Single Assignment Compiler, Single Assignment Architecture
Future Gated Single Assignment Form
(Supported by NSF Grant CCF- 1116551)

Shuhan Ding   John Earnest   Soner Önder

Michigan Technological University

February 18, 2014
- FGSA
- Congruence Classes
- Efficiently Computing FGSA
- Experimental Analysis
- Executing FGSA
- Conclusion
Future Gated Single Assignment
Motivation

- A balance of work must be struck between compilers and microarchitectures
- Close collaboration can simplify both
- A shared program representation can support this
FGSA

- Single-Assignment representation
- Directly usable by optimization algorithms or microarchitectures
- Executable semantics
A Simple CFG

\[ x = \ldots \]
\[ \neg P \]
\[ x = \ldots \]
\[ Q \]
\[ \neg Q \]

\[ \ldots = x \]
A Simple CFG: SSA

\[ x_1 = \ldots \]

\[ x_2 = \ldots \]

\[ x_3 = \phi(x_1, x_2) \]

\[ x_4 = \phi(x_3, x_2) \]

... = x_4
The Predicated Function $\psi$

$$\psi_{P_1, P_2, \ldots P_n}(x_1, x_2, \ldots x_n)$$
Path Expressions

\[ P \lor (\neg P \land Q) \]

\[ \neg Q \]

\[ \neg P \]

\[ P \]

\[ \neg P \]
A Simple CFG: FGSA

\[ x_1 = \ldots \]
\[ x_2 = \ldots \]
\[ x_3 = \psi_P(x_1, x_2) \]
\[ \ldots = x_3 \]
Congruence Classes
Congruence Classes

\[ x_1 = \ldots \]

\[ x_2 = \ldots \]

\[ x_3 = \psi P(x_1, x_2) \]

\[ \ldots = x_3 \]

\[ \langle D, U \rangle \rightarrow \langle \{x_1, x_2\}, \{x_3\} \rangle \]
Gated Congruence Classes

\[
\langle D, U \rangle_g \rightarrow \langle \{ P : x_1, \neg P : x_2 \}, \{ x_3 \} \rangle
\]
Minimal Path Expressions for Gating Functions

\[ x_1 = \ldots \]
\[ x_2 = \ldots \]
\[ x_3 = \psi_P(x_1, x_2) \]
Theorem 1
Given $CC = \langle \{d_1, d_2\}, U \rangle$ and path expressions $p_1$ for $d_1$, $p_2$ for $d_2$, the gating predicate expression for $d_1$ is given by $g_1 = \neg p_2 \land p_1$ if there exists a path on which $d_2$ kills $d_1$, and $g_1 = p_1$ otherwise.
Efficiently Computing FGSA
Overview

To compute FGSA we find all congruence classes by applying a bidirectional interval analysis algorithm:

1. Scan each block to identify local CCs
2. Process the entire graph by repeatedly applying T1 and T2 transformations until the graph is reduced to a single node
   - As necessary, split irreducible cores using $T_R$
3. Place gating functions
Perform a backwards linear scan to coalesce together CCs. CCs which are neither upwards or downwards visible are complete.
Local CC computation

$$CC_{u1} = \langle \emptyset, \{x_{u1}\} \rangle$$

... = $x_{u1}$

$\downarrow$

$\uparrow$

upward visible

$\downarrow$

$\downarrow$

downward visible

$$CC_{d2} = \langle \{x_{d2}\}, \emptyset \rangle$$

$$CC_{d1} = \langle \{x_{d1}\}, \{x_{u2}, x_{u3}\} \rangle$$
Acyclic Regions and T2

- Candidates for T2 have exactly one predecessor
- The successors of the selected node become successors of the chosen node’s predecessors, and edges are chained and merged
Edge Chaining

\[ xd_1 = \ldots \]

\[ \uparrow \emptyset \quad \downarrow \emptyset \]

\[ \emptyset \]

\[ \uparrow \emptyset \quad \downarrow \langle \{xd_1\}, \emptyset \rangle \]

\[ \uparrow \langle \emptyset, \{xu_1\} \rangle \quad \downarrow \emptyset \]

\[ \emptyset \]

\[ \ldots \]
Edge Chaining

\[ \uparrow \emptyset \]
\[ \downarrow \langle \{ xd_1 \}, \emptyset \rangle \]
\[ \downarrow \langle \emptyset, \{ xu_1 \} \rangle \]
\[ \downarrow \emptyset \]
Edge Chaining

\[ \uparrow \emptyset \]
\[ \downarrow \langle \{ x_d \}, \{ x_u \} \rangle \]
Edge Merging

\[
\begin{align*}
\uparrow \emptyset & \quad \downarrow \langle \{xd_2\}, \emptyset \rangle \\
\downarrow \langle \{xd_2\}, \emptyset \rangle & \quad \uparrow \emptyset & \quad \downarrow \langle \{xd_1\}, \emptyset \rangle
\end{align*}
\]
Edge Merging

\[ \uparrow \emptyset \]
\[ \downarrow \langle \{ x_{d_1}, x_{d_2} \}, \emptyset \rangle \]
Cyclic Regions and T1

- Candidates for T1 are nodes with a self-pointing back edge.
- The back-edge is merged with the node’s definitions and as necessary we introduce a gating function guarded by a read-once predicate to select from values which flow into the loop and loop-carried values.
Read-Once Predicates

Definition 1
The read-once predicate is a special predicate which becomes false once it is read.

- Used to create gating predicates for cyclic code
Loop Carried Value

\[ x_1 = \ldots \]
\[ \rho = true \]

\[ x_2 = \psi_\rho(x_1, x_3) \]

\[ \ldots = x_2 \]
\[ x_3 = \ldots \]

\[ \neg P \]
The Exit Function

Definition 2
The exit function $\eta(d_i)$ returns the last value of an iteratively executed definition $d_i$. 
Exit Value

\[ x_1 = \ldots \]
\[ \rho = \text{true} \]

\[ x_2 = \psi_{\rho}(x_1, x_3) \]

\[ \ldots \Rightarrow x_2 \]
\[ x_3 = \ldots \]

\[ \neg P \]

\[ \ldots = \eta_{\neg P}(x_3) \]
Irreducible Graphs and $T_R$

Sometimes we will encounter an irreducible subgraph while performing T1/T2 transformations. In this case, we must convert the graph into a reducible one.

**Definition 3**
An entrance of an irreducible loop is defined as a node such that there exits a path from the Shared External Dominator (SED) to the node that contains no other nodes in the loop.
$T_R$ Example

\[ x_1 = \ldots \]
\[ y_1 = \ldots \]
\[ \ldots = x_u \]
\[ y_2 = \ldots \]
\[ x_2 = \ldots \]
\[ \ldots = y_u \]
$T_R$ Example

\[ x_1 = \ldots \]
\[ y_1 = \ldots \]

\[ x_2 = \ldots \]

\[ y_2 = \ldots \]

\[ \ldots = x_u \]
\[ y_2 = \ldots \]

\[ \ldots = y_u \]
$T_R$ Example

\[ x_1 = \ldots \]
\[ y_1 = \ldots \]
\[ \rho_1 = true \]

\[ \neg P \]
\[ P \]

\[ x_2 = \ldots \]

\[ \ldots = x_u \]
\[ y_2 = \ldots \]

\[ \ldots = y_u \]
$T_R$ Example

$x_1 = ...$
$y_1 = ...$
$\rho_1 = true$

$\neg P$

$P$

$x_2 = ...$

$W = P \lor \neg \rho_1$

$\neg P$

$\neg W$

$... = x_u$
$y_2 = ...$

$... = y_u$
Gating Function Construction

- Compute gating predicates from path predicates and reduced reachability information computed during T1/T2
- Gating functions are inserted at the LCDOM node of any uses in the CC
- Definitions which appear below the gating function are marked as a *future value*
Definition 4
When instructions $i$ and $j$ are true dependent on each other and the instruction order is reversed, the true dependency becomes a future value and is marked on the source operand with the subscript $f$. 
Complexity of FGSA Construction

Given a program, let the number of nodes, edges, user defined variables and instructions be \( N, E, V \) and \( I \) respectively.

- Local CC computation scans each instruction in each node for each variable. Thus, time complexity per variable is \( \frac{O(I)}{V} \).
- During CC propagation edge-chaining runs for each node with a single predecessor \( (O(N)) \), edge-merging runs over edges in the graph \( (O(E)) \) and runtime for \( T1 \) is bounded by \( O(N) \).
- For each CC definition \( (O(N) \) CCs containing \( O(N) \) definitions each as a loose bound), we must query the reduced reachable sets some number of times \( \sum_{CC_i} |CC_i \cdot D| \).

Loose bound for time complexity is \( \frac{O(I)}{V} + O(N + E) + O(N^2) \).

Expected overall time complexity is \( \frac{O(I)}{V} + O(N + E) \).
Experimental Analysis
Methodology

- Compute the number of gated CCs and compare with the number of $\phi$ functions constructed in SSA
- SPEC CINT2000 test suite with `-O3` optimizations
- GCC generates SSA via Cytron’s Algorithm
  - Tested with and without $\phi$-pruning
- Data collected per function in each benchmark
Summary

- Comparing CCs with pruned $\phi$s, we observe a maximum reduction of 67.5\% from a function in 186.crafty and an average reduction of 7.7\%.
- CCs consisting of two definitions are dominant, accounting for at least 62\% in all the benchmarks.
- CCs consisting of more than four definitions account for $\leq 13.38\%$ in worst-case benchmarks.
- Median predicate expression length in the whole suite is $\leq 2$.
- Predicate expressions longer than eight elements make up $< 10\%$ of the CCs.
Executing FGSA
Executing FGSA

- Traditional architectures (via inverse transformation)
- Control-flow architectures supporting future values
- Demand-driven architectures...
int a = 0;
for(int b = 1; b < 16; b++) {
    a += 1 << b;
}
... = a;
Demand-Driven Interpretation

\[
\begin{align*}
    a_1 &= 0 \\
    b_1 &= 1 \\
    \rho_1 &= true \\
    \rho_2 &= true
\end{align*}
\]

\[
\begin{align*}
    a_2 &= \psi_{\rho_1}(a_1, a_3) \\
    b_2 &= \psi_{\rho_2}(b_1, b_3) \\
    c_1 &= 1 \ll b_2 \\
    a_3 &= a_2 + c_1 \\
    b_3 &= b_2 + 1 \\
    P &= b_3 < 16
\end{align*}
\]

\[
\begin{align*}
    x_3 &= \eta_{\neg P}(a_3)
\end{align*}
\]
Demand-Driven Interpretation

\[
\begin{align*}
    a_1 &= 0 \\
    b_1 &= 1 \\
    \rho_1 &= true \\
    \rho_2 &= true
\end{align*}
\]

\[
\begin{align*}
    a_2 &= \psi_{\rho_1}(a_1, a_3) \\
    b_2 &= \psi_{\rho_2}(b_1, b_3) \\
    c_1 &= 1 \ll b_2 \\
    a_3 &= a_2 + c_1 \\
    b_3 &= b_2 + 1 \\
    P &= b_3 < 16 \\
    \neg P \\
    x_3 &= \eta_{\neg P}(a_3)
\end{align*}
\]
Demand-Driven Interpretation

\[
\begin{align*}
a_1 &= 0 \\
b_1 &= 1 \\
\rho_1 &= \text{true} \\
\rho_2 &= \text{true}
\end{align*}
\]

\[
\begin{align*}
a_2 &= \psi_{\rho_1}(a_1, a_3) \\
b_2 &= \psi_{\rho_2}(b_1, b_3) \\
c_1 &= 1 < b_2 \\
a_3 &= a_2 + c_1 \\
b_3 &= b_2 + 1 \\
P &= b_3 < 16 \\
\neg P
\end{align*}
\]

\[
x_3 = \eta_{\neg P}(a_3)
\]
Demand-Driven Interpretation

\[ a_1 = 0 \]
\[ b_1 = 1 \]
\[ \rho_1 = \text{true} \]
\[ \rho_2 = \text{true} \]

\[ a_2 = \psi_{\rho_1}(a_1, a_3) \]
\[ b_2 = \psi_{\rho_2}(b_1, b_3) \]
\[ c_1 = 1 \ll b_2 \]
\[ a_3 = a_2 + c_1 \]
\[ b_3 = b_2 + 1 \]
\[ P = b_3 < 16 \]

\[ \neg P \]

\[ x_3 = \eta_{\neg P}(a_3) \]
Demand-Driven Interpretation

\[ a_1 = 0 \]
\[ b_1 = 1 \]
\[ \rho_1 = \text{true} \]
\[ \rho_2 = \text{true} \]

\[ a_2 = \psi_{\rho_1}(a_1, a_3) \]
\[ b_2 = \psi_{\rho_2}(b_1, b_3) \]
\[ c_1 = 1 \ll b_2 \]
\[ a_3 = a_2 + c_1 \]
\[ b_3 = b_2 + 1 \]
\[ P = b_3 < 16 \]

\[ \neg P \]

\[ x_3 = \eta_{\neg P}(a_3) \]
Demand-Driven Interpretation

\[ a_1 = 0 \]
\[ b_1 = 1 \]
\[ \rho_1 = true \]
\[ \rho_2 = true \]

\[ a_2 = \psi_{\rho_1}(a_1, a_3) \]
\[ b_2 = \psi_{\rho_2}(b_1, b_3) \]
\[ c_1 = 1 \ll b_2 \]
\[ a_3 = a_2 + c_1 \]
\[ b_3 = b_2 + 1 \]

\[ P = b_3 < 16 \]

\[ \neg P \]

\[ x_3 = \eta_{\neg P}(a_3) \]
Demand-Driven Interpretation

\[ a_1 = 0 \]
\[ b_1 = 1 \]
\[ \rho_1 = true \]
\[ \rho_2 = true \]

\[ a_2 = \psi_{\rho_1}(a_1, a_3) \]
\[ b_2 = \psi_{\rho_2}(b_1, b_3) \]
\[ c_1 = 1 \ll b_2 \]
\[ a_3 = a_2 + c_1 \]
\[ b_3 = b_2 + 1 \]
\[ P = b_3 < 16 \]

\[ \neg P \]

\[ x_3 = \eta_{\neg P}(a_3) \]
Conclusion
Overview of FGSA

- A static-single-assignment IR with executable semantics
- Densely represents use-def relationships with gated congruence classes
- Can be efficiently computed using a series of T1/T2 transformations
- Construction handles irreducible graphs without exponential code expansion
- Convenient both for optimization and direct execution by hardware
Future Work

▶ Formal analysis, adaptation and implementation of well-known optimizations using this representation
▶ Development of micro-architectures that take advantage of FGSA
▶ Exploration of alternative forms of execution under this paradigm
Questions?
## CCs vs $\phi$-functions over REAL.

<table>
<thead>
<tr>
<th></th>
<th>vars</th>
<th>phis</th>
<th>ccs</th>
<th>% Reduction</th>
<th>Max</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>3715</td>
<td>624</td>
<td>514</td>
<td>42.86</td>
<td>100</td>
<td>8.85</td>
</tr>
<tr>
<td></td>
<td>3715</td>
<td>4401</td>
<td>514</td>
<td>8.85</td>
<td>100</td>
<td>69.76</td>
</tr>
<tr>
<td>175.vpr</td>
<td>16648</td>
<td>1309</td>
<td>1092</td>
<td>61.11</td>
<td>100</td>
<td>7.39</td>
</tr>
<tr>
<td></td>
<td>16648</td>
<td>15773</td>
<td>1092</td>
<td>7.39</td>
<td>100</td>
<td>81.26</td>
</tr>
<tr>
<td>176.gcc</td>
<td>125212</td>
<td>15810</td>
<td>14206</td>
<td>66.67</td>
<td>100</td>
<td>4.8</td>
</tr>
<tr>
<td></td>
<td>125212</td>
<td>152079</td>
<td>14206</td>
<td>4.8</td>
<td>100</td>
<td>72.98</td>
</tr>
<tr>
<td>181.mcf</td>
<td>899</td>
<td>161</td>
<td>117</td>
<td>60</td>
<td>100</td>
<td>12.17</td>
</tr>
<tr>
<td></td>
<td>899</td>
<td>666</td>
<td>117</td>
<td>12.17</td>
<td>100</td>
<td>63.92</td>
</tr>
<tr>
<td>186.crafy</td>
<td>14341</td>
<td>1485</td>
<td>1226</td>
<td>67.47</td>
<td>100</td>
<td>10.55</td>
</tr>
<tr>
<td></td>
<td>14341</td>
<td>15972</td>
<td>1226</td>
<td>10.55</td>
<td>100</td>
<td>79.77</td>
</tr>
<tr>
<td>197.parser</td>
<td>18720</td>
<td>2887</td>
<td>2653</td>
<td>50</td>
<td>100</td>
<td>6.08</td>
</tr>
<tr>
<td></td>
<td>18720</td>
<td>25656</td>
<td>2653</td>
<td>6.08</td>
<td>100</td>
<td>60.59</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>20330</td>
<td>1789</td>
<td>1656</td>
<td>50</td>
<td>100</td>
<td>2.83</td>
</tr>
<tr>
<td></td>
<td>20330</td>
<td>16578</td>
<td>1656</td>
<td>2.83</td>
<td>100</td>
<td>77.97</td>
</tr>
<tr>
<td>255.vortex</td>
<td>36585</td>
<td>1913</td>
<td>1747</td>
<td>50</td>
<td>100</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td>36585</td>
<td>16151</td>
<td>1747</td>
<td>1.9</td>
<td>100</td>
<td>77.97</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>3598</td>
<td>342</td>
<td>286</td>
<td>50</td>
<td>100</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>3598</td>
<td>2421</td>
<td>286</td>
<td>12</td>
<td>100</td>
<td>71.21</td>
</tr>
<tr>
<td>300.twolf</td>
<td>21676</td>
<td>2653</td>
<td>1991</td>
<td>64.91</td>
<td>100</td>
<td>10.22</td>
</tr>
<tr>
<td></td>
<td>21676</td>
<td>34162</td>
<td>1991</td>
<td>10.22</td>
<td>100</td>
<td>81.18</td>
</tr>
</tbody>
</table>
## Number of definitions in CCs

<table>
<thead>
<tr>
<th></th>
<th>ccs</th>
<th>2defs%</th>
<th>3defs%</th>
<th>4defs%</th>
<th>4⁹⁺ defs%</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>514</td>
<td>78.79</td>
<td>11.87</td>
<td>4.28</td>
<td>5.06</td>
</tr>
<tr>
<td>175.vpr</td>
<td>1092</td>
<td>81.32</td>
<td>7.97</td>
<td>7.97</td>
<td>2.75</td>
</tr>
<tr>
<td>176.gcc</td>
<td>14206</td>
<td>76.95</td>
<td>10.14</td>
<td>4.65</td>
<td>8.26</td>
</tr>
<tr>
<td>181.mcf</td>
<td>117</td>
<td>68.38</td>
<td>27.35</td>
<td>1.71</td>
<td>2.56</td>
</tr>
<tr>
<td>186.crafy</td>
<td>1226</td>
<td>62.07</td>
<td>14.52</td>
<td>10.03</td>
<td>13.38</td>
</tr>
<tr>
<td>197.parser</td>
<td>2653</td>
<td>79.80</td>
<td>16.66</td>
<td>2.41</td>
<td>1.13</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>1656</td>
<td>79.71</td>
<td>8.33</td>
<td>7.13</td>
<td>4.83</td>
</tr>
<tr>
<td>255.vortex</td>
<td>1747</td>
<td>87.58</td>
<td>5.15</td>
<td>3.15</td>
<td>4.12</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>286</td>
<td>80.42</td>
<td>12.24</td>
<td>5.59</td>
<td>1.75</td>
</tr>
<tr>
<td>300.twolf</td>
<td>1991</td>
<td>76.49</td>
<td>10.90</td>
<td>9.94</td>
<td>2.66</td>
</tr>
</tbody>
</table>
## Length of CC Predicate Expressions

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>median</th>
<th>average</th>
<th>% &gt; 4</th>
<th>% &gt; 8</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>1</td>
<td>1.98</td>
<td>12.5</td>
<td>0.4</td>
<td>13</td>
</tr>
<tr>
<td>175.vpr</td>
<td>1</td>
<td>2.06</td>
<td>7.1</td>
<td>1.4</td>
<td>31</td>
</tr>
<tr>
<td>176.gcc</td>
<td>2</td>
<td>3.79</td>
<td>20.3</td>
<td>9.2</td>
<td>132</td>
</tr>
<tr>
<td>181.mcf</td>
<td>1</td>
<td>1.97</td>
<td>6.0</td>
<td>1.7</td>
<td>9</td>
</tr>
<tr>
<td>186.crafty</td>
<td>2</td>
<td>3.15</td>
<td>16.7</td>
<td>6.1</td>
<td>95</td>
</tr>
<tr>
<td>197.parser</td>
<td>2</td>
<td>2.27</td>
<td>12.9</td>
<td>1.3</td>
<td>83</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>1</td>
<td>2.50</td>
<td>12.6</td>
<td>5.3</td>
<td>31</td>
</tr>
<tr>
<td>255.vortex</td>
<td>1</td>
<td>2.01</td>
<td>11.2</td>
<td>3.4</td>
<td>17</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>1</td>
<td>1.71</td>
<td>4.6</td>
<td>1.4</td>
<td>15</td>
</tr>
<tr>
<td>300.twolf</td>
<td>1</td>
<td>2.23</td>
<td>8.1</td>
<td>3.5</td>
<td>32</td>
</tr>
</tbody>
</table>