Approximating Inclusion-based Points-to Analysis

Rupesh Nasre
Indian Institute of Science, Bangalore, India
nasre@csa.iisc.ernet.in

Abstract
It has been established that achieving a points-to analysis that is scalable in terms of analysis time typically involves trading off analysis precision and/or memory. In this paper, we propose a novel technique to approximate the solution of an inclusion-based points-to analysis. The technique is based on intelligently approximating pointer- and location-equivalence across variables in the program. We develop a simple approximation algorithm based on the technique. By exploiting various behavioral properties of the solution, we develop another improved algorithm which implements various optimizations related to the merging order, proximity search, lazy merging and identification frequency. The improved algorithm provides a strong control to the client to trade off analysis time and precision as per its requirements. Using a large suite of programs including SPEC 2000 benchmarks and five large open source programs, we show how our algorithm helps achieve a scalable solution.

1. Introduction
Points-to analysis enables several compiler optimizations and remains an important static analysis technique. A flow-insensitive analysis ignores the control flow in the program and, in turn, assumes that the statements could be executed in any order. A context-insensitive analysis ignores the calling context of a statement while computing the points-to information. We deal with flow-insensitive, context-insensitive points-to analysis in this work.

Since alias analysis is undecidable [16], approximation is necessary for a scalable implementation. We propose an online approximation technique based on the notion of pointer- and location-equivalences [6]. The two kinds of equivalences are originally defined as offline optimizations; we extend those to a dynamic setting. We use auxiliary functions \( \text{ptsto}(p) \) to denote the points-to set of \( p \) and \( \text{ptdby}(o) \) to denote the set of pointers pointing to \( o \).

**Pointer-equivalence (PE)** Two pointers \( p_1 \) and \( p_2 \) are pointer-equivalent, if \( \text{ptsto}(p_1) = \text{ptsto}(p_2) \).

**Location-equivalence (LE)** Two objects \( o_1 \) and \( o_2 \) are location-equivalent, if \( \text{ptdby}(o_1) = \text{ptdby}(o_2) \).

Literature defines both these kinds of equivalences in an exact form. We generalize the definitions to cover approximations. Using the approximate versions of equivalences, we develop an algorithm to compute safe but approximate points-to information while quantifying the precision-loss.

Major contributions of this paper are as below.

- Two notions: approximate pointer- and location-equivalence across program variables.
- A simple new sound approximation algorithm for inclusion-based points-to analysis.
- Several optimizations which help achieve a scalable solution and an improved approximation algorithm.
- Evaluation using 21 programs to illustrate the promise of a scalable solution.

2. Preliminaries

2.1 Points-to Analysis
A typical (and efficient) way of computing the fixpoint of a flow-insensitive inclusion-based points-to analysis is to iteratively build a constraint graph \( G(V, E) \) where \( V \) is a set of nodes representing variables in the input constraints and \( E \) is a set of directed edges across nodes. An edge from a node \( u \) to \( v \) indicates a transfer (copy) of points-to information from \( u \) to \( v \). Edges are added using copy, load and store constraints. An outline of the iterative analysis using constraint graph is given in Algorithm 1.

2.2 Running Example
Consider the following points-to statements.

\[
\begin{align*}
& a = &u; a = &v; a = &w; a = &x; b = &y; c = &z; u = &v; \\
& v = &u; p = &a; q = &b; r = &a; d = a; *p = c; *q = c;
\end{align*}
\]
Algorithm 1 Points-to Analysis using Constraint Graph.

Require: set 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 fixpoint

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Line</th>
<th>Constraint</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>a = &amp;v</td>
<td>ptsto(a) = {u}</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>a = &amp;w</td>
<td>ptsto(a) = {u,v}</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>a = &amp;x</td>
<td>ptsto(a) = {u,v,w}</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>b = &amp;y</td>
<td>ptsto(b) = {y}</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>c = &amp;z</td>
<td>ptsto(c) = {z}</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>u = &amp;v</td>
<td>ptsto(u) = {v}</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>v = &amp;u</td>
<td>ptsto(v) = {u}</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>p = &amp;a</td>
<td>ptsto(p) = {a}</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>q = &amp;b</td>
<td>ptsto(q) = {b}</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>r = &amp;a</td>
<td>ptsto(r) = {a}</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>d = a</td>
<td>α ← d</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>p = c</td>
<td>c ← a</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>n = c</td>
<td>c ← a</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>ptsto(a) = {u,v,w,x,z}, ptsto(b) = {y,z}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ptsto(d) = {u,v,w,x,z}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ptsto(p) = {a}, ptsto(q) = {b}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ptsto(r) = {a}, ptsto(u) = {v}, ptsto(v) = {u}</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Points-to Analysis using Constraint Graph.

Algorithm 1 works on these constraints as given in Table 1. The analysis requires two iterations to reach the fixpoint. The column titled Line denotes the line number of Algorithm 1 under execution. The last column indicates any changes (addition of an edge or modification to the points-to information at the nodes) done to the constraint graph as an effect of processing a constraint.

At the end of the analysis, the fixpoint contains the following points-to information (18 points-to pairs).

\[
\begin{align*}
ptsto(a) &= \{u, v, w, x, z\}, 
ptsto(b) &= \{y, z\}, 
ptsto(c) &= \{z\}, 
ptsto(d) &= \{u, v, w, x, z\}, 
ptsto(p) &= \{a\}, 
ptsto(q) &= \{b\}, 
ptsto(r) &= \{a\}, 
ptsto(u) &= \{v\}, 
ptsto(v) &= \{u\} 
\end{align*}
\]

(1)

In this example, since \(ptsto(a) = ptsto(d) = \{u, v, w, x, z\}\), pointers \(a\) and \(d\) are PE. Since \(ptdb(y) = ptdb(x) = \{a, d\}\), objects \(w\) and \(x\) are LE.

2.3 Approximate Equivalences

**approximate pointer-equivalence (APE)** Pointers \(p_1\) and \(p_2\) are approximate pointer-equivalent with similarity \(\alpha\) if \(\text{sim}(ptsto(p_1), ptsto(p_2)) \geq \alpha, 0 \leq \alpha \leq 1\) for some function \(\text{sim}()\) which finds set similarity.

**approximate location-equivalence (ALE)** Objects \(o_1\) and \(o_2\) are approximate location-equivalent with similarity \(\beta\) if \(\text{sim}(ptdb(y_1), ptdb(y_2)) \geq \beta, 0 \leq \beta \leq 1\)

The function \(\text{sim}()\) can be defined in various ways. A customary definition of similarity between points-to sets is as below, which we use throughout this text.

\[
\text{sim}(s_1, s_2) = \frac{s_1 \cap s_2}{s_1 \cup s_2} \quad (2)
\]

When the points-to sets are implemented using sparse bitmaps, the function \(\text{sim}()\) has \(O(n)\) time-complexity where \(n\) is the number of variables in the program. An advantage of using the above definition for \(\text{sim}()\) is that it can be easily extended to more than two sets.

\[
\text{sim}(s_1, s_2, \ldots, s_n) = \frac{s_1 \cap s_2 \cap \ldots \cap s_n}{s_1 \cup s_2 \cup \ldots \cup s_n} \quad (3)
\]

When \(\alpha = 1\), APE equals PE. When \(\beta = 1\), ALE equals LE. When \(\alpha = 0\), the corresponding pointers never point to the same object. When \(\beta = 0\), the corresponding objects are never pointed to by the same pointer. This information may be used for extracting parallelism or for further optimizations. In general, the fractions \(\alpha\) and \(\beta\) could be used for speculative optimizations. As an example, the work on probabilistic points-to analysis [10] can be benefited by incorporating the definitions of APE and ALE.

**Example.** Defining similarity across points-to sets using Equation 2 and setting its threshold fractions \(\alpha\) and \(\beta\) to be 0.5, we get the following approximations.

- \(b\) and \(c\) are APE with \(\text{sim}(ptsto(b), ptsto(c)) = 0.5\).
- \(x\) and \(z\) are ALE with \(\text{sim}(ptdby(x), ptdby(z)) = 0.5\).
- Since \(w\) and \(x\) are LE, \(w\) and \(z\) are also ALE with \(\text{sim}(ptdby(w), ptdby(z)) = 0.5\).
- Note that \(a\) and \(c\) are not APE (for \(\alpha = 0.5\)) as \(\text{sim}(ptsto(a), ptsto(c)) = 0.25 < 0.5\).
- Also, \(y\) and \(z\) are not ALE (for \(\beta = 0.5\)) as \(\text{sim}(ptdb(y), ptdby(z)) = 0.25 < 0.5\).

2.4 Merging the Approximates

Once two variables are identified as equivalent, the nodes corresponding to them in the constraint graph can be merged.

Merging nodes \(n_1\) and \(n_2\) into \(n\) involves the following steps.

1. \(ptsto(n_1) = ptsto(n_2)\), and shrinking the sets \(ptsto(n_1)\) and \(ptsto(n_2)\).
2. incoming\((n) = incoming(n_1) \cup incoming(n_2)\), and removing the edges incoming\((n_1)\) and incoming\((n_2)\).
3. outgoing\((n) = outgoing(n_1) \cup outgoing(n_2)\), and removing the edges outgoing\((n_1)\) and outgoing\((n_2)\).
4. removing the nodes \(n_1\) and \(n_2\).
5. renaming \(n_1\) and \(n_2\) by \(n\) in the points-to constraints.

Node merging is a frequent operation in graph algorithms and can be performed in amortized \(O(1)\) time. Merging of nodes reduces the number of variables tracked during the analysis and helps achieve scalability.
3. Approximate Points-to Analysis Algorithm

In the example in the previous section, we identified APEs and ALEs after the complete analysis. However, the main advantage of adding approximation is to reduce analysis time and storage requirements during the analysis. Therefore, the analysis needs to make intelligent assumptions about the relationships between variables based on partial points-to sets.

Using this notion of calculating similarity online, we now present our naive approximation algorithm, shown in Algorithm 2. Compared to the exact analysis of Algorithm 1, the approximation algorithm has two more input parameters, \( \alpha \) and \( \beta \), to identify the pointer- and location-equivalence thresholds and an additional code sequence between Lines 5 and 11. The for-loop is a nested loop running over each unique pair of variables \((x, y)\). Lines 6 and 7 compute similarities for pointer- and location-equivalence using function \( \sim() \). The similarities are checked against the input thresholds to identify APEs and ALEs. If a similarity measure is above a threshold, the nodes corresponding to the variables \( x \) and \( y \) are merged (Section 2.4). This process is repeated in every iteration after points-to information is propagated. Identification of the approximates is done prior to processing load and store constraints (Line 12) to reduce the overhead of the node merging.

Various actions taken by Algorithm 2 on our running example with \( \alpha = 0.5 \), \( \beta = 0.7 \) are detailed in Table 2. The processing prior to the repeat-until-loop is similar to the exact algorithm. In Iteration 1 after points-to information propagation, the code for finding similarities across variables is executed. It identifies sim(\( \text{ptsto}(a), \text{ptsto}(d) \)) = 1.0 ≥ \( \alpha \). Thus, \( a \) and \( d \) are identified as APEs and their nodes are merged. Similarly, the nodes \( p \) and \( x \) are also merged. In the same iteration, sim(\( \text{ptdby}(w), \text{ptdby}(x) \)) = 1.0 ≥ \( \beta \). Thus, \( w \) and \( x \) are identified as ALEs and their nodes are merged. Note that sim(\( \text{ptdby}(u), \text{ptdby}(v) \)) = 0.66 < \( \beta \). Thus, \( u \) and \( v \) are not merged. Line 12 of Iteration 1 evaluates load and store constraints and adds edges from \( c \) to the merged node \([a, d]\) and to node \( b \). Iteration 2 begins with propagating points-to information across edges updating ptsto([a,d]) and ptsto(b) to additionally contain object \( z \). Once again, the code for finding the equivalences is triggered which identifies \( b \) and \( c \) as APEs since \( \sim(\text{ptsto}(b), \text{ptsto}(c)) = 0.5 \geq \alpha \). Note again that \( y \) and \( z \) are not identified as ALEs because \( \sim(\text{ptdby}(y), \text{ptdby}(z)) = 0.5 < \beta \). Line 12 of Iteration 2 does not add any more edges, but since nodes \( b \) and \( c \) were merged, a new edge \( b \rightarrow [a, d] \) was implicitly added. Hence, another iteration is necessary. Iteration 3 simply propagates object \( y \) from merged node \([b, c]\) to \([a, d]\). No further equivalences are identified and the complex constraints do not add more edges, suggesting a fixpoint. Thus, the final points-to information computed by Algorithm 2 is:

\[
\begin{align*}
\text{ptsto}(a) & = \{u, v, w, x, z\}, \\
\text{ptsto}(b) & = \{y, z\}, \\
\text{ptsto}(c) & = \{y, z\}, \\
\text{ptsto}(d) & = \{u, v, w, x, z\}, \\
\text{ptsto}(p) & = \{a\}, \\
\text{ptsto}(q) & = \{b\}, \\
\text{ptsto}(r) & = \{a\}, \\
\text{ptsto}(u) & = \{v\}, \\
\text{ptsto}(v) & = \{u\}
\end{align*}
\]

(4)

Note that compared to the points-to information computed by the exact Algorithm 1 which computed 18 points-to pairs (see Label 1), the approximation algorithm computed only one additional pair (\( c \) pointing to \( y \)) with moderate values of \( \alpha \) and \( \beta \). In fact, for \( \alpha = 0.6, \beta = 0.7 \), Algorithm 2 computes the same information as that by Algorithm 1.

Note that Algorithm 2 required one more iteration compared to the exact version over the running example. In fact, this naive algorithm cannot be practically used for a scalable implementation. The issue is mainly due to the nested for-loop at Line 5 that has quadratic complexity. Further, the naive approximation algorithm does not exploit any properties of the solution behavior. We tackle these issues in our improved approximation algorithm next.

4. Optimized Approximation Algorithm

Eager versus Lazy Merging

Algorithm 2 performs the identification of approximate equivalent variables and their merging together, i.e., in an eager manner. By decoupling
the two tasks and performing the merging operation lazily after identification of all the potential equivalences can lead to more opportunities for merging. For instance, consider the following points-to information.

\[
\begin{align*}
\text{ptsto}(p_1) &= ab, \text{ptsto}(p_2) = bc, \text{ptsto}(p_3) = cd, \\
\text{ptsto}(p_4) &= de, \text{ptsto}(p_5) = ef, \text{ptsto}(p_6) = fg
\end{align*}
\]

Eager merging with \(\alpha = 0.5\) produces the following merged variables: \([a,b], [b,c], [a,b], [a,c],[b,c]\). However, lazy merging would wait for the identification of all possible equivalent variables: \([a,b], [b,c], [a,c],[b,c]\) and merge all of them together to produce a single merged variable \([a,b,c]\).

### Merging Order

In Algorithm 2, the order in which nodes are merged affects future equivalence opportunities. An example illustrating this behavior is described in Table 3. Initially, \text{ptsto}(a) = \{x,y,z\}, \text{ptsto}(b) = \{x,y,z\}, \text{ptsto}(c) = \{x\} and \(\alpha = 0.5, \beta = 0.5\). Depending upon the merging order, we get different results. In the first case, when the order is \((a,b),(c,a),(b,c)\), only a and b are merged, whereas in the second case, when the order is \((a,c),(b,c),(a,b)\), only \(a\) and \(b\) are merged, whereas in the second case, when the order is \((a,c),(b,c),(a,b)\), only \(a\) and \(b\) are merged, whereas in the second case, when the order is \((a,c),(b,c),(a,b)\), only \(a\) and \(b\) are merged, whereas in the second case, when the order is \((a,c),(b,c),(a,b)\), only \(a\) and \(b\) are merged, whereas in the second case, when the order is \((a,c),(b,c),(a,b)\), only \(a\) and \(b\) are merged, whereas in the second case, when the order is \((a,c),(b,c),(a,b)\), only \(a\) and \(b\) are merged.

#### Proximity

It is rare that two nodes that are far apart in the constraint graph will have similar points-to information. Typically, approximately similar nodes have common ancestors from which the points-to information flows. Thus, nodes with similar points-to information would often be in close proximity with each other. We exploit this property to improve the similarity checking algorithm by considering only those nodes that are \(k\)-reachable from the node under question, i.e., nodes that are reachable by traversing at most \(k\)-edges. This optimization drastically improves the analysis time without reducing much precision.

### Equivalence Identification Frequency

Instead of toying with the parameters \(\alpha\) and \(\beta\), a client may specify maximum memory \(\text{MAXMEM}\) allowed to store points-to information and let the analysis make the best use of it to achieve maximum possible precision. We deal with this requirement by altering the equivalence identification frequency. In Algorithm 2, we check for approximate equivalences in every iteration. Instead, the identification can be triggered when the amount of memory consumed exceeds \(\text{MAXMEM}\). Decreasing values of similarity (1.0, 0.9, 0.8, ...) can then be tried to find approximate equivalences until the amount of consumed memory reduces to an allowed threshold. This kind of gradual reduction in the similarity thresholds makes a judicious use of the available memory without reducing too much precision.

### 4.1 The Algorithm

Based on the above optimizations, we design a new and improved points-to analysis based on identifying approximate equivalences. The analysis is shown in Algorithm 3. To avoid clutter, we have removed the code corresponding to equivalence identification frequency using input parameter \(\text{MAXMEM}\). A client can use the input parameters to control the algorithm behavior to suit its needs.

Similar to Algorithm 2, Algorithm 3 starts with processing address-of constraints (Line 1) and adding edges to the constraint graph corresponding to the copy constraints (Line 2). The \texttt{repeat-until} loop at Line 3 iteratively computes sound approximate points-to information until a fixpoint. As before, propagation of points-to information across edges in the constraint graph is done in Line 4 and new edges are added by processing complex constraints in Line 62.

Lines 5 to 28 (Lines 28 to 53) process constraints if the merging strategy MT is lazy (eager). In each case, the \texttt{for}-loops at Line 6 and Line 29 operate on points-to sets having size above an input threshold \(\text{MINPTSTO}\). The inner \texttt{for}-loops at Lines 8 and 31 only check for nodes that are in proximity \(k\) (\(k\)-reachable). Computation of similarities (APE and ALE) using function \texttt{sim()} is done at Lines 9–10 and Lines 32–33. For lazy merging strategy, we need to cache the pairs that are getting processed in Line 11.

In case of eager merging, depending upon the merging order, the nodes in \(k\)-proximity are ordered (Lines 34–38). Next, the necessary pairs of nodes are checked for similarity (Lines 15 and 39). The next step of merging nodes begins now. Depending upon the value of \text{MAXMERGES} (to limit cascaded merges), \(x\) and \(y\) are merged (Lines 17 and 42). The rest of the code in the \texttt{for}-loops is bookkeeping. In the end, in Lines 54–61, the values of the similarity thresholds are updated adaptively to balance precision and analysis time. This process is repeated until a fixpoint. The fixpoint computes an over-approximation of that computed by Andersen’s exact analysis.

### 5. Experimental Evaluation

We evaluate our approach using 16 SPEC 2000 C/C++ benchmarks and five large open source programs, namely \texttt{httpd}, \texttt{sendmail}, \texttt{ghostscript}, \texttt{gdb} and \texttt{wine-server}. We use LLVM framework [14] for our experiments. Table 5 (Appendix A) lists the benchmarks characteristics. All the experiments are carried out on an Intel Xeon machine with 2 GHz clock, 4 MB L2 cache and 3 GB RAM.

Our approximate analysis stores points-to information in sparse bitmaps. The baseline approach uses a field-insensitive version of Andersen’s points-to analysis [1]. It also performs optimizations like offline variable substitution.
Algorithm 3 Optimized Approximate Points-to Analysis

```
Require: set of points-to constraints, pointer-equivalence threshold α, location-equivalence threshold β, merging order MO, merging type MT, merging limit MAXMERGES, proximity k, maximum memory MAXMEM, adaptive threshold AD, minimum pointee size MINPTSTO
Process address-of constraints
Add edges to constraint graph G using copy constraints
repeat
Propagate points-to information in G
for each changed variable x and ptsto(x).size ≥ MINPTSTO do
  k-proximity = get nodes k-reachable from x
  simx = simx(ptsto(x), ptsto(y))
  cached-pairs = cached-pairs ∪ (x,y)
end for
for each variable pair (x, y) ∈ cached-pairs do
  if x.nmerges ≤ MAXMERGES and y.nmerges ≤ MAXMERGES then
    merge(x, y)
  end if
  ++x.nmerges
  ++y.nmerges
end for
for each variable x and ptsto(x).size ≥ MINPTSTO do
  k-proximity = get nodes k-reachable from x
  simx = simx(ptsto(x), ptsto(y))
end for
if MO = increasing similarity then
  sort k-proximity on simx
else if MO = decreasing similarity then
  reverse sort k-proximity on simx
end if
if x.nmerges ≥ α or y.nmerges ≥ β then
  ++iterationmergsα
else
  ++iterationmergsβ
end if
for each variable x and ptsto(x).size ≥ MINPTSTO do
  if x.nmerges ≤ MAXMERGES and y.nmerges ≤ MAXMERGES then
    merge(x, y)
  end if
  ++x.nmerges
  ++y.nmerges
end for
for each variable pair (x, y) ∈ cached-pairs do
  if x.nmerges ≥ α or y.nmerges ≥ β then
    ++iterationmergsα
  else
    ++iterationmergsβ
  end if
end for
end if
until fixpoint
```

[17] and online cycle elimination [4]. The analysis times and memory requirement are given in Table 4. On average, it requires 14 seconds and 365 MB memory for analyzing each benchmark. The following sections show the effect of our approximate analysis in terms of analysis time, memory and precision loss normalized to the base analysis.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (s)</th>
<th>Memory (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>6.546</td>
<td>1209037</td>
</tr>
<tr>
<td>perlbench</td>
<td>2.345</td>
<td>669104</td>
</tr>
<tr>
<td>vortex</td>
<td>1.445</td>
<td>9271</td>
</tr>
<tr>
<td>eon</td>
<td>2.446</td>
<td>382731</td>
</tr>
<tr>
<td>parser</td>
<td>0.889</td>
<td>7526</td>
</tr>
<tr>
<td>gap</td>
<td>2.777</td>
<td>86646</td>
</tr>
<tr>
<td>vpr</td>
<td>0.601</td>
<td>975</td>
</tr>
<tr>
<td>crafty</td>
<td>0.595</td>
<td>470</td>
</tr>
<tr>
<td>mesa</td>
<td>2.045</td>
<td>2517</td>
</tr>
<tr>
<td>ammp</td>
<td>0.575</td>
<td>584</td>
</tr>
<tr>
<td>rwolf</td>
<td>0.686</td>
<td>2297</td>
</tr>
<tr>
<td>gzip</td>
<td>0.456</td>
<td>38</td>
</tr>
<tr>
<td>bzip2</td>
<td>0.396</td>
<td>29</td>
</tr>
<tr>
<td>mcf</td>
<td>0.382</td>
<td>32</td>
</tr>
<tr>
<td>equake</td>
<td>0.436</td>
<td>32</td>
</tr>
<tr>
<td>art</td>
<td>0.485</td>
<td>15</td>
</tr>
<tr>
<td>sendmail</td>
<td>3.311</td>
<td>752933</td>
</tr>
<tr>
<td>httpd</td>
<td>4.447</td>
<td>869314</td>
</tr>
<tr>
<td>ghostscript</td>
<td>84.497</td>
<td>1652383</td>
</tr>
<tr>
<td>gdb</td>
<td>174.355</td>
<td>1797623</td>
</tr>
<tr>
<td>wine-server</td>
<td>4.456</td>
<td>162949</td>
</tr>
<tr>
<td>average</td>
<td>14.01</td>
<td>365071.71</td>
</tr>
</tbody>
</table>

Figure 1: Effect of a representative configuration

Figure 1 illustrates the effect of our approximate points-to analysis over each benchmark for a representative configuration: α = β = 0.8. Left y-axis represents normalized time and memory, while right y-axis represents precision loss. The bars represent time, the dotted line represents memory and the dashed line represents precision loss. All the quantities are better if smaller.

We observe that analysis time and memory vary largely in tandem. The precision is mainly guided by the amount of memory consumed and varies in proportion to it. Especially for large benchmarks (e.g., gcc, perlbench, httpd, ghostscript, gdb and wine-server) the improvement in analysis time is significant for smaller values of α, β. The configuration α = β = 0.8 improves analysis time by 5%, memory requirement by 16% and reduces precision by a small 3.2%.

The effect of optimizations is discussed in the appendix.
6. Related Work

The area of points-to analysis is rich in literature. See [8] for a survey. We mention only the most relevant related work.

Most scalable algorithms proposed are based on unification [18][5]. Steensgaard[18] proposed an almost linear time algorithm that has been shown to scale to millions of lines of programs. However, precision of unification based approaches has always been an issue. Inclusion based approaches [1] that work on subsupersetion of points-to sets rather than a bidirectional similarity offer a better precision at the cost of cubic complexity. Due to the enormous growth of code bases to billions of code lines, the complexity of an inclusion-based approach has been proven to be limiting. Hence, several techniques [2][7][13][19] have been proposed to improve upon the original work by Andersen. [2] extracts similarity across points-to sets while [19] exploits similarity across contexts to make brilliant use of BDD to store information in a succinct manner. The idea of bootstrapping [11] first reduces the problem by partitioning the set of pointers into disjoint alias sets using a fast and less precise algorithm (e.g., [18]) and later running more precise analysis on each of the partitions. Complete context-sensitivity requires huge amount of storage and analysis time. Hence, approximate representations were introduced to trade off precision for scalability. [3] proposed one level flow. [12] unified contexts, while [15] hashed contexts to alleviate the need to store complete contexts.

Various enhancements have also been made for the inclusion-based analyses: online cycle elimination [4] to break dependence cycles on the fly and offline variable substitution [17] to reduce the number of pointers tracked.

All the points-to analyses proposed in literature are approximate since precise points-to analysis is NP-Hard [9]. The upper bound of precision of our approximate analysis is that obtained by inclusion-based points-to analysis. To the best of our knowledge, ours is the first work that focuses on computing explicit similarity across points-to sets.

7. Conclusion

In this paper, we generalized the notions of pointer- and location-equivalences to include approximation. This allowed us to merge more nodes in the constraint graph improving the analysis time and storage requirements. By controlling the similarity thresholds, we illustrated how analysis time and precision can be traded off against each other. Based on identifying and merging approximate pointer equivalences and approximate location equivalences, we developed a simple algorithm to approximate the solution of an inclusion-based points-to analysis. We argued that the naive algorithm is inefficient and by exploiting the solution behavior of the analysis, we proposed several optimizations related to the merging order, cascaded merging, proximity search, lazy merging, identification frequency and adaptive thresholds. Based on these optimizations, we developed another improved algorithm and evaluated its effectiveness using a large suit of programs including SPEC 2000 benchmarks and five large open source programs. We illustrated that our approximation algorithm gives a good handle to a client to control analysis time and precision. We believe that this work would help the compiler community achieve scalable analyses and transformations.

References

Table 5: Benchmark characteristics. KLOC is the kilo lines of unprocessed source code. Total Inst is the total number of three address code instructions in LLVM intermediate format after optimizing at -O2 level. Pointer Inst is the total number of pointer instructions that get processed by the analysis. Func is the number of functions in each program.

A. Additional Experimental Evaluation

A.1 Effect of $\alpha$

Figure 2 illustrates the effect of the configuration parameter $\alpha$ on analysis time and precision for $\beta = 0.8$. Memory requirement and precision vary non-linearly with increasing $\alpha$. This happens because for smaller values of $\alpha$ (less than 0.5) the opportunities for merging go on rapidly increasing. This results in a large number of APE pointers, reducing precision. In fact, for very small values of $\alpha$ (less than 0.2), the cost of merge operation starts dominating, resulting in marginal increase in the analysis time.

A.2 Effect of $\beta$

Figure 3 illustrates the effect of the configuration parameter $\beta$ for $\alpha = 0.8$. We observe that the effect of $\beta$ on analysis time, memory requirement and precision have similar nature as that of $\alpha$ but the changes are smaller for $\beta$. For instance, for $\alpha = 0.8$, analysis time varies from 93% to 98% for different values of $\beta$ whereas for $\beta = 0.8$, analysis time varies from 83% to 98% for different values of $\alpha$ (Figure 2). Thus, we conclude that APE is more significant than ALE.

A.3 Eager and Lazy Merging

The observations so far are based on eager merging. Recall that in lazy merging, identification of equivalences and merge operation are decoupled and the merging is performed at the end of finding all the potential equivalences. Figure 4 shows the effect of the two merging strategies for configurations $\alpha = \beta = 0.2, 0.4, 0.6, 0.8, 0.9, 1.0$. The first box shows the analysis time of eager merging while the latter shows it for lazy merging. Lazy merging improves the analysis time within 8%. This happens because lazy merging
involves lesser number of merges. Therefore, the memory requirement also improves by a similar margin resulting in corresponding reduction in precision.

### A.4 Effect of Merge Order

Figure 5 shows the effect of various merging orders followed in an eager merging strategy. The first box represents analysis time for no order amongst nodes (random). The other two boxes represent respectively the non-increasing (rsorted) and non-decreasing (sorted) orders amongst nodes. The configurations illustrated are for \( \alpha = \beta = 0, 0.2, 0.4, 0.6, 0.8, 0.9, 1.0 \). We observe that an rsorted order reduces analysis time by around 10% while a sorted order requires around 8% more time than a random order. There are proportionate effect in memory requirements and precision. This happens because an rsorted order merges a node with another *most similar* node resulting in overall lesser opportunities for merging. In contrast, a sorted order merges a node with another *least similar* node resulting in overall more opportunities for merging reducing the precision.

### A.5 Effect of Proximity

While other configuration parameters and strategies result in a trade-off between analysis time and precision, the proximity value has the ability to reduce analysis time drastically affecting little amount of precision. Recall that a proximity search at a node compares its similarity with nodes only in proximity. Figure 6 shows the effect of a proximity search for proximity = *infinity* (original analysis), 100, 50, 40, 30, 20, 10, 5. The results suggest that comparing similarity only with a very small number of neighboring nodes is sufficient to gain complete benefits of approximations. Specifically, for proximity = 10, the normalized analysis time is 0.62, memory requirement is 0.87 and precision loss is only 2.6%.

It should be emphasized that the graphs show only the average values. For small benchmarks (e.g., art, equake, mcf, etc.), proximity = 5 is also sufficient, whereas for large benchmarks (e.g., ghostscript, gdb, sendmail, etc.), proximity = 50 is a better choice to get overall benefits (not shown in the figure).

### A.6 Equivalence Identification Frequency

One of the biggest advantages of our approximate analysis is that a client can pre-specify a fixed amount of maximum memory to be used by our points-to analysis and get as much precision as possible within the limit. Figure 7 shows the effect on analysis time and precision when the limit MAXMEM is set to various values ranging over *infinity* (original analysis), 1 GB, 100 MB, 50 MB, 10 MB, 1 MB, 500 KB, 100KB. We observe that the analysis time reduces sharply with decreasing MAXMEM. For instance, for MAXMEM = 1 MB, the normalized analysis time is 0.73 with a precision loss of 36%.

As in the case of proximity, it should be emphasized here too that the observations are averaged across benchmarks and different values of MAXMEM suffice for different benchmarks. For instance, for large benchmarks like gdb, even a value of 1 GB of MAXMEM results in 34% of precision loss (not shown).