Pluggable Abstract Domains for Analyzing Embedded Software

Nathan Cooprider and John Regehr
University of Utah
School of Computing
Goal

- Aggressive static analysis of embedded C programs
  - Use compiler technology
  - Perform optimizations
    - Code size reduction
    - Reducing duty cycle
    - RAM compression
Solution

- Conditional X propagation (cXprop)
  - X is a pluggable abstract value domain

- Handles all of C
- Interprocedural dataflow analysis
- Analyzes concurrent programs
- Multiple code transformations

- Example benefit:
  - 9.2% average code size reduction in TinyOS applications
Contributions

- Clean domain interface for pluggable domains
- Quantitative comparison of abstract domains
- New method for analyzing interrupt-driven concurrency
- Pointer analysis intermixed with dataflow analysis
Results of abstract interpretation

```plaintext
a= x= y= z=

a=&x

opaque

z=5 *a=77

z=2 *a=13

a=&y

x

T

*a=42

F

*z=5 *a=77

z=1

z=0

da=

x=

y=

z=
```

```plaintext
a= x= y= z=

a=&y

opaque

z=2 *a=13

a=&y

x

T

*a=42

F

*z=5 *a=77

z=1

z=0
```

```plaintext
a= x= y= z=

a=&y

opaque

z=2 *a=13

a=&y

x

T

*a=42

F

*z=5 *a=77

z=1

z=0
```
Parity domain

\[ a = \{ \&x \} \]
\[ x = \perp \]
\[ y = \perp \]
\[ z = \perp \]

\[ a = \&x \]
\[ z = 5 \]
\[ *a = 77 \]
\[ z = 2 \]
\[ *a = 13 \]
\[ a = \&y \]

\[ *a = 42 \]
\[ x = \text{odd} \]
\[ y = \text{even} \]
\[ z = \perp \]

\[ a = \{ \&y \} \]
\[ x = \text{odd} \]
\[ y = \text{even} \]
\[ z = \perp \]

\[ a = \{ \&y \} \]
\[ x = \text{odd} \]
\[ y = \perp \]
\[ z = \perp \]
Constant domain

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

\[ a = \&y \]
\[ x = \perp \]
\[ y = \perp \]
\[ z = \perp \]
Value-set domain

\[ a = \{\&x\} \]
\[ x = \bot \]
\[ y = \bot \]
\[ z = \bot \]

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

\[ a = \&y \]
\[ x = \{13, 77\} \]
\[ y = \{42\} \]
\[ z = \{2, 5\} \]
Bitwise domain

\[ a = \{&x\} \]
\[ x = 11111111 \]
\[ y = 11111111 \]
\[ z = 11111111 \]

\[ a = \{&x\} \]
\[ x = 0 \perp 0011101 \]
\[ y = 001001010 \]
\[ z = 0000000 \]

\[ a = \{&y\} \]
\[ x = 0 \perp 0011101 \]
\[ y = 00101010 \]
\[ z = 0000000 \]

\[ a = \{&y\} \]
\[ x = 0 \perp 001101 \]
\[ y = 00101010 \]
\[ z = 0000000 \]

\[ a = \{&y\} \]
\[ x = 0 \perp 001101 \]
\[ y = 00101010 \]
\[ z = 0000000 \]

\[ a = \{&y\} \]
\[ x = 0 \perp 001101 \]
\[ y = 00101010 \]
\[ z = 0000000 \]

\[ a = \{&y\} \]
\[ x = 0 \perp 001101 \]
\[ y = 00101010 \]
\[ z = 0000000 \]

\[ a = \{&y\} \]
\[ x = 0 \perp 001101 \]
\[ y = 00101010 \]
\[ z = 0000000 \]
Interval domain

\[ a = \{&x\} \]
\[ x = \bot \]
\[ y = \bot \]
\[ z = \bot \]

\( a = \&x \) 
\[ z = 2 \]
\[ *a = 13 \]

\[ z = 5 \]
\[ *a = 77 \]

\[ z = 1 \]

\[ a = \{&y\} \]
\[ x = \bot \]
\[ y = \bot \]
\[ z = [2, 5] \]

\( a = \{&y\} \) 
\[ x = \bot \]
\[ y = [26, 42] \]
\[ z = [1, 1] \]

\[ a = \{&y\} \]
\[ x = \bot \]
\[ y = [26, 42] \]
\[ z = [2, 5] \]
cXprop components

Abstract domain

CIL Core

CIL's feature interface

cXprop's domain interface
Constant and value-set domains

- Constant domain
- Transfer functions
  - Special cases
  - Handle \( \perp \)
  - Call CIL's constant folder
- Value-set domain
- User-defined set size
- Quadratic-time transfer functions using brute force
Bitwise and interval domains

- Bitwise domain
- Vectors of three-valued bits
  - 0, 1, ⊥
- Regehr and Duongsaad's transfer functions

- Interval domain
- Upper and lower bound on range of values
- Regehr and Duongsaad's transfer functions
Sensor network programming

- Component-based
  - Block-box reuse
- Interrupt-driven
- Very constrained environment

- cXprop features added for sensor network application analysis
  - Concurrency analysis
  - Interrupt bit modeling
  - Atomic section removal
Modeling concurrency in cXprop

- Analysis of interrupt-driven concurrency
- Three types of program data
  - Unshared data
  - Unprotected shared data
  - Protected shared data
Concurrency
**cXprop's transformations**

- **Three primary transformation modes**
  - **Normal conditional constant propagation**
    
    ```
    x = 1;
    if (flag == 0) x++;
    y = x + 1;
    
    x = 1;
    if (flag == 0) x=2;
    y = x + 1;
    ```

- **Assert program state**
    
    ```
    x = 1;
    if (flag == 0) x++;
    y = x + 1;
    
    x = 1;
    assert (x==1); if (flag == 0) { assert(x==1); x++; }
    assert (x==1 || x==2); y = x + 1;
    ```

- **Dynamic dataflow information**
    
    ```
    x = 1;
    cXprop_dynamic_meet(& cxp_glob_var_0, x);
    if (flag == 0) { x++; 
    cXprop_dynamic_meet(& cxp_glob_var_1, x); }
    y = x + 1;
    cXprop_dynamic_meet(& cxp_glob_var_2, y);
    ```
Validation of cXprop

- Specific test cases
- Benchmarks
  - SPEC 2000: gzip, mcf, bzip2
  - MiBench: basicmath, patricia, FFT
  - TinyOS 1.x: blink, cnttoledsandrfm, hfs, rfmtoleds, surge, taskapp, acoustic, agilla, ecc
  - TinyOS 2.x: null, basestation
  - Miscellaneous: rc4, dhrystone, yacr2
- Random program generator
## Analysis times

<table>
<thead>
<tr>
<th>Benchmark (loc)</th>
<th>Constant</th>
<th>Value-set</th>
<th>Bitwise</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>blink (1852)</td>
<td>0.29</td>
<td>0.78</td>
<td>0.75</td>
<td>0.82</td>
</tr>
<tr>
<td>rfmtoleds (7785)</td>
<td>6.88</td>
<td>46.9</td>
<td>54.9</td>
<td>95.2</td>
</tr>
<tr>
<td>surge (10653)</td>
<td>18.1</td>
<td>138</td>
<td>172</td>
<td>299</td>
</tr>
<tr>
<td>acoustic (13804)</td>
<td>32.5</td>
<td>193</td>
<td>253</td>
<td>360</td>
</tr>
<tr>
<td>agilla (34414)</td>
<td>786</td>
<td>2990</td>
<td>2770</td>
<td>5570</td>
</tr>
</tbody>
</table>

Times are in seconds

- **Analysis time contributors**
  - Size of concretization sets
  - Transfer functions
  - Dataflow representation
Code size reduction results

![Graph showing code size reduction results for different domains: Constant Domain, Value Set Domain, Bitwise Domain, and Interval Domain. The x-axis represents different applications: cnttoledsandrfin, blink, basestation, hfs, acoustic, and taskapp. The y-axis represents code size as a percentage of the original nesC code size. The graph illustrates the reduction in code size for each application across different domains.]
Comparing abstract domains

- Goal: Apples-to-apples comparison across domains
- Information known metric

\[
\text{# of information bits} \overset{\text{def}}{=} \log_2 j - \log_2 k
\]

\( j = \text{total # of representable values} \)
\( k = \# \text{ of possible values} \)

- total information known for a program = 
  total # of information bits / total # of bits possible
Example information bit computation

- Consider a 32-bit int with abstract value of 
  \{4, 17, 19, 312\}

- There are \(2^{32}\) representable values

- There are 4 possible values

- \(\log_2 2^{32} - \log_2 4 = 30\) information bits
Comparison of information known

![Bar chart showing comparison of information known across different domains. The x-axis represents various domains, and the y-axis represents the percentage of information known. The domains are color-coded: Constant Domain (red), Value Set Domain (green), Bitwise Domain (yellow), Interval Domain (purple), and Combined Domain (black). The bars indicate the percentage of information known for each domain across different applications.]
Conclusion

- Interprocedural dataflow analysis based on abstract interpretation
- Clean domain interface for pluggable domains
- Quantitative comparison of abstract domains
  - No universal winner
- Additions to handle concurrency in interrupt-driven embedded systems

- You can download cXprop here:
  
  http://www.cs.utah.edu/~coop/research/cxprop
Pluggable Abstract Domains for Analyzing Embedded Software

Nathan Cooprider and John Regehr
University of Utah
School of Computing
Necessary assumptions for cXprop

- Precise and sound dataflow analysis of C is nearly impossible
- Assumptions about
  - Memory model
  - External calls
  - Floating point
  - Inline assembly
  - Order of evaluation
  - Concurrency
- Consider store to array where index is indeterminate
  - `big_array[⊥] = 42;`
  - Sound analysis must drop all dataflow facts
  - Could overwrite return address
Uses of cXprop

- Compare abstract domains
- Optimize C code
  - Reduce code size
  - Lower duty cycle
  - Identify unused bits
- Clean up after CCured
- Facilitate other analyses
cXprop's abstract domain interface

- Ocaml
  - Do not need to know CIL internals
  - Parity domain implemented in under an hour

- 32 transfer functions
  - `mult(abstract value, abstract value, type) ➔ abstract value`
  - `shiftrt(abstract value, abstract value, type) ➔ abstract value`

- 6 backwards functions
  - `backwards_eq (abstract value, abstract value, type, abstraction function) ➔ (boolean * (abstract value * abstract value) * (abstract value * abstract value))`

- 11 utility functions
  - widen, concretize, abstract, meet, . . .
FLID removal

The chart shows the percentage of checks removed across various benchmarks with different compiler optimizations.

- Benchmark names: BlinkTask_Mica2, GenericBase_Mica2, CntToLedsAndRfm_Mica2, SenseToRfm_Mica2, Surge_Mica2, HighFrequencySampling_Mica2
- Compiler optimizations: gcc, CCured optimizer + gcc, CCured optimizer + cXprop + gcc, CCured optimizer + inlining + cXprop + gcc
- Y-axis: Checks removed (%)
- X-axis: Benchmarks

The results indicate that gcc can remove a significant number of checks in most benchmarks, with additional optimizations showing improvements in specific scenarios.