Goals for lecture

• Justify treating real-time design problem as optimization problem
• Example problem to illustrate specification and design
• Tractable algorithm design (NP-completeness in a nutshell)
• Detail on design representations
• Sensor network motivations
• NesC overview

Optimization

Thinking of a design problem in terms of optimization gives design team members objective criterion by which to evaluate the impact of a design change on quality.

• Still need to do a lot of hacking
• Know whether its taking you in a good direction

Example problem

• Richland, Washington’s Hanford Reservation plutonium finishing facility
• July 1988 facility’s last reactor, Reactor N, put into cold standby due the nation’s surplus of plutonium
• Was used for processing weapons-grade fissile material

The value of formality: Optimization and costs

• The design of a real-time system is fundamentally a cost optimization problem
• Minimize costs under constraints while meeting functionality requirements
  – Slight abuse of notation here, functionality requirements are actually just constraints
• Why view problem in this manner?
• Without having a concrete definition of the problem
  – How is one to know if an answer is correct?
  – More subtly, how is one to know if an answer is optimal?

Simple example

• Ensure that a wireless data display 300 m away from a temperature sensor always displays the correct temperature with a lag of, at most, 100 ms.
• Wireless broadcasts reach 100 m with high probability and 200 m with very low probability.
• There are two, evenly distributed, rebroadcast nodes between the sensor and the data display.
• Functional requirements?
• Constraints?
• Costs?

Example problem

• Currently holds 11.0 metric tons of plutonium-239 and 0.6 metric tons of uranium-235
  – The two fissile materials most commonly used in nuclear weapons
• Even without refining, a small quantity of either would convert conventional explosives into weapons capable of causing long-term damage far beyond their blast radii
• Ongoing provisions for security required

Homework index

1 Hanford security network design . . . . . . . . 12
2 Reading assignment (18 January) . . . . . . . . 54
Example problem

- Build perimeter security network
- Functional requirements?
- Constraints?
- Costs?

Example tasks

- Sense audio
- Compress it
- Determine whether it is unusual
- Sense, compress, and stream video
- Analyze information from region to determine most promising messages to forward, given network contention

Example constraints

- Data rate
- Dependencies between tasks
- Price
- Lifetime of battery-powered devices
- Etc.

Hanford security network design

- By 18 January, working with your lab partner, provide
  - A paragraph formalizing the real-time system design goals
  - A paragraph giving an overview of the design you propose
- Keep it within a page. We want you thinking about this and learning but you should focus on the lab assignment.
- Have questions? Do research. The Hanford Reservation is real.
  - Post to the newsgroup if you get stuck.

Lab one

- Subversion working for everybody?
- Access to mailing list?
- Anybody stuck on development?

NP-completeness

- Scheduling is central to real-time systems design and research
- Tractable algorithm design is central to scheduling
- Many (but not all) interesting and useful scheduling problems are NP-complete
- We need to understand what this means, at least at a high level

Recall that sorting may be done in $O(n \lg n)$ time

DFS $\in O(\langle V \rangle + |E|)$, BFS $\in O(\langle V \rangle)$, Topological sort $\in O(\langle V \rangle + |E|)$

There also exist exponential-time algorithms: $\Omega(2^{|V|})$, $O(2^n)$, $O(3^n)$
NP-completeness

For $t(n) = 2^n$ seconds

- $t(1) = 2$ seconds
- $t(10) = 17$ minutes
- $t(20) = 12$ days
- $t(50) = 35,702,052$ years
- $t(100) = 40,196,936,841,331,500,000,000$ years

NP-completeness

- What is NP? Nondeterministic polynomial time.
- A computer that can simultaneously follow multiple paths in a solution space exploration tree is nondeterministic. Such a computer can solve NP problems in polynomial time.
- Nobody has been able to prove either
  
  $P \neq NP$

  or

  $P = NP$

Basic complexity classes

- $P$ solvable in polynomial time by a computer (Turing Machine)
- $NP$ solvable in polynomial time by a nondeterministic computer
- $NP$-complete converted to other $NP$-complete problems in polynomial time

How to deal with hard problems

- What should you do when you encounter an apparently hard problem?
- Is it in $NP$-complete?
- If not, solve it
- If so, then what?

  Determine whether all encountered problem instances are constrained.

  Wonderful when it works.

NP-completeness

- There is a class of problems, $NP$-complete, for which nobody has found polynomial time solutions
- It is possible to convert between these problems in polynomial time
- Thus, if it is possible to solve any problem in $NP$-complete in polynomial time, all can be solved in polynomial time
- Unproven conjecture: $NP \neq P$

If we define $NP$-complete to be a set of problems in $NP$ for which any problem's instance may be converted to an instance of another problem in $NP$-complete in polynomial time, then

$P \subseteq NP \Rightarrow NP$-complete $\cap P = \emptyset$

Hard ($NP$-complete) scheduling problems

- Uniprocessor scheduling with hard deadlines and release times
- Uniprocessor scheduling to minimize tardy tasks
- Multiprocessor scheduling
  - Easy if all tasks are identical
- Multiprocessor precedence constrained scheduling
- Multiprocessor preemptive scheduling
- etc.

One example

Terminology

- Book’s terminology fine, others also exist
- Different groups — different terminology
- Not confusing, terse definitions provided
- Book on jobs, tasks: Jobs discrete, tasks groups of related jobs
- Other sources: Tasks discrete, hierarchical

Additional terminology

- Or vs. And data dependencies
- Conditionals
  - Doesn’t help hard real-time unless perfect path correlation
  - Can help soft real-time

Terminology

- Scheduling, allocation, and assignment
- Scheduling central but not only thing
- Book treats scheduling as combination of scheduling and assignment
- More fine-grained definitions exist

Substantial quirks

1. Every processor is assigned to at most one job at any time
   - O.K.
2. Every job is assigned at most one processor at any time
   - Broken
3. No job scheduled before its release time
   - O.K., but the whole notion of absolute release times is broken for some useful classes of real-time systems.
4. Etc.

Design representations

- Introduction
- Software oriented
  - ANSI-C
  - SystemC
  - Other SW language-based, e.g., Ada
- Hardware oriented
- Graph based
- Resource description

Specification language requirements

- Specify constraints on design
- Indicate system-level building blocks
- To allow flexibility in compilation/synthesis, must be abstract
  - Specify implementation details only when necessary (e.g., HW/SW)
  - Concentrate on requirements, not implementation
  - Make few assumptions about platform

ANSI-C advantages

- Huge code base
- Many experienced programmers
- Efficient means of SW implementation
- Good compilers for many SW processors
ANSI-C disadvantages

- Little implementation flexibility
  - Strongly SW oriented
  - Makes many assumptions about platform
- Little (volatile)/no built-in support for synchronization
  - Especially fine-scale HW synchronization
- Doesn’t directly support specification of timing constraints

Other SW language-based

- Numerous competitors
- Numerous languages
  - ANSI-C, C++, and Java are most popular starting points
- In the end, few can survive
- SystemC has broad support

SystemC

Advantages
- Support from big players
  - Synopsys, Cadence, ARM, Red Hat, Ericsson, Fujitsu, Infineon Technologies AG, Sony Corp., STMicroelectronics, and Texas Instruments
- Familiar for SW engineers

Disadvantages
- Extension of SW language
  - Not designed for HW from the start
- Compiler available for limited number of SW processors
  - New

Design representations

- Software oriented
- Hardware oriented
  - VHDL
  - Verilog
  - Esterel
- Graph based
- Resource description

VHDL

Advantages
- Supports abstract data types
- System-level modeling supported
- Better support for test harness design

Disadvantages
- Requires extensions to easily operate at the gate-level
- Difficult to learn
- Slow to code

Verilog

Advantages
- Easy to learn
- Easy for small designs

Disadvantages
- Not designed to handle large designs
- Not designed for system-level

Verilog vs. VHDL

- March 1995, Synopsys Users Group meeting
- Create a gate netlist for the fastest fully synchronous loadable 9-bit increment-by-3 decrement-by-5 up/down counter that generated even parity, carry and borrow
- 5 / 9 Verilog users completed
- 0 / 5 VHDL users competed

Does this mean that Verilog is better?

Maybe, but maybe it only means that Verilog is easier to use for simple designs.

Esterel

- Easily allows synchronization among parallel tasks
- Works at a high level of abstraction
  - Doesn’t require explicit enumeration of all states and transitions
- Recently extended for specifying datapaths and flexible clocking schemes
- Amenable to theorem proving
- Translation to RTL or C possible
- Commercialized by Esterel Technologies
Design representations

- Software oriented
- Hardware oriented
- Graph based
  - Dataflow graph (DFG)
  - Synchronous dataflow graph (SDFG)
  - Control flow graph (CFG)
  - Control dataflow graph (CDFG)
  - Finite state machine (FSM)
  - Petri net
  - Periodic vs. aperiodic
  - Real-time vs. best effort
  - Discrete vs. continuous timing
  - Example from research
- Resource description

Dataflow graph (DFG)

- Nodes are tasks
- Edges are data dependencies
- Edges have communication quantities
- Used for digital signal processing (DSP)
- Often acyclic when real-time
- Can be cyclic when best-effort

Synchronous dataflow graph (SDFG)

- Nodes are tasks
- Supports conditionals, loops
- No communication quantities
- SW background
- Often cyclic

Control flow graph (CFG)

- Supports conditionals, loops
- Supports communication quantities
- Used by some high-level synthesis algorithms

Control dataflow graph (CDFG)

- Supports conditionals, loops
- Supports communication quantities
- Used by some high-level synthesis algorithms

Finite state machine (FSM)

- Normally used at lower levels
- Difficult to represent independent behavior
  - State explosion
- No built-in representation for data flow
  - Extensions have been proposed
- Extensions represent SW, e.g., co-design finite state machines (CFSMs)

Petri net

- Graph composed of places, transitions, and arcs
- Tokens are produced and consumed
- Useful model for asynchronous and stochastic processes
- Places can have priorities
- Not well-suited for representing dataflow systems
- Timing analysis quite difficult
- Large flat graphs difficult to understand
- Real-time use: Specification and formal timing verification
M/D/3/2: Markov arrival, deterministic service delay,

From A. Zimmermann’s token game demonstration.

NesC

Events

- Provided by interface
- Used to signal command completion
- Interrupt tasks
- Require concurrency control (atomic blocks)

NesC

- View as ANSI C with additional layer
- Specify interfaces between components
- Centers on commands and events
- Commands
  - Provided by interface, do things
  - Non-blocking, split-phase (response from events)
  - Call down
  - E.g., transmit data

NesC

- Tasks: Interrupted only by events, no normal preemption
- Asynchronous code: can be reached by interrupt handlers
- Synchronous code: can be reached only from tasks
- Not the only option

Summary

- Justify treating real-time design problem as optimization problem
- Example problem to illustrate specification and design
- Tractable algorithm design (NP-completeness in a nutshell)
- Detail on design representations
- Sensor network motivations
- NesC overview

Reading assignment (18 January)

  - Chapter 1
  - Chapter A5: Sequencing and scheduling
  - Chapter 3
  - Chapter 4