Wave Propagation and Deep Propagation for Pointer Analysis

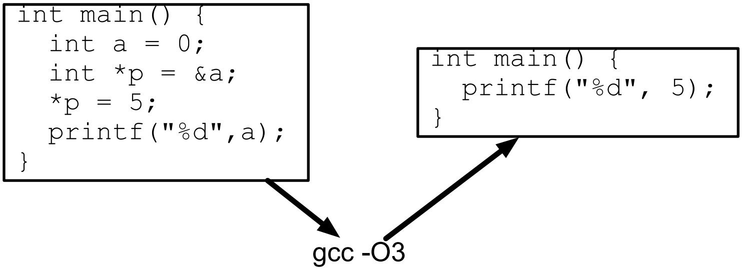
Fernando Magno Quintão Pereira

February 15, 2009

Outline

Outline

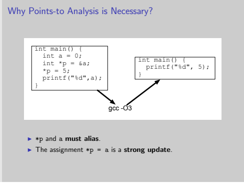
Why Points-to Analysis is Necessary?



```
int main() {
  int a = 0;
  int *p = &a;
  *p = 5;
  printf("%d",a);
}
```

```
int main() {
  printf("%d", 5);
}
```

- `*p` and `a` **must alias**.
- The assignment `*p = a` is a **strong update**.



Why Points-to Analysis is Necessary?

```
int main() {
  int a = 0;
  int *p = &a;
  *p = 5;
  printf("%d",a);
}
```

gcc -O3

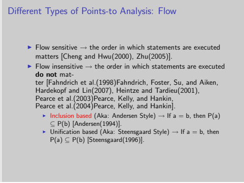
```
int main() {
  printf("%d", 5);
}
```

- `*p` and a **must alias**.
- The assignment `*p = a` is a **strong update**.

Many imperative programming languages use aliasing. Aliasing is used, for instance, to avoid copying whole data structures when passing arguments to functions. Although a powerful feature, aliasing complicates the job of optimizing compilers. The objective of alias analysis is to discover which names in a program refer to the same location. The main motivation is to enable compiler optimizations, but it also finds applications in security, data race detection, etc.

Different Types of Points-to Analysis: Flow

- Flow sensitive → the order in which statements are executed matters [Cheng and Hwu(2000), Zhu(2005)].
- Flow insensitive → the order in which statements are executed **do not** mat-
ter [Fahndrich et al.(1998)Fahndrich, Foster, Su, and Aiken, Hardekopf and Lin(2007), Heintze and Tardieu(2001), Pearce et al.(2003)Pearce, Kelly, and Hankin, Pearce et al.(2004)Pearce, Kelly, and Hankin].
  - Inclusion based (Aka: Andersen Style) → If a $=$ b, then P(a) $\subseteq$ P(b) [Andersen(1994)].
  - Unification based (Aka: Steensgaard Style) → If a $=$ b, then P(a) $\subseteq$ P(b) [Steensgaard(1996)].



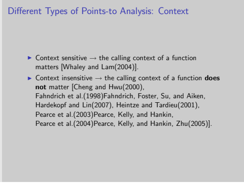
Different Types of Points-to Analysis: Flow

- Flow sensitive → the order in which statements are executed matters [Cheng and Hwu(2000), Zhu(2005)].
- Flow insensitive → the order in which statements are executed **do not** mat-
ter [Fahndrich et al.(1998)Fahndrich, Foster, Su, and Aiken, Hardekopf and Lin(2007), Heintze and Tardieu(2001), Pearce et al.(2003)Pearce, Kelly, and Hankin, Pearce et al.(2004)Pearce, Kelly, and Hankin].
  - Inclusion based (Aka: Andersen Style) → If a = b, then P(a) ⊆ P(b) [Andersen(1994)].
  - Unification based (Aka: Steensgaard Style) → If a = b, then P(a) ⊆ P(b) [Steensgaard(1996)].

There are different types of pointer analysis with regard to flow and context sensitiveness. Flow insensitive algorithms [Fahndrich et al.(1998)Fahndrich, Foster, Su, and Aiken, Hardekopf and Lin(2007), Heintze and Tardieu(2001)] ignore the order of statements in a program, contrary to flow sensitive analyses [Cheng and Hwu(2000), Zhu(2005)]. 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 [Steensgaard(1996)] is the most famous representative, assume that the locations pointed by both variables are the same; thus, trading precision by speed.

Different Types of Points-to Analysis: Context

- Context sensitive $\rightarrow$ the calling context of a function matters [Whaley and Lam(2004)].
- Context insensitive $\rightarrow$ the calling context of a function **does not** matter [Cheng and Hwu(2000), Fahndrich et al.(1998)Fahndrich, Foster, Su, and Aiken, Hardekopf and Lin(2007), Heintze and Tardieu(2001), Pearce et al.(2003)Pearce, Kelly, and Hankin, Pearce et al.(2004)Pearce, Kelly, and Hankin, Zhu(2005)].


Different Types of Points-to Analysis: Context

- Context sensitive → the calling context of a function matters [Whaley and Lam(2004)].
- Context insensitive → the calling context of a function **does not** matter [Cheng and Hwu(2000), Fahndrich et al.(1998)Fahndrich, Foster, Su, and Aiken, Hardekopf and Lin(2007), Heintze and Tardieu(2001), Pearce et al.(2003)Pearce, Kelly, and Hankin, Pearce et al.(2004)Pearce, Kelly, and Hankin, Zhu(2005)].

Context sensitive analyses distinguish the different calling contexts of a function [Whaley and Lam(2004)].

Inclusion Based Points-to Analysis

Statement	Name	Constraint	Meaning
`a = &b`	Base	$a \supseteq \{b\}$	$b \in P(a)$
`a = b`	Simple	$a \supseteq b$	$P(a) \supseteq P(b)$
`a = *b`	Complex 1	$a \supseteq *b$	$\forall v \in P(b), P(a) \supseteq P(v)$
`*a = b`	Complex 2	$*a \supseteq b$	$\forall v \in P(a), P(b) \supseteq P(v)$

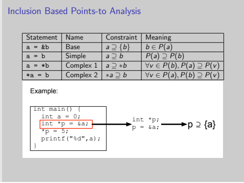
Example:

```
int main() {
  int a = 0;
  int *p = &a;
  *p = 5;
  printf("%d",a);
}
```

$\longrightarrow$

```
int *p;
p = &a;
```

$\longrightarrow$ $p \supseteq \{a\}$



Inclusion Based Points-to Analysis

Statement	Name	Constraint	Meaning
`a = &b`	Base	$a \supseteq \{b\}$	$b \in P(a)$
`a = b`	Simple	$a \supseteq b$	$P(a) \supseteq P(b)$
`a = *b`	Complex 1	$a \supseteq *b$	$\forall v \in P(b), P(a) \supseteq P(v)$
`*a = b`	Complex 2	$*a \supseteq b$	$\forall v \in P(a), P(b) \supseteq P(v)$

Example:

```
int main() {
  int a = 0;
  int *p = &a;
  *p = 5;
  printf("%d",a);
}
```

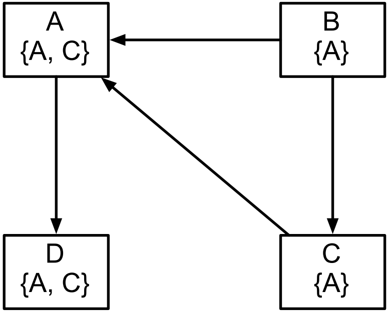
```
int *p;
p = &a;
```

p ⊇ {a}

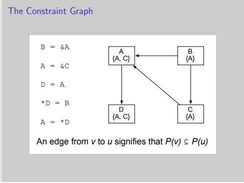
Constraints are derived from statements involving variable assignment or parameter passing in the program that is being analyzed. There are basically four types of constraints: base, simple, complex 1 and complex 2. We can convert the statements in a program into these basic constraints via trivial transformations.

The Constraint Graph

```
B  =  &A

A  =  &C

D  =  A

*D  =  B

A  =  *D
```



An edge from *v* to *u* signifies that *P(v)* ⊆ *P(u)*


The Constraint Graph

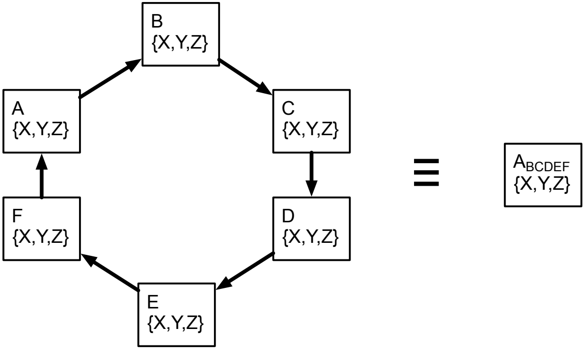
```
B = &A

A = &C

D = A

*D = B

A = *D
```

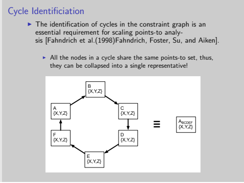
An edge from *v* to *u* signifies that *P(v)* ⊆ *P(u)*

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.

Cycle Identificiation

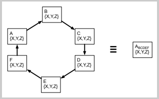
- The identification of cycles in the constraint graph is an essential requirement for scaling points-to analysis [Fahndrich et al.(1998)Fahndrich, Foster, Su, and Aiken].
  - All the nodes in a cycle share the same points-to set, thus, they can be collapsed into a single representative!





Cycle Identificiation

- The identification of cycles in the constraint graph is an essential requirement for scaling points-to analysis [Fahndrich et al.(1998)Fahndrich, Foster, Su, and Aiken].
  - All the nodes in a cycle share the same points-to set, thus, they can be collapsed into a single representative!

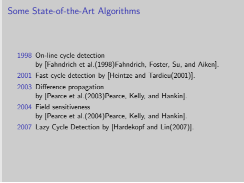


$$\begin{tikzcd} & B\ \{X,Y,Z\} \arrow[dr] & & \\ A\ \{X,Y,Z\} \arrow[ur] & & C\ \{X,Y,Z\} \arrow[d] & \equiv\quad A_{BCDEF}\ \{X,Y,Z\} \\ F\ \{X,Y,Z\} \arrow[u] & & D\ \{X,Y,Z\} \arrow[dl] & \\ & E\ \{X,Y,Z\} \arrow[ul] & & \end{tikzcd}$$

The 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.

Some State-of-the-Art Algorithms

1998	On-line cycle detection by [Fahndrich et al.(1998)Fahndrich, Foster, Su, and Aiken].
2001	Fast cycle detection by [Heintze and Tardieu(2001)].
2003	Difference propagation by [Pearce et al.(2003)Pearce, Kelly, and Hankin].
2004	Field sensitiveness by [Pearce et al.(2004)Pearce, Kelly, and Hankin].
2007	Lazy Cycle Detection by [Hardekopf and Lin(2007)].



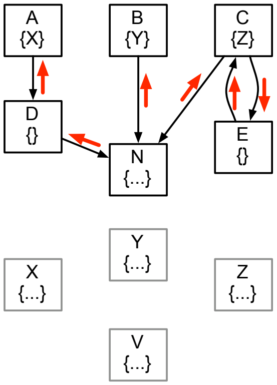
Some State-of-the-Art Algorithms

1998 On-line cycle detection by [Fahndrich et al.(1998)Fahndrich, Foster, Su, and Aiken].
2001 Fast cycle detection by [Heintze and Tardieu(2001)].
2003 Difference propagation by [Pearce et al.(2003)Pearce, Kelly, and Hankin].
2004 Field sensitiveness by [Pearce et al.(2004)Pearce, Kelly, and Hankin].
2007 Lazy Cycle Detection by [Hardekopf and Lin(2007)].

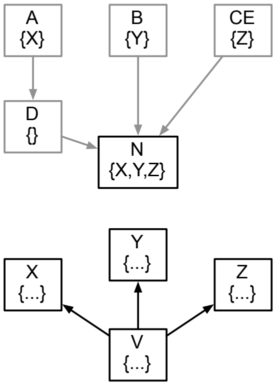
Fahndrich *et al.* proposed one of the first algorithms to detect cycles on-line, that is, while complex constraints are being processed. Since then, many new algorithms have been proposed. Heintze and Tardieu describe an algorithm that can analyze C programs with over one million lines of code in a few seconds. Pearce *et al.* have also introduced important contributions to this field. The algorithms designed by Pearce *et al.* constitute the core of GCC's points-to solver. 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. The points-to solver used in LLVM was implemented after this last algorithm.

Example: Heintze-Tardieu processing *N = V

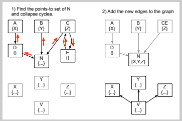
1) Find the points-to set of N and collapse cycles.



2) Add the new edges to the graph



Example: Heintze-Tardieu processing *N = V



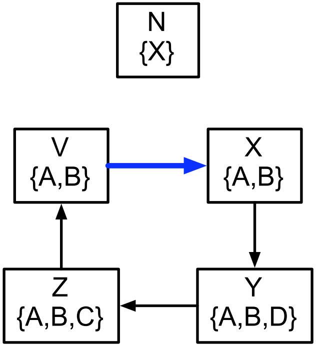
Heintze-Tardieu compute points-to sets on demand. In order to process a constraint such as *N = V, the algorithm first computes the points-to set of node N. To do this, it traverses the constraint graph backwards, copying the points-to set of any node reachable from N into the points-to set of N. During this traversal, the algorithm collapses cycles, if any is found. After it has computed the points-to set of N, it uses adds a new edge from V to each element of the new set.

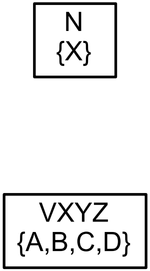
Heintze-Tardieu was an improvement on previous algorithms, but it still computes redundant work every time it reprocesses a constraint such as *N = V, because points-to set are repropagated from the transitive closure of the predecessors of N into N itself.

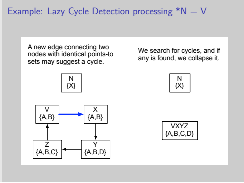
Example: Lazy Cycle Detection processing *N = V

A new edge connecting two nodes with identical points-to sets may suggest a cycle.

We search for cycles, and if any is found, we collapse it.






Example: Lazy Cycle Detection processing *N = V

A new edge connecting two nodes with identical points-to sets may suggest a cycle.

We search for cycles, and if any is found, we collapse it.

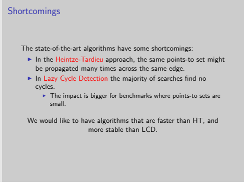
All the nodes in a cycle share the same points-to set. The lazy cycle detection approach searches for cycles every time an edge is added to the constraint graph connecting two vertices that have the same points-to set. Cycles are not always found. Indeed, only a small fraction of searches results in hits. Nevertheless, the effort pays off if the constraint graph is too big, or if the average size of the points-to sets is too large.

Shortcomings

The state-of-the-art algorithms have some shortcomings:

- In the Heintze-Tardieu approach, the same points-to set might be propagated many times across the same edge.
- In Lazy Cycle Detection the majority of searches find no cycles.
    - The impact is bigger for benchmarks where points-to sets are small.

We would like to have algorithms that are faster than HT, and more stable than LCD.



Shortcomings

The state-of-the-art algorithms have some shortcomings:

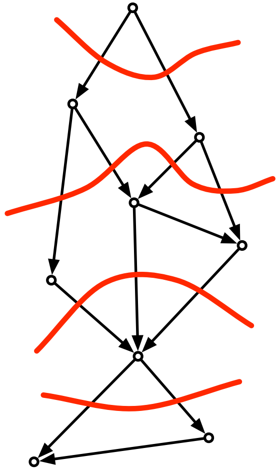
- In the Heintze-Tardieu approach, the same points-to set might be propagated many times across the same edge.
- In Lazy Cycle Detection the majority of searches find no cycles.
    - The impact is bigger for benchmarks where points-to sets are small.

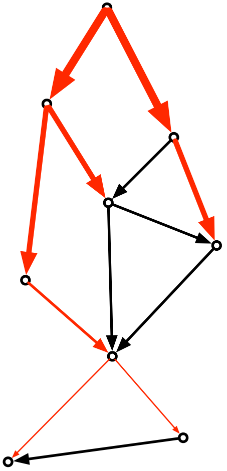
We would like to have algorithms that are faster than HT, and more stable than LCD.

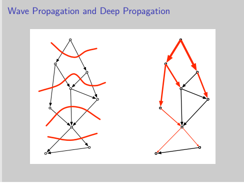
LCD is like the heavy-artillery of points-to analysis. For large benchmarks, less than 1% of the searches will find actual cycles. This work pays off when the average size of points-to sets is large, because finding cycles avoids having to propagate these big chunks of data. However, we rarely compile very large programs, e.g, how many times have you compiled the linux kernel today? For smaller benchmarks, the extra searches for cycles eclipses the saved effort of propagating points-to sets.

Outline

Wave Propagation and Deep Propagation







We present two new algorithms for pointer analysis. We will call them wave propagation and deep propagation. They have the same common starting point: Nuutila's [Nuutila and Soisalon-Soininen(1994)] algorithm for finding strongly connected components. But the similarities between these two algorithms stop there. Wave propagation is to deep propagation as breadth first search is to depth first search. In the wave propagation method information is propagated in waves, in the same order in which nodes are visited by the breadth first search. In deep propagation the information is propagated as a flow, similar to depth first search.

Nuutila's Algorithm

The starting point of our two algorithms is Nuutila's algorithm for finding strongly connected components in directed graphs [Nuutila and Soisalon-Soininen(1994)].

- Published in 1994 by Nuutila and Soisalon-Soininen.
- Traverses the graph once, in contrast to Tarjan's more popular algorithm that traverse the graph twice.
- Used by Pearce-Kelly-Hankin in their inclusion based pointer analysis solver.
- Produces the topological ordering of the target graph as a side effect.

Nuutila's Algorithm

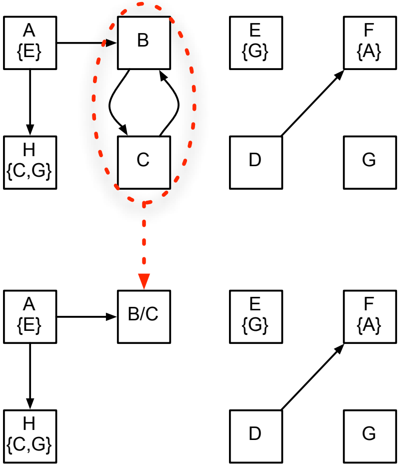
The starting point of our two algorithms is Nuutila's algorithm for finding strongly connected components in directed graphs [Nuutila and Soisalon-Soininen(1994)].

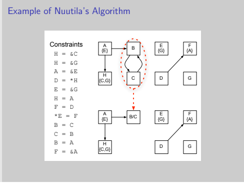
- Published in 1994 by Nuutila and Soisalon-Soininen.
- Traverses the graph once, in contrast to Tarjan's more popular algorithm that traverse the graph twice.
- Used by Pearce-Kelly-Hankin in their inclusion based pointer analysis solver.
- Produces the topological ordering of the target graph as a side effect.

Example of Nuutila's Algorithm

Constraints

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





Example of Nuutila's Algorithm

Constraints

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

This example shows finding and removal of strongly connected components. A strongly connected component of a directed graph is a set of nodes that are reachable from each other. In the left we have a set of 12 constraints, and in the upper we have the constraint graph that would be produced for these constraints. Nuutila's algorithm discovers that nodes B and C are part of the same connected component. We can collapse these nodes into a single node, because they will have always the same points-to set. The collapsed graph is shown in the lower part of the figure.

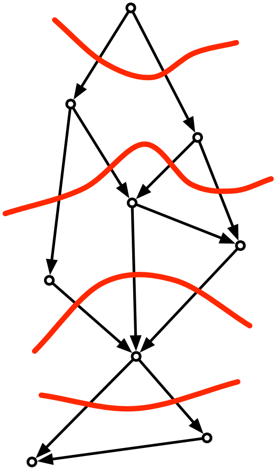
Outline

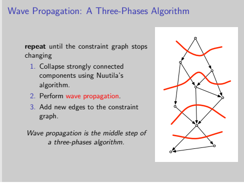
Wave Propagation: A Three-Phases Algorithm

repeat until the constraint graph stops changing

1. Collapse strongly connected components using Nuutila's algorithm.
2. Perform wave propagation.
3. Add new edges to the constraint graph.

Wave propagation is the middle step of a three-phases algorithm.




Wave Propagation: A Three-Phases Algorithm

repeat until the constraint graph stops changing
1. Collapse strongly connected components using Nuutila's algorithm.
2. Perform wave propagation.
3. Add new edges to the constraint graph.

Wave propagation is the middle step of a three-phases algorithm.

Calling the whole algorithm wave propagation is indeed an abuse of language. Wave propagation is the name of the middle step of a three-phases, iterative algorithm that solves inclusion based pointer analysis.

What is Wave Propagation?

Wave propagation is the propagation of points-to set in an acyclic constraint graph in topological order, starting from the predecessor of all the nodes.

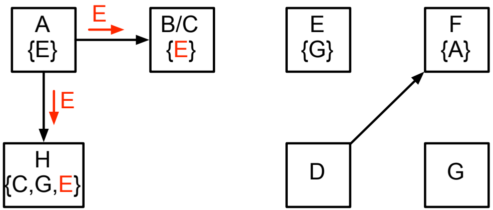
After wave propagation, if u is reachable from v, then $PTS(v) \subseteq PTS(u)$.

Optimization: cache the points-to information already propagated from a node. Call it the PTS_{old}. Only propagate the difference current - old.

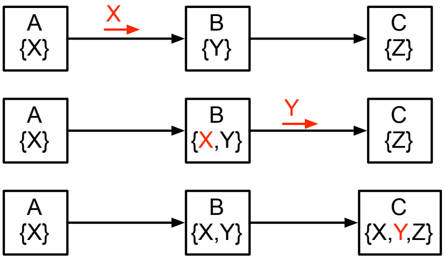
Invariant: if $(u, v) \in G$, then $PTS_{old}(u) \subseteq PTS(v)$.

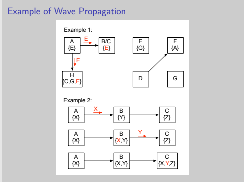
Example of Wave Propagation

Example 1:



Example 2:





The first example shows how wave propagation would take place in the constraint graph seen in our running program. In this case, only the points-to set of node A would be propagated to its successors. The second example illustrates how the topological ordering plus the acyclicity of the constraint graph helps us to save work. Each edge is visited only once, and after the propagation we guarantee that if a node u is reachable from a node v, then the points-to set of v us a subset of the points-to set of v.

The Insertion of New Edges

Complex constraints cause the insertion of new edges in the constraint graph.

- $l = *r$: add edge (u, l) for each node $u \in PTS(r)$
- $*l = r$: add edge (r, u) for each node $u \in PTS(l)$

After inserting a new edge (x, y) we copy the old points-to set of x into the current points-to set of y to preserve the invariant that $PTS_{old}(x) \subseteq PTS(y)$ for any edge (x, y).

The Insertion of New Edges

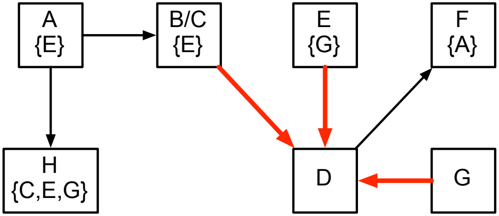
Complex constraints cause the insertion of new edges in the constraint graph.

- $l = *r$: add edge (u, l) for each node $u \in PTS(r)$
- $*l = r$: add edge (r, u) for each node $u \in PTS(l)$

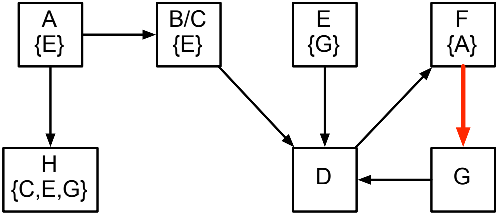
After inserting a new edge (x, y) we copy the old points-to set of x into the current points-to set of y to preserve the invariant that $PTS_{old}(x) \subseteq PTS(y)$ for any edge (x, y).

Example of Edge Insertion

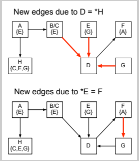
New edges due to D = *H



New edges due to *E = F



Example of Edge Insertion



Our running example has two complex constraints: $D = *H$ and $*E = F$. These constraints force the insertion of new edges in the constraint graph. First, let's consider $D = *H$. Node H contains three names in its points-to set: $\{C, E, G\}$; thus, we must insert three new edges into the constraint graph: (C, D), (E, D) and (G, D).

Outline

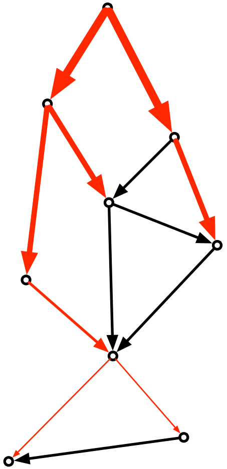
Deep Propagation

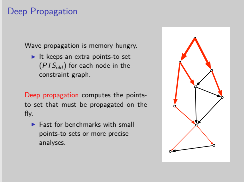
Wave propagation is memory hungry.

- It keeps an extra points-to set (PTS_{old}) for each node in the constraint graph.

Deep propagation computes the points-to set that must be propagated on the fly.

- Fast for benchmarks with small points-to sets or more precise analyses.





Deep Propagation

Wave propagation is memory hungry.

- It keeps an extra points-to set (PTS_{old}) for each node in the constraint graph.

Deep propagation computes the points-to set that must be propagated on the fly.

- Fast for benchmarks with small points-to sets or more precise analyses.

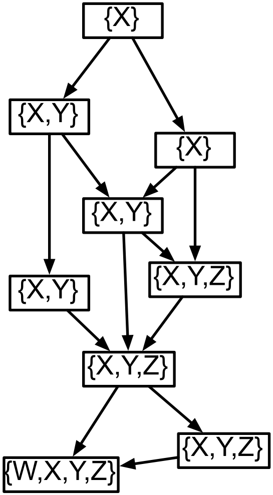
The deep propagation method uses far less memory than wave propagation. It is also fast for settings where the average size of the points-to sets is small. It is the fastest algorithm that we know for benchmarks such as vim, SPEC perl and SPEC gcc, e.g, applications with less than 100K lines of code.

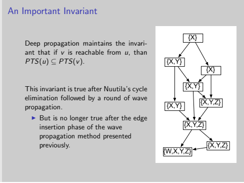
An Important Invariant

Deep propagation maintains the invariant that if v is reachable from u, than $PTS(u) \subseteq PTS(v)$.

This invariant is true after Nuutila's cycle elimination followed by a round of wave propagation.

- But is no longer true after the edge insertion phase of the wave propagation method presented previously.



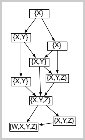


An Important Invariant

Deep propagation maintains the invariant that if v is reachable from u, than $PTS(u) \subseteq PTS(v)$.

This invariant is true after Nuutila's cycle elimination followed by a round of wave propagation.

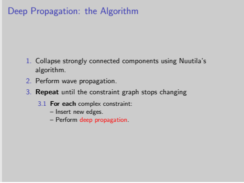
- But is no longer true after the edge insertion phase of the wave propagation method presented previously.



The figure in the left illustrates our invariant in an acyclic graph. Put in another words, the invariant means that nodes that have many predecessors tend to have fatter points-to sets. In this figure, the node in the bottom has the largest points-to set, which includes the points-to set of every other node.

Deep Propagation: the Algorithm

1. Collapse strongly connected components using Nuutila's algorithm.
2. Perform wave propagation.
3. **Repeat** until the constraint graph stops changing
   3.1 **For each** complex constraint:
   – Insert new edges.
   – Perform deep propagation.



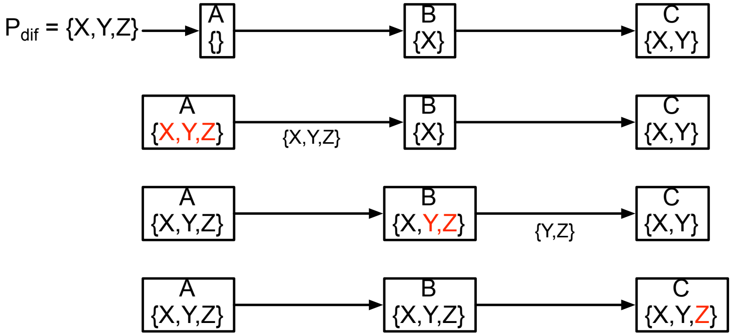
Deep Propagation: the Algorithm

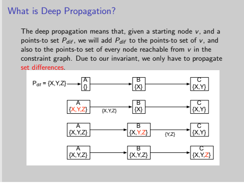
1. Collapse strongly connected components using Nuutila's algorithm.
2. Perform wave propagation.
3. **Repeat** until the constraint graph stops changing
   3.1 **For each** complex constraint:
      – Insert new edges.
      – Perform deep propagation.

Notice that steps 1 and 2 are the same as in the wave propagation method; however, they are only executed once, in contrast to the previous algorithm.

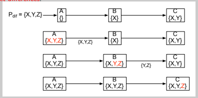
What is Deep Propagation?

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. Due to our invariant, we only have to propagate set differences.




What is Deep Propagation?

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. Due to our invariant, we only have to propagate set differences.



Let's recall the invariant of the deep propagation algorithm: if v is reachable from u, than $PTS(u) \subseteq PTS(v)$. Because of this invariant, when propagating points-to set across the constraint graph, we do not have to propagate the whole set, but only those points that are not present in the node that is been visited. This also means that we can finish the propagation much before visiting every reachable node in the constraint graph. The propagation will finish when the different to be propagated becomes the empty set.

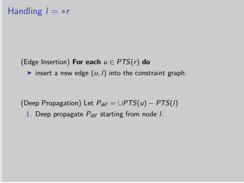
Handling $l = *r$

(Edge Insertion) **For each** $u \in PTS(r)$ **do**

- insert a new edge (u, l) into the constraint graph.

(Deep Propagation) Let $P_{dif} = \cup PTS(u) - PTS(l)$

1. Deep propagate P_{dif} starting from node l.



Handling $l = *r$

(Edge Insertion) **For each** $u \in PTS(r)$ **do**

- insert a new edge (u, l) into the constraint graph.

(Deep Propagation) Let $P_{dif} = \cup PTS(u) - PTS(l)$

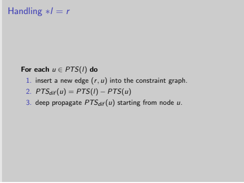
1. Deep propagate P_{dif} starting from node l.

The nodes in the points-to set of node r contribute to the new points-to set of node l. Because of our invariant, we do not need to keep a cache of the last points-to set propagated. This will be the different between the current points-to set of node l and the new points-to set that will be added to l.

Handling $*l = r$

For each $u \in PTS(l)$ **do**

1. insert a new edge (r, u) into the constraint graph.
2. $PTS_{dif}(u) = PTS(l) - PTS(u)$
3. deep propagate $PTS_{dif}(u)$ starting from node u.


Handling $*l = r$

For each $u \in PTS(l)$ **do**

1. insert a new edge (r, u) into the constraint graph.
2. $PTS_{dif}(u) = PTS(l) - PTS(u)$
3. deep propagate $PTS_{dif}(u)$ starting from node u.

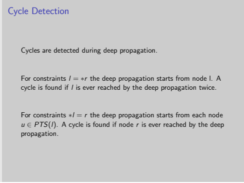
Complex 2 constraints are handled in a different way than complex 1 constraints. In the case of complex 1 constraints, only one deep propagation will happen. In the case of complex two constraints such as $*l = r$, one deep propagation will be triggered for each node in the points-to set of node l. Again, our invariant avoids unnecessary propagations. Even though many deep propagations might be fired, we only need to add new information to the points-to set of each visited node once. The paper describes with more detail this optimization.

Cycle Detection

Cycles are detected during deep propagation.

For constraints $l = *r$ the deep propagation starts from node l. A cycle is found if l is ever reached by the deep propagation twice.

For constraints $*l = r$ the deep propagation starts from each node $u \in PTS(l)$. A cycle is found if node r is ever reached by the deep propagation.



Cycle Detection

Cycles are detected during deep propagation.

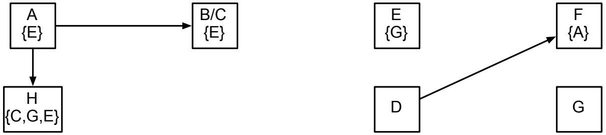
For constraints $l = *r$ the deep propagation starts from node l. A cycle is found if l is ever reached by the deep propagation twice.

For constraints $*l = r$ the deep propagation starts from each node $u \in PTS(l)$. A cycle is found if node r is ever reached by the deep propagation.

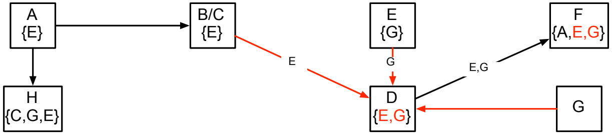
The deep propagation method is not guaranteed to eliminate all the cycles in the target constraint graph. Omissions happen because a node only invokes the deep propagation routine on its successors if there are points to propagate.

Deep Propagation: Running Example

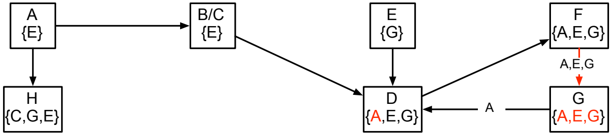
Constraint graph after initial set-up (Nuutila plus WP)



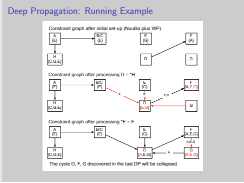
Constraint graph after processing D = *H



Constraint graph after processing *E = F



The cycle D, F, G discovered in the last DP will be collapsed.

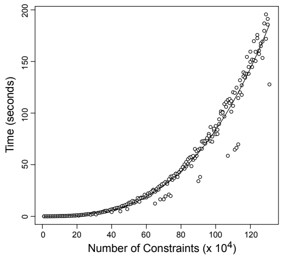

Deep Propagation: Running Example

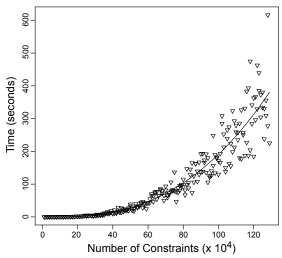
The cycle D, F, G discovered in the last DP will be collapsed.

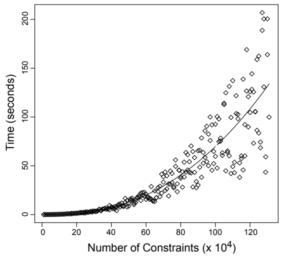
This slide shows how the deep propagation algorithm performs on our running example. In this case, deep propagation is triggered twice. The first call handles the constraint $D = *H$. The algorithm inserts edges connecting the nodes in the points-to set of H to D. After that, the new points-to set of node D is computed, and it is then propagated to nodes reachable from D; in this case, only node F. The second call handles the constraint $*E = F$. We insert an edge connecting F to G, the only node present in the points-to set of E. The deep propagation that follows the edge insertion finds a cycle containing nodes F, D and G. The cycle will be collapsed into F after the propagation of points-to sets.

Outline

Asymptotic Behavior: HT, PKH and LCD

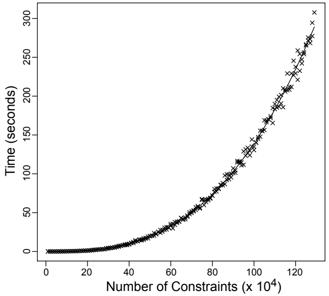


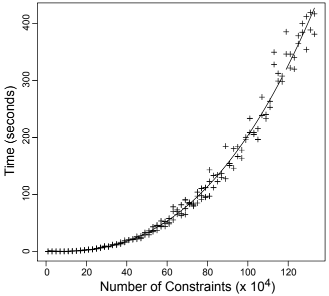




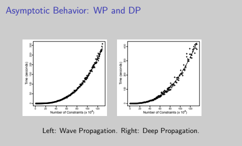
Top-Left: Heintze-Tardieu.
Top-Right: Pearce-Kelly-Hankin.
Bottom: Lazy Cycle Detection.

Asymptotic Behavior: WP and DP





Left: Wave Propagation. Right: Deep Propagation.

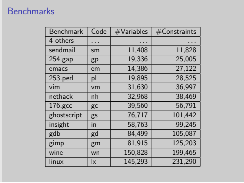

Asymptotic Behavior: WP and DP

Left: Wave Propagation. Right: Deep Propagation.

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.

Benchmarks

Benchmark	Code	#Variables	#Constraints
4 others	...	...	...
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

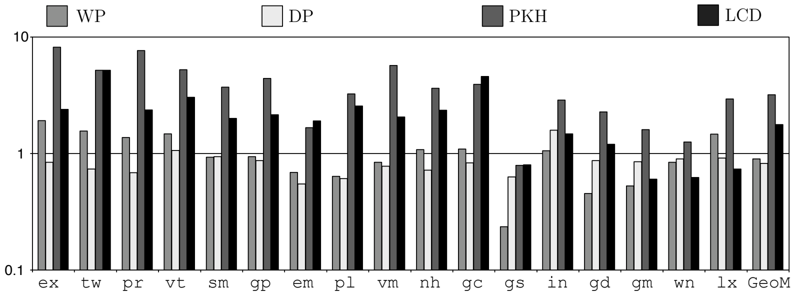


Benchmarks

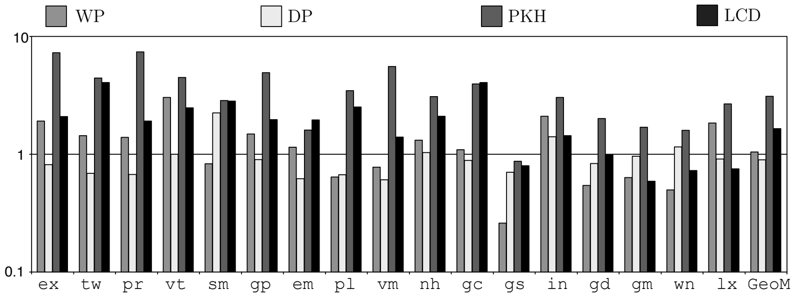
Benchmark	Code	#Variables	#Constraints
4 others	...	...	...
sendmail	sm	11,408	11,828
254.gap	gp	19,336	25,005
emacs	em	14,386	27,122
253.perl	pl	19,895	28,525
vim	vm	31,630	36,997
nethack	nh	32,968	38,469
176.gcc	gc	39,560	56,791
ghostscript	gs	76,717	101,442
insight	in	58,763	99,245
gdb	gd	84,499	105,087
gimp	gm	81,915	125,203
wine	wn	150,828	199,465
linux	lx	145,293	231,290

In order to measure how the proposed algorithms perform in constraint graphs extracted from actual programs, we have used the benchmarks presented in, plus 12 benchmarks kindly provided to us by Ben Hardekopf, which include the six biggest integer programs in SPEC 2000. 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. The number of constraints includes all the constraint types - base, simple and complex - found in the programs after off-line variable substitution.

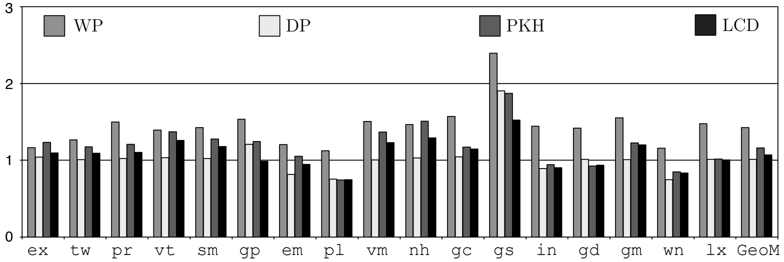
Running Time: Intel/Mac OS X



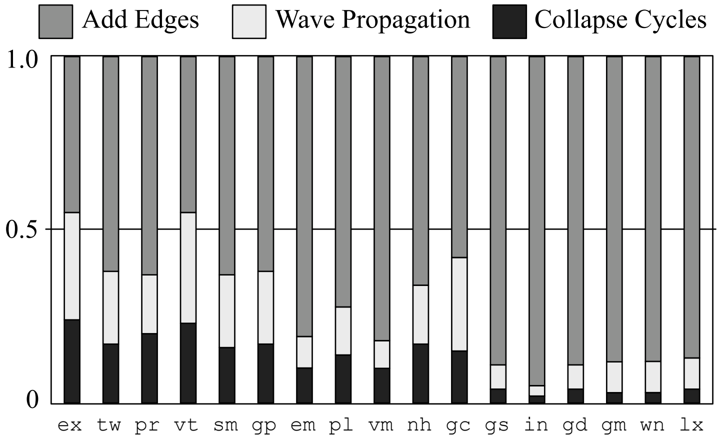
Running Time: AMD/Linux



Memory Consumption: AMD/Linux



Time Division of WP Phases



L. O. Andersen.

Program Analysis and Specialization for the C Programming Language.

PhD thesis, DIKU, University of Copenhagen, 1994.

B.-C. Cheng and W.-M. W. Hwu.

Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation.

In *PLDI*, pages 57–69, 2000.

M. Fahndrich, J. S. Foster, Z. Su, and A. Aiken.

Partial online cycle elimination in inclusion constraint graphs.

In *PLDI*, pages 85–96, 1998.

B. Hardekopf and C. Lin.

The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code.

In *PLDI*, pages 290–299, 2007.

N. Heintze and O. Tardieu.

Ultra-fast aliasing analysis using CLA: A million lines of C code in a second.

In *PLDI*, pages 254–263, 2001.

E. Nuutila and E. Soisalon-Soininen.

On finding the strongly connected components in a directed graph.

Inf. Process. Lett., 49(1):9–14, 1994.

D. J. Pearce, P. H. J. Kelly, and C. Hankin.

Online cycle detection and difference propagation for pointer analysis.

In *SCAM*, pages 3–12, 2003.

D. J. Pearce, P. H. J. Kelly, and C. Hankin.

Efficient field-sensitive pointer analysis for C.

In *PASTE*, pages 37–42, 2004.

B. Steensgaard.

Points-to analysis in almost linear time.

In *POPL*, pages 32–41, 1996.

J. Whaley and M. S. Lam.

Cloning-based context-sensitive pointer alias analysis using binary decision diagrams.

In *PLDI*, pages 131–144, 2004.

J. Zhu.

Towards scalable flow and context sensitive pointer analysis.

In *DAC*, pages 831–836, 2005.