5.1).

In summary, our dominator-based points-to analysis offers significant performance benefits over the state-of-the-art methods.

5.3 Constraint Graph Statistics

We now present our results on applying the dynamic pointer equivalence method on the constraint graph. Figure 5 shows the percentage of the pointer-equivalent variables detected by various methods for our suite of benchmarks. Total number of pointer-equivalent variables (i.e., 100%) is calculated by a separate analysis that exhaustively checks for pointer equivalent variables, i.e., pointers with the same points-to set, in the constraint graph and computes the number of pairs of such pointer equivalent variables. Then, the percentage of pointer-equivalent variables detected by a method is calculated as

\[
\text{Percentage} = \frac{\text{no. of pointer-equivalent pairs detected by the method}}{\text{no. of pointer-equivalent pairs present in constraint graph}} \times 100
\]

Offline Variable Substitution (ovs) [9] is able to detect only 19% of the total pointer equivalence. Hash-based Value Numbering (HVN) with deReference and Union (hvu) is a powerful offline optimization technique for detecting pointer-equivalent variables [11], and it detects a superset of that detected by ovs. It is also able to detect only a quarter of the actual available pointer equivalence. Both these methods are offline, i.e., they are executed prior to running pointer analysis. Online cycle detection (ocd) [24] is an online method that periodically identifies and collapses cycles on the fly, while points-to analysis is in progress. We implemented the algorithm with the cycle detection done in every analysis iteration (same as in Algorithm 3). We observe that ocd is able to detect more
than half of pointer equivalence. However, it fails to detect a significant portion of pointer equivalence in the program. This happens not because of the frequency of cycle detection, but because cycle detection cannot exploit the constraint graph structure beyond a strongly connected component. When we combine hru + ocd, their combined effect is only 58%. Our observations about hru are in close agreement with those mentioned by Hardekopf and Lin [11]. Our results support our thesis that a combination of offline techniques and online cycle detection have inherent limitation in detecting the pointer equivalence prevalent in programs.

Running our Algorithm 3 improves the pointer equivalence detection percentage to almost 70% (ocd + doms). doms alone is able to detect additional 15% of the pointer-equivalence. When combined with an offline analysis phase of hru, the resultant analysis hru + ocd + doms is able to detect almost 75% of the total pointer equivalence. This suggests that existing methods can be combined with our dominator-based analysis for significant benefits.

We observe that around 25% of the pointer-equivalences are not detected by our algorithm. This happens because our algorithm does not try to detect all the variables that are reachable from the same start nodes. Two nodes in the constraint graph that do not have a common dominator nor are in a cycle can still be pointer equivalent if they are reachable from the same set of start (address-of) nodes (see Figure 3). Detecting such nodes is costly in our experience and it reduces the benefits of our optimizations.

In summary, the dominator-based pointer-equivalence algorithm is able to detect significantly larger number of pointer equivalences compared to the previous methods.

6. Related Work

Surveys on pointer analysis techniques are presented by Hind and Pioli [14] and Nasre [20].

Inclusion based algorithms incur cubic computational complexity. A naive implementation of Andersen’s analysis turns out to be inefficient in practice. Therefore, several novel techniques have been developed to improve upon the original Andersen’s analysis [3, 13, 19, 32]. Binary Decision Diagrams (BDD) [3, 32] are used to store points-to information in a succinct manner. Although the space reduction using BDD is significant, it also incurs a performance penalty over sparse bitmaps, since accessing and merging points-to information involve a complex logic. The idea of bootstrapping [15] uses a divide-and-conquer strategy to first divide the large problem of pointer analysis by partitioning the set of pointers into disjoint alias sets using a fast and less precise algorithm (e.g., [31]) and later, a more precise algorithm analyzes each partition. Due to the small partition sizes, the overall analysis scales well with the program size. The analysis over the alias partitions can be done in parallel. Nasre et al. [21] convert points-to constraints into a set of linear equations and solve it using a standard linear solver. Storing complete calling context information achieves a good precision, but at the cost of storage and analysis time. For a complete context-sensitive analysis, potentially, the storage requirement and the analysis time can be exponential in the number of functions in the program making it non-scalable. Therefore, approximate representations have been introduced to trade off precision for scalability. Das [6] proposed one level flow while Lattner et al. [17]

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (seconds)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>anders-ci</td>
<td>bddlci-ci</td>
</tr>
<tr>
<td>176.gcc</td>
<td>151.618</td>
<td>5.154</td>
</tr>
<tr>
<td>253.perlbench</td>
<td>65.969</td>
<td>3.078</td>
</tr>
<tr>
<td>254.gap</td>
<td>1.457</td>
<td>2.282</td>
</tr>
<tr>
<td>255.vortex</td>
<td>29.625</td>
<td>2.339</td>
</tr>
<tr>
<td>177.mesa</td>
<td>0.831</td>
<td>2.020</td>
</tr>
<tr>
<td>186.crafty</td>
<td>6.889</td>
<td>2.527</td>
</tr>
<tr>
<td>300.twolf</td>
<td>0.465</td>
<td>1.290</td>
</tr>
<tr>
<td>175.vpr</td>
<td>0.453</td>
<td>1.528</td>
</tr>
<tr>
<td>252.eon</td>
<td>1.029</td>
<td>2.231</td>
</tr>
<tr>
<td>188.ammp</td>
<td>0.372</td>
<td>1.347</td>
</tr>
<tr>
<td>197.parser</td>
<td>0.614</td>
<td>2.056</td>
</tr>
<tr>
<td>164.gzip</td>
<td>0.221</td>
<td>0.955</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>0.199</td>
<td>0.889</td>
</tr>
<tr>
<td>181.mcf</td>
<td>0.175</td>
<td>1.228</td>
</tr>
<tr>
<td>183.equake</td>
<td>0.176</td>
<td>0.856</td>
</tr>
<tr>
<td>179.art</td>
<td>0.167</td>
<td>0.643</td>
</tr>
<tr>
<td>httpd</td>
<td>58.624</td>
<td>1.856</td>
</tr>
<tr>
<td>sendmail</td>
<td>37.276</td>
<td>1.521</td>
</tr>
<tr>
<td>ghostscript</td>
<td>425.362</td>
<td>343.579</td>
</tr>
<tr>
<td>gdb</td>
<td>852.622</td>
<td>758.473</td>
</tr>
<tr>
<td>wine-server</td>
<td>62.545</td>
<td>45.512</td>
</tr>
<tr>
<td>average</td>
<td>80.785</td>
<td>56.235</td>
</tr>
</tbody>
</table>

Table 3. Comparison with context-insensitive algorithms.
unified contexts while Nasre et al. [23] hashed contexts to alleviate the need to store the complete context information.

Inclusion based analysis can also be improved using several novel enhancements proposed in literature. Online cycle elimination [8] breaks dependence cycles amongst pointer variables on the fly. Offline variable substitution [29] operates over constraints prior to the constraint evaluation to find out pointer equivalent variables and rewrites constraints with the reduced set of variables to improve the analysis time. Hardekopf and Lin [11] provide a suite of offline analyses based on Hash-based Value Numbering to further improve the effectiveness of offline methods.

Wave and Deep Propagation techniques [25] perform a breadth-wise and depth-wise propagation of points-to information in a constraint graph. Various techniques proposed for worklist management [16] also identify heuristics to reach the fixed-point faster. Prioritized constraint evaluation [22] dynamically orders constraints to produce useful edges early in the constraint graph. Although all of these techniques work on the constraint graph, they are orthogonal to our approach. Our dominator based technique is more comprehensive and provides more opportunities to identify pointer equivalent variables.

However, dominators have been extensively used in other data-flow analysis of programs. Cytron et al. [5] introduced the notion of dominance frontiers to efficiently compute static single assignment (SSA) form and control dependence graph of programs, and showed that their storage requirement is usually linear in the program size. Burke and Torczon [4] used dominators to avoid unnecessary recompilation of modules after a change to the source code. They described several methods to reduce the number of recompiations based on the trade-off between compilation time and the number of spurious recompiations. Agrawal and Horgan [1] reduced the cost of dynamic slicing by using dominators in the program dependence graphs. Ours is the first work that uses dominators for points-to analysis.

7. Conclusion

We defined a new type of dynamic pointer equivalence in this work based on the notion of dominators. Based on this notion, we developed a new context-sensitive points-to analysis which exploits the structure of the constraint graph to improve opportunities to detect more pointer-equivalent variables. We showed that the algorithm is sound and as precise as an inclusion-based analysis. We argued that detecting dominator-based pointer equivalence is critical for improving the efficiency of pointer analysis. Using a suite of benchmarks, we showed that our algorithm performs significantly better than the state-of-the-art methods. We believe that dominator-based points-to analysis will find applications in several optimizations.

References


