Parallel Flow-Sensitive Pointer Analysis by Graph-Rewriting

Vaivaswatha Nagaraj
R. Govindarajan
Indian Institute of Science, Bangalore

PACT 2013
Outline

- Introduction
- Background
- Flow-sensitive graph-rewriting formulation
- Implementation and Results
- Conclusion
Outline

- Introduction
- Background
- Flow-sensitive graph-rewriting formulation
- Implementation and Results
- Conclusion
Flow-sensitive pointer analysis

1. $x = &a$
   $y = &b$

2. $z = x$
   z points-to {?}

3. $z = y$
   z points-to {?}

4. (Blank)

5. (Blank)
Flow-sensitive pointer analysis

1. \( x = \&a \)
   \( y = \&b \)
   \( x \text{ points-to } a \)
   \( y \text{ points-to } b \)

2. \( z = x \)
   \( x \text{ points-to } a \)
   \( y \text{ points-to } b \)
   \( z \text{ points-to } a \)

3. \( z = y \)
   \( x \text{ points-to } a \)
   \( y \text{ points-to } b \)
   \( z \text{ points-to } b \)

4. (Resulting state)
Flow-sensitive vs flow-insensitive

1. $x = &a$
   $y = &b$
   - $x$ points-to $a$
   - $y$ points-to $b$

2. $z = x$
   - $x$ points-to $a$
   - $y$ points-to $b$
   - $z$ points-to \{a, b\}

3. $z = y$
   - $x$ points-to $a$
   - $y$ points-to $b$
   - $z$ points-to \{a, b\}

4. $z$ points-to \{a, b\}
Outline

- Introduction
- Background
  - Staged flow-sensitive analysis
  - Graph-rewriting
- Flow-sensitive graph-rewriting formulation
- Implementation and Results
- Conclusion
Staged flow-sensitive pointer analysis

[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO’11]

- Scales to large programs
- Faster, less precise analysis used to speed up the primary analysis
Staged flow-sensitive pointer analysis

[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO’11]

• Scales to large programs
• Faster, less precise analysis used to speed up the primary analysis

Our goal: Parallelize the staged flow-sensitive pointer analysis
Staged flow-sensitive analysis

[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO’11]

In a traditional analysis, points-to info of both $a$ & $b$ will be propagated to both 2 and 3.
Staged flow-sensitive analysis

$\begin{align*}
\text{Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO’11} \\
\end{align*}$
Staged flow-sensitive analysis

Points-to info of only 
b is required here

Points-to info of only 
a is required here

\[ x_1 \text{ points-to } \{a,b\} \]
\[ a \text{ points-to } \{p,q\} \]
\[ b \text{ points-to } \{r,s\} \]
Staged flow-sensitive analysis

[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO’11]

How does staged analysis work?

(1) Use a fast analysis to get less precise points-to information

\[ \begin{align*}
  x_1 & \rightarrow \{ a, b \} \\
  z_1 & \rightarrow \{ a \} \\
  y_1 & \rightarrow \{ b \} \\
  w_1 & \rightarrow \{ c \}
\end{align*} \]
Staged flow-sensitive analysis

[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO’11]

How does staged analysis work?

1. Use a fast analysis to get less precise points-to information

\[ x_1 \rightarrow \{a,b\} \]

\[ z_1 \rightarrow \{a\} \]

\[ y_1 \rightarrow \{b\} \]

\[ w_1 \rightarrow \{c\} \]

2. Use the less precise info to find variables potentially referenced at indirections

\[ *x_1 = u_1 \]

\[ a_1 = \chi(a_0) \]

\[ b_1 = \chi(b_0) \]

3. \[ *v_1 = *y_1 \]

4. \[ *w_1 = t_2 \]

\[ c_2 = \chi(c_1) \]

5. \[ \mu(a_1) \]

\[ \mu(b_1) \]
Staged flow-sensitive analysis

[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO’11]

How does staged analysis work?

Flow-sensitivity - All variables in SSA form

1. \( *x_1 = u_1 \)

2. \( v_1 = *y_1 \)

3. \( w_1 = *z_1 \)

4. \( *w_1 = t_2 \)

\[ a_1 = \chi(a_0) \]
\[ b_1 = \chi(b_0) \]
\[ c_2 = \chi(c_1) \]
Staged flow-sensitive analysis

[Flow-sensitive pointer analysis for millions of lines of code – Ben Hardekopf. et al. - CGO’11]

How does staged analysis work?

1. \[ *x_1 = u_1 \]
2. \[ v_1 = *y_1 \]
3. \[ w_1 = *z_1 \]
4. \[ *w_1 = t_2 \]

Flow-sensitivity -
All variables in SSA form

a_1 = \chi(a_0) 

b_1 = \chi(b_0) 

\mu(a_1) 

\mu(b_1) 

\mu(a_1) 

\mu(b_1) 

Propagation of information is thus optimized

(3) Build def-use chains and propagate only along these chains
Parallelizing the staged analysis

Can the staged flow-sensitive pointer analysis be parallelized and scaled to multiple cores?
Towards a parallel algorithm

• Flow-insensitive pointer analysis has already been parallelized¹

• Parallelization of flow-insensitive analysis involves transforming it to a graph-rewriting problem – expose amorphous data-parallelism

¹[Parallel Inclusion-based Points-to Analysis - Mario Mendez-Lojo, et al. - OOPSLA’10]
Towards a parallel algorithm

- Flow-insensitive pointer analysis has already been parallelized\#.
- Parallelization of flow-insensitive analysis involves transforming it to a graph-rewriting problem – expose *amorphous data-parallelism*.

**Our goal:** Parallelize the staged flow-sensitive pointer analysis.

Formulate it as a graph-rewriting problem.

\# [Parallel Inclusion-based Points-to Analysis - Mario Mendez-Lojo, et al. - OOPSLA’10]
Outline

• Introduction
• Background
  – Staged flow-sensitive analysis
  – Graph-rewriting
• Flow-sensitive graph-rewriting formulation
• Implementation and Results
• Conclusion
Graph-rewriting

[Parallel Inclusion-based Points-to Analysis - Mario Mendez-Lojo, et al. - OOPSLA’10]

Points-to constraint

\[ x = \&a \]

\[ y = x \]

Constraint graph

\[ x \]

\[ a \]

\[ p \]

\[ x \]

\[ y \]

\[ c \]
Graph-rewriting

[Parallel Inclusion-based Points-to Analysis - Mario Mendez-Lojo, et al. - OOPSLA'10]

Points-to constraint

\[ x = &a \]
\[ y = x \]

Constraint graph

\[ x \rightarrow p \rightarrow a \]
\[ x \rightarrow c \rightarrow y \]

Apply rewrite-rule

Example: copy rewrite rule
Outline

- Introduction
- Background
- Flow-sensitive graph-rewriting formulation
- Implementation and Results
- Conclusion
Graph-rewriting formulation

Graph:
Nodes: Variables in the program
Edges: Points-to, copy, load, store, etc ... 
Rewrite rules?

\[ *x_1 = u_1 \]
\[ a_1 = \chi(a_0) \]
\[ b_1 = \chi(b_0) \]
\[ v_1 = *y_1 \]
\[ w_1 = *z_1 \]
\[ \mu(a_1) \]
\[ \mu(b_1) \]
\[ *w_1 = t_2 \]
\[ c_2 = \chi(c_1) \]
Challenges

Directly using the flow-insensitive graph formulation for flow-sensitive analysis leads to the following challenges

- **Spurious edges** – leads to imprecision

- **Strong and weak** updates at store constraints are not handled
Directly using the flow-insensitive graph formulation for flow-sensitive analysis leads to the following challenges

- **Spurious edges** – leads to imprecision
  - Solution: potential edges
- **Strong and weak** updates at store constraints are not handled
  - Solution: *klique* nodes
Outline

- Introduction
- Background
- Flow-sensitive graph-rewriting formulation
  - Problem of spurious edges
  - Handling strong and weak updates
- Implementation and Results
- Conclusion
Spurious edges

- Load rule for flow-insensitive analysis:

Points-to constraints

\[ y_1 = &a \]

\[ x_1 = *y_1 \]
Spurious edges

- Load rule for flow-insensitive analysis:

\[ y_1 = \&a \]
\[ x_1 = \ast y_1 \]
Spurious edges

- Load rule for flow-insensitive analysis:

Points-to constraints

\[ y_1 = \&a \]
\[ x_1 = \ast y_1 \]

Formulate graph

Apply load rule
Spurious edges

- Load rule for flow-insensitive analysis:

  Points-to constraints

  \[ y_1 = &a \]
  \[ x_1 = *y_1 \]

  Formulate graph

  \[ y_1 \]
  \[ \rightarrow \]
  \[ a \]
  \[ x_1 \]
  \[ \rightarrow \]
  \[ l \]

  Apply load rule

  \[ y_1 \]
  \[ \rightarrow \]
  \[ a \]
  \[ x_1 \]
  \[ \rightarrow \]
  \[ l \]

- Loss of precision when used for flow-sensitive analysis:

  \[ x_1 = *y_1 \]
  \[ \mu(a_1) \]

  \[ z_1 = *y_1 \]
  \[ \mu(a_2) \]
Spurious edges

- Load rule for flow-insensitive analysis:

Points-to constraints

\[ y_1 = \&a \]
\[ x_1 = *y_1 \]

Formulate graph

\[ y_1 \rightarrow a \]
\[ x_1 \rightarrow l \]

Apply load rule

\[ y_1 \rightarrow a \]
\[ x_1 \rightarrow l \]
\[ c \]

- Loss of precision when used for flow-sensitive analysis:

\[ x_1 = *y_1 \]
\[ \mu(a_1) \]

\[ z_1 = *y_1 \]
\[ \mu(a_2) \]
Spurious edges

- Load rule for flow-insensitive analysis:

Points-to constraints

\[ y_1 = \&a \]
\[ x_1 = *y_1 \]

Formulate graph

\[ y_1 \quad p \quad a \]
\[ x_1 \quad l \]

Apply load rule

\[ y_1 \quad p \quad a \]
\[ x_1 \quad l \]
\[ c \]

- Loss of precision when used for flow-sensitive analysis:

\[ x_1 = *y_1 \]
\[ \mu(a_1) \]

\[ z_1 = *y_1 \]
\[ \mu(a_2) \]

\[ y_1 \quad l \]
\[ x_1 \]

\[ a_1 \]
\[ a_2 \]
Spurious edges

- Load rule for flow-insensitive analysis:

Points-to constraints

\[
\begin{align*}
y_1 &= \&a \\
x_1 &= \ast y_1
\end{align*}
\]

Formulate graph

\[
\begin{align*}
y_1 &\rightarrow a \\
x_1 &\rightarrow p \\
x_1 &\rightarrow l
\end{align*}
\]

Apply load rule

\[
\begin{align*}
y_1 &\rightarrow a \\
x_1 &\rightarrow p \\
x_1 &\rightarrow l
\end{align*}
\]

- Loss of precision when used for flow-sensitive analysis:

\[
\begin{align*}
x_1 &= \ast y_1 \\
\mu(a_1) \\
z_1 &= \ast y_1 \\
\mu(a_2)
\end{align*}
\]
Spurious edges

- Load rule for flow-insensitive analysis:

\[
y_1 = &a \\
x_1 = *y_1
\]

Points-to constraints

Formulate graph

Apply load rule

- Loss of precision when used for flow-sensitive analysis:

\[
x_1 = *y_1 \\
\mu(a_1) \\
z_1 = *y_1 \\
\mu(a_2)
\]
Spurious edges

- Load rule for flow-insensitive analysis:

Points-to constraints

\[ y_1 = &a \]
\[ x_1 = *y_1 \]

Formulate graph

\[ y_1 \rightarrow p \rightarrow a \]
\[ x_1 \rightarrow l \]

Apply load rule

\[ y_1 \rightarrow p \rightarrow a \]
\[ x_1 \rightarrow l \]

- Loss of precision when used for flow-sensitive analysis:

Points-to constraints

\[ x_1 = *y_1 \]
\[ \mu(a_1) \]
\[ z_1 = *y_1 \]
\[ \mu(a_2) \]

Spurious edge

\[ c \]
Spurious edges

- Load rule for flow-insensitive analysis:

Points-to constraints

\[ y_1 = \&a \]
\[ x_1 = \star y_1 \]

Formulate graph

Apply load rule

- Loss of precision when used for flow-sensitive analysis:
Spurious edges

- Load rule for flow-insensitive analysis:

Points-to constraints

\[ y_1 = \&a \]
\[ x_1 = *y_1 \]

Formulate graph

Apply load rule

- Loss of precision when used for flow-sensitive analysis:

\[ x_1 = *y_1 \]
\[ z_1 = *y_1 \]
\[ \mu(a_1) \]
\[ \mu(a_2) \]
Solution: Potential edges

- A *potential* edge of type $t$ means that, there could be an actual edge of type $t$, between the same two nodes
- *Potential* edges are added during initial graph construction
- Some rewrite rules are modified to look for potential edges
Solution: Potential edges

- Modified load rule:

\[ y_1 = \&a \]
\[ x_1 = ^*y_1 \]
Solution: Potential edges

- Modified load rule:

\[ y_1 = \&a \]
\[ x_1 = \ast y_1 \]

Formulate graph

\[ y_1 \rightarrow a \]
\[ x_1 \]
\[ p \]
\[ l \]
\[ p_c \]

Modified load rule

\[ y_1 \rightarrow a \]
\[ x_1 \]
\[ p \]
\[ l \]
\[ c \]

- Modified rule applied to flow-sensitive analysis:

\[ x_1 = \ast y_1 \]
\[ \mu(a_1) \]
\[ z_1 = \ast y_1 \]
\[ \mu(a_2) \]

\[ x_1 \rightarrow a_1 \]
\[ p \]
\[ p_c \]
\[ \mu(a_1) \]
\[ y_1 \rightarrow a_2 \]
\[ p \]
\[ p_c \]
\[ \mu(a_2) \]
Solution: Potential edges

- Modified load rule:

\[ y_1 = \&a \]
\[ x_1 = *y_1 \]

Formulate graph

Modified load rule

- Modified rule applied to flow-sensitive analysis:

\[ x_1 = *y_1 \]
\[ \mu(a_1) \]
\[ z_1 = *y_1 \]
\[ \mu(a_2) \]
Outline

• Introduction
• Background
• Flow-sensitive graph-rewriting formulation
  – Problem of spurious edges
  – Handling strong and weak updates
• Implementation and Results
• Conclusion
Strong and weak updates

As the flow-sensitive analysis progresses:

\[ *x_1 = u_1 \]
\[ a_1 = \chi(a_0) \]
\[ b_1 = \chi(b_0) \]
Strong and weak updates

As the flow-sensitive analysis progresses:

How to update $a_1$?

\begin{align*}
\mathbf{x}_1^* &= \mathbf{u}_1 \\
\mathbf{a}_1 &= \chi(\mathbf{a}_0) \\
\mathbf{b}_1 &= \chi(\mathbf{b}_0)
\end{align*}
Strong and weak updates

As the flow-sensitive analysis progresses:

\[ *x_1 = u_1 \]
\[ a_1 = \chi(a_0) \]
\[ b_1 = \chi(b_0) \]

How to update \( a_1 \)?

\( x_1 \) points-to \( \{ a_1 \} \)
Strong and weak updates

As the flow-sensitive analysis progresses:

\[ *x_1 = u_1 \]
\[ a_1 = \chi(a_0) \]
\[ b_1 = \chi(b_0) \]

How to update \( a_1 \)?

Strong Update

\( x_1 \) points-to \( \{a_1\} \)

\( a_1 \leftarrow u_1 \)
Strong and weak updates

As the flow-sensitive analysis progresses:

\[ x_1 = u_1 \]
\[ a_1 = \chi(a_0) \]
\[ b_1 = \chi(b_0) \]

How to update \( a_1 \)?

Strong Update

\( x_1 \) points-to \( \{a_1\} \)

\[ a_1 \leftarrow u_1 \]

\( x_1 \) points-to \( \{a_1, b_1\} \)
Strong and weak updates

As the flow-sensitive analysis progresses:

\[ \mathbf{x}_1 = \mathbf{u}_1 \]

- \[ a_1 = \chi(a_0) \]
- \[ b_1 = \chi(b_0) \]

How to update \( a_1 \)?

**Strong Update**

\( x_1 \) points-to \( \{a_1\} \)

\[ a_1 \leftarrow u_1 \]

**Weak Update**

\( x_1 \) points-to \( \{a_1, b_1\} \)

\[ a_1 \leftarrow u_1 \cup a_0 \]
Strong and weak updates

\[ *x_1 = u_1 \]
\[ a_1 = \chi(a_0) \]
\[ b_1 = \chi(b_0) \]

*update* \( a_1 \) as

\[ \text{if } x_1 \text{ points-to } a \]
\[ \text{then} \]
\[ a_1 \leftarrow u_1 \]
\[ \text{end if} \]

\[ \text{if } x_1 \text{ points-to } b \]
\[ \text{then} \]
\[ a_1 \leftarrow a_0 \]
\[ \text{end if} \]
Strong and weak updates

Points-to set of $a_1$ depends on whether $x_1$ points-to $b_1$.

update $a_1$ as

\[ *x_1 = u_1 \]
\[ a_1 = \chi(a_0) \]
\[ b_1 = \chi(b_0) \]

if $x_1$ points-to a then
\[ a_1 \leftarrow u_1 \]
end if

if $x_1$ points-to b then
\[ a_1 \leftarrow a_0 \]
end if
Strong and weak updates

Points-to set of $a_1$ depends on whether $x_1$ points-to $b_1$

if $x_1$ points-to $a$
then
  $a_1 \leftarrow u_1$
end if

if $x_1$ points-to $b$
then
  $a_1 \leftarrow a_0$
end if
PACT'13

Strong and weak updates

\[ \star \chi_1 = u_1 \]
\[ a_1 = \chi(a_0) \]
\[ b_1 = \chi(b_0) \]

Solution: Connect all variables on the LHS of \( \chi \), within a store. Use a \textit{klique} node as the common connection.

\begin{center}
\begin{tikzpicture}
  \node (a1) at (0,0) [circle, draw] {a_1};
  \node (b1) at (0,-1) [circle, draw] {b_1};
  \node (k) at (0,-0.5) [circle, draw, fill=green] {k};
  \draw [->] (a1) -- (k);
  \draw [<-] (b1) -- (k);
\end{tikzpicture}
\end{center}

\textit{klique} node \( \Leftrightarrow \) store constraint
Strong and weak updates

\[
\begin{align*}
*x_1 &= u_1 \\
a_1 &= \chi(a_0) \\
b_1 &= \chi(b_0)
\end{align*}
\]

**update** \(a_1\) as

\[
\begin{cases}
\text{if } x_1 \text{ points-to } a & \text{then} \\
& a_1 \leftarrow u_1 \\
\end{cases}
\]

\[
\begin{cases}
\text{if } x_1 \text{ points-to } b & \text{then} \\
& a_1 \leftarrow a_0 \\
\end{cases}
\]
Strong and weak updates

\[
\begin{align*}
* x_1 & = u_1 \\
a_1 & = \chi(a_0) \\
b_1 & = \chi(b_0)
\end{align*}
\]

update \( a_1 \) as

\[
\begin{align*}
\text{if } x_1 \text{ points-to } a \\
\quad a_1 & \leftarrow u_1 \\
\text{end if}
\end{align*}
\]

\[
\begin{align*}
\text{if } x_1 \text{ points-to } b \\
\quad a_1 & \leftarrow a_0 \\
\text{end if}
\end{align*}
\]

Rewrite rule to update \( a_1 \):

![Diagram showing the rewrite rule and its effect on updating \( a_1 \).]
Strong and weak updates

\[ *x_1 = u_1 \]
\[ a_1 = \chi(a_0) \]
\[ b_1 = \chi(b_0) \]

update \( a_1 \) as

if \( x_1 \) points-to a
then
\[ a_1 \leftarrow u_1 \]
end if

if \( x_1 \) points-to b
then
\[ a_1 \leftarrow a_0 \]
end if
Strong and weak updates

*\( x_1 \) = \( u_1 \)

\( a_1 = \chi(a_0) \)

\( b_1 = \chi(b_0) \)

update \( a_1 \) as

if \( x_1 \) points-to \( a \)
then
\( a_1 \leftarrow u_1 \)
end if

if \( x_1 \) points-to \( b \)
then
\( a_1 \leftarrow a_0 \)
end if

Rewrite rule to update \( a_1 \): (2 steps)

Note: The rewrite rule shown here is not complete.
Strong and weak updates

*\( x_1 = u_1 \)

\( a_1 = \chi(a_0) \)

\( b_1 = \chi(b_0) \)

Note: The rewrite rule shown here is not complete.

Rewrite rule to update \( a_1 \): (2 steps)

if \( x_1 \) points-to \( a \)
then
  \( a_1 \leftarrow u_1 \)
end if

if \( x_1 \) points-to \( b \)
then
  \( a_1 \leftarrow a_0 \)
end if
Strong and weak updates

\[ *x_1 = u_1 \]
\[ a_1 = \chi(a_0) \]
\[ b_1 = \chi(b_0) \]

update \( a_1 \) as

if \( x_1 \) points-to \( a \)
then
\[ a_1 \leftarrow u_1 \]
end if

if \( x_1 \) points-to \( b \)
then
\[ a_1 \leftarrow a_0 \]
end if

Handled
Applying the rewrite rules

- Rewrite rules can be applied in any order, and to any node.
- Rewrite rules are applied until fixed point.
- At any time, there may be multiple nodes ready for rewrite rule application, allowing parallel application of rewrite rules.
Outline

- Introduction
- Background
- Flow-sensitive graph-rewriting formulation
- Implementation and Results
- Conclusion
Implementation details

- Intel threading building blocks (TBB) used to manage parallel workload
- A dual-worklist based approach to keep track of nodes for processing
- A hash table based concurrent data structure (concurrent_unordered_set), provided by TBB used to represent graph edges
Machine configuration

- 4-socket machine
- 2.0 GHz, 8-core processor on each socket
- 64GB memory
- Debian GNU/Linux 6.0
- Intel Threading Building Blocks – 4.0
**Benchmarks**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of graph nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larger (lines of code) programs from SPEC2006</td>
<td>13k - 414k</td>
</tr>
<tr>
<td>Ex – text processor</td>
<td>19k</td>
</tr>
<tr>
<td>Nethack – text based game</td>
<td>222k</td>
</tr>
<tr>
<td>Sendmail – email server</td>
<td>71k</td>
</tr>
<tr>
<td>SVN – revision control system</td>
<td>6439k</td>
</tr>
<tr>
<td>Vim – text editor</td>
<td>1265k</td>
</tr>
</tbody>
</table>

All of these have been used in previous pointer analysis experiments
Scaling

The diagram shows the speedup of different programs under various thread counts. The x-axis represents different programs, and the y-axis represents the speedup. The legend indicates the speedup for 1 thread, 2 threads, 4 threads, 6 threads, and 8 threads. For example, the vim program shows a significant speedup with 8 threads, reaching up to 6.96x.
Outline

- Introduction
- Background
- Flow-sensitive graph-rewriting formulation
- Implementation and Results
- Conclusion
Conclusion

- First parallelization of precise flow-sensitive pointer analysis
- Flow-sensitive pointer analysis as a graph-rewriting problem – easy to take advantage of amorphous data parallelism
- Scaling of up to 6.9x, for 8 threads shown
Acknowledgement

- PACT student travel grant
- GARP student grant - IISc
- Ben Hardekopf and Mario Mendez-Lojo
- Rupesh Nasre
- HPC lab members – SERC – IISc
Thank you

Questions?
Backup: recent results – combine worklists

The graph shows the speedup for different workloads running on varying numbers of threads. The workloads include ex, 254.gap, 176.gcc, nethack, 253.perl, 197.parser, sendmail, vim, svn, and 255.vortex.

The speedup varies across workloads and the number of threads used. For example, the speedup for 'vim' is significantly higher compared to other workloads, especially when using 8 threads.

Key observations:
- 1 thread: Generally lower speedups.
- 2 threads: Moderate speedups.
- 4 threads: Higher speedups, especially for 'vim' and 'svn'.
- 6 threads: Further increase in speedup, with 'vim' showing a 5.38x speedup.
- 8 threads: Significant speedup, with 'vim' reaching 6.08x speedup.

This indicates that combining worklists can lead to substantial performance improvements, especially with multiple threads.
Backup: – Comparison with SFS
Backup: Program size

TABLE I: Number of nodes of each type

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

TABLE I: Average number of nodes in each worklist

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

(a) .. (h) are worklists for each rewrite rule
Backup: - Future work

- Explore different orders of applying rewrite rules
- Reordering nodes for locality might give better scaling
- Extend for context-sensitivity
- Using concurrent sparse-bit-vectors for representing edges may improve performance