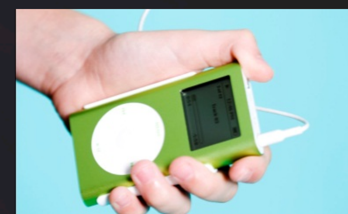
MAKING HIGH-CONFIDENCE SYSTEMS LOW-COST

Alex Dean
Center for Efficient, Scalable and Reliable Computing
Dept. of ECE
North Carolina State University

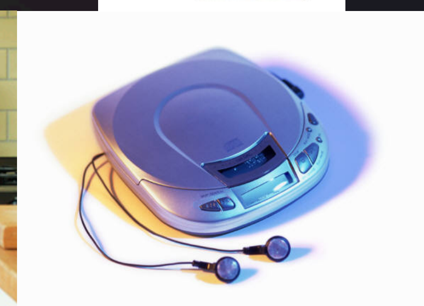
Embedded and Cyber-Physical Systems





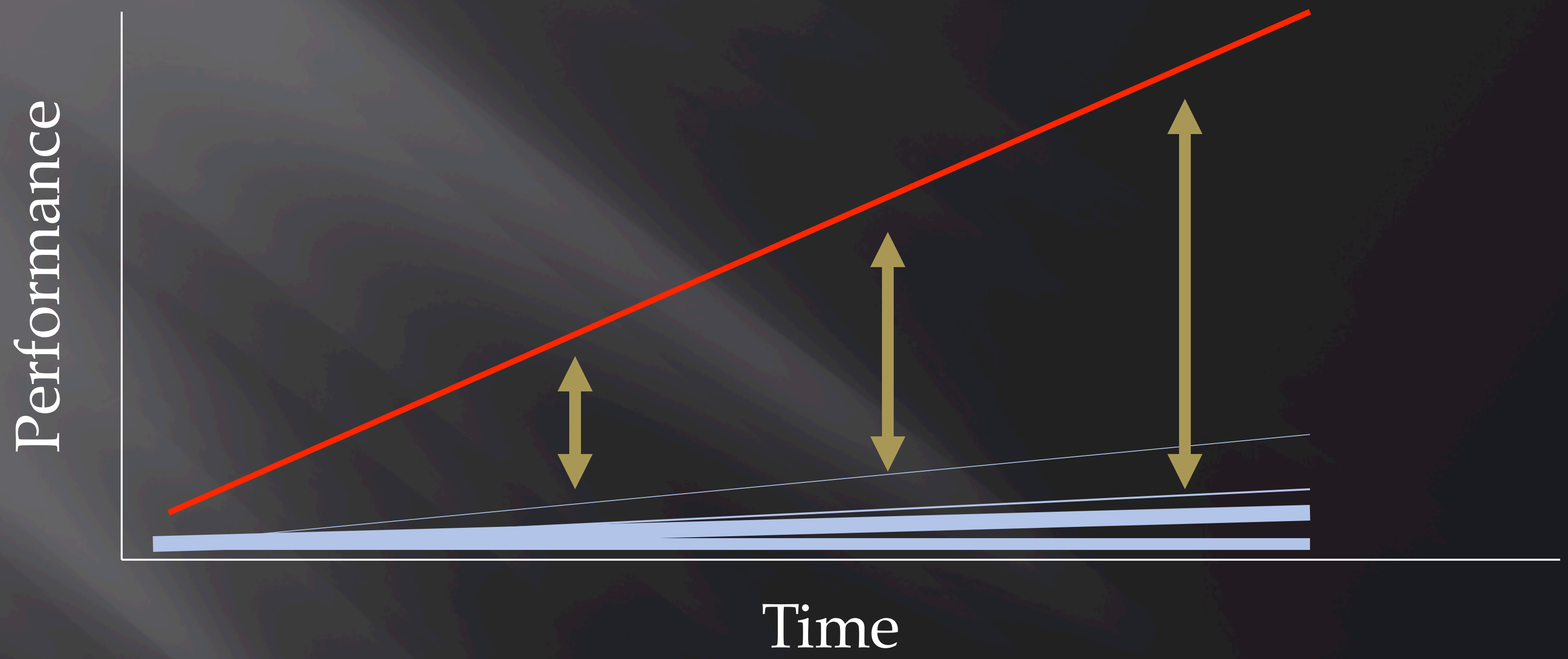








Abstractions May Not Scale



- Growing gap between high-performance CPUs and embedded MCUs
- Abstractions which work at top don't necessarily work at bottom

My Industry Perspective

- Past 10 years doing > 50 in-depth firmware design reviews for embedded systems industry
  - Thermostat, furnace/AC/heat pump control, Jacuzzi control
  - Electric motor control and protection
  - Switching power supplies for telecom, servers, PCs, etc.
    - Buck, boost, inverter, etc.
  - Process control: pressure & flow meters (mag, ultrasonic, etc.), oil/gas metering & mixing, high pressure regulator
  - Synchronous AC transfer (generator, etc.)
  - Water heater controllers, pump controllers

Rigorous Reviews

- Inspect requirements, architecture, detailed design, source code, object code. List potential issues.
- On-site day-long visit for nearly all reviews to discuss issues, screen for risks
- Afterwards, write detailed report to document review

Who is this person? Background

- You haven't sailed the Chesapeake until you've run aground in the Chesapeake
  - I build systems to keep my feet grounded
- Research: How to make embedded systems *fast*, *responsive* and *energy-efficient* by combining techniques from *compilers*, *real-time systems, operating systems*, *computer architecture* and *switching power supplies*
- 3 embedded systems courses
  - Programing MCU and peripherals in C (16-bit MCU)
  - Analyzing and optimizing for speed, energy and responsiveness (16 bit MCU w/RTOS, ARM Cortex-A8 w/ Linux)
  - Performance analysis and optimization of complex embedded systems (ARM Cortex-A8 , Linux)

System Characteristics

- Mostly 8 and 16 bit processors, but some 32 bitters
- Memory size from 4 kB to 256 MB
- CPU speed from 400 kHz to 600 MHz
- Mostly superloop+ISRs, some non-preemptive task schedulers, some RTOSs, a couple with Linux
- Digital electronic parts costs from $2 to $1500
- Various embedded networks (wired and wireless)

Requirements

- Business Requirements
  - Must *sell* - Acceptable price (BOM)
    - Competitive market forces prices down
  - Must *get to market on time* - Acceptable development schedule (NRE)

- Technical Requirements
  - Must *work* - Correct functionality and timing
    - Typically reactive and real-time

Development Effort

- Need to deliver a product on time with given staff
- Risk reduction - go with proven technology rather than novel one
- Development team size = 1, 2 or maybe 3 for large projects
- Staff are experts in domain area, good in implementation, rarely current with academic research
- Won't get it perfect, but want it to work well enough (robustness)

Maintenance Effort

- How do you patch a deeply embedded system?
  - May have no internet connection
  - Very narrow pipe
  - Physically return the device?
  - Send out a technician?

- Complexity of implementing bootloader, and resulting authentication requirements
- Look at all the effort at rooting phones, tablets, game consoles

Cost Constraints

- BOM cost -> System resource constraints
  - Embedded computer
  - Communications and network
- NRE cost -> Schedule constraints
  - Development effort

System Resources

- Embedded computer
  - RAM, ROM
  - Compute cycles
  - Power, energy
  - File system
- Communications
  - Uses Embedded control network
  - Limited Bandwidth, packet size
  - MAC, predictability

Observation: Fragmentation

- Major CPU architectures
  - x86
  - ARMv7
  - Power
- Major MCU architectures
  - PIC
  - 68HC11
  - Coldfire
  - MSP430
  - AVR
  - 8051
  - ARMv3
  - ARM Cortex-M0
  - M16C
  - C6x DSP
  - Blackfin DSP
  - 56Fx DSP

- Standard MCU tools
  - Good C compilers, optimizers, debuggers
- Rare MCU tools
  - Stack depth bounding
  - Static timing analysis
  - Code coverage of tests
- Good news: gcc

Observation: Abstraction Mismatches

- Can't use resource-rich mentality
- Constraints
  - Market pressures -> cost
  - Cost, power, energy -> limitations on memory, speed, size,
  - Developer expertise - generalists rather than specialists
- Mismatches
  - Java, abstraction, processing throughput (same skills don't scale across 10,000x (400 kHz to 4 GHz) performance difference)
  - Threads, timesharing, multiuser, multiprocess
  - Multicore
  - Virtual memory

Practical Design Approaches

- Programming language
- Scheduling approaches
- Mixed-criticality system implementation
- Run-time monitoring
- Power and energy efficiency
- Networking

- *Growing mismatch between deeply embedded computing and everything else*

Programming Language

- Abstration
  - Java and other abstract, portable languages
- Disadvantages
  - Much greater requirements for MHz and memory
  - Abstraction reduces performance and ability to access hardware
  - Real-time requirements hard to meet

Practical Approach

- Programs written mostly in C (a little C++)
  - Good tool infrastructure available for constrained hardware
  - Good language due to proximity to hardware, not much obfuscating abstraction
  - Developer expertise

Scheduling Approaches

- Goal: create a predictably responsive system
- Abstraction
  - Use fully-preemptive prioritized scheduling
- Preemption complications
  - Requires more RAM (stack per thread)
  - Introduces data race conditions
  - Requires more sophisticated development tools

Practical Approach

- Usually no preemptive task scheduling
  - Most common: superloop + ISRs
  - Less common: non-preemptive tasks + ISRs. State machines for long tasks.
  - Least common: fully-preemptive tasks + ISRs (RTOS)

- *Never underestimate the power of an ISR*

Mixed-Criticality Systems

- Goal: Isolate high-criticality processing from rest of system
- Abstractions
  - Time
    - Use fully-preemptive prioritized scheduling
  - Space
    - Use processes with hardware-based memory protection

Practical Approach

- Architect the system carefully, with practical partitioning
- Time
  - Rely on "scheduler" - Use ISRs for most critical operations, then high-priority tasks (if supported)
- Space
  - Practice data hiding, modular program design, etc.
  - Protect critical data with complement or block CRC, verify before use

Run-Time Monitoring

- Abstraction
  - Not covered except by specialists
- Goal: Ensure system is either running correctly or else is disabled
- Typical approach: Watchdog timer
  - Only detects serious timing errors
- Enhance with safety invariants
  - Must be easy to compute and test
  - Detect corrupted variables, missed deadlines

Power and Energy Efficiency

- Why:
  - Limited energy source - battery, supercapacitor
  - Limited power source - energy harvesting (photovoltaics, etc.), 4-20 mA current loop
  - Limited cooling - high temperature, fanless device -> limited power dissipation.
- Strategy
  - Energy: Run fast when needed, sleep when possible
  - Power: Run slow when needed, sleep when possible

Abstracted Approach

- Abstraction: extensive support, standardized platforms
  - Dynamic voltage and frequency scaling with multiple OPPs
  - Multiple independent voltage domains (cost for power converters, signal level translators)
  - Linux standard device drivers, power management callbacks
  - cpufreq interface with governors

Practical Approaches

- Use efficient code (optimize heavily)
- Select good components - low power, sleep modes, etc.
- Mechanisms
  - Manually scale processor speed with clock divider & oscillator
  - Shut off unused peripherals, including network and radio. May need to write own code to do this.
- Optimization and debugging are difficult without good tool support

Networking

- Goals:
  - Predictable performance for small messages (up to 16 bytes of data in packet)
- Challenges:
  - Limited bandwidth, if any
  - Network protocol stacks require independent threads of control. Need to make this work with limited scheduling environment
  - No support for authentication

Example: Automotive Remote Keyless Entry System

- Constraints: size, weight, cost, battery life
- Early 1990s: fixed codes used
  - Major problem with vehicle theft
- Colleagues at United Technologies developed efficient rolling code implementation on 6805 microcontroller
  - Had to argue for the extra nickel to buy an MCU with 64 bytes of RAM (rather than 32), enabling more secure algorithm
- Later MCU makers added hardware support to accelerate these devices, given large market

Summary

- Need to think creatively
- Some abstractions aren't practical for deeply-embedded systems, but still need system to work

Thanks for your attention!

alex_dean@ncsu.edu

http://www.cesr.ncsu.edu/agdean