Symbolic Pointer Analysis

Jianwen Zhu

Electrical and Computer Engineering
University of Toronto

November 11th, 2002

jzhu@eecg.toronto.edu
http://www.eecg.toronto.edu/~jzhu
Outline

- Background
- New Formalism
- New Efficiency
- Engineering and Results
- Conclusion
Synthesis from C-like Languages

- Traditional high-level synthesis
  - Regretfully not embraced by design community
  - Complexity
  - Quality
- Commercial “C Synthesis” tools today
  - RTL in C flavor
- SystemC/SpecC
  - Primarily used for system-level modeling
The Pervasive Pointers

- All of them are pointers!
  - C: addresses of global, local and heap block
  - C++: plus addresses of class objects
  - Java: reference to class objects
  - Function pointers
  - Virtual methods, Interfaces

- Complex data structures

- Candidates for hardware synthesis
  - Multimedia
  - Networking
  - 3-D Graphics
The Evasive Pointers

- Runtime values **unknown at compile time**
- Pointers maybe **aliases** to each other
- Evil for optimization
  - Dependency test for parallelization
  - Memory bank partition
  - Memory sharing

```plaintext
*p = a + b;
    ...
    ...
    c = *q + d;
```
State of the Art

- **Pointer Analysis Problem**
  - Determine program state at every program point
  - Cares only about pointer values
  - Undecidable problem

- **Hind 2001**: “75 papers, 9 PhD thesis”
- **Flow and Context-insensitive**
  - Steensgaard’96
  - Andersen’94
- **Flow and Context-sensitive**
  - Wilson and Lam’95
  - Liang and Harrold’01
- **Applications in CAD**
  - Semeria and De Micheli’01
  - Panda et. al.’01
  - Zhu’01

- **Context and Flow sensitivity**
Sources of Inefficiency

- Aggressive optimization needs accurate analysis
  - Context-sensitive + Flow-sensitive + more!

- Best available algorithms have exponential complexity
  - Cost for summarize procedure call
    - Wilson’s partial transfer function
    - Liang’s parameterized summary
  - Cost for propagating call effect at call site
  - Redundant program state representation
Outline

- Background
- New Formalism
- New Efficiency
- Engineering and Results
- Conclusion
Program Modeling

- Representing instructions
  - Only two types interesting
    - store dest, src
    - call dest, [src₁ ... srcₙ]
  - dest, srcᵢ = ⟨block, level⟩
  - Example:

```
char *g, a;
void main() {
    call alloc, [p];
    call getg, [q, g];
    store g, &a;
}
getg([r₁, g₁]) {
    store t, &g;
    others
    call alloc [*t];
    store *r, *t;
}
alloc([f₁]) {
    store *f, &m;
}
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>store t &amp;g;</td>
<td>store ⟨t, 0⟩, ⟨g, −1⟩</td>
</tr>
<tr>
<td>store *r, *t;</td>
<td>store ⟨r, 1⟩, ⟨t, 1⟩</td>
</tr>
<tr>
<td>call getg, [q, g];</td>
<td>call ⟨getg, 0⟩, [⟨q, 0⟩, ⟨g, 0⟩]</td>
</tr>
<tr>
<td>store *f, &amp;m;</td>
<td>store ⟨f, 1⟩, ⟨m, −1⟩</td>
</tr>
</tbody>
</table>
Point-to Graph

- Captures program state \( \langle V, E \rangle \)
- Vertices \( V \)
  - Global block
  - Local block
  - Heap block
  - Procedure block
  - Initial block (\( \lambda \) state@callsite)
- Edges \( E \)
  - \( \langle u, v \rangle \in E \Rightarrow \) the content of
    block \( u \) may be the address of
    location \( v \)

Basic algorithms
- state query
- evaluating store
- evaluating call
A Symbolic Alternative

- Key observation: edge set of a graph captures a Relation
- Big idea: represent relation using Boolean function
- Define Boolean space: domain and range space
- Encoding memory locations
  - initials $\mapsto$ Boolean variable
  - Others $\mapsto$ minterms

<table>
<thead>
<tr>
<th>Block</th>
<th>domain</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$\bar{x}_1\bar{x}_2\bar{x}_3$</td>
<td>$\bar{x}_1\bar{x}_2\bar{x}_3$</td>
</tr>
<tr>
<td>g</td>
<td>$\bar{x}_1\bar{x}_2\bar{x}_3$</td>
<td>$\bar{x}_1\bar{x}_2\bar{x}_3$</td>
</tr>
<tr>
<td>p</td>
<td>$\bar{x}_1\bar{x}_2\bar{x}_3$</td>
<td>$\bar{x}_1\bar{x}_2\bar{x}_3$</td>
</tr>
<tr>
<td>q</td>
<td>$\bar{x}_1\bar{x}_2\bar{x}_3$</td>
<td>$\bar{x}_1\bar{x}_2\bar{x}_3$</td>
</tr>
<tr>
<td>t</td>
<td>$x_1^<em>\bar{x}_2\bar{x}_3^</em>$</td>
<td>$x_1\bar{x}_2\bar{x}_3$</td>
</tr>
<tr>
<td>r</td>
<td>$x_1^<em>\bar{x}_2\bar{x}_3^</em>$</td>
<td>$x_1\bar{x}_2\bar{x}_3$</td>
</tr>
<tr>
<td>f</td>
<td>$x_1^<em>\bar{x}_2\bar{x}_3^</em>$</td>
<td>$x_1\bar{x}_2\bar{x}_3$</td>
</tr>
<tr>
<td>m</td>
<td>$x_1^<em>\bar{x}_2\bar{x}_3^</em>$</td>
<td>$x_1\bar{x}_2\bar{x}_3$</td>
</tr>
<tr>
<td>fl</td>
<td>$y_1^*$</td>
<td>$y_1$</td>
</tr>
<tr>
<td>gl</td>
<td>$y_2^*$</td>
<td>$y_2$</td>
</tr>
<tr>
<td>rl</td>
<td>$y_3^*$</td>
<td>$y_3$</td>
</tr>
</tbody>
</table>
Symbolic Replacement of Point-to Graph

\[ x_1^* x_2^* \bar{x}_3^* y_1 \]
\[ + \quad y_1^* x_1 x_2 x_3 \]

(a) alloc

(b) getg

(c) main

12
Symbolic State Query

Graph query: **r

```
 r
 / \
a  b
 \ /
c d e
```

Symbolic query:

\[
S = r^*a + r^*b + a^*c \\
+ a^*d + b^*d + b^*e \\
L1 = a + b \\
L2 = S \cdot \text{mirror}(L1) \\
= c + d + e
\]

Algorithm 1  State query.

```
spaQueryState = func(sp, state, from, level) : SpaDD {
    if( level == 0 ) return from ;
    return bddAndAbstract(sp, state, bddMirror(sp, spaQueryState(sp, state, from, level-1))) ;
}
```
Symbolic Evaluation of Stores

Examples

- \( S_0 = r^* r_1 + g^* m + g^* g_1 \)
- \( t = \& g : \Delta = t^* g \)
- \( *r = *t : \Delta = r_1^* m + r_1^* g_1 \)

Algorithm 2  State update.

```plaintext
spaUpdateState = func( 
    spa, state, dst, src 
) : SpaDD 
{
    return bddOr( state, bddAnd( 
        bddMirror( spa, 
            spaQueryState( spa, state, 
                dst.blk.range, dst.level ) ), 
            spaQueryState( spa, state, 
                src.blk.range, src.level+1 ) ) ) ;
}
```
Symbolic Evaluation of Calls

Examples

- Callee alloc with tf = $f1*m$
- At callsite of getg:
  $$f1*m|_{f1/p} = p^*m$$
- At callsite of main:
  $$f1*m|_{f1/g} = g^*m$$

Algorithm 3 Evaluate call.

```plaintext
spApply = func(  
  spa, state, srcs, proc, tf  
) : SpaDD  
{  
  var proj : SpaDD \rightarrow SpaDD;  
  build projection;  
  return bddOr(  
    state, bddCompose( spa, tf, proj )  
  ) ;  
}  
```

alloc  getg  main
Outline

- Background
- New Formalism
- New Efficiency
- Engineering and Results
- Conclusion
Binary Decision Diagram (BDD)

Summary
- Established a Boolean formalism for the manipulation of Point-to-relation
- Sounds elegant, how efficient?

Efficiency derive from Bryant’s ROBDD
- Rooted directed graph based on Shannon expansion
- Small size for large amount of functions
- Canonical
- State query = Image computation?
BDD Representation of Point-To Graph
BDD Seems to be Larger, Why Bother?

- Scale matters: the more #edges we have, the simpler the BDD!
- Symbolic states can be shared among program points!
A Comment on Complexity

Let $G_1$ and $G_2$ be two BDDs

<table>
<thead>
<tr>
<th>operation</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>bddAnd</td>
<td>$O(</td>
</tr>
<tr>
<td>bddOr</td>
<td>$O(</td>
</tr>
<tr>
<td>bddCompose</td>
<td>$O(</td>
</tr>
<tr>
<td>practically</td>
<td>$O(</td>
</tr>
<tr>
<td>bddMirror</td>
<td>$O(</td>
</tr>
</tbody>
</table>

Compound efficiency
- Intra-procedural space sharing
- Inter-procedural space sharing
- Implicit batch processing
- Dynamic programming

Scalability
- gimp: 7M LOC, 131552 variables
- 18 Boolean variables
An Example of Batch Processing

- **Explicit evaluation**

- **Implicit evaluation**

\[ y_1 = y_2 \]

\[ *y_1 = y_2 \]
Outline

- Background
- New Formalism
- New Efficiency
- Engineering and Results
- Conclusion
Engineering

- Relaxing simplifying assumptions
  - Records and classes
  - Arrays
  - Alias test of initial blocks
  - Strong and weak update
  - Recursive functions

- Engineering a fast algorithm
  - Partitioning of Boolean space
  - Fast `bddMirror` operation
    Consistent variable ordering between domain and range space
  - Fast `bddCompose` operation
    Use single variable for initials and predicates
  - Caching of BDD operation
A Context-Sensitive Flow-Insensitive Validation

- Algorithm
  - Bottom-up evaluation of procedures
  - Use \textit{spaUpdateState} for store instruction
  - Use \textit{spaApply} for call instruction
  - Needs to iterate until fixed-point is reached

- Omissions
  - Field independent
  - No location set
  - Hardwired libraries
  - Ignore setjmp/longjmp
Experimental Results

- Standard benchmark from McGill and Landi
- While LOC is not large, invocation graph can be very large
- All finished in seconds

<table>
<thead>
<tr>
<th>Name</th>
<th>Source</th>
<th>LOC</th>
<th>#procs</th>
<th>density</th>
<th>Run Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01.qsort</td>
<td>McGill</td>
<td>325</td>
<td>8</td>
<td>24.1%</td>
<td>0.10</td>
</tr>
<tr>
<td>06.matx</td>
<td>McGill</td>
<td>350</td>
<td>7</td>
<td>13.5%</td>
<td>0.13</td>
</tr>
<tr>
<td>15.trie</td>
<td>McGill</td>
<td>358</td>
<td>13</td>
<td>23.4%</td>
<td>0.21</td>
</tr>
<tr>
<td>04.bisect</td>
<td>McGill</td>
<td>463</td>
<td>9</td>
<td>9.7%</td>
<td>0.10</td>
</tr>
<tr>
<td>17.bintr</td>
<td>McGill</td>
<td>496</td>
<td>17</td>
<td>8.8%</td>
<td>0.13</td>
</tr>
<tr>
<td>05.eks</td>
<td>McGill</td>
<td>1202</td>
<td>30</td>
<td>4.0%</td>
<td>0.20</td>
</tr>
<tr>
<td>08.main</td>
<td>McGill</td>
<td>1206</td>
<td>41</td>
<td>20.9%</td>
<td>1.33</td>
</tr>
<tr>
<td>09.vor</td>
<td>McGill</td>
<td>1406</td>
<td>52</td>
<td>28.6%</td>
<td>5.54</td>
</tr>
<tr>
<td>allroots</td>
<td>Landi</td>
<td>227</td>
<td>7</td>
<td>1.3%</td>
<td>3.02</td>
</tr>
<tr>
<td>football</td>
<td>Landi</td>
<td>2354</td>
<td>58</td>
<td>1.8%</td>
<td>2.38</td>
</tr>
<tr>
<td>compiler</td>
<td>Landi</td>
<td>2360</td>
<td>40</td>
<td>5.1%</td>
<td>5.3</td>
</tr>
<tr>
<td>assembler</td>
<td>Landi</td>
<td>3446</td>
<td>52</td>
<td>16.6%</td>
<td>10.63</td>
</tr>
<tr>
<td>simulator</td>
<td>Landi</td>
<td>4639</td>
<td>111</td>
<td>6.3%</td>
<td>4.03</td>
</tr>
</tbody>
</table>
Conclusion

- Pointer analysis is a crucial problem for C-based synthesis

- Contribution
  - Boolean algebra as new Formalism for pointer analysis
  - Efficient algorithms for fundamental symbolic pointer evaluation
  - Validation of new concept

- Future work
  - A scientific, comparative study of algorithm efficiency and scalability
  - Towards better precision
  - Towards faster speed
  - Towards application