Data-flow Analysis for Interrupt-driven Microcontroller Software

Nathan Cooprider

Advisor: John Regehr

Dissertation defense
School of Computing
University of Utah
Data-flow Analysis for Interrupt-driven Microcontroller Software

- A whole program analysis
- Targeting embedded C programs
- Suitable for use in a compiler
Microcontrollers (MCUs)

- 10 billion units / year
- $12.5 billion market in 2006
- Cheap
- Resource constrained
- e.g. Wireless sensor networks
  - Mica2 mote
    - ATmega 128L (4 MHz 8-bit MCU)
    - 128 kB code, 4 kB data SRAM
Problem

- Resources are constrained
- Software outlives hardware
  - Code reuse leads to bloat
- Low-level code confuses analysis
  - Interrupt-driven concurrency
  - Device register access
Solution

- Traditional data-flow analysis
  - Not adequate precision for MCU software
- New techniques to increase precision
  - Deal with concurrency
  - Track volatile data
- Use in code transformations
  - Optimizations

Thesis statement
Contributions

• Analysis techniques
  – Interatomic concurrent data-flow (ICD)
  – Tracking data through volatile variables
• Tool – cXprop
• Applications
  – Practical memory safety – Safe TinyOS
  – Offline RAM Compression
• Open-source OS for WSNs
• Written in nesC
  – Dialect of C
• Concurrency
  – Tasks and interrupts
  – No threads
  – Atomic sections
Abstract interpretation

ICD

Volatile tracking

Conditional x propagation

Pointer analysis

Safe TinyOS

RAM compression
Abstract interpretation

switch (x) {
    . . .
    break;
    case 42: case 7: case -1:
        if (x < 0)
            x *= -1;
        x++;
        if (x == 0)
            assert(0);
    break;
    . . .
Abstract interpretation

switch (x) {
    ...  
    break;
    case 42: case 7: case -1:
        if (x < 0)
            x *= -1;
    x++;  
    if (x == 0)
        assert(0);
    break;
    ...  
}

• Abstract domain
  - Abstract values
  - Form poset
    - Subset relation (⊆)
  - Lattice
    - Undefined (T)
    - Unknown (⊥)

• Data-flow analysis
  - Transfer functions
  - Merging (∪)
  - Fixed point
Abstract interpretation

• Abstract domain
  – Abstract values
  – Form poset
    • Subset relation ($\subseteq$)
  – Lattice
    • Undefined ($\top$)
    • Unknown ($\bot$)

• Data-flow analysis
  – Transfer functions
  – Merging ($\cup$)
  – Fixed point

x < 0
{42,7} ∪ {42,7,1}
{42,7} + +
{43,8,2}

x == 0
{43,8,2} ∪ T

x* = 1;
{1}

assert(0);
{42,7,-1}
{-1}
Abstract interpretation

ICD

Volatile tracking

Conditional x propagation

Pointer analysis

Safe TinyOS

RAM compression
Interrupt-driven concurrency

- Problems
  - C statements not necessarily atomic

```c
x = 0x4242;
```

```asm
ldi r24, 0x42
ldi r25, 0x42
```

Interrupt
Interrupt-driven concurrency

- Problems
  - C statements not necessarily atomic
  - Preempts sequential control flow
- Complicated control flow
- Synchronization
  - One flow does not “break” another
  - Bad synchronization happens
    - Difficult or impossible to reason about
    - Must deal with conservatively (⊥)

A race
Related work

• **Thread-based concurrency**

• **Leveraging race detection**

• **Formal semantics**
Race detection

- Lockset analysis - standard technique
  - Lock status = interrupt enable bit status
  - Only one lock – no lock aliasing
  - nesC uses lexical nesting

- Data classification
  - Unshared – accessed only from main
  - Shared – accessed from interrupts
Race detection

- Accessed without locking
- Written in shared or unlocked unshared code
- Accessed in shared code

- Data classification
  - Unshared – accessed only from main
  - Shared – accessed from interrupts
Race detection case analysis

Interrupt

Write

Racing

Interrupt or task

Reads

Atomic section
Data classification

- Data
  - Static (Global)
    - Stack
      - Sequential
      - Unshared 50%
    - Shared
      - Concurrent
        - Racing 6%
        - Not racing 44%
Atomic interleaving

Published at LCTES 2006

Interatomic Concurrent Data-flow
Volatile

- C type qualifier – `volatile int`
- Special case of C’s memory model
  - Read value may change “randomly”
  - Write may affect system state
- E.g., racing data, device registers
- Behavior opaque at C level
- Prevents compiler optimizations
Tracking volatile RAM

- Locate variables backed by RAM
- Introduce concurrency information
  - Interatomic concurrent dataflow
- Have sound approximation of mutators
  - Behavior not opaque at system level
- Safely analyze volatile variables in RAM
Tracking volatile device registers

- Hardware registers
  - Memory mapped I/O
  - Hardware not actually random (volatile)
- Can track using MCU-specific information
  - OK to track individual bits
    - Instead of whole register
    - Interrupt bit of status register

Volatile tracking
Pointer analysis

- Points-to sets – must and may alias
  - Two pluggable domains
  - Subtleties from context-insensitivity
- Targets:
  - Device registers
  - Scalars
  - Structs
  - Arrays
  - not-NUL
  - Heap

\[
\begin{array}{c}
\{&x\} \\
\{&y\} \\
\{&z\} \\
\{\text{NULL}\} \\
\{&x,&y\} \\
\{&x,&z\} \\
\{&x,\text{NULL}\} \\
\{&y,&z\} \\
\{&y,\text{NULL}\} \\
\{&z,\text{NULL}\} \\
\text{not-NUL} \\
\text{Heap}
\end{array}
\]

\[\text{Pointer analysis}\]
Conditional X propagation

- Pluggable abstract domains
  - From conditional constant propagation
- Clean domain interface
  - Transfer functions
  - Abstract interpretation utility functions

Diagram:
- Conditional X propagation
  - Abstract domain
  - Analysis
Domains

Constant

Bitwise

Interval

Conditional X propagation
Safe TinyOS

RAM compression

Abstract interpretation

Conditional x propagation

Pointer analysis

Volatile tracking

ICD
Struct splitter
Inliner
Cleaner

Fixed point computation
Value-flow
Pointer-flow
ICD
Volatile tracking

• Constant propagation
• Dead code elimination
• Dead data elimination

Transformations
Cleaner

Implemented as a CIL extension
Suppose we have a WSN...
Suppose we have a WSN...

• What happened?
  – State got corrupted
  – Hard to debug
    • Limited visibility into executing systems
    • Difficult to replicate complex bugs

• Memory safety can
  – Catch all pointer and array bounds errors
    • Before they corrupt state
  – Provide a choice of recovery action
    • Display error message or reboot
Safe TinyOS

- Modify TinyOS to work with Deputy
- Enforce Deputy’s safety model under concurrency
- Reduce overhead

Deputy: existing solution for making C safe

Expand into system safety

Published at SenSys 2007
Safe TinyOS toolchain

int post(val_t* buf, int n);

cXprop

run modified nesC compiler
enforce safety using Deputy
deal with concurrency
compress
Safe TinyOS app

TinyOS code
Annotate
Safe TinyOS code

Modify TinyOS to work with Deputy
Enforce Deputy’s safety model under concurrency
Reduce overhead

cXprop

Safe TinyOS app

int post(val_t* buf, int n);
Concurrency

- Deputy enforces safety in sequential code
- cXprop avoids extraneous protection
  - Only racing variables need protection
Code size

- Safe, verbose error messages in ROM
- Safe, error messages reduced to integers
- Safe, error messages reduced to integers, and optimized by cXprop
- Unsafe and optimized by cXprop

The chart shows the change in code size for different applications, with the x-axis representing the applications (Geometric mean, RadioSenseToLeds, BaseStation, MViz, AntiTheft) and the y-axis representing the percentage change in code size (0% to 300%).
Safe, verbose error messages in ROM
Safe, error messages reduced to integers
Safe, error messages reduced to integers, and optimized by cXprop
Unsafe and optimized by cXprop
A closer look at RAM usage

- On-chip RAM for MCUs expensive
  - Kilobytes, not megabytes or gigabytes
  - Data in SRAM – 6 transistors / bit
  - SRAM can dominate power consumption of a sleeping chip
A closer look at RAM usage

• On-chip RAM for MCUs expensive
  
  On-chip RAM is persistently scarce in tiny MCU-based systems

• Is RAM used efficiently?
  – Performed value profiling for MCU apps
    • Apps already heavily tuned for RAM usage
  – Result: Average byte stores four values!
Offline RAM compression

- Automated sub-word packing for statically allocated scalars, pointers, structs, arrays
  - No heap on targeted MCUs
  - Trades ROM and CPU cycles for RAM

Published at PLDI 2007
Method

\( x \overset{\text{def}}{=} \) variable that occupies \( n \) bits

\( V_x \overset{\text{def}}{=} \) conservative estimate of value set

\[ \left\lfloor \log_2 |V_x| \right\rfloor < n \Rightarrow \text{RAM compression possible} \]

\( C_x \overset{\text{def}}{=} \) another set such that \( |C_x| = |V_x| \)

\( f_x \overset{\text{def}}{=} \) bijection between \( V_x \) and \( C_x \)

\[ n - \left\lfloor \log_2 |C_x| \right\rfloor \Rightarrow \text{bits saved through compression of } x \]
Example Compression

void (*function_queue[8])(void);
Example Compression

void (*function_queue[8])(void);

\( n = \text{size of a function pointer} = 16 \text{ bits} \)
Example Compression

\[ x \] \[ V_x \]

&function_A
&function_B
&function_C
NULL
Example Compression

\[ n = 16 \text{ bits} \]

\[ |V_x| = 4 \]

\[ \lceil \log_2 |V_x| \rceil < n \]

\[ 2 < 16 \]
Example Compression

$$f_x \overset{\text{def}}{=} V_x \text{ to } C_x \overset{\text{def}}{=} \text{compression}$$

$$f_x^{-1} \overset{\text{def}}{=} C_x \text{ to } V_x \overset{\text{def}}{=} \text{decompression}$$
Example Compression

$V_x = \{\text{cloud}, \text{cloud}, \text{cloud}, \text{no match}\}$

$f_x \triangleq$ compression

$\text{table scan}$

$f_x^{-1} \triangleq$ decompression

$\text{table lookup}$
Example Compression

\[ V_x = \{ \text{clouds} \} \]

128 bits reduced to 16 bits
112 bits of RAM saved
RAM compression results

The bar chart illustrates the change in RAM compression for different datasets. The x-axis represents the datasets: GenericBase, RfmToLeds, CntToLedsAndRfm, Ident, TinyDB. The y-axis represents the change in compression percentage. The bars are color-coded: red for duty cycle, yellow for code size, and blue for data size. The chart shows the relative change in compression for each dataset.
RAM compression results

- **cXprop (no compression)**
  - 10% RAM reduction
  - 20% ROM reduction
  - 5.9% duty cycle reduction

- **Compression**
  - 22% RAM reduction
  - 3.6% ROM reduction
  - 29% duty cycle increase

**Tradeoffs**
Abstract interpretation

ICD

Volatile tracking

Conditional x propagation

Pointer analysis

Safe TinyOS

RAM compression
Conclusion

• Interatomic concurrent data-flow
• Volatile data may be tracked
• Better analysis \( \Rightarrow \) more optimizations
  – Safe TinyOS – practical memory safety
  – RAM compression – 22% RAM reduction

http://www.cs.utah.edu/~coop/research/cxprop/
http://www.cs.utah.edu/~coop/safetinyos/
http://www.cs.utah.edu/~coop/research/ccomp/

Thank you
Cost/Benefit Ratio

\[ \sum C_i \left( A_i + B_i V \right) \]

- \( C \) is the access profile
- \( A, B \) are platform-specific costs
- \( V \) is the cardinality of the value set

\[ S_u - S_c \]

- \( S_u \) is the original size
- \( S_c \) is the compressed size
Turning the RAM Knob

Percent change

Percent of compressible RAM compressed

duty cycle
code size
data size

0%
Turning the RAM Knob

Percent change

Percent of compressible RAM compressed

duty cycle  code size  data size

10%
Turning the RAM Knob

Percent change

Percent of compressible RAM compressed

duty cycle
code size
data size

20%
Turning the RAM Knob

Percent change vs. Percent of compressible RAM compressed

- duty cycle
- code size
- data size

30%
Turning the RAM Knob

![Graph showing percent change in duty cycle, code size, and data size as percent of compressible RAM compressed increases.]
Turning the RAM Knob

![Graph showing percent change in duty cycle, code size, and data size against percent of compressible RAM compressed. The graph indicates a 50% compression at the 50% mark.]
Turning the RAM Knob

- duty cycle
- code size
- data size

Percent change vs. Percent of compressible RAM compressed

60%
Turning the RAM Knob

- Duty cycle
- Code size
- Data size

Percent change vs. Percent of compressible RAM compressed.
Turning the RAM Knob

Percent change

Percent of compressible RAM compressed

duty cycle
code size
data size

80%
Turning the RAM Knob

- duty cycle
- code size
- data size

90%
Turning the RAM Knob

Percent change vs. Percent of compressible RAM compressed

- duty cycle
- code size
- data size

Graph showing changes in RAM metrics as a function of RAM compression percentage.
Turning the RAM Knob

Percent change

Percent of compressible RAM compressed

duty cycle
code size
data size

95%
Future work

- Triggering and sequencing
  - Timer interrupt handler
  - Fire
  - Trigger
  - Fire

- Caching compressed values
  - decompress x
  - decompress x
  - decompress x
More related work

• Safe TinyOS

• Offline RAM compression
PAG

• Program Analysis Generator
  – Domain specific language input describes
    • Domain lattice
    • Transfer functions
    • Language-describing grammar
    • Fixed point solution method
  – Data-flow analyzer as output
• Does not deal with concurrency
• Used to evaluate fixed point solutions
Feature comparison

- cXprop
- No concurrency
- No assume nesC
- No arrays
- Address taken
- No volatiles

Reduction in code size

GEO_MEAN
RadioSenseToLeds
BaseStation
AntiTheft_Nodes
MViz
Domain comparison

- constant
- valueset
- interval
- bitwise
- multiinterval

Reduction in code size

GEO_MEAN
RadioSenseToLeds
BaseStation
AntiTheft_Nodes
MViz
Resource reduction

- **ROM**: 12%
- **duty cycle**: 8.3%
- **static RAM**: 2.5%
- **stack RAM**: 1.8%

**GEO_MEAN**
- **ROM**: 1.8%
- **duty cycle**: 12%
- **static RAM**: 2.5%
- **stack RAM**: 8.3%

**RadioSenseToLeds**
- **ROM**: 1.8%
- **duty cycle**: 8.3%
- **static RAM**: 2.5%
- **stack RAM**: 1.8%

**BaseStation**
- **ROM**: 12%
- **duty cycle**: 8.3%
- **static RAM**: 2.5%
- **stack RAM**: 1.8%

**AntiTheft_Nodes**
- **ROM**: 12%
- **duty cycle**: 8.3%
- **static RAM**: 2.5%
- **stack RAM**: 1.8%

**MViz**
- **ROM**: 12%
- **duty cycle**: 8.3%
- **static RAM**: 2.5%
- **stack RAM**: 1.8%
Atomic interleaving

Published at LCTES 2006
Context insensitivity

a is a global variable

foo
int x = 7;
bar(&x);

bar(int *y)
goo(y);

goo(int *z)
*z = 42;
a = *z;

a = {27}
x = {7,42}

a = {27}
y = {&x}

a = {7,27,42}
z = {&x}
Benchmark descriptions

• AVR ATmega128 code
• TinyOS
• 3,000-26,000 lines of C code
• Analysis times - seconds to an hour
• Metrics
  – Duty cycle
    • % of time processor is on
    • Obtained from Avrora
      – Cycle-accurate simulator for WSNs
  – Code size and data size
Wireless sensor networks

- 10 billion units/year
- $12.5 billion market in 2006
- Cheap
- Resource constrained

- Mica2 mote
  - ATmega 128L (4 MHz 8-bit MCU)
  - 128 KB code, 4 KB data SRAM