Reliable embedded system design and synthesis

Conventional software testing

- Implement and test
  Number of tests bounded but number of inputs huge
  Imperfect coverage

Model checking

- Use finite state system representation
  Use exhaustive state space exploration to guarantee desired properties hold for all possible paths
  Guarantees properties
  Difficulty with variables that can take on many values
  Symbolic techniques can improve this
  Difficulty with large number of processes

Overcoming barriers to use

- Automatic abstraction techniques permitting use on more complex systems
  Difficult problem
  Target moderate-complexity systems where reliability is important
    Medical devices
    Transportation devices
  Electronic commerce applications
  Give users a high-level language that is actually easier to use than their current language, and provide a path to a language used in existing model checkers

Cross-talk

- Shielding
- Bus encoding

Particle impact

- Temporal redundancy
- Structural redundancy
- Voltage control
Reliable embedded system design and synthesis

Realtime systems

Scheduling

Homework

Algorithm correctness

Appropriate responses to transient faults

Appropriate responses to permanent faults

Random background offset charge

Improvements to fabrication
Temporal redundancy
Structural redundancy

Temperature-induced timing faults

Preemptive throttling
Global planning

Checkpointing: a tool for robustness in the presence of transient faults

Periodically store system state
On fault detection, roll back to known-good state
Should system-wide or incremental, as-needed restores be used?
When should checkpoints be taken?

Electromigration

Reduce temperature
Reduce current
Spatial redundancy

Example lifetime failure aware synthesis flow

Use temperature reduction and spatial redundancy to increase system MTTF
System MTTF: the expected amount of time an MPSoC will operate, possibly in the presence of component faults, before its performance drops below some designer-specified constraint or it is no longer able to meet it functionality requirements

Manufacturing defects

Spatial redundancy

Motivating example for reliability optimization

Solution I

Solution II

Reliability optimization flow
Accurate reliability models
Efficient system-level reliability models
Efficient fault detection and recovery solutions
Optimization

Many reliability techniques attempt to deal with arbitrary fault processes.
However, the properties of the fault process most significant for a particular application may be important.
Considering them can allow more efficient and reliable designs.

Task arrival times can be predicted.
Static (compile-time) analysis possible.
Allows good resource usage (low processor idle time proportions).
Sometimes designers shoehorn dynamic problems into static formulations allowing a good solution to the wrong problem.

Task arrival times unpredictable.
Static (compile-time) analysis possible only for simple cases.
Even then, the portion of required processor utilization efficiency goes to 0.693.
In many real systems, this is very difficult to apply in reality (more on this later).
Use the right tools but don’t over-simplify, e.g.,

We assume, without loss of generality, that all tasks are independent.

If you do this people will make jokes about you.

More slack in implementation
Timing may be suboptimal without being incorrect
Problem formulation can be much more complicated than hard real-time
Two common (and one uncommon) methods of dealing with non-trivial soft real-time system requirements
Set somewhat loose hard timing constraints
Informal design and testing
Formulate as optimization problem

Each task (or group of tasks) executes repeatedly with a particular period.
Allows some nice static analysis techniques to be used.
Matches characteristics of many real problems...
... and has little or no relationship with many others that designers try to pretend are periodic.
Periodic → Single-rate

- One period in the system.
- Simple.
- Inflexible.
- This is how a lot of wireless sensor networks are implemented.

Periodic → Multirate

- Multiple periods.
- Can use notion of circular time to simplify static (compile-time) schedule analysis E. L. Lawler and D. E. Wood.
- Co-prime periods leads to analysis problems.

Periodic → Other

- It is possible to have tasks with deadlines less than, equal to, or greater than their periods.
- Results in multi-phase, circular-time schedules with multiple concurrent task instances.
- If you ever need to deal with one of these, see me (take my code).
  This class of scheduler is nasty to code.

Definitions

- Task
  - Processor
  - Graph representations
  - Deadline violation
  - Cost functions

Processor

- Processors execute tasks
- Distributed systems: contain multiple processors
  - Inter-processor communication has impact on system performance
  - Communication is challenging to analyze
- One processor type: Homogeneous system
- Multiple processor types: Heterogeneous system

Task/processor relationship

WC exec time (s)

<table>
<thead>
<tr>
<th>WC exec time (s)</th>
<th>Tooth</th>
<th>Road</th>
<th>FIR</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tooth</td>
<td>1.7E-6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Road</td>
<td>35E-9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIR</td>
<td>4.1E-6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix</td>
<td>310E-3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IBM PowerPC 405 GP 266 MHz
IDT79RC32364 100 MHz
Imaginationchip 40 MHz

Relationship between tasks, processors, and costs

E.g., power consumption or worst-case execution time
Central areas of real-time study

- Allocation, assignment and scheduling
  Operating systems and scheduling
  Distributed systems and scheduling
  Scheduling is at the core of real-time systems study

Allocation, assignment, and scheduling

- In order to efficiently and (when possible) optimally minimize
  - Price, power consumption, soft deadline violations
  - Providing guarantees whenever possible
  For all the different classes of real-time problem classes
  This is what I did for a Ph.D.

Operating systems and scheduling

How does one best design operating systems to
Support sufficient detail in workload specification to allow good control, e.g., over scheduling, without increasing design error rate
Design operating system schedulers to support real-time constraints?
Support predictable costs for task and OS service execution

Distributed systems and scheduling

How does one best dynamically control
The assignment of tasks to processing nodes...
... and their schedules
for systems in which computation nodes may be separated by vast distances such that
Task deadline violations are bounded (when possible)...
... and minimized when no bounds are possible
This is part of what Professor Dinda did for a Ph.D.

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?
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.

Know whether your design changes are taking you in a good direction.

**Optimization**

Given a set of tasks, a cost function, and a set of resources, decide the exact time each task will execute on each resource.

**Problem definition**

- **Discrete vs. continuous timing**
  - System-level: Continuous
    - Operations are not small integer multiples of the clock cycle
  - High-level: Discrete
    - Operations are small integer multiples of the clock cycle
  - Implications:
    - System-level scheduling is more complicated...
    - However, high-level also very difficult.
    - Can we solve this by quantizing time? Why or why not?

- **Real-time – Best effort**
  - Why make decisions about system implementation statically?
    - Allows easy timing analysis, hard real-time guarantees
    - If a system doesn’t have hard real-time deadlines, resources can be more efficiently used by making late, dynamic decisions
    - Can combine real-time and best-effort portions within the same specification
    - Reserve time slots
      - Take advantage of slack when tasks complete sooner than their worst-case finish times

- **Hard deadline – Soft deadline**
  - Tasks may have hard or soft deadlines
    - Hard deadline
      - Task must finish by given time or schedule invalid
    - Soft deadline
      - If task finishes after given time, schedule cost increased

- **Unconstrained – Constrained resources**
  - Unconstrained resources
    - Additional resources may be used at will
  - Constrained resources
    - Limited number of devices may be used to execute tasks

**Discrete vs. continuous timing**

- **System-level: Continuous**
  - Operations are not small integer multiples of the clock cycle
- **High-level: Discrete**
  - Operations are small integer multiples of the clock cycle

**Implications:**

- System-level scheduling is more complicated...
- However, high-level also very difficult.
- Can we solve this by quantizing time? Why or why not?

**Real-time – Best effort**

- Why make decisions about system implementation statically?
  - Allows easy timing analysis, hard real-time guarantees
  - If a system doesn’t have hard real-time deadlines, resources can be more efficiently used by making late, dynamic decisions
  - Can combine real-time and best-effort portions within the same specification
    - Reserve time slots
      - Take advantage of slack when tasks complete sooner than their worst-case finish times
Uni-processor
- All tasks execute on the same resource
- This can still be somewhat challenging
  However, sometimes in P

Multi-processor
- There are multiple resources to which tasks may be scheduled
  Usually N^P-complete

Homogeneous processors
- All processors are the same type

Heterogeneous processors
- There are different types of processors
  Usually N^P-complete

Free – Expensive communication
- Free communication
  Data transmission between resources has no time cost
- Expensive communication
  Data transmission takes time
  Increases problem complexity
  Generation of schedules for communication resources necessary
  Usually N^P-complete

Homogeneous – Heterogeneous tasks
- Homogeneous tasks: All tasks are identical
- Heterogeneous tasks: Tasks differ

One-shot – Periodic
- One-shot: Assume that the task set executes once
  Periodic: Ensure that the task set can repeatedly execute at some period

Single rate – Multirate
- Single rate: All tasks have the same period
- Multirate: Different tasks have different periods
  Complicates scheduling
  Can copy out to the least common multiple of the periods (hyperperiod)

Periodic graphs
- System hyperperiod = 60 ms
  2 copies
- System hyperperiod = 60 ms
  2 copies
- System hyperperiod = 60 ms
  3 copies
- System hyperperiod = 60 ms
  3 copies
Aperiodic/sporadic graphs

- No precise periods imposed on task execution
- Useful for representing reactive systems
- Difficult to guarantee hard deadlines in such systems
- Possible if minimum inter-arrival time known

Aperiodic to periodic

Can design periodic specifications that meet requirements posed by aperiodic/sporadic specifications

- Some resources will be wasted

Example:

- At most one aperiodic task can arrive every 50 ms
- It must complete execution within 100 ms of its arrival time

Non-preemptive – Preemptive

- Non-preemptive: Tasks must run to completion
- Ideal preemptive: Tasks can be interrupted without cost
- Non-ideal preemptive: Tasks can be interrupted with cost

Off-line – On-line

Off-line
- Schedule generated before system execution
- Stored, e.g., in dispatch table. For later use
- Allows strong design/synthesis/compile-time guarantees to be made
- Not well-suited to strongly reactive systems

On-line
- Scheduling decisions made during the execution of the system
- More difficult to analyze than off-line
- Making hard deadline guarantees requires high idle time
- No known guarantee for some problem types
- Well-suited to reactive systems

Hardware-software co-synthesis scheduling

Automatic allocation, assignment, and scheduling of system-level specification to hardware and software

Scheduling problem is hard

- Hard and soft deadlines
- Constrained resources, but resources unknown (cost functions)
- Multi-processor
- Strongly heterogeneous processors and tasks
- No linear relationship between the execution times of a tasks on processors

Expensive communication

- Complicated set of communication resources
- Precedence constraints
- Periodic
- Multirate
- Strong interaction between \(NP\)-complete allocation-assignment and \(NP\)-complete scheduling problems
- Will revisit problem later in course if time permits
Difficult real-world scheduling problem
- Not multirate
- Discrete notion of time
- Generally less heterogeneity among resources and tasks
What scheduling algorithms should be used for these problems?

## Scheduling methods

### Clock-driven scheduling

- Clock-driven: Pre-schedule, repeat schedule
- Music box:
  - Periodic
  - Multi-rate
  - Heterogeneous
  - Off-line
  - Clock-driven

### Weighted round-robin

- MILP
- Force-directed
- Frame-based
- PGA
- RMS

**Weighted round-robin:** Time-sliced with variable time slots

### Priority-driven

- Impose linear order based on priority metric
  - Earliest start time (EST)
  - Latest start time
    - Danger! LST also stands for least slack time.
  - Shortest execution time first (SETF)
  - Longest execution time first (LETF)
  - Slack (LFT - EFT)

### List scheduling

- Assigns priorities to nodes
  - Sequentially schedules them in order of priority
  - Usually very fast
  - Can be high-quality
  - Prioritization metric is important
Prioritization

As soon as possible (ASAP)
As late as possible (ALAP)
Slack-based
Dynamic slack-based
Multiple considerations

As late as possible (ALAP)

From deadlines, topological sort on the precedence graph
Propagate execution times, taking the min at reconverging paths
Consider precedence-constraint satisfied tasks
Schedule in order of increasing latest start time (LST)

Multiple considerations

Nothing prevents multiple prioritization methods from being used
Try one method, if it fails to produce an acceptable schedule, reschedule with another method

EDF, LST optimality

EDF optimal if zero-cost preemption, uniprocessor assumed
  Why?
  What happens when preemption has cost?
Same is true for slack-based list scheduling in absence of preemption cost

Effective release times

Ignore the book on this
  Considers simplified, uniprocessor, case
Use EFT, LFT computation
Example?

Breaking EDF, LST optimality

Non-zero preemption cost
Multiprocessor
Why?
Multi-rate tricks

Contract deadline
Usualy safe
Contract period
Sometimes safe
Consequences?

MILP scheduling

\[ P \text{ the set of tasks} \]
\[ t_{\text{max}} \text{ maximum time} \]
\[ \text{start}(p, t) = 1 \text{ if task } p \text{ starts at time } t, 0 \text{ otherwise} \]
\[ D \text{ the set of execution delays} \]
\[ E \text{ the set of precedence constraints} \]
\[ t_{\text{start}}(p) = \sum_{t=0}^{t_{\text{max}}} t \cdot \text{start}(p, t) \text{ the start time of } p \]

Force directed scheduling

Calculate EST and LST of each node
Determine the force on each vertex at each time-step
Force: Increase in probabilistic concurrency
Self force
Predecessor force
Successor force

Predecessor and successor forces

**pred** all predecessors of node under consideration
**succ** all successors of node under consideration
predecessor force
\[ B = \sum_{b \in \text{pred}} \sum_{t \in F_S} D_t \cdot \delta D_t \]
successor force
\[ C = \sum_{c \in \text{succ}} \sum_{t \in F_F} D_t \cdot \delta D_t \]
Intuition

total force: \( A + B + C \)

Schedule operation and time slot with minimal total force
Then recompute forces and schedule the next operation
Attempt to balance concurrency during scheduling

Implementation: Frame-based scheduling

Break schedule into (usually fixed) frames
Large enough to hold a long job
- Avoid preemption
- Evenly divide hyperperiod
Scheduler makes changes at frame start
Network flow formulation for frame-based scheduling
Could this be used for on-line scheduling?

Rate monotonic scheduling (RMS)

Single processor
Independent tasks
Differing arrival periods
Schedule in order of increasing periods
No fixed-priority schedule will do better than RMS
Guaranteed valid for loading \( \leq \ln 2 = 0.69 \)
For loading \( > \ln 2 \) and \( < 1 \), correctness unknown
Usually works up to a loading of 0.88

Main idea
- 1973, Liu and Layland derived optimal scheduling algorithm(s)
  for this problem
- Schedule the job with the smallest period (period = deadline)
  first
  - Analyzed worst-case behavior on any task set of size \( n \)
  - Found utilization bound: \( U(n) = n \cdot (2^{1/n} - 1) \)
    - \( 0.828 \) at \( n = 2 \)
    - \( \text{as } n \to \infty, U(n) \to \log 2 = 0.693 \)
  - Result: For any problem instance, if a valid schedule is possible,
    the processor need never spend more than 31% of its time idle
**Rate monotonic scheduling**

Constrained problem definition
Over-allocation often results
However, in practice utilization of 85%–90% common
Lose guarantee
If phases known, can prove by generating instance

**Critical instants**

Main idea:
A job’s critical instant a time at which all possible concurrent
higher-priority jobs are also simultaneously released
Useful because it implies latest finish time

**Proof sketch for RMS utilization bound**

- Consider case in which no period exceeds twice the shortest period
- Find a pathological case: in phase
  - Utilization of 1 for some duration
  - Any decrease in period/deadline of longest-period task will cause deadline violations
  - Any increase in execution time will cause deadline violations

**Proof sketch for RMS utilization bound**

Same true if execution time of high-priority task reduced

\[ e' = p_i + 1 - p_i - \epsilon \]

In this case, must increase other \( \epsilon \) or leave idle for \( 2 \cdot \epsilon \)

\[ e'' = e' + 2 \epsilon \]

\[ U'' - U = \frac{3 \epsilon}{p_i} + \frac{2 \epsilon}{p_k} \]

Again, \( p_k < 2 \rightarrow U'' > U \)

Sum over execution time/period ratios

**Notes on RMS**

- DMS better than or equal RMS when deadline \( \neq \) period
- Why not use slack-based?
- What happens if resources are under-allocated and a deadline is missed?
**Mixing on-line and off-line**

- Book mixes off-line and on-line with little warning
  - Be careful, actually different problem domains
- However, can be used together
  - Superloop (cyclic executive) with non-critical tasks
  - Slack stealing
  - Processor-based partitioning

**Vehicle routing**

- Low-price, slow, ARM-based system
- Long-term shortest path computation
- Greedy path calculation algorithm available, non-preemptable
  - Don’t make the user wait
    - Short-term next turn calculation
    - 200 ms timer available

**Bizarre scheduling idea**

- Scheduling and validity checking algorithms considered so far operate in time domain
  - This is somewhat strange idea
  - Think about it and tell/email me if you have any thoughts on it
  - Could one very quickly generate a high-quality real-time off-line multi-rate periodic schedule by operating in the frequency domain?
  - If not, why not?
  - What if the deadlines were soft?

**Example problem: Static scheduling**

- What is an FPGA?
  - Why should real-time systems designers care about them?
- Multiprocessor static scheduling
  - No preemption
  - No overhead for subsequent execution of tasks of same type
  - High cost to change task type
  - Scheduling algorithm?

**Compression references (for next class)**


**Project proposals**

Due 12:00 Sunday
A one-page project description
Ideally, you will have some preliminary results or ideas based on reading papers or doing analysis already
Lecture on data compression in embedded system design
A real, graded quiz