Wave Propagation and Deep Propagation for Pointer Analysis

Fernando Magno Quintão Pereira
UCLA
fernando@cs.ucla.edu

Daniel Berlin
Google DC
dannyb@google.com

Abstract

This paper describes two new algorithms for solving inclusion based points-to analysis. The first algorithm, the Wave Propagation Method, is a modified version of an early technique presented by Pearce et al.*; however, it greatly improves on the running time of its predecessor. The second algorithm, the Deep Propagation Method, is a more light-weighted analysis, that requires less memory. We have compared these algorithms with three state-of-the-art techniques by Hardekopf-Lin, Heintze-Tardieu and Pearce-Kelly-Hankin. Our experiments show that Deep Propagation has the best average execution time across a suite of 17 well-known benchmarks, the lowest memory requirements in absolute numbers, and the fastest absolute times for benchmarks under 100,000 lines of code. The memory-hungry Wave Propagation has the fastest absolute running times in a memory rich execution environment, matching the speed of the best known points-to analysis algorithms in large benchmarks.*

1. Introduction

Two variables are said to alias if they address overlapping storage locations. Aliasing is a key trait in many imperative programming languages such as C, C++ and Java, and it is used, for instance, to avoid copying entire data structures during parameter passing in function calls. Although a powerful feature, aliasing comes with a price: it makes it hard for compilers to reason about programs, and it may hinder many potential optimizations, such as partial redundancy elimination [9]. The traditional solution adopted by compilers to deal with this problem is alias analysis. This type of analysis provides to the optimizing compiler information about which memory locations may alias, which locations will never alias, and which locations must always alias. Although precise alias analysis is a NP-complete problem [8], compilers can use imprecise results with great benefits [7]. The most aggressive compiler optimizations tend to require whole program analysis, and one of the biggest challenges of this decade has been scaling such analyses for large programs [3], [5], [7].

In this paper we present two new algorithms for Andersen style [1] inclusion based points-to analysis. The first is called the *Wave Propagation* method. This algorithm is an evolution of the technique introduced by Pearce *et al* [13, Fig.3], and it greatly improves on the overall running time, predictability and scalability of its predecessor. The second algorithm is named the Deep Propagation method. It presents very small overhead when compared to other points-to solvers, in terms of memory usage and preprocessing time. This makes this algorithm an attractive option for analyzing small to average size programs with up to 100K lines of code. Both algorithms rely on elegant invariants that simplify their design and make them competitive with state-of-the-art solvers already described in the literature.

In the next section we describe points-to analysis with greater detail and touch some related works. In Section 3 we introduce the wave propagation method, and in Section 4 we discuss the deep propagation technique. Section 5 describes experiments supporting both algorithms and Section 6 concludes this paper and indicates future research directions.

2. Background

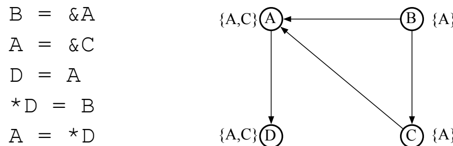
There are different types of pointer analysis with regard to flow and context sensitiveness. Flow insensitive algorithms [3], [5], [7] ignore the order of statements in a program, contrary to flow sensitive analyses [2], [18]. Context sensitive analyses distinguish the different calling contexts of a function [17]. Flow and context insensitive analyses are further divided between inclusion based and unification based. The former variation, when facing an assignment such as $a = b$, assumes that the locations pointed by b are a subset of the locations pointed by a. The unification based analyses, in which the Steensgard's Algorithm [15] is the most famous representative, assume that the locations pointed by both variables are the same; thus, trading precision by speed. Although flow and context sensitive analyses produce more precise results, for many purposes the accuracy provided by the flow and context insensitive analyses is regarded as sufficient. For instance, popular compilers such as the Gnu C Compiler (GCC) and LLVM [10] use inclusion based flow and context insensitive analyses. The algorithms provided in this paper fit in this category.

Andersen's dissertation [1] contains one of the first descriptions of the inclusion based points-to analysis problem,

which he specifies using typing rules. This seminal work has inspired research in many different directions. Later works have attempted to improve the precision of Andersen's analysis or, as in our case, have attempted to speed up the algorithms used to solve constraint sets. Constraints are derived from statements involving variable assignment or parameter passing in the program that is being analyzed. There are basically four types of constraints, which are enumerated in the table below, taken from [5]:

Statement	Name	Constraint
`a = &b`	Base	$a \supseteq \{b\}$
`a = b`	Simple	$a \supseteq b$
`a = *b`	Complex 1	$a \supseteq *b$
`*a = b`	Complex 2	$*a \supseteq b$

Complex constraints represent variable dereferencing. A constraint such as $a \supseteq *b$ signifies that for any variable v, if v is in the points-to set of b, then the points-to set of v is a subset of the points-to set of a. The analogous $*a \supseteq b$ signifies that for any variable v, if v is in the points-to set of a, then the points-to set of b is a subset of the points-to set of v. The input of the Andersen style points-to analysis problem is a collection of constraints. The output is a conservative assignment of variables to point-to set that satisfies the constraints. Solving the points-to problem amounts to computing the transitive closure of the *constraint graph*. This graph has one vertex for each variable in the constraint set, and it has one edge connecting variable v to variable u if the points-to set of v is a subset of the points to set of u. In the figure below we show a simple program, and its constraint graph, augmented with a solution to the points-to problem.



These constraints are normally solved iteratively: complex constraints cause new edges to be added to the constraint graph, forcing points to be propagated across nodes. The process is repeated until no more changes are detected. By the end of the nineties, it was clear that the identification of cycles was an essential requirement for scaling points-to analysis. All the nodes in a cycle are guaranteed to have the same points-to set, and thus they can be collapsed together. Fahndrich *et al.* [3] designed an algorithm that detects and removes cycles while complex constraints are being processed. Since then, many new algorithms have been proposed. Heintze and Tardieu [7] describe an algorithm that can analyze C programs with over one million lines of code in a few seconds. Pearce *et al.* have also introduced important contributions to this field [12], [13]. Finally, in 2007 Hardekopf and Lin presented two techniques that considerably improve the state-of-the-art solvers: Lazy Cycle Detection and Hybrid Cycle Detection [5]. In addition to on-line cycle detection, points-to analyses rely on preprocessing of constraints for scalability. Two important preprocessing methods are Off-Line Variable Substitution [14], and the HVN family of algorithms [6]. Both the on-line and off-line techniques have seen large use in actual production compilers. The algorithms designed by Pearce *et al.* [12], [13] constitute the core of GCC's points-to solver. This compiler also employs off-line cycle detection [5] and variable substitution [14] plus the HU algorithm described in [6]. The points-to solver used in LLVM was implemented after [5]. In this paper we compare our algorithms with well tuned implementations of [5], [7] and [13].

3. Wave Propagation

The wave propagation method is a modification of the algorithm proposed by Pearce *et al* in [13, Fig.3]. Our algorithm detaches from the original technique by separating the insertion of new edges in the constraint graph and the propagation of points-to sets. The propagation of points-to sets, which we call *Wave Propagation*, takes place in an acyclic constraint graph. The absence of cycles allows us to propagate points-to information in topological order, so that only set differences need to be propagated. Once this phase finishes, we have the invariant that the points-to set of a node v includes the points-to sets of all the nodes n that precede v in the constraint graph. These three phases - collapsing of cycles, points-to propagation and insertion of new edges - are repeated until no more changes are detected in the constraint graph, as shown in Algorithm 1.

Algorithm 1 The Wave Propagation Method. **Input**: a Constraint Graph $G = (V, E)$. **Output**: a mapping of nodes to points-to sets.

1: **repeat**
2: changed $\leftarrow$ false
3: Collapse Strongly Connected Components in G (Algorithm 2)
4: Perform Wave Propagation in G (Algorithm 4)
5: Add new edges to G (Algorithm 5)
6: **if** a new edge has been added to G **then**
7: changed $\leftarrow$ true
8: **end if**
9: **until** changed = False

The first part of Algorithm 1 consists in finding and collapsing the nodes of the constraint graph that are part of cycles. Following previous algorithms [5], [13], we use Nuutila's [11] approach for finding strongly connected components, which is an improvement on an earlier algorithm proposed by Tarjan et al [16]. This method runs in linear

Algorithm 2 Collapse the Strongly Connected Components of G. **Input**: a constraint graph $G = (V, E)$.

Ensure: G is acyclic after nodes have been visited and SCC components have been collapsed.

```
1: I ← 0
2: for all v such that D(v) = ⊥ do
3:    visit node v (Algorithm 3)
4: end for
5: for all v such that R(v) ≠ v do
6:    unify(v, R(v))
7: end for
```

Algorithm 3 Visit node v. **Input**: a constraint node v.

```
 1: I ← I + 1
 2: D(v) ← I
 3: R(v) ← v
 4: for all w such that (v, w) ∈ E do
 5:    if D(w) = ⊥ then
 6:       visit node w
 7:    end if
 8:    if w ∉ C then
 9:       R(v) ← (D(R(v)) < D(R(w))) ? R(v) : R(w)
10:    end if
11: end for
12: if R(v) = v then
13:    C ← C ∪ {v}
14:    while S ≠ ∅ do
15:       let w be the node on the top of S
16:       if D(w) ≤ D(v) then
17:          break
18:       else
19:          remove top from S
20:          C ← C ∪ {w}
21:          R(w) ← v
22:       end if
23:    end while
24:    push v into T
25: else
26:    push v into S
27: end if
```

time on the number of edges in the constraint graph and, as pointed by Pearce *et al* [13], it has the beneficial side effect of producing a topological ordering of the target graph for free. The pseudo-code for this phase is shown in Algorithms 2 and 3. We use the following data structures:

- $\mathcal{D}$: **map of** V **to** $\{1, \ldots, |V|\} \cup \perp$, associates the nodes in V to the order in which they are visited by Nuutila's algorithm. Initially, $\mathcal{D}(v) = \perp$.

- $\mathcal{R}$: **map of** V **to** V, associates each node in a cycle to the representative of that cycle. Initially $\mathcal{R}(v) = v$.

- $\mathcal{C}$: **subset of** V, holds the nodes that are part of a known strongly connected component. Initially $C = \emptyset$.

- $\mathcal{S}$: **stack of** V, holds the nodes that are in a cycle, but have not yet been inserted into $\mathcal{C}$. Initially empty.

- $\mathcal{T}$: **stack of** V, holds the nodes of V that are representatives of strongly connected components. $\mathcal{T}$ keeps the nodes in topological order, that is, the top node has no predecessors. Initially empty.

After collapsing cycles we perform one round of wave propagation, which, for any node v, sends the points-to set of v to all its neighbors. If the constraint graph is acyclic, and the order of propagations is the topological ordering of the nodes, then we guarantee that any node w reachable from a node v will contain the points-to-set of node v. Fortunately, the topological ordering of the constraint nodes, stored in the stack $\mathcal{T}$, is a byproduct of Nuutila's technique (see line 24 of Algorithm 3). The wave propagation phase is detailed in Algorithm 4. Our algorithm stores two points-to sets for each node v. The first set, $P_{cur}(v)$, is the current points-to set of v, and, once Algorithm 1 stops iterating, it is the result of the pointer analysis for v. The second set, $P_{old}(v)$, holds the points-to information that has been sent from v since the last iteration of the wave propagation. We keep track of $P_{old}(v)$ to avoid re-sending the whole current points-to set of v during each iteration of our algorithm. We maintain the invariant that $P_{old}(u) \subseteq P_{cur}(v)$, for any edge (u, v) in the constraint graph.

Algorithm 4 Perform wave propagation in G. **Input**: a constraint graph $G = (V, E)$.

Require: G is acyclic.
Ensure: $P_{cur}(v) \subseteq P_{cur}(w)$ if w is reachable from v.

```
1: while T ≠ ∅ do
2:    v ← pop node on top of T
3:    P_dif ← P_cur(v) − P_old(v)
4:    P_old(v) ← P_cur(v)
5:    for all w such that (v, w) ∈ E do
6:       P_cur(w) ← P_cur(w) ∪ P_dif
7:    end for
8: end while
```

We add new edges to the constraint graph in the last phase of the proposed algorithm. New edges are added due to the evaluation of complex constraints. This step is illustrated in Algorithm 5. We keep track of $P_{cache}(c)$, the last collection of points used in the evaluation of complex constraint c. This optimization reduces the number of edges that must be checked for inclusion in G. If an edge (u, v) is added to the graph, then we must copy $P_{old}(u)$ into $P_{cur}(v)$, to maintain the invariant discussed in the previous paragraph.

We will use the program in Figure 1 to illustrate the algorithms presented in this paper. The order in which statements are executed is not important to us, because our analysis is flow insensitive. Figure 2 outlines the first iteration of Algorithm 1 on that program. Only the current points-to set of each node is shown. During the search for strongly

Algorithm 5 Add new edges to G. **Input**: a constraint graph $G = (V, E)$, a list of constraints $c_1, c_2, \ldots, c_m$.

1: **for all** Constraint $c = l \supseteq *r$ **do**
2: $\quad P_{new} \leftarrow P_{cur}(r) - P_{cache}(c)$
3: $\quad P_{cache}(c) \leftarrow P_{cache}(c) \cup P_{new}$
4: $\quad$ **for all** $v \in P_{new}$ **do**
5: $\quad\quad$ **if** $(v, l) \notin E$ **then**
6: $\quad\quad\quad E \leftarrow E \cup \{(v, l)\}$
7: $\quad\quad\quad P_{cur}(l) \leftarrow P_{cur}(l) \cup P_{old}(v)$
8: $\quad\quad$ **end if**
9: $\quad$ **end for**
10: **end for**
11: **for all** Constraint $c = *l \supseteq r$ **do**
12: $\quad P_{new} \leftarrow P_{cur}(l) - P_{cache}(c)$
13: $\quad P_{cache}(c) \leftarrow P_{cache}(c) \cup P_{new}$
14: $\quad$ **for all** $v \in P_{new}$ **do**
15: $\quad\quad$ **if** $(r, v) \notin E$ **then**
16: $\quad\quad\quad E \leftarrow E \cup \{(r, v)\}$
17: $\quad\quad\quad P_{cur}(v) \leftarrow P_{cur}(v) \cup P_{old}(r)$
18: $\quad\quad$ **end if**
19: $\quad$ **end for**
20: **end for**

```
H = &C        E = &G        B = C
H = &G        H = A         C = B
A = &E        F = D         B = A
D = *H        *E = F        F = &A
```

Figure 1. The example program.

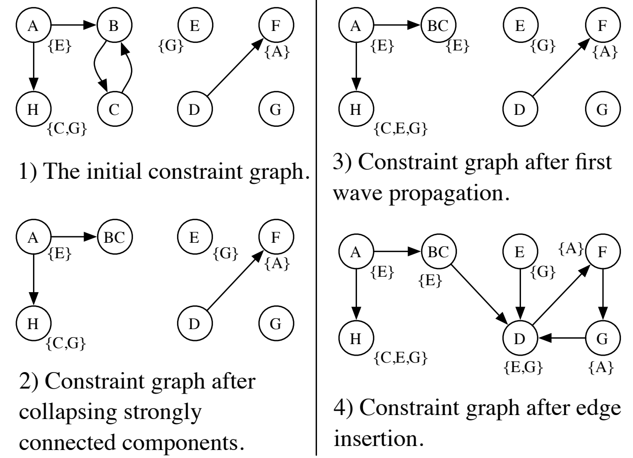


Figure 2. One iteration of the wave propagation method.

connected components in Algorithm 2, the cycle formed by nodes B and C is collapsed into a single node. In the following step, e.g: Algorithm 4, we propagate the points-to sets across the acyclic constraint graph. Finally, Algorithm 5 creates the new edges produced by the constraints $D = *H$ and $*E = F$. The analysis will finish in the next iteration, which we omit from the example.

Complexity Analysis The collapsing of strongly connected components is $O(V^2)$ for dense graphs, where V is the number of nodes in the constraint graph. The wave propagation phase and the insertion of new edges are $O(V^3)$.

4. Deep Propagation

The wave propagation method is very memory intensive: it keeps a cache of points-to information already processed for both nodes and constraints. The Deep Propagation method addresses this shortcoming. This new algorithm maintains the invariant that, if a node w is reachable from a node v, then the points-to set of w contains the points-to set of v. This condition is true after the collapsing of strongly connected components followed by the wave propagation step discussed in the previous section, and that is the starting point for the deep propagation approach, as shown in Algorithm 6. Notice that in the algorithm presented in Section 3 this property holds after a round of wave propagation, but it is no longer true after the insertion of new edges performed by Algorithm 5.

Algorithm 6 The Deep Propagation Points-to Solver. **Input**: a Constraint Graph $G = (V, E)$. **Output**: a mapping of nodes to points-to sets.

1: Collapse Strongly Connected Components in G (Algorithm 2).
2: Perform Wave Propagation in G (Algorithm 4).
3: **repeat**
4: $\quad$ changed $\leftarrow$ false
5: $\quad$ **for all** Constraint $c = l \supseteq *r$ **do**
6: $\quad\quad P_{new\ pts} \leftarrow \emptyset$
7: $\quad\quad P_{new\ edges} \leftarrow P_{cur}(r) - P_{cache}(c)$
8: $\quad\quad P_{cache}(c) \leftarrow P_{cache}(c) \cup P_{new\ edges}$
9: $\quad\quad$ **for all** $v \in P_{new\ edges}$ **do**
10: $\quad\quad\quad$ **if** $(v, l) \notin E$ **then**
11: $\quad\quad\quad\quad E \leftarrow E \cup \{(v, l)\}$
12: $\quad\quad\quad\quad P_{new\ pts} \leftarrow P_{new\ pts} \cup P_{cur}(v)$
13: $\quad\quad\quad$ **end if**
14: $\quad\quad$ **end for**
15: $\quad\quad P_{dif} \leftarrow P_{new\ pts} - P_{cur}(l)$
16: $\quad\quad$ Deep propagate P_{dif} from l with stop point l
17: $\quad\quad$ Unmark black nodes and unify gray nodes with l
18: $\quad$ **end for**
19: $\quad$ **for all** Constraint $c = *l \supseteq r$ **do**
20: $\quad\quad P_{new\ edges} \leftarrow P_{cur}(l) - P_{cache}(c)$
21: $\quad\quad P_{cache}(c) \leftarrow P_{cache}(c) \cup P_{new\ edges}$
22: $\quad\quad$ **for all** $v \in P_{new\ edges}$ **do**
23: $\quad\quad\quad$ **if** $(r, v) \notin E$ **then**
24: $\quad\quad\quad\quad E \leftarrow E \cup \{(r, v)\}$
25: $\quad\quad\quad\quad P_{dif} \leftarrow P_{cur}(r) - P_{cur}(v)$
26: $\quad\quad\quad\quad$ deep propagate P_{dif} from v with stop point r
27: $\quad\quad\quad$ **end if**
28: $\quad\quad$ **end for**
29: $\quad\quad$ unmark black nodes and unify gray nodes with r
30: $\quad$ **end for**
31: **until** changed = False

In algorithm 6 we compute, for each complex constraint, the set of nodes that must be *deep propagated* through the constraint graph. The deep propagation means that, given a starting node v, and a points-to set P_{dif}, we will add P_{dif} to the points-to set of v, and also to the points-to set of every node reachable from v in the constraint graph. Algorithm 6 is divided in two parts. The first part, shown in lines 5 to 18, handles complex 1 constraints. Given a constraint such as $l \supseteq *r$, our algorithm computes the new points-to set P_{dif} that must be propagated from node l. The poinst-to set of every variable v recently added to the points-to set of node r contributes to P_{dif}. However, due to our invariant, nodes reachable from l already contain l's current points-to set; thus, we can remove $P_{cur}(l)$ from P_{dif} in line 15 of our algorithm, before the deep propagation begins. The second part of our algorithm, given in lines 19 to 30, handles complex 2 constraints such as $*l \supseteq r$. We must deep propagate to each node v recently added to the points-to set of l every node in the points-to set of r that is not already present in the points-to set of v. As in the wave propagation method, we keep the points-to set $P_{cache}(c)$ of nodes processed for each complex constraint c, to avoid dealing with edges already added to the constraint graph.

The core of deep propagation is the recursive procedure detailed in Algorithm 7. This procedure receives three parameters: a node v, a points-to set P_{dif} and a node s, which is called the *stop point*. The objective of the deep propagation is to guarantee that the set P_{dif} be part of the points-to set of every node reachable from v. However, not every node reachable from v needs to be visited by the deep propagation routine: this traversal can stop if a node that already contains P_{dif} is visited, due to our invariant. This invariant also allow us to reduce the size of P_{dif} during successive calls of the deep propagation method, as we do in line 6 of Algorithm 7. Because this difference is computed on the fly during deep propagation, we do not have to keep the P_{old} sets used in the algorithm from Section 3. The node called stop point is used to identify cycles. As we observe in lines 16 and 26 of Algorithm 6, this is the node where the deep propagation effectively starts. If the stop point is ever reached by a recursive call of deep propagation, then we know that a cycle has been found.

Set operations such as those executed in lines 6 and 8 of Algorithm 7 are relatively expensive - they are linear on the number of nodes in the constraint graph, and we would like to dodge them as much as possible. Therefore, in order to avoid testing if P_{dif} is already part of the current points-to set of a node, we mark the nodes already visited by the deep propagation traversal with one of two colors: *black* or *gray*. A node v is marked gray if there is a path from v to the stop point, otherwise v is marked black. Set operations are applied only to uncolored nodes. Figure 3 illustrates two iterations of the deep propagation routine.

Complexity Analysis One call of Algorithm 7 executes two linear set operations; thus it is $O(V)$. Each of the $O(V)$ nodes in the constraint graph is visited by Algorithm 7 at most $O(V)$ times, that is, as long as its points-to set can be augmented. Therefore, the final complexity of this algorithm is $O(V^3)$.

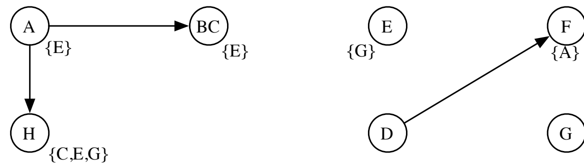
Algorithm 7 The Deep Propagation Routine. **Input**: the point-to set P_{dif} that must be propagated, the node v that is been visited and the stop point s. **Output**: true if stop point s is reachable from v, and false otherwise.

Require: $P_{cur}(v) \subseteq P_{cur}(w)$ if w is reachable from v.
Ensure: $P_{cur}(v) \subseteq P_{cur}(w)$ if w is reachable from v.
1: **if** v is gray **then**
2:    **return** True
3: **else if** v is black **then**
4:    **return** False
5: **end if**
6: $P_{new} \leftarrow P_{dif} - P_{cur}(v)$
7: **if** $P_{new} \neq \emptyset$ **then**
8:    $P_{cur}(v) \leftarrow P_{dif} \cup P_{new}$
9:    changed $\leftarrow$ True
10:    **for all** w such that $(v, w) \in E$ **do**
11:       **if** $w = s$ **or** deep propagate P_{new} from w with stop point s returns true **then**
12:          mark v gray
13:          **return** True
14:       **else**
15:          mark v black
16:          **return** False
17:       **end if**
18:    **end for**
19: **else if** $(v, s) \in E$ **then**
20:    mark v gray
21:    **return** True
22: **else**
23:    mark v black
24:    **return** False
25: **end if**

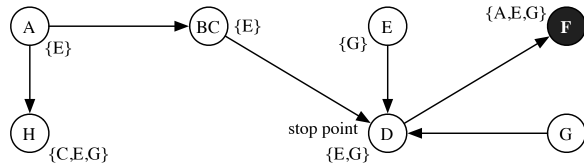
5. Experimental Results

We have performed a number of experiments in order to verify the efficiency of the proposed algorithms. We have run tests in two different machines. The first is a 2.4GHz Intel Core 2 Duo computer running Mac OS X version 10.5.4, with 2 GB of SDRAM. The other machine is a 1GHz four processors Dual-Core AMD Opteron(tm) running Linux Ubuntu 6.06.2, with 8G of memory. We compare our algorithm with the three inclusion based solvers implemented by Ben Hardekopf and previously used in [5]. The algorithms that we have used, all freely available on-line, are listed below:

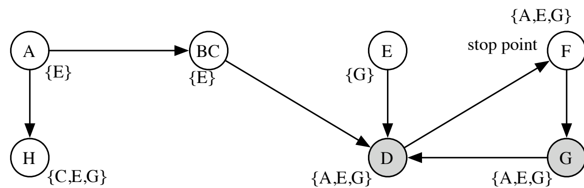
- *Lazy Cycle Detection* [5] (LCD): this is the most scalable algorithm for solving inclusion based points-to analysis. Basically, this algorithm searches for cycles every time it detects that the points-to set of a node v equals the points-to set of one of its successor nodes w.



Acyclic constraint graph after initial wave propagation.

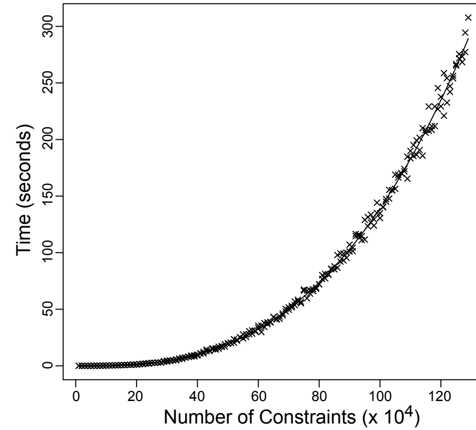


Constraint graph after processing D = *H, and deep propagation from D with stop point D.



Constraint graph after processing *E = F, and deep propagation from G with stop point F. The gray nodes will be collapsed into the stop point F.

Figure 3. Deep propagation in action.



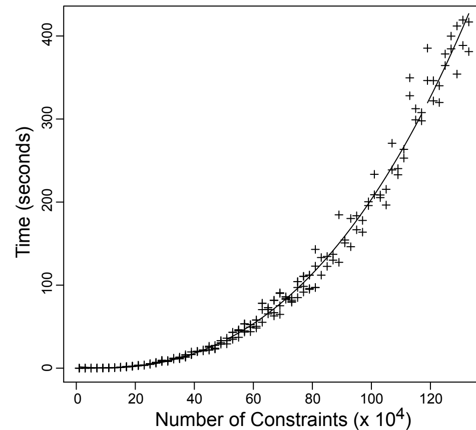


Figure 4. Asymptotic behavior: (Top) Wave Propagation, (Bottom) Deep Propagation.

To mitigate the number of searches that do not result in cycles, it executes at most one search per edge in the constraint graph.

- *Heintze-Tardieu* [7] (HT). This is the first massively scalable solver presented in the literature. It propagates points on demand, in a depth first fashion, in a way similar to the deep propagation method; however, it does not keep the invariant of that algorithm. Thus, it has to propagate entire points-to sets.
- *Pearce-Kelly-Hankin* [13] (PKH). The base of the current points-to solver used in GCC. It relies on a week topological ordering of the target graph to avoid searching for cycles across the entire space of nodes.

All the programs were compiled with GCC 4.0.1 at the -O3 optimization level, and use the same data-structure to represent points-to sets: the `bitmap` library from GCC.

5.1. Asymptotic Behavior

In order to verify the stability and the asymptotic behavior of each of the available algorithms, we have run them on a collection of 216 random constraint graphs. To produce these graphs, we generate random constraints, using the average proportion of constraints that we found in actual programs: 14% of base constraints, 49% of simple constraints, 25% of complex 1 constraints and 12% of complex 2 constraints. For this particular experiment we have used random constraint graphs because it is difficult to find a collection of benchmarks containing a gradually increasing number of constraints. Notice that these graphs are different from the constraint graphs that we would obtain from actual programs. The existence of edges in the constraint graphs of actual programs does not follow a normal distribution; instead, we observe that some special nodes tend to span or collect many edges. Furthermore, real constraint graphs tend to be more dense than our random graphs. For instance, after components had been collapsed using the deep propagation method, the random constraint graphs contain, on average, 0.87 edges per node. In comparison, the graphs produced in the same way for our real benchmarks range in density from 0.78 edges per nodes (sendmail) to 139.581 edges per node (wine). Another difference is the average size of the points-to set produced by the random graphs, which tend to be 5-10 times bigger than the average sizes observed in actual programs. Nevertheless, the random constraints give us an idea about the asymptotic behavior of each solver.

The results obtained from running the algorithms on the

Table 1. The Set of Benchmarks used in our experiments.

Benchmark	Short	#Variables	#Constraints
ex	`ex`	3419	3,933
300.twolf	`tw`	4,697	4,849
197.parser	`pr`	5,055	6,491
255.vortex	`vt`	8,262	8,746
sendmail	`sm`	11,408	11,828
254.gap	`gp`	19,336	25,005
emacs	`em`	14,386	27,122
253.perl	`pl`	19,895	28,525
vim	`vm`	31,630	36,997
nethack	`nh`	32,968	38,469
176.gcc	`gc`	39,560	56,791
ghostscript	`gs`	76,717	101,442
insight	`in`	58,763	99,245
gdb	`gd`	84,499	105,087
gimp	`gm`	81,915	125,203
wine	`wn`	150,828	199,465
linux	`lx`	145,293	231,290

MacOS environment are displayed in Figures 4 and 5. Each figure shows the line produced by fitting a polynomial of degree three on the data points. In order to measure the stability of each algorithm, we computed the variance of each point in relation to the regression curve. We observe that, for these constraint graphs, the Wave Propagation approach is the most stable, with an average variance of 4.31 seconds per constraint graph. The variance found for the other algorithms, in increasing order, is 8.819 for Heintze-Tardieu, 12.33 for Deep Propagation, 20.28 for Lazy Cycle Detection and 39.19 for the Pearce-Kelly-Hankin algorithm.

5.2. Running Time

In order to measure how the proposed algorithms perform in constraint graphs extracted from actual programs, we have used the benchmarks presented in [5], plus 12 benchmarks provided to us by Ben Hardekopf, which include the six largest integer programs in SPEC 2000. Table 1 shows the benchmarks. We will be using the short names in the second column to refer to each program. The constraints in these benchmarks are *field-insensitive*, that is, different variables in the same `struct` are treated as the same name. All the algorithms used in our tests are tuned to perform well with field-insensitive input constraints. The benchmarks have been preprocessed with an off-line variable substitution analysis [14]. The number of constraints includes all the constraint types - base, simple and complex - found in the programs after off-line variable substitution.

Figure 6 compares the running time of the five algorithms in the Intel/MacOS setting. All the results are normalized to the Heintze-Tardieu (HT) algorithm. Deep Propagation has the lowest geometric-mean, 0.82 of HT. The Wave Propagation method has the second lowest geometric-mean: 0.90 of HT. The average for the other algorithms are 1.77 for LCD, and 3.19 for PKH. Notice that Lazy Cycle Detection and Wave Propagation tend to outperform Deep Propagation for bigger benchmarks. For the three largest benchmarks, gimp, wine and linux, we have the following geometric means: DP = 0.87, WP = 0.89, LCD = 0.65 and PKH = 1.81.

Figure 7 compares the five algorithm in the AMD/Linux execution environment. We have observed small changes on the relative execution times for some of the benchmarks. In particular, Wave Propagation outperforms Lazy Cycle Detection for Wine, the most time consuming benchmark. Also the Heintze-Tardieu algorithm has the second best geometric-mean. However, the overall time pattern remains the same: the Deep Propagation method presents the best geometric-mean: 0.89 of HT. The other means are WP = 1.04, LCD = 1.65 and PKH = 3.09. Considering only the three largest benchmarks, we have: WP = 0.83, DP = 1.0, LCD = 0.69 and PKH = 1.97.

Table 2 gives the absolute running time of all the algorithms in the Intex/MacOS environment, and Table 3 indicates the equivalent numbers for the AMD/Linux setting. Overall, the MacOS environment shows the fastest times. The total running time, in seconds, of all the algorithms in the MacOS Setting is LCD = 1,656.99, WP = 2,094.13, DP = 2,086.52, HT = 2,376.07 and PKH = 3,733.37. Although LCD has worse geometric mean than DP or WP, its absolute running time in this environment is better, because it produces the fastest results for Wine and the Linux kernel, the biggest benchmarks. On the other hand, the absolute running times in the AMD/Linux platform show the wave propagation with the best times: WP = 2,783.22, LCD = 2,817.8, HT = 3,709.01, DP = 3,916.68 and PKH = 6,565.51. We speculate that the relative variation between WP and LCD is due to the amount of free memory in the different machines, which explains the memory-hungry WP outperforming LCD on the machine with larger RAM. We have also observed that the average size of points-to sets plays a main role in the running time of the algorithms. LCD shows better relative performance in benchmarks with large points-to sets, such as wine, where we have found nodes with over 16,000 aliases.

The chart in Figure 8 closes this section showing how the execution time is divided among the three phases of the wave propagation method. Notice that, although the edge insertion phase and the wave propagation phase have the same complexity, the latter is much faster in actual applications. As it can be noted, the insertion of new edges in the constraint graph accounts for over 89% of the total running time of the algorithm for the largest benchmarks. This difference happens because the search for cycles is linear on the number of edges of the constraint graph, and because the wave propagation phase only propagates differences between points-to sets.

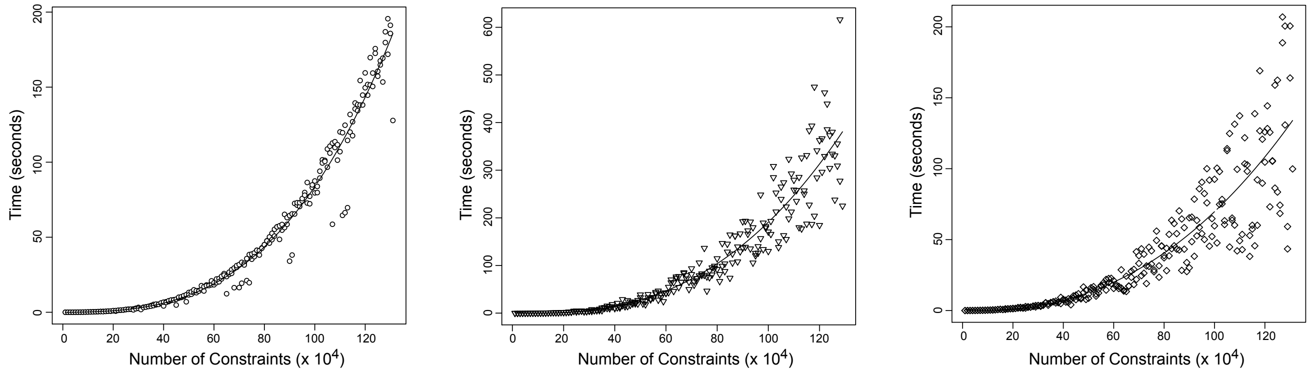


Figure 5. Asymptotic behavior: (Left) Heintze-Tardieu, (Middle) Pearce-Kelly-Hankin (Right) Lazy Cycle Detection.

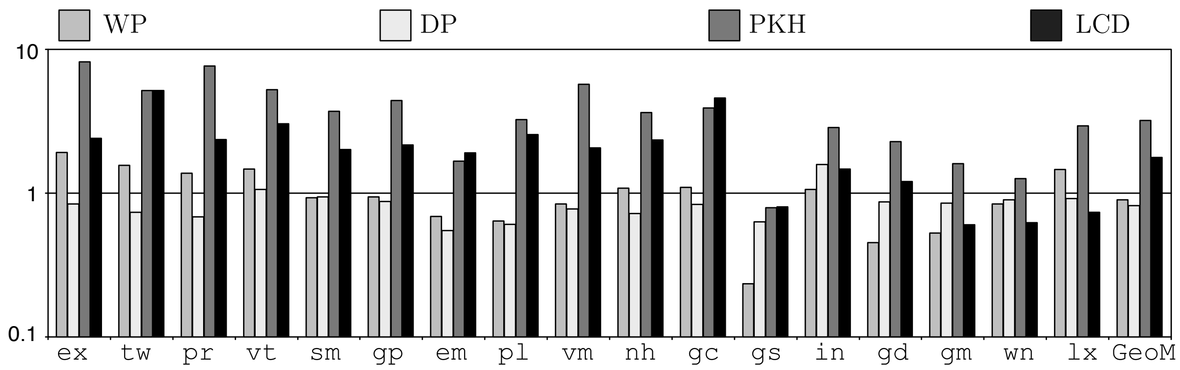


Figure 6. Running time comparison between the five solvers (Intel running MacOSX 10.4.1).

Table 2. The execution time of each algorithm (sec) on the Intel/MacOS setting.

	WP	DP	HT	LCD	PKH
ex	0.010	0.004	0.005	0.012	0.041
tw	0.023	0.011	0.015	0.075	0.078
pr	0.056	0.028	0.041	0.097	0.315
vt	0.034	0.024	0.023	0.070	0.120
sm	0.106	0.107	0.114	0.229	0.422
gp	0.316	0.293	0.336	0.725	1.480
em	0.997	0.793	1.445	2.756	2.408
pl	0.895	0.853	1.402	3.589	4.551
vm	1.810	1.673	2.150	4.442	12.25
nh	0.167	0.111	0.154	0.362	0.559
gc	0.813	0.619	0.742	3.397	2.910
gs	51.50	138.07	219.15	175.81	173.91
in	67.47	45.05	42.54	62.68	122.06
gd	32.74	62.86	72.32	87.11	164.87
gm	33.36	53.92	63.21	38.13	101.65
wn	1,327.3	1,423.5	1,578.3	983.6	1987.5
lx	560.0	349.7	382.4	281.1	1,126.2
Tot	2,055.1	2,099.5	2,364.8	1,644.8	3,701.3

Table 3. The execution time of each algorithm (sec) on the AMD/Linux setting.

	WP	DP	HT	LCD	PKH
ex	0.021	0.009	0.020	0.025	0.103
tw	0.022	0.011	0.016	0.061	0.079
pr	0.064	0.032	0.046	0.084	0.338
vt	0.044	0.029	0.028	0.071	0.131
sm	0.148	0.213	0.151	0.269	0.514
gp	0.642	0.397	0.434	0.867	2.064
em	2.407	1.300	2.098	4.094	3.357
pl	1.262	1.326	1.924	4.818	6.683
vm	3.399	2.757	3.926	6.161	21.971
nh	0.277	0.189	0.213	0.441	0.643
gc	1.081	0.880	0.985	3.886	3.84
gs	90.14	244.20	346.24	277.08	301.68
in	131.45	88.24	62.75	90.12	190.42
gd	67.38	103.77	106.10	123.05	283.42
gm	64.69	98.24	102.21	60.38	173.65
wn	1,191.4	2,769.5	2,396.7	1,754.3	4,013.4
lx	1,227.9	605.7	666.61	501.69	1,784.2
Tot	2,783.2	3,916.7	3,709.0	2,817.8	6,565.5

5.3. Memory Usage

Figure 9 compares memory consumption among the five tested algorithms. HT has the lowest geometric-mean across the benchmark suite. The next algorithm, DP, uses 1% more memory on average than HT. The memory usage for the

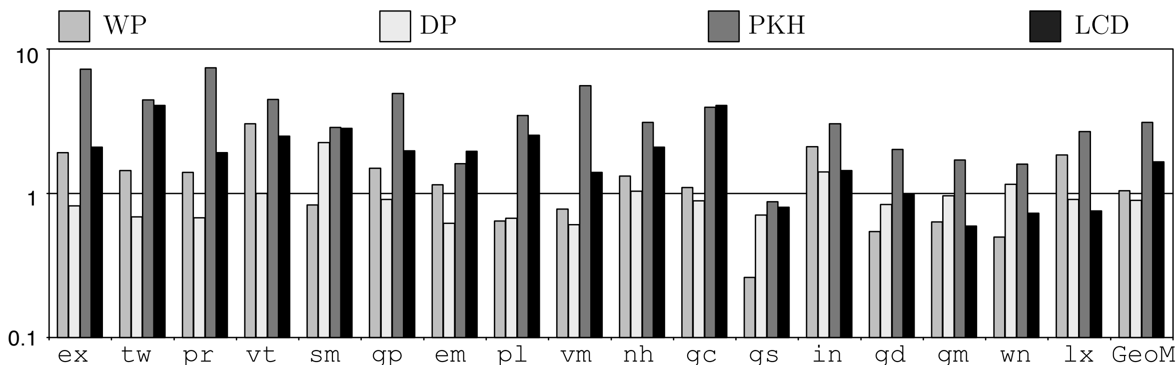


Figure 7. Running time comparison between the five solvers (AMD Opteron running Linux Ubuntu 6.06.2).

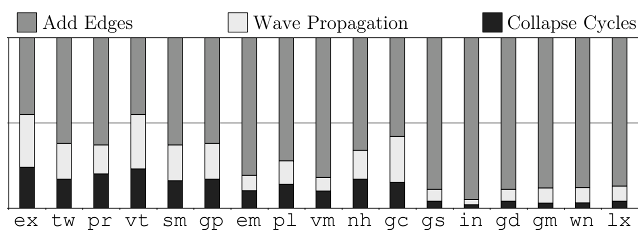


Figure 8. Time division in the wave propagation method in the Intel/MacOS setting.

other algorithms, relative to HT, is LCD = 1.07, PKH = 1.16 and WP = 1.42. Memory is mostly necessary to store points-to information. All the implementations use the same `bitmap` library of GCC to represent points-to sets. Wave propagation demands more memory because it stores an extra bitset per variable, P_{old} in Algorithm 4, plus an extra bitset per complex constraint, P_{cache} in Algorithm 4. Wine is the most memory intensive benchmark. The amount of memory that each algorithm needs to process this benchmark is: DP = 1,561M, LCD = 1,750M, PKH = 1,778M, HT = 2,095M and WP = 2,421M. Because the numbers for Wine dominate all the other benchmarks, although HT had the lowest geometric mean, in absolute terms DP was the most economical algorithm. If we sum up the memory required by each algorithm to process all the benchmarks, we get: DP = 3,954M, LCD = 4,121M, PKH = 4,255M, HT = 4,328M and WP = 5,881M.

5.4. Summary of Experiments

Table 4 summarizes all the results described in this Section. *GT* stands for geometric mean of running time, *AT* stands for absolute running time, *OSX* stands for MacOS/Intel, *LX* stands for linux/AMD, *GM* stands for geometric mean of memory consumption and *AM* stands for absolute memory consumption.

Table 4. Summary of Experiments

	DP	WP	LCD	HT	PKH
GT OSX	0.82 (*)	0.90	1.77	1.0	3.19
GT LX	0.89 (*)	1.04	1.65	1.0	3.09
AT OSX	2,099	2,055	1,645 (*)	2,364	3,701
AT LX	3,917	2,783 (*)	2,818	3,709	6,566
GM	1.01	1.42	1.07	1.0 (*)	1.16
AM	3,954 (*)	5,881	4,121	4,328	4,255
Variance	12.33	4.31 (*)	20.28	8.82	39.19

It is interesting to compare the new algorithms, deep and wave propagation, with the state-of-the-art lazy cycle detector. Our experiments show that LCD is the fastest algorithm for large benchmarks, although WP outperforms it in memory rich execution environments. We have observed that both wave and particularly deep propagation tend to be faster than lazy cycle detection in settings with smaller points-to sets per node, that is, the more precise the constraints, the faster the new algorithms are with relation to LCD.

6. Conclusion and Future Work

This paper has presented two new algorithms for solving Andersen based points-to analysis: the Wave Propagation and the Deep Propagation methods. As discussed in Section 5, these algorithms improve the current state of the art in many different directions. All our implementations are available on-line. As future work, we intend to pursue a parallel implementation of the wave propagation method. Cycle elimination complicates the parallelization of points-to solvers, because this optimization may force the locking of a linear number of nodes in the constraint graph to avoid data races. The wave propagation algorithm separates the detection of cycles from the propagation of points-to

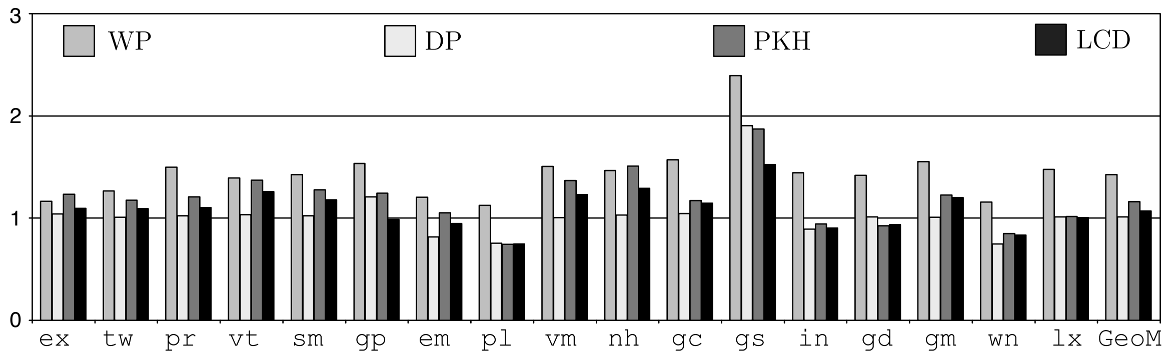


Figure 9. Memory usage in AMD Opteron running Linux Ubuntu 6.06.2.

sets. The detection of strongly connected components, has a well-known parallel implementation [4]. Once connected components are discovered, each of them can be collapsed by a different thread. The insertion of edges, which accounts for most of the execution time of the algorithm according to Figure 8, requires the locking of a constant number of nodes per thread.

Acknowledgment

Ben Hardekopf provided the benchmarks used in our experiments. We thank Ben Hardekopf, Tianwei Sheng, Sifei Zhong and the anonymous reviewers for helpful comments on early drafts of this paper. This research was funded by Google. Fernando Pereira is also sponsored by the Brazilian Ministry of Education under grant number 218603-9.

References

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[2] Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI*, pages 57–69, 2000.

[3] Manuel Fahndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI*, pages 85–96, 1998.

[4] Alan Gibbons. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[5] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299, 2007.

[6] Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *SAS*, pages 265–280, 2007.

[7] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *PLDI*, pages 254–263, 2001.

[8] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, 1997.

[9] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred C. Chow. Partial redundancy elimination in SSA form. *ACM Trans. Program. Lang. Syst.*, 21(3):627–676, 1999.

[10] Chris Lattner and Vikram S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[11] Esko Nuutila and Eljas Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Inf. Process. Lett.*, 49(1):9–14, 1994.

[12] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Online cycle detection and difference propagation for pointer analysis. In *SCAM*, pages 3–12, 2003.

[13] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for C. In *PASTE*, pages 37–42, 2004.

[14] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *PLDI*, pages 47–56, 2000.

[15] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.

[16] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[17] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.

[18] Jianwen Zhu. Towards scalable flow and context sensitive pointer analysis. In *DAC*, pages 831–836, 2005.