Pluggable Domains for C Dataflow Analysis

Nathan Cooprider and John Regehr

{coop, regehr}@cs.utah.edu

School of Computing, University of Utah
Goal

- Aggressive static analysis of embedded C programs
  - Use compiler technology
    - not model checking technology
  - Perform optimizations
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 (this is old, we do better now)
Initial 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
Uses of cXprop

• Compare abstract domains

• Optimize C code
  – Reduce code size
  – Lower duty cycle
  – Identify unused bits

• Clean up after CCured

• RAM compression
C : A million and one ways . . .

• To shoot yourself (and our analysis) in the foot
  - Stack manipulations
  - External calls
  - Floating point
  - Order of evaluation
  - Concurrency
  - Missing returns
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
cXprop components

Abstract domain

CIL Core

CIL's feature interface

cXprop's domain interface
CIL

- **C Intermediate Language** – developed at UCB
- Cleans up C to a few core constructs
  - removes syntactic sugar (like “->” notation)
  - arrays become pointers
  - all loops become while loops
- Works on **real** programs
  - handles ANSI-C, Microsoft C, and GNU C
  - SPEC 95, linux kernel, 🦀, bzip
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, 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`, . . .
Results of abstract interpretation

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

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

\[ x \]
\[ F \]
\[ \ast a=26 \]
\[ F \]
\[ z=0 \]
\[ T \]
\[ z=1 \]

\[ a=\&x \]
\[ z=2 \]
\[ \ast a=13 \]
\[ a=\&y \]

\[ a= \]
\[ x= \]
\[ y= \]
\[ z= \]
Parity domain

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

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

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

\[
\begin{aligned}
a &= \&x \\
\text{opaque} &\quad \begin{cases} 
T : z = 5 & \quad \begin{cases} 
* a = 77 \\
\text{a} = \&y \\
\end{cases} \\
F &\quad \begin{cases} 
* a = 13 \\
\text{a} = \&y \\
\end{cases}
\end{cases}
\end{aligned}
\]

\[
\begin{aligned}
a &= \{\&y\} \\
x &= \bot \\
y &= \bot \\
z &= \bot
\end{aligned}
\]
Value-set domain

\( a = \{\&x\} \)
\( x = \perp \)
\( y = \perp \)
\( z = \perp \)

\( a = \&x \)
\( x = \perp \)
\( y = \perp \)
\( z = \perp \)

opaque

\( z = 2 \)
\( \ast a = 13 \)

\( a = \&y \)

\( a = \{\&y\} \)
\( x = \{13, 77\} \)
\( y = \{42\} \)
\( z = \{2, 5\} \)

\( a = \{\&y\} \)
\( x = \{13, 77\} \)
\( y = \perp \)
\( z = \{2, 5\} \)

\( a = \{\&y\} \)
\( x = \{13, 77\} \)
\( y = \{42\} \)
\( z = \{1\} \)
Bitwise domain

- $a = \{&x\}$
- $x = 11111111$
- $y = 11111111$
- $z = 11111111$

- $a = \{&x\}$
- $x = 0\bot001101$
- $y = 00101010$
- $z = 00000\bot\bot\bot$

- $a = \{&y\}$
- $x = 0\bot001101$
- $y = 00101010$
- $z = 000000\bot\bot$

- $a = \{&y\}$
- $x = 0\bot001101$
- $y = 00101010$
- $z = 0000000\bot$

- $a = \{&x\}$
- $x = 11111111$
- $y = 11111111$
- $z = 11111111$
Constant and value-set domains

- Constant domain
- Transfer functions
  - Special cases
  - Handle $\bot$
  - Call CIL's constant folder
- Value-set domain
- User-defined set size
- Quadratic-time transfer functions using brute force
let mult d1 d2 tp =
match d1, d2 with
  Constant(z), _
  | _, Constant(z) when (isZero z) ->
    Constant (zero)
  | Bottom, _
  | _, Bottom -> Bottom
  | Constant(e1), Constant(e2) ->
    conc_to_abs (BinOp(Mult,e1,e2,tp))

int tricky () {
  int x = 1;
  int count = 0;
  do {
    int b = x;
    if (b != 1)
      x = 2;
    count += x;
  } while (count < 10);
  return x;
}
Bitwise and interval domains

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

- Interval domain
- Upper and lower bound on range of values
- Regehr and Duongsa's transfer functions
Bitwise

```cpp
int tricky () {
    int x = 1;
    int count = 0;
    do {
        int b = x;
        if (b != 1)
            x = 2;
        count += x;
    } while (count < 10);
    return x;
}
```

let lnot (d, dk) tp =
if ((dk = no_bottoms) && (d = I.zero))
    then TbTrue
else if (I.logand d dk) <> I.zero then TbFalse
    else TbBottom
```
Software and hardware platform

- TinyOS
  - libraries of code for sensor networks
  - idioms are conducive to static analysis
    - static memory allocation model
  - written in nesC
    - unsafe dialect of C that compiles to C
- Mica2 from Crossbow
  - ATmega128 8-bit processor
  - 4 KB RAM, 128 KB flash
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
    
    \[
    \begin{array}{c}
    x = 1; \\
    \text{if (flag == 0) } x++; \\
    y = x + 1; \\
    \end{array}
    \quad \rightarrow \quad
    \begin{array}{c}
    x = 1; \\
    \text{if (flag == 0) } x=2; \\
    y = x + 1; \\
    \end{array}
    \]

  – Assert program state
    
    \[
    \begin{array}{c}
    x = 1; \\
    \text{if (flag == 0) } x++; \\
    y = x + 1; \\
    \end{array}
    \quad \rightarrow \quad
    \begin{array}{c}
    x = 1; \\
    \text{assert (x==1); \text{if (flag == 0) } \{ \text{assert(x==1); x++; } \}} \\
    \text{assert (x==1 || x==2); y = x + 1;}
    \end{array}
    \]

  – Dynamic dataflow information
    
    \[
    \begin{array}{c}
    x = 1; \\
    \text{cXprop\_dynamic\_meet(& cxp\_glob\_var\_0, x);} \\
    \text{if (flag == 0) } \{ x++; \\
    \text{cXprop\_dynamic\_meet(& cxp\_glob\_var\_1, x);}\}
    \end{array}
    \quad \rightarrow \quad
    \begin{array}{c}
    x = 1; \\
    \text{cXprop\_dynamic\_meet(& cxp\_glob\_var\_0, x);} \\
    \text{if (flag == 0) } \{ x++; \\
    \text{cXprop\_dynamic\_meet(& cxp\_glob\_var\_1, x);}\}
    \end{array}
    \]

\]
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>rfmtreeleds (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

![Code size reduction results graph]

- Constant Domain
- Value Set Domain
- Bitwise Domain
- Interval Domain

<table>
<thead>
<tr>
<th>Code size as a percentage of original nesC code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
</tr>
<tr>
<td>cnttoledsandrfm</td>
</tr>
</tbody>
</table>
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
Linear vs. logarithmic

These are the possible values for an eight-bit variable
We know it is not 42

Taking away one is not very useful
We know it is odd

Taking away half is real progress
Iterations later...

How about now?
We know it is not nine

Taking away one is still not very meaningful
We know it is less than nine

So we use the log scale
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
Demo
Application 1:

Safe TinyOS
A conflict in embedded systems

- Tiny embedded systems coded in C
- These are used in important and even safety critical systems
- Developers are extremely reluctant to change languages
- Developers are extremely reluctant to introduce time, space, and memory overhead
Our contribution

• Type and memory safety with little overhead
  – Lots of engineering required
  – Must be fast if people are going to use it
  – Low or no costs
    • 2% average decrease in duty cycle
    • 8% average increase in code size
    • 6% average increase in data size

• Safety with minimal programmer impact
  – Safe TinyOS
  – works on unmodified legacy C code
Safety for embedded systems

- Type and memory safety
  - early detection of bugs
  - observability and predictable consequences of run-time faults
- CCured is the starting point
  - Safe C dialect developed at Berkeley
  - Translates a C program into a safe C program by inserting safety checks
- Example of null check:
  ```c
  _CHECK_NULL((void *)next_task, 0xAA);
  (*((next_task))());
  ```
Initial drawbacks of CCured

• Overhead
  – All these checks slow down the program and use up memory

• CCured library
  – Does not fit on motes if unmodified
  – First estimation at fitting onto Mica2
    • over 1KB of RAM
    • over 3KB of ROM
Addressing the drawbacks

- One-time, manual changes to CCured library
  - remove OS and x86 dependencies
  - drop garbage collector
- Refactor hardware accesses
- Protect non-atomic accesses to fat-pointers
- Compress the error messages
- Optimize
Optimize with an inliner

• Inlining can reduce size
• Inlining introduces some context sensitivity
• Parameterize inlining decisions
  – sweet spot in between
    • maximally reduce size AFTER optimization
Optimize with cXprop

• Interprocedural dataflow analysis
• Analyzes concurrent programs
  – Removal of nested atomic sections
• Simultaneous pointer analysis
• Aggressive dead code elimination
  – whole program
Safe TinyOS Toolchain

- run nesC compiler
- refactor HW accesses
- run CCured + concurrency safety
- compress error messages
- run inliner
- run cXprop
- run gcc

tailored CCured runtime library
Average 2% decrease in duty cycle via Safe TinyOS
Average 8% increase in code size via Safe TinyOS
Average 6% increase in data size via Safe TinyOS
Application 2:

RAM compression
Observations

• RAM is used inefficiently
• Low-end systems persistently need RAM
  - Sensor networks use Harvard-architectures
• Resource reduction benefit is 0 after it “fits”
• Manual optimization is difficult

• So we combine cXprop with RAM compression
RAM compression

- Original declaration

```c
typedef struct {
    void (*tp)(void);
} TOSH_sched_entry_T;
volatile TOSH_sched_entry_T TOSH_queue[8];
```

- Compression table for task queue

```c
unsigned short const __attribute__((__progmem__)) __valueset_3[4] =
{NULL, &BlinkTaskM$processing, &TimerM$HandleFire, &TimerM$signalOneTimer};,
```

- Structure for holding compressed values

```c
struct __compressed {
    char f9[2] ;
    unsigned char f0 : 3 ;
    ...;
};
```

- Original code for reading task queue

```c
func = TOSH_queue[old_full].tp;
```

- Code for reading the compressed queue

```c
__tmp = __array_read (__compressed.f9, old_full, 2);
func = __finv_16 (__valueset_3, __tmp);
```
“Current” cXprop optimizations
cXprop with RAM compression
Tradeoff graphs

Trading RAM for ROM ———
Trading RAM for duty cycle ————

Trading RAM for ROM ———
Trading RAM for duty cycle ————

Percent change in code size
Percent change in duty cycle
Percent of compressible RAM compressed
Percent of compressible RAM compressed
Another demo
(sort of)
Conclusion

- Interprocedural dataflow analysis based on abstract interpretation
- Clean domain interface for pluggable domains
- Quantitative comparison of abstract domains
- Additions to handle concurrency in interrupt-driven embedded systems
- Cool applications
  - Safe TinyOS and RAM compression
- You can download cXprop here: http://www.cs.utah.edu/~coop/research/cxprop