HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems

John Regehr       Alastair Reid
School of Computing, University of Utah
• Hoist makes it significantly easier to do static analysis of embedded software
  – E.g. TinyOS
• Automatically derives transfer functions for analyzing object code
  – This is new
  – Hoisted transfer functions are maximally precise
  – Brute-force approach that works well for small architectures
### Before Hoist:

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>M</th>
<th>T</th>
<th>W</th>
<th>T</th>
<th>F</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
</tr>
<tr>
<td></td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
</tr>
<tr>
<td></td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
</tr>
<tr>
<td></td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
</tr>
<tr>
<td></td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
</tr>
<tr>
<td></td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
</tr>
<tr>
<td></td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
</tr>
<tr>
<td></td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
</tr>
<tr>
<td></td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
</tr>
<tr>
<td></td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
</tr>
<tr>
<td></td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
</tr>
</tbody>
</table>

### After:

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>M</th>
<th>T</th>
<th>W</th>
<th>T</th>
<th>F</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
</tbody>
</table>

3
Use Static Analysis to Eliminate...

• Concurrency errors
• Deadline misses
• Stack overflow
• Language-level errors
  – Array bound violations
  – Null pointer dereferences
  – Numerical problems
• Everything else Jim Larus talked about!
r0 = ??11?001
r1 = ?00110??

and r0, r1

r0 = ??11?001
r1 = ?00110??
Abstract Transfer Functions

\[
\begin{align*}
\text{Bitwise} & \quad \text{Interval} \\
\text{??11?001} & \quad [3 \ldots 6] \\
\& \quad ?00110?? & \quad + \quad [36 \ldots 60] \\
= \quad ?001?00? & \quad = \quad [39 \ldots 66] \\
\text{??11?001} & \quad [3 \ldots 6] \\
+ \quad ?00110?? & \quad & \quad + \quad [36 \ldots 60] \\
= \quad ... & \quad = \quad ...
\end{align*}
\]
Transfer Functions can be Hard

- Domain / operation mismatch
- Condition codes – input and output
- Hard to know where precision matters
- Lots of transfer functions: \# domains * \# instructions * \# architectures

- **Result**: Wasted time, bugs, imprecision
Hoist Contributions

• Derive transfer functions with
  – Near-zero developer effort
  – Maximal precision
  – Sufficient performance
  – High confidence in correctness
• Extract complete result table for instruction
  – Dest register + cond codes

• Ideas:
  – No high-level model of instruction
  – Brute force
• Generate complete abstract transfer function

• Ideas:
  – Recursive decomposition of abstract domain
  – Speedup through dynamic programming

Extract results

Hoist into abstract domain

Encode as BDD

Generate code

Test
- Binary decision diagrams can compactly represent many functions
- Encode transfer function as vector of BDDs
- Ideas:
  - Variable ordering matters
  - Operation ordering matters
- Turn BDD into code implementing the transfer function
• Probabilistically or exhaustively verify
  – Correctness
  – Maximal precision
• Original result table is ground-truth
Hoisting Atmel AVR Architecture

• Up to 45 minutes to Hoist a bitwise operation
• Up to 34 hours to Hoist an interval operation
• Dominated by BDD library
• Parallelizes trivially across operations
Performance at Analysis Time

• Analyze programs that ship with TinyOS for worst-case stack depth
  – Analysis time increases from 8.3s to 8.9s for the program that takes longest to analyze
Precision in Bitwise Domain

• Fed random bitwise values to Hoisted and hand-written operations
  – 59% more known bits in result register
  – 130% more known bits in condition codes

• Analyzed 26 TinyOS programs
  – 8% more known bits in result register
  – 40% more known bits in condition codes

• Hand-written operations had been tuned for months
Twist #1: Pseudo-Unary Ops

• Problem:
  – xor 0?10??11, 0?10??11 == 0?00??00
  However:
  – xor r3, r3 == 00000000
  – Oops! Maximal precision doesn’t help here

• Solution: Create a pseudo-unary version of each binary operation
  – E.g. xor₁, sub₁, and₁, or₁
  – Without these, analysis fails miserably
  – Not fun to implement these by hand
Twist #2: Interacting Domains

• If a register contains 
  \[160..210\] and \texttt{???11011}

• We can show that it actually contains 
  \[187..187\] and 10111011

• In general: Use Hoist to create a reduced product of the interval and bitwise domains
  – [Cousot & Cousot 79] says this is impossible
  – For finite domains we can brute-force it
  – Maximally precise
Elephant in the Closet

• Hoist does not scale to machines bigger than 8 bits
  – 8 bit is important: Many architectures, huge sales volume, used in critical systems

• Current work
  – Replace BDDs with high-level symbolic representation
  – Gain scalability but lose many other advantages of Hoist
Conclusions

• Reduce barriers to entry for analyzing embedded software

• Hoist generates transfer functions for interval and bitwise domains
  – Near-zero specification effort, maximal precision

• We use Hoisted operations in day-to-day development / use of our static analyzer
  – Biggest benefit is never wondering if the transfer functions are the problem