Generic Slicing on Machine Code

Marc Schlickling
schlickling@cs.uni-sb.de
http://rw4.cs.uni-sb.de/~schlickling
Embedded Systems (ES) are widely used

- Many systems of daily use: handy, handheld, pda, …
- Safety critical systems: airbag control, flight control system, …

- Rapidly growing complexity of software in ES
- But what happens, if such a system crashes?
Embedded Systems (1)

Hello?

- Embedded Systems (ES) are widely used
  - Many systems of daily use: handy, handheld, pda, ...
  - Safety critical systems: airbag control, flight control system, ...
- Rapidly growing complexity of software in ES
- But what happens, if such a system crashes?
Embedded Systems (ES) are widely used
- Many systems of daily use: handy, handheld, pda, ...
- Safety critical systems: airbag control, flight control system, ...

Rapidly growing complexity of software in ES

But what happens, if such a system crashes?
Embedded Systems (ES) are widely used
- Many systems of daily use: handy, handheld, pda, ...
- Safety critical systems: airbag control, flight control system, ...

Rapidly growing complexity of software in ES

But what happens, if such a system crashes?
Embedded Systems (2)

- ES subject to strict safety constraints
  - Strict timing constraints
    - Execution has to be always fast enough
  - Strict resource constraints
    - No stack overflow during run time
  - Strict spacial partitioning
    - Different tasks run within their own memory spaces

➢ Urgent need to guarantee these characteristics
Guaranteeing safety constraints (1)

- Not only possible at source code level
  - Caused by compilers
    - Optimizations
    - Transformations

- Complex due to the use of integrated modular platforms
  - E.g. IMA (integrated modular avionics)
    - Many components sharing the same resource
    - Used in the new Airbus A380
Guaranteeing safety constraints (2)

- Programs checking such constraints
  - aiT
    - Worst case execution time (WCET) prediction
  - StackAnalyzer
    - Worst case stack usage determination
- Analyses are done on real executable
  - Due to the complexity of analyzed software, some help of the developers required
Appliance of slicing

- Program understanding
  - Hard at machine code level

- Traceability
  - Is a branch in the control flow taken or not?
  - Ensure, that a specific variable in the program is not changed by other parts
    - E.g. co-driver airbag module
In this talk

- Interprocedural backward slicing on machine code
- Based on data flow
  - Data dependencies
    - Only flow dependencies required
  - Control dependencies
- Setup
  - Textual representation of a control flow graph + some additional attributes
  - Results of a value analysis
    - Bounds of memory accesses
Data flow analysis/Abstract interpretation

- A technique for collecting run-time information about data in programs without actually executing them
- Based on a control flow graph

Abstract interpretation: computation on abstract values
Control Flow Graph & CRL

start with main;
routine main: name="main"
{ entry b0: id="b0"
  { edges to b1/f , b2/t;
   contains {
     0x00  "ldr r0,sp,0": src1="al",dst2=""
     src3="sp", src4="Mem", guard="always";
     0x04  "mov r1,1": src1="al",dst2=""
     src3="1", guard="always";
     0x08  "cmp r1,r0": src1="al",src2=""
     src3="r0",dst4="CPSR",
     guard="always";
     0x0C  "bgt b2": src1="gt",src2="0x14",
     dst3="PC",guard="conditional";
   }
  }
}
Program Analyzer Generator (PAG)

- Analogous to flex and bison
- For generating efficient program analyzers
- CRL frontend available to translate CRL files into data structures describing the CFG
- Interface to read results from other analyses and using them in the current one
Reconstruct data dependencies

- Simple reaching definition analysis
- Hardware-specific problems
  - Handling of composed registers / register aliasing
    - Architectures provide registers or register parts under different names
  - Guarded execution
    - Execution of an instruction depends on a condition
      - Instruction is definitively executed: must-update
      - Instruction may be executed: may-update
Reconstruct control dependencies

- Computed by two consecutive analyses
  - Control point analysis (cp)
    - Forward directed
    - Collecting all nodes with more than one successor
  - Post dominator analysis (pdom)
    - Backward directed

- Combination of the results
  - For a node n, all nodes, n is dependent on, get computed
    by: \( \{ m | m \in \text{dfi}_{cp}(n): n \notin \text{dfi}_{pdom}(m) \} \)
Model memory accesses

- Consider the following example with slicing criterion \( C=(5,R1) \)

\[
\begin{align*}
&M[0] = 1; \\
&M[1] = 2; \\
&R1 = M[0]; \\
&R2 = M[1]; \\
&M[2] = R2;
\end{align*}
\]

\[
\begin{align*}
&M[0] = 1; \\
&M[1] = 2; \\
&R1 = M[0]; \\
&R2 = R1 + R2;
\end{align*}
\]

- Memory consists of many cells
- Partitioning into fixed cells is not possible

- Dynamic model of memory necessary
Dynamic memory tree

- Binary tree
- Each inner node consists of an interval and two successors
- Each leaf consists of an interval and a set of program points
- Interval corresponds to bounds of different memory cells
- Start a forward analysis with \( ([0x0, \infty], \{\}) \)
Transfer function

- For instruction $n$ writing to the memory cells $[a,b]$

\[
[a,d] 
\quad \text{must-update} \quad \text{old}
\]

\[
[a,d] 
\quad \text{may-update} \quad \text{old} 
\]

\[
[a,b] \quad \{n\} \quad \text{old}
\]

\[
[b+1,d] \quad \text{old} 
\]

\[
[a,b] \quad \text{old} \cup \{n\} 
\]

\[
[b+1,d] \quad \text{old} 
\]
Combine function

- Take the leafs of one tree and add it to the second one by doing a may-update

```
(a, d)
   /   \
(a, b)   (c, d)
    \     /     \
      \   combine
        /   \     \
(a, f)     (a, d)
              /   \
            (e, f)
```

```
(a, b)  (c, d)
    /     \
    \     \
      \   y
```

```
(a, d)  (a, f)
   /   \
(a, b)  (c, d)
    \     \
      \   y
```

{m}  {n}  {x}  {y}

{m, x}  {n, x}
Slicing the program (1)

- Already seen:
  - Reconstruction of data dependencies
  - Reconstruction of control dependencies
  - Dynamic model of the memory system

- Follows next:
  - Using these information to compute approximations to statement-minimal slices
Slicing the program (2)

- Precomputation of data, control and memory dependencies

- while no abort
  - wait for slicing criterion \((n,V)\)
  - \(\text{workset} = \text{Data}(n,V) \cup \text{Mem}(n) \cup \text{Data}(	ext{Ctrl}(n)) \cup \text{Mem}(	ext{Ctrl}(n))\)
  - \(\text{visited} = \{(n,V)\} \cup \text{Ctrl}(n)\)
  - while \(\text{workset} \neq \emptyset\)
    - \((x,W) \leftarrow \text{workset}\)
    - \(\text{visited} = \text{visited} \cup \{(x,W)\} \cup \text{Ctrl}(x)\)
    - \(\text{workset} = \text{workset} \setminus \{(x,W)\} \cup \{\text{Data}(x,W) \cup \text{Mem}(x,W) \cup \text{Data}(	ext{Ctrl}(x)) \cup \text{Mem}(	ext{Ctrl}(x)) \setminus \text{visited}\}\)
  - \(\text{slice} = \{ n \mid (n,_) \in \text{visited} \}\)
Experimental results (1)

- Usability depends on
  - Time for analyzing the executable
    - Depends on the analyzed executable
      - #instructions, #calls, #memory accesses
  - Time for computing a slice
    - Depends on the chosen criterion
  - Preciseness of calculated slices
    - Also depends on the chosen slicing criterion
    - Hard to verify
Experimental results (2)

Characteristics of some test files

<table>
<thead>
<tr>
<th></th>
<th>minmax</th>
<th>fac</th>
<th>prime</th>
<th>dry2_1</th>
<th>st30</th>
</tr>
</thead>
<tbody>
<tr>
<td>routines</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>17</td>
<td>163</td>
</tr>
<tr>
<td>calls</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>32</td>
<td>309</td>
</tr>
<tr>
<td>instructions</td>
<td>114</td>
<td>24</td>
<td>119</td>
<td>773</td>
<td>3820</td>
</tr>
<tr>
<td>loops</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>50</td>
</tr>
<tr>
<td>loads</td>
<td>4</td>
<td>2</td>
<td>20</td>
<td>296</td>
<td>984</td>
</tr>
<tr>
<td>stores</td>
<td>4</td>
<td>2</td>
<td>10</td>
<td>140</td>
<td>877</td>
</tr>
</tbody>
</table>
Experimental results (3)

- Measured precomputation times (in seconds)

<table>
<thead>
<tr>
<th></th>
<th>minmax</th>
<th>fac</th>
<th>prime</th>
<th>dry2_1</th>
<th>st30</th>
</tr>
</thead>
<tbody>
<tr>
<td>cfg</td>
<td>0.03</td>
<td>0.02</td>
<td>0.03</td>
<td>0.17</td>
<td>0.5</td>
</tr>
<tr>
<td>data</td>
<td>0.02</td>
<td>0.03</td>
<td>0.04</td>
<td>0.25</td>
<td>173.2</td>
</tr>
<tr>
<td>cp</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.1</td>
<td>0.4</td>
</tr>
<tr>
<td>pdom</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.07</td>
<td>0.4</td>
</tr>
<tr>
<td>mem</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.32</td>
<td>1.2</td>
</tr>
<tr>
<td>(\sum)</td>
<td>0.07</td>
<td>0.1</td>
<td>0.13</td>
<td>0.82</td>
<td>177.5</td>
</tr>
</tbody>
</table>
Experimental results (4)

Variation from statement-minimal slices

<table>
<thead>
<tr>
<th></th>
<th>minmax criterion 1</th>
<th>minmax criterion 2</th>
<th>minmax criterion 3</th>
<th>fac criterion 1</th>
<th>fac criterion 2</th>
<th>fac criterion 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>minimal</td>
<td>8</td>
<td>6</td>
<td>19</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>computed</td>
<td>10</td>
<td>6</td>
<td>20</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>
Conclusions

- Generic solution for the slicing of executable programs
  - Only a few architecture specific parameters required
    - Registers
    - Register aliases
- Dynamic model of the memory
- Currently support for ARM, PPC5xx and PPC755 Architectures
- Successfully integrated in the aiT framework
  - Interactive visualization with aiSee3
Generic Slicing on Machine Code
Generic Slicing on Machine Code
Generic Slicing on Machine Code