Problems in Interrupt-Driven Software

Stack Overflow
Interrupt Overload

Robyn Evelyn
Wei Jiao
Truong Nghiem

Overview of Presentation

- Paper: “Safe and Structured Use of Interrupts in Real-Time and Embedded Software”
  - Interrupt Definitions and Semantics
  - Problems in Interrupt-Driven Software
    - ✓Stack Overflow
    - ✓Interrupt Overload - with additional reference to “Preventing Interrupt Overload”
    - ✓Guidelines for Interrupt Driven Embedded Software
- Paper: “Eliminating Stack Overflow by Abstract Interpretation”
- Paper: “Memory Overflow Protection for Embedded Systems using Runtime Checks, Reuse and Compression”
- Paper: “Multi Task Stack Sharing for Embedded Systems”

INTRODUCTION

Interrupts versus polling

- Reduce latency and overhead of event detection
- Reduce energy consumption
- Relatively non-portable across compilers and hardware platforms
- Prone to software errors which are difficult to detect

DEFINITIONS

Interrupt - Hardware-supported asynchronous transfer of control to an interrupt vector

Interrupt Vector - Dedicated location in memory that specifies address execution jumps to

Interrupt Handler - Code that is reachable from an interrupt vector

Interrupt Controller - Peripheral device that manages interrupts for the processor

Pending - Firing condition met and noticed but interrupt handler has not began to execute

Interrupt Latency - Time from interrupt's firing condition being met and start of execution of interrupt handler

Nested Interrupt - Occurs when one interrupt handler preempts another

Reentrant Interrupt - Multiple invocations of a single interrupt handler are concurrently active

SEMANTICS

- Semantics of interrupts differ across hardware platforms and embedded compilers
  - Execution of instructions
    - Interrupts - typically executed atomically
    - Slow instructions - typically executed non-atomically

      However, not always the case: some architectures use single non-interruptible instructions. Why? To reduce code size

      Note: Such instructions increase latency.

      Tradeoff between latency and code size

  - Amount of state that is saved depends on the processor architecture

    68HC11 (CISC) - all registers saved
    AVR (RISC) - program counter only

  - Support for interrupt handlers depends on the embedded compiler

    Most support interrupt handlers that look like normal functions but with a non-portable *pragma* indicating that code generator create an interrupt *prologue* and *epilogue* for the function.

INTERRUPT-DRIVEN SOFTWARE

A system is *interrupt-driven* when a significant amount of its processing is initiated by interrupts

PROBLEMS ENCOUNTERED

- Avoiding stack overflow
- Meeting real-time deadlines
    - Dealing with interrupt overload

STACK OVERFLOW

- During execution stacks grow and shrink
- If stack becomes larger than memory allocated to it, RAM becomes corrupted and system malfunctions
- How much memory should the programmer allocate for the stack?
    - Overprovisioning: wastes memory that could be used for other purposes
    - Underprovisioning: system is prone to stack overflow
- How does the developer decide on the stack size?
    - *Testing-based* approach - empirical data from simulated or actual runs used as a guide
    - *Analysis-based* approach - some form of counting push, pop and call instructions along different paths; needs to be automatic

STACK OVERFLOW

Analysis vs. Testing approaches to sizing the stack

- goal is to *bound* the stack size by finding a path through the program that produces the worst-case stack behavior

Testing-Based

- Run the system to see how big the stack gets
- System should be tested under heavy, diverse loads
- Real system - Initialize stack memory to known values and see how many get overwritten
- Treats system as a black box
- Can miss paths through the code

Analysis-Based

- Looks at flow of control through system
- Goal is to find the path that pushes the maximum amount of data onto the stack
- Complex - involves main function and interrupt handlers
- Often overestimates maximum stack size - sometimes gives infinite worst-case stack depth
- Much faster than testing
- Possible to produce a guaranteed upper bound on stack depth - *stack-safe* systems

worst depth seen in testing ≤ true worst depth ≤ analytic worst depth

STACK OVERFLOW

- Gaps between worst depth seen in testing and analytic worst depth can be narrowed through hard work
  - ★ Clever tests to include missing paths
  - ★ Run tests for longer
  - ★ Analysis can take causal relationships between interrupt handlers into account

<u>Stack-depth Analysis</u>

Control Flow Graph (CFG) - representation of possible movement of program counter through system's code

- ★ Straightforward - cyclic executive; single control graph
- ★ Use of analyzer - recursion, indirect calls, RTOS
- ★ Simple analyzer - push and pop instructions only
- ★ More sophisticated analyzer - *alloca* instructions

Safe and Structured Use of Interrupts in Real-Time and Embedded Software
&
Preventing Interrupt Overload

INTERRUPT OVERLOAD

- Embedded systems particularly interrupt-driven
  - constant micromanagement of peripherals
  - processors can sleep until interrupt arrives
  - ⊙ Polling - performs well under overload but inefficient during underload

- *Interrupt overload*: Condition where external interrupts signaled frequently enough to cause other activities running on the processor to be starved

  - Case 1: Apollo Guidance Computer (AGC) - first recognizably modern embedded system, used in real-time by astronaut pilots to collect and provide flight information, and to automatically control all of the navigational functions of the Apollo spacecraft. First moon landing - flood of radar data overloaded CPU on Lunar Landing Module

- There is a need to bound the maximum interrupt arrival rates
  - not easy; may require reasoning about complex physical systems

Safe and Structured Use of Interrupts in Real-Time and Embedded Software
&
Preventing Interrupt Overload

Source	*Max. Interrupt Freq. (Hz)*
knife switch bounce	333
loose wire	500
toggle switch bounce	1 000
rocker switch bounce	1 300
serial port @115 kbps	11 500
10 Mbps Ethernet	14 880
CAN bus	15 000
I2C bus	50 000
USB	90 000
100 Mbps Ethernet	148 800
Gigabit Ethernet	1 488 000

Table 1. Potential sources of excessive interrupts for embedded processors. The top part of the table reflects the results of experiments and the bottom part presents numbers that we computed or found in the literature.

Preventing Interrupt Overload

- Switches can generate surprisingly high-frequency events
  - Embedded systems are prone to unforseen "switches"
  - Debounce switch using low-pass filter in hardware or software

- Preventing interrupt overload - stopping processor from handling interrupts when developer-specified conditions are met.
  - filtering using hardware
  - scheduling interrupts in software; involves controlling the interrupt's enable bit (*software scheduler)*

Safe and Structured Use of Interrupts in Real-Time and Embedded Software
&
Preventing Interrupt Overload

- Two types of *software scheduler*
  - *Strict*
  - *Bursty*

  Strict Software Scheduler

  - Enforces *minimum interarrival time* between interrupts or *maximum interrupt frequency*
  - Interrupt prologue modified to clear the interrupt's enable bit. Sets up a one-shot timer to expire one interarrival time in the future; when timer expires the handler re-enables the interrupt
  - Incurs some overhead; number of interrupts handled is doubled

  Bursty Software Scheduler

  - Lower overhead but weaker isolation
  - Disables the interrupt only after a burst of interrupt requests has been observed
  - Requires *maximum burst size* and *maximum arrival rate* for bursts
  - Interrupt prologue modified to increment counter. When counter reaches max burst size, clear interrupt's enable bit
  - Counter is cleared by a periodic timer set to the frequency of burst arrival rate

Safe and Structured Use of Interrupts in Real-Time and Embedded Software
&
Preventing Interrupt Overload

Scheduling multiple interrupt sources

- Strict Scheduler
  - Simply replicate software for each interrupt
- Bursty Scheduler
  - Opportunities for optimization
    - ✓ Use single periodic timer to clear counters for multiple interrupt sources
  - Choose burst size that strikes a balance between overhead and protection from overload

Modeling Interrupt Schedulers

Parameter	*Cost (cycles)*
t_{int}	79
t_{poll}	4
t_{setup}	5
t_{expire}	79
t_{flip}	5
t_{count}	12
t_{clear}	5

Table 2. Overhead constants for the ATmega103L with TinyOS. A cycle is 250 ns.

Safe and Structured Use of Interrupts in Real-Time and Embedded Software
&
Preventing Interrupt Overload

Parameter	*Cost (cycles)*
t_{int}	79
t_{poll}	4
t_{setup}	5
t_{expire}	79
t_{flip}	5
t_{count}	12
t_{clear}	5

Table 2. Overhead constants for the ATmega103L with TinyOS. A cycle is 250 ns.

t_work = worst-case execution time of processing one unit of work
t_arrival = minimum interarrival time
t_int = overhead of taking interrupt as opposed to polling
t_poll = cost of polling
t_setup = time taken to set up the one-shot timer
t_expire = overhead for one-shot or periodic timer to expire
t_count = overhead of incrementing counter
t_clear = cost of clearing counter
t_flip = overhead of setting or clearing interrupt enable flag

Note that t_work is the only one under the control of the programmer

Goal: to compute the WCET (C) and the worst-case inter-arrival time (T), real-time parameters for each interrupt source

Safe and Structured Use of Interrupts in Real-Time and Embedded Software
&
Preventing Interrupt Overload

- Does not address the computation of WCET of generic code, a well-studied problem

Pure Interrupts

- T = 0
- low-priority work is starved
- corresponds to stuck level-triggered interrupt

Pure Polling

- polling driven by timer that expires every t_arrival
- worst-case: work from device must be processed at each expiration
- T = t_arrival
- C = t_expire + t_poll + t_work

Strict Software Scheduler

- model scheduler as pair of tasks; one for interrupt handler, one for timer interrupt
- both tasks have T = t_arrival
- Interrupt handler: C = t_int + t_flip + t_setup + t_work
- Timer: C = t_expire + t_flip

Bursty Software Scehduler

- model as pair of tasks
- N = burst size

Safe and Structured Use of Interrupts in Real-Time and Embedded Software
&
Preventing Interrupt Overload

Bursty Software Scehduler

- model as pair of tasks
- N = burst size
- both tasks have T = t_arrival
- Interrupt handler: C = N*(t_int + t_work + t_count) + t_flip
- Burst of interrupts is modeled as single task arrival
- Timer: C = t_expire + t_clear + t_flip

Basic strategies for avoiding interrupt overload

- Keep interrupt handlers short
- Bound arrival rates of interrupts - may involve studying the interrupting device
- Reduce worst-case arrival rate
- Poll for events rather than using interrupts
  - adds processor overhead even when there are no events to process
  - consider switching between interrupts and polling depending on system load
    - switching will create additional overhead
- Use an interrupt scheduler - hardware or software that limits maximum arrival rate of an interrupt source

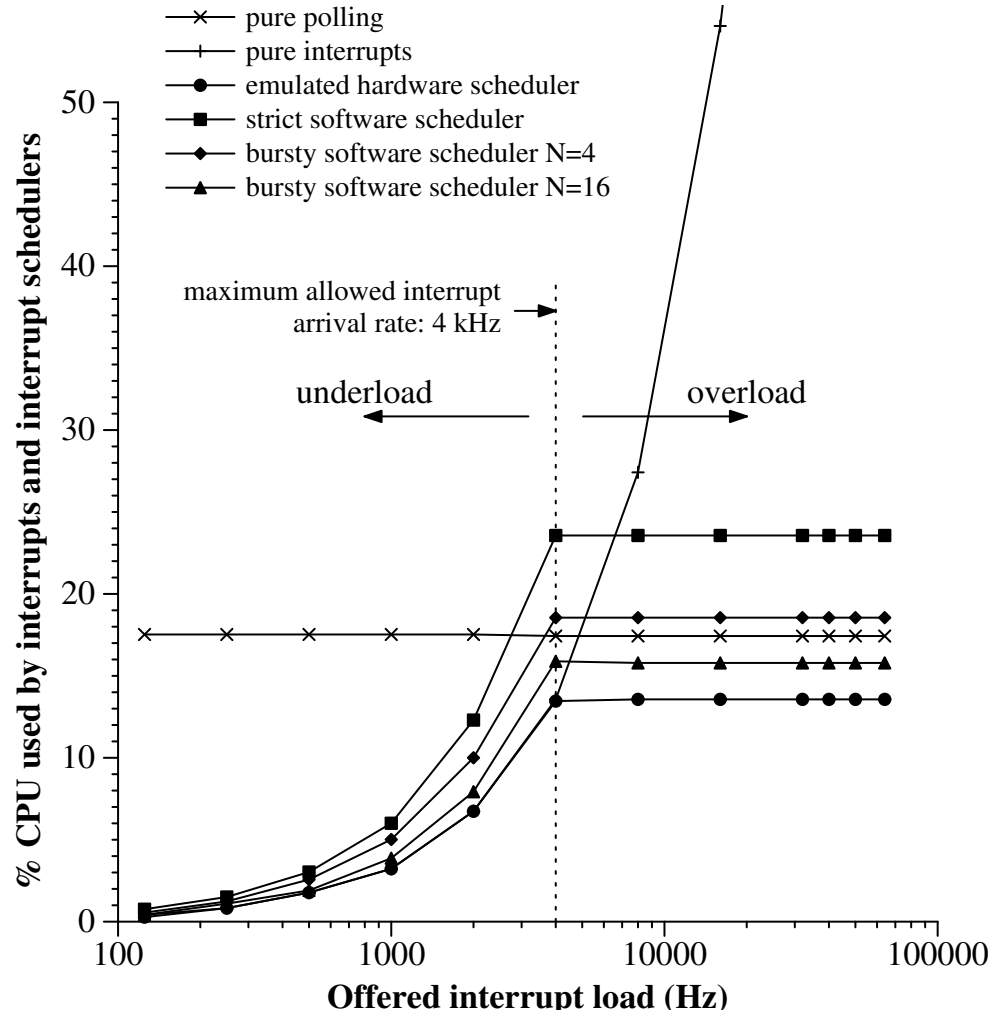

Figure 1. Comparing the performance of different interrupt schedulers when interrupt handlers perform no work

Safe and Structured Use of Interrupts in Real-Time and Embedded Software
&
Preventing Interrupt Overload

General guidelines for Interrupt-Driven Embedded Software

Scheduling

- Scheduling discipline must be specified - preemptive or non-preemptive priority-based scheduling. Include response-time equations
- Never permit two different interrupt handlers to preempt each other
- Reasons for preemption identified. Strive to eliminate useless preemption

Callgraph

- Identify callgraphs for system; one per interrupt, one for main, one for each thread
- Avoid recursive loops

Time Correctness

- Determine maximum arrival rate of each interrupt source.
- Determine deadline for each interrupt and cost of missing a deadline
- Determine WCET for each interrupt. Consider also the WCET of the non-time-sensitive part of interrupt handler
- Determine longest amount of time for which interrupts are disabled (used as *blocking term* in schedulability equations)

Stack Correctness

- Develop a *stack model*

Safe and Structured Use of Interrupts in Real-Time and Embedded Software
&
Preventing Interrupt Overload

General guidelines for Interrupt-Driven Embedded Software

Stack Correctness

- Develop a *stack model;* identify effect of interrupts on each stack
- Determine stack budget - worst-case amount of RAM available for interrupts
- Worst-case stack memory usage for each interrupt
- Overall worst-case stack depth

Concurrency Correctness

- Reachability analysis to determine which data structures are reachable from which interrupt handlers
- Automatic data structures must be unshared
- Protect shared variables