Static Analysis of Embedded C

John Regehr
University of Utah

Joint work with Nathan Cooprider
Motivating Platform: TinyOS

• Embedded software for wireless sensor network nodes
• Has lots of SW components for MAC, routing, collection, data acq., etc.
• Runs on highly constrained MCUs
• Goal of TinyOS is to get MCU into lowest possible power state as soon as possible
TinyOS Execution Model

• Application = interrupts + tasks + scheduler
  – Interrupt handlers natively supported by the HW platform
  – Tasks are deferred function calls
    • No blocking – tasks are not threads
    • Scheduled FIFO
    • TinyOS scheduler is about 100 lines of code
  – Concurrency control by disabling interrupts
TinyOS Constraints

• Energy
  – 2 AA batteries == ~4 Watt-hours

• SRAM for data storage
  – 0.5 KB – 10 KB
  – SRAM can dominate power consumed by sleeping MCU

• Flash for code and read-only data
  – 4 KB – 256 KB
nesC – the TinyOS language

• C dialect with
  – Static component model
  – Race condition detection
  – Generics

• nesC compiler
  – Input: Collection of nesC components
  – Output: Monolithic C program

• We analyze compiler output
Thesis

1. Embedded codes contain significant structure not exploited by compilers and other existing analyzers

2. Static analysis based on a language model and a system model can uncover and exploit this structure

3. Analysis results are useful
Using Analysis Results

• To support program transformations
  – Optimizing Type-Safe TinyOS (SenSys 2007)
    • Reduce CPU overhead from 24% to 5%
    • Reduce code size overhead from 35% to 13%
  – Offline RAM compression (PLDI 2007)
    • Reduce SRAM usage of TinyOS applications by 23%

• To cheaply discover program facts supporting verification?
What’s different about this work vs. what you see at PLDI and ICSE?

• It’s easier…
  – Scalability not so important
  – Simple locking model
  – We know the platform

• It’s harder…
  – Unstructured interrupt-driven concurrency
  – Direct hardware access
  – We know the platform
Analysis Overview

• Flow sensitive
• Path and context insensitive
• Scalars analyzed using value sets
  – Bounded-size sets of concrete values
• Pointers analyzed using pointer sets
  – Bounded-size sets of pointers + specific support for non-null
  – Supports both must and may alias analysis
Analysis Overview 2

• Precede analysis with function inlining pass
  – Compensates somewhat for context insensitivity

• Arrays summarized as single abstract value
• What do you get by applying an analyzer for sequential C code to TinyOS applications?
  – Can analyze local variables
    • Assumption: No pointers across stacks
  – Cannot generally analyze globals
    • Many are touched by both main() and interrupts
Analyzing Global Variables

• “Synchronous” global state == Not used for communication between concurrent flows
  – Sequential semantics apply

• Problem:
  – Conservative identification of synchronous variables requires pointer analysis
  – Many global variables are pointers
  – Circular dependency between analyses
Finding Synchronous State

• Solution: Integrate
  1. Pointer analysis
  2. Synchronous / asynchronous classification

• Next question: How to analyze variables that are not synchronous?
Analyzing Asynchronous State 1

• Strawman algorithm:
  1. Find program points where interrupts cannot be shown to be disabled
  2. Add a flow edge from each such point to the start of each interrupt handler (and back)

• This is overkill: Too many extra flow edges ⇒ long analysis time

• This is unsound: C statements are not atomic operations
Analyzing Asynchronous State 2

• Better algorithm:
  1. Find blocks that execute with interrupts disabled (“atomic blocks” in nesC)
  2. Conservatively find “racing” variables == Asynchronous and stored-to outside of atomic blocks
  3. Analyze non-racing variables only
     - Racing == homebrew synchronization protocol
     - Cannot reuse nesC compiler’s race detection which is unsound
Finding Atomic Blocks

- nesC guarantees that atomic blocks have lexical scope
  - This makes the problem easy

- When analyzing code not output by nesC...
  1. Conservatively check for lexical interrupt use
  2. If yes, all is good (this is often the case)
  3. If no, context sensitive analysis is required
     - We don’t do that
Analyzing Asynchronous State 3

• Add a flow edge from the end of each atomic section to the start of each interrupt handler
• Add a flow edge from the end of each interrupt handler back to the end of each atomic section
• Claim: This is a sound model for dataflow analysis of non-racing asynchronous global variables
Analyzing Racing Variables

• Need to know the underlying atomic memory operations
  – Exploit knowledge of compiler and target
    • Easy in some cases
    • E.g. word sized, word aligned scalar variables
    • Difficult for compound variables

• When atomic operations are known, add flow edge to interrupts after each one
  – When not known, treat racing variable as ≺
Variable Classification Results

• For 15 applications totaling 145 Kloc
  – 12 TinyOS apps for AVR-based MicaZ platform
  – 3 other applications for AVR MCUs

• Conservative classification:
  – 57% variables synchronous
  – 34% variables asynchronous and not racing
  – 8% variables racing
Analyzing Volatile Variables

• In C volatiles are opaque
  – “may change in ways unknown to the impl.”
• However: We are not analyzing “C” but rather “C + processor model”
  – We can perform dataflow analysis through some volatile locations
• First – For each volatile location:
  – Is it backed by SRAM or by a device register?
Analyzing Volatile SRAM

- **volatile** used to prevent loads and stores from being added, eliminated, or reordered
- **Claim:** Dataflow analysis can ignore the volatile qualifier for variables in SRAM
  - **Basis:** We have a sound model of all possible mutators
    - No DMA on these architectures
  - Volatiles opaque at the language level but not the system level
Analyzing Device Registers

- Treat registers as SRAM except...
  - Load from a bit in a hardware register may return:
    - Fixed value
    - Last value stored
    - Undeterminable value
  - Device register accesses may also have exploitable side effects – more on this soon
Analyzing Device Registers

- Currently we analyze interrupt control bits
- Plan to pursue this further
  - E.g. to infer predicates on device state
    - “ADC is powered up, enabled, and configured to deliver repeating interrupts”
- Would be nice to have a tool for translating device register specifications into program analysis code
• Embedded codes have “hidden” indirect control flow relations
  – Interrupts:
    1. ADC driver code requests a conversion
    2. ADC completion interrupt fires
  – TinyOS tasks (deferred procedure calls):
    1. Task or interrupt posts a task
    2. TinyOS `main()` loop dispatches the task
• Idea: Represent the hidden state and exploit it to increase analysis precision
• Add models of pending interrupts and tasks to abstract machine state
  – Abstract interrupt and task schedulers decide when to fire these
  – These replace the default models where:
    • Any interrupt may fire any time interrupts are enabled
    • Any task may fire any time scheduler is reached
  – Effect is to prune edges from the flow graph
• Example of causal relations:

- This work is preliminary and ongoing
Analyzer Big Picture

• Integrated into a single analysis pass:
  – Control flow graph discovery
  – Atomic block detection
  – Synchronous / asynchronous / racing analysis
  – Value set analysis
  – Pointer set analysis
  – Dead code detection
  – (Soon) Indirect control flow analysis
Reduction in code size by using analysis results to drive constant propagation, dead code elimination, etc.

Reduction in RAM consumption by using analysis results to drive offline RAM compression

• Each result is geometric mean of improvements over same 15 AVR applications
• Baseline: No concurrency analysis, no analysis of volatile SRAM, address-taken pointer analysis
Improvement: 21% code, 23% data

Improvement: 9% code, 6% data
Lessons

• Integrated analyses a necessary evil
  – Lots of interactions to keep track of ⇒ No fun to design and implement

• Precise analysis of low-level codes requires knowledge of HW and SW platform properties
Conclusion

• High-precision static analysis of MCU codes is possible and useful

• Open source cXprop tool
  – http://www.cs.utah.edu/~coop/reserach/cxprop/