Low-Level and Embedded Programming (2)

Real-Time and Embedded Systems (M)

Lecture 19

UNIVERSITY
of
GLASGOW

Lecture Outline

- Hardware developments
- Implications on system design
  - Low-level programming
  - Automatic memory management
  - Timing
  - Concurrency


EMBEDDED COMPUTING

Absolutely Positively on Time: What Would It Take?

Edward A. Lee, University of California, Berkeley

For embedded computing to realize its full potential, we must reinvent computer science.

- Considerations for new system architectures

Low-Level and Embedded Programming

- Real time and embedded systems programming differs from conventional desktop applications programming
  - Must respect timing constraints
  - Must interact with environment
  - Often very sensitive to correctness and robust operation
  - Often very sensitive cost, weight, or power consumption

- Implications to consider:
  - Proofs of correctness, schedulability tests, etc.
    - Must consider system implementation issues, not just theory
  - Limited resources available
    - Low level programming environments typical
    - Require high awareness of system issues; interaction with hardware
    - Cannot necessarily depend on “common” language, operating system, or hardware features being present

Yes, but…

- Continued advances in hardware
  - Moore's "law" shows no sign of abating for some years yet
  - Increasing use of system-on-a-chip designs
    - Processor, memory, I/O devices integrated into a single chip package
  - Performance of low-cost embedded hardware increasing rapidly
- Where are corresponding advances in software?
  - Desirable to raise abstraction level
    - Ease program development and increase productivity
    - Modern software engineering techniques
    - High(er) level languages
      - E.g. Real Time Java
  - Simplify proofs of correctness

- How to improve real time & embedded systems implementation?

Evolution of Real Time Systems

- Use increased system performance to enable:
  - Language support for low-level programming
    - Interrupt handling
    - Device access
  - Language support for automatic memory management
    - Real time garbage collection
  - Language support for timing
    - Timed periodic threads
    - Timed statements/timing annotations
  - Language support for concurrency
    - Problems with threads
    - Problems with synchronisation

- Use increased hardware performance to offset reduced software efficiency, gain programmer productivity

Interrupt Handling

- Interrupt handling highly machine/operating system dependent
- Few systems support linking user code into interrupt handlers
  - Ada Real Time Systems annex a notable exception:

```
package Ada.Interrupts is
   type Interrupt_Id is ...;
   type Parameterless_Handler is access protected procedure;

   function Is_Reserved(Interrupt:Interrupt_Id) return Boolean;
   function Is_Attached(Interrupt:Interrupt_Id) return Boolean;

   function Current_Handler(Interrupt:Interrupt_Id) return Parameterless_Handler;

   procedure Attach_Handler(Handler:Parameterless_Handler, Interrupt:Interrupt_Id);
   procedure Detach_Handler(Interrupt:Interrupt_Id);

   ...
end Ada.Interrupts;
```

  - Possible to provide similar standard facilities in other languages
    - Some overhead since must vector through hardware abstraction layer; reasonable and safe to implement on microkernel
    - Could eliminate platform-specific hooks, allow portable code

Device Drivers

- Seen various approaches to low-level device access
  - C-style: simple and expressive, non-portable
  - Ada: verbose, precise specification, portable

- Can language support help?
  - Clear that object-oriented ideas useful for device families:
    - MacOS X I/O Kit – object oriented device drivers using a subset of C++
    - Linux uses object-based approach for many drivers, implemented in C
      - Higher performance, but MacOS X drivers easier to write
  - Ability to cleanly define inheritance and sub-class relationships, timeouts, state machines, and interrupt handlers at language level likely beneficial
    - Given wide range of embedded hardware, might be appropriate to sacrifice some performance for ease of development

Memory Management

- Strong distrust of managed languages, garbage collection, in real time systems community
  - E.g. Real Time Java memory model augmented with non-collected zones, manual memory management

- But: memory management problems abound
  - Memory leaks
  - Unpredictable memory allocation performance
    - Calls to `malloc()` can vary in execution time by several orders of magnitude
  - Memory corruption and buffer overflows

- Can garbage collection and memory protection help?

Garbage Collection

- Traditional algorithms not suitable
  - Triggered at unpredictable times
  - Unpredictable collection delays since move objects to avoid fragmentation
- Real time garbage collection still an active research area
  - Two basic approaches:
    - Work based: every request to allocate an object or assign an object reference does some garbage collection; amortise collection cost with allocation cost
    - Time based: schedule an incremental collector as a periodic task
      - Easier to prove correctness
      - More predictable behaviour

D. Frampton, D. F. Bacon, P. Cheng, and D. Grove, "Generational Real-Time Garbage Collection: A Three-Part Invention for Young Objects", Proceedings 21st European Conference on Object Oriented Programming, Berlin, Germany, July 2007.

  - Obtain timing guarantees only by limiting amount of garbage that can be collected in a given interval
    - Implication: user must indicate maximum memory consumption and allocation rate, to determine cost of the garbage collector
    - Workable solutions exist for many periodic applications; same issue as certain scheduling algorithms placing constraints on application design

Memory Protection

- Traditional memory protection unpredictable ⇒ problematic
  - Slows context switch and system call times
  - Requires illegal access traps and handler
    - Unpredictable
    - Difficult to implement error recovery

- Can guarantee safety without hardware protection:
  - Strongly typed language, checked array bounds, no pointer arithmetic
    - Closer to Java than to C
    - E.g. Singularity from Microsoft Research
      - Majority of system written in extended C# and .Net, small microkernel in C++
      - http://research.microsoft.com/os/singularity/
  - Much verification done at compile time; reduces run-time unpredictability
    - Higher overhead than current systems, but not excessive

Timing Annotations

- How to ensure predictable timing?
  - Extensive scheduler theory, proofs of schedulability
  - Introduce abstractions for timed threads into the language
    - E.g. Real Time Java
  - Add timing annotations to language, let compiler determine schedulability
    - Compiler *much* better at counting cycles than a human, due to complex processor architectures
    - Likely feasible to estimate worst-case execution time for many embedded codes; compare with task timing annotations
      - Computationally hard in general due to loops, etc.
      - Equivalent to halting problem for arbitrary code
      - Real systems often much more constrained: hard real time systems *required* to be provably correct
    - Helps debugging if not proving correctness

Timing Annotations

- To what extent possible to annotate timing requirements?
  - Properties of periodic tasks straight forward
  - Aperiodic/sporadic tasks harder, but often meaningful statistics
  - But what about low-level behaviour?
    - Annotate that an expression should take no more than *x* milliseconds
    - System call/library function timing
- What are hidden timing behaviours of system?
  - Scheduler and system call overhead
  - **`malloc()`**/**`free()`**, garbage collection
  - Cache, memory hierarchy, memory protection
  - Speculative execution, pipelining, super-scalar and out-of-order execution
- Programmers cannot count cycles; yet many still program as if it were possible – need compiler help

Support for Concurrency

- Concurrency increasingly important
  - Trends in microprocessor design
  - Asynchronous interactions with outside world

  - Threads and synchronisation primitives problematic
    - Low level model
    - Easy to make mistakes
    - Hard to reason about performance/correctness

  - Are there alternative architectures which avoid these issues?
    - Implicit concurrency; execution models which hide complexity
    - Functional and/or message passing algorithms
      - e.g. Ericsson AXD301 160 Gbps ATM switch has claimed 99.9999999% uptime and is (mostly) written in the Erlang functional programming language

Joe Armstrong, "Making reliable distributed systems in the presence of software errors", PhD thesis, Kungliga Tekniska Höskolan, Stockholm, December 2003, http://www.sics.se/~joe/thesis/armstrong_thesis_2003.pdf

Reliability Through Clarity

- State and requirements hidden in existing code
  - Need to infer high-level goals from low-level implementation
- Yet Moore's law continues
  - Performance increasing for fixed price point, power consumption
- Better languages/libraries would allow programmers to express high-level goals, system to check implementation meets them
  - Requires paradigm shift away from current implementation strategies
  - Beginning to happen with Real Time Java; realisation that platforms both powerful and cost effective

Tim Sweeney, "The Next Mainstream Programming Language", Keynote at the 33rd Symposium on Principles of Programming Languages, Charleston, Janary 2006. http://www.cs.princeton.edu/~dpw/popl/06/Tim-POPL.ppt

Questions or Discussion?

Summary

- Development in hardware
- Implications on system design
  - Low-level programming
  - Automatic memory management
  - Timing
  - Concurrency
- Considerations for new system architectures

Further reading: Gay *et al*, “The nesC Language: A Holistic Approach to Networked Embedded Systems”, Proc. PLDI’03.