Static Analysis of Embedded C Code

John Regehr
University of Utah

Joint work with Nathan Cooprider

- Relevant features of C code for MCUs
  - Interrupt-driven concurrency
  - Direct hardware access
  - Whole program analysis is feasible
- TinyOS is the specific target
  - But we’ve analyzed many other codes
  - Techniques should generalize to systems with threads and heap – but we haven’t done this

Why Analyze Low-Level C?

- To support program transformations
  - Eliminating type safety checks (SenSys 2007)
    - Reduced CPU overhead from 24% to 5%
    - Reduced code size overhead from 35% to 13%
  - Offline RAM compression (PLDI 2007)
    - Transformation that reduces SRAM usage of TinyOS applications by 22%
- To cheaply discover program facts supporting verification?

Thesis

1. Embedded codes contain significant structure not exploited by 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

- What does a dataflow analyzer for sequential C code buy you?
  - Can analyze local variables
    - Assumption: No pointers across stacks
  - Cannot generally analyze globals

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

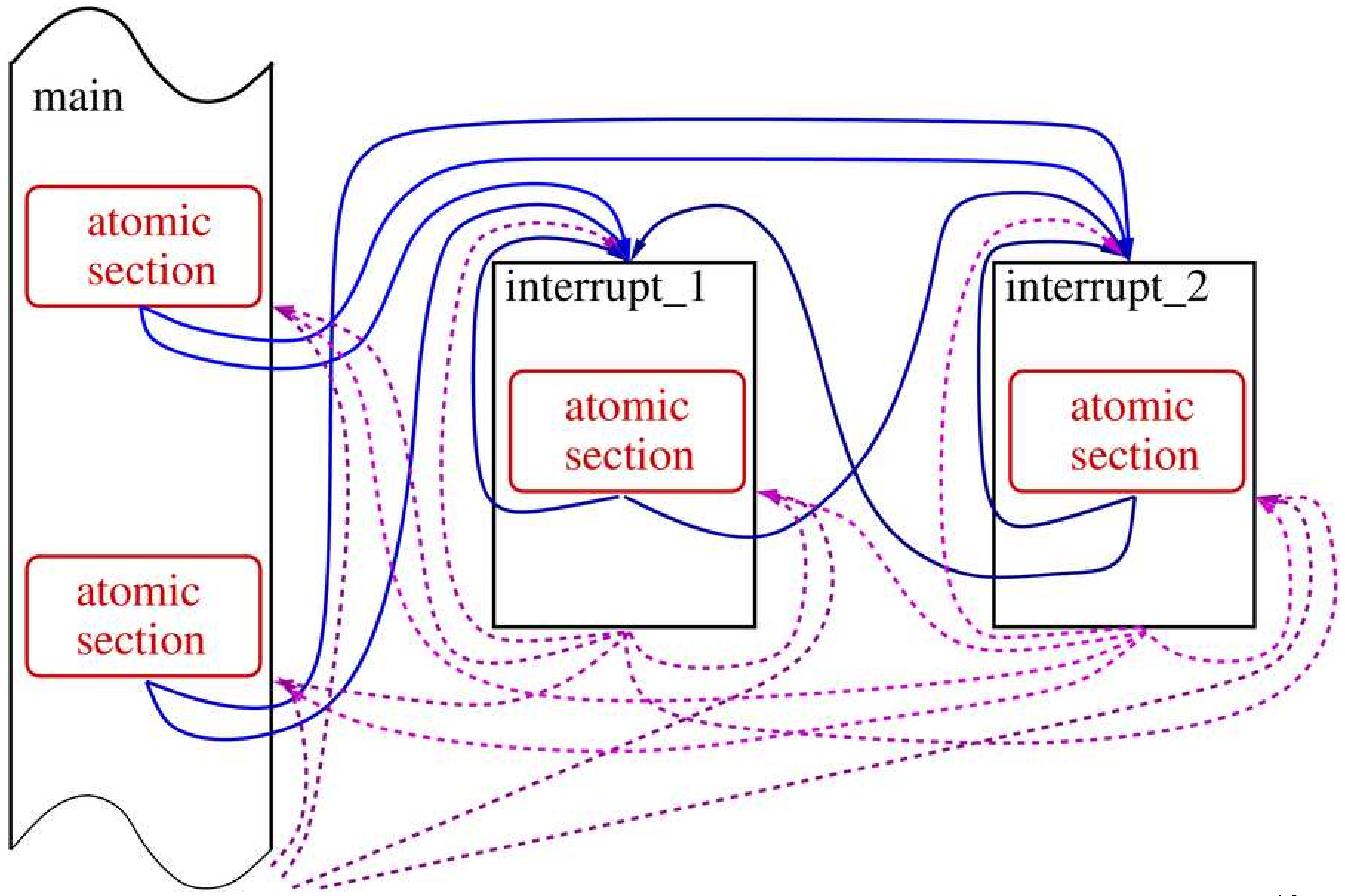
- But how to analyze asynchronous variables?

Analyzing Asynchronous State 1

- Strawman algorithm:
  1. Find program points where interrupts are enabled
  2. Add a flow edge from each such point to the start of each interrupt handler (and back)
- This is overkill: So many extra flow edges $\Rightarrow$ long analysis time
- This is unsound: C statements are not atomic operations

Analyzing Asynchronous State 2

- Better algorithm:
  1. Conservatively find “racing” variables == Asynchronous and touched with interrupts enabled
     - These use homebrew synchronization protocols
     - Don’t try to analyze them
  2. Analyze non-racing variables by adding flow edges from program points that may enable interrupts to interrupt handlers (and back)


main
atomic section
atomic section
interrupt_1
atomic section
interrupt_2
atomic section

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 $\perp$

Variable Classification Results

- For 11 TinyOS 1.x applications totaling
  - 88 Kloc
  - 1352 global variables

- Conservative classification:
  - 56% variables synchronous
  - 37% variables asynchronous and not racing
  - 7% 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

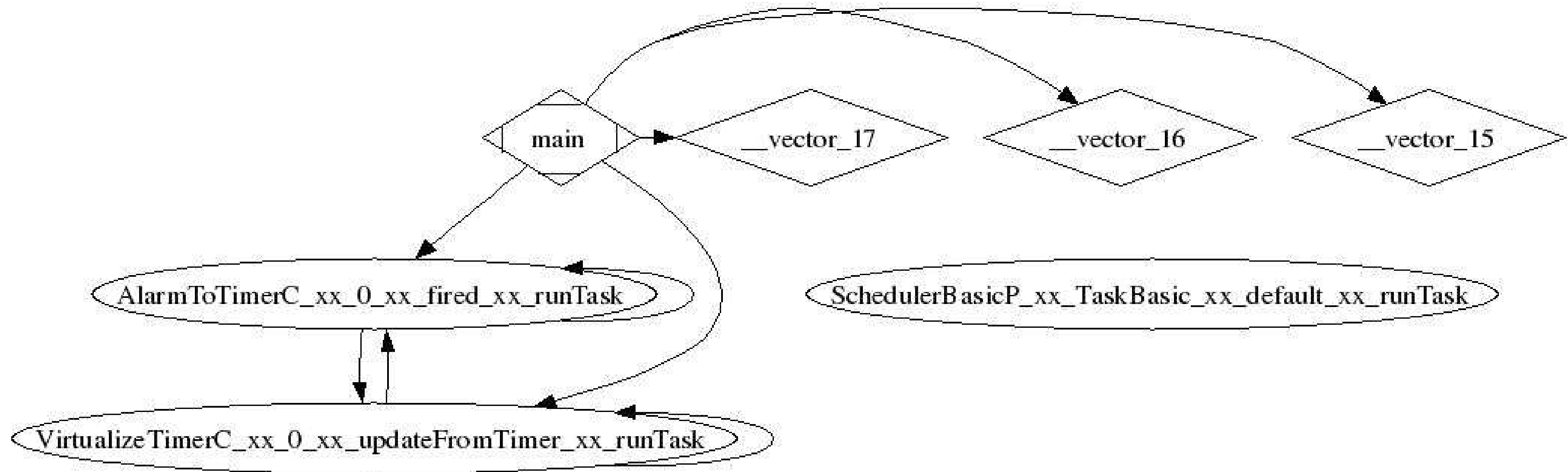
Analyzing Device Registers

- 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 disk
- 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

- These are all integrated into a single analysis pass:
  - Control flow graph discovery
  - Interrupt-enabled analysis
  - Synchronous / asynchronous / race analysis
  - May / must alias analysis
  - Value set analysis (for scalar types)
  - Dead code detection
  - (Soon) Indirect control flow analysis

Lessons

- Integrated analyses a necessary evil
  - Lots of interactions to keep track of $\Rightarrow$ 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/research/cxprop/