Embedded System Design and Synthesis

Robert Dick

http://robertdick.org/esds/
Office: EECS 2417-E
Department of Electrical Engineering and Computer Science
University of Michigan




Outline

Types of reliability

- Algorithm correctness: Does the specification have the desired properties?
- Robustness in the presence of transient faults: Can the system continue to operate correctly despite temporary errors?
- Robustness in the presence of permanent faults: Can the system continue to operate correctly in the presence of permanent errors?

Section outline

1. Reliable embedded system design and synthesis
    - Algorithm correctness
    - Appropriate responses to transient faults
    - Appropriate responses to permanent faults

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

Critical barriers to use

- For simple systems, manual proofs possible
- For very complex systems, state space exploration intractable
- May require new, more formal, specification language

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

Section outline

1. Reliable embedded system design and synthesis
    - Algorithm correctness
    - Appropriate responses to transient faults
    - Appropriate responses to permanent faults

Cross-talk

- Shielding
- Bus encoding

Particle impact

- Temporal redundancy
- Structural redundancy
- Voltage control

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?

Section outline

1. Reliable embedded system design and synthesis
    - Algorithm correctness
    - Appropriate responses to transient faults
    - Appropriate responses to permanent faults

Electromigration

- Reduce temperature
- Reduce current
- Spatial redundancy

Manufacturing defects

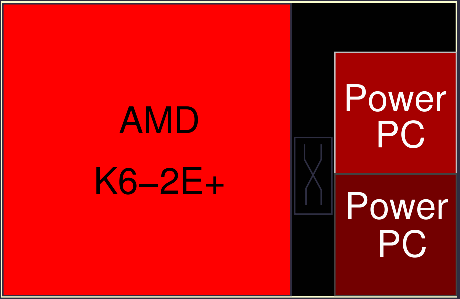
- Spatial redundancy

Example lifetime failure aware synthesis flow

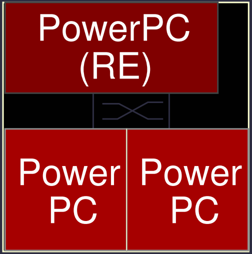
Changyun Zhu, Z. P. Gu, Robert P. Dick, and Li Shang. Reliable multiprocessor system-on-chip synthesis. In *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, pages 239–244, October 2007

- 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

Motivating example for reliability optimization

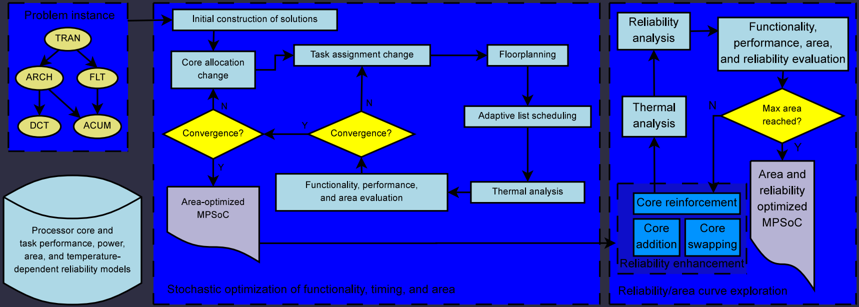


Solution I

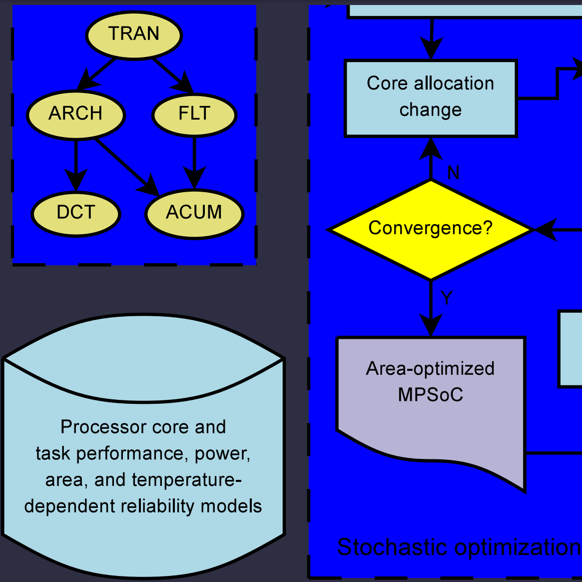


Solution II

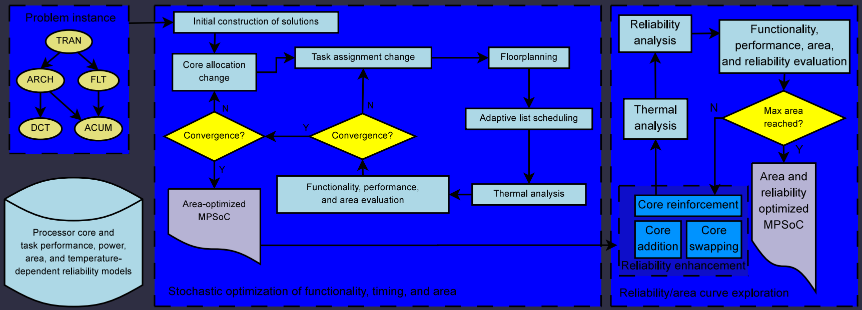
Reliability optimization flow



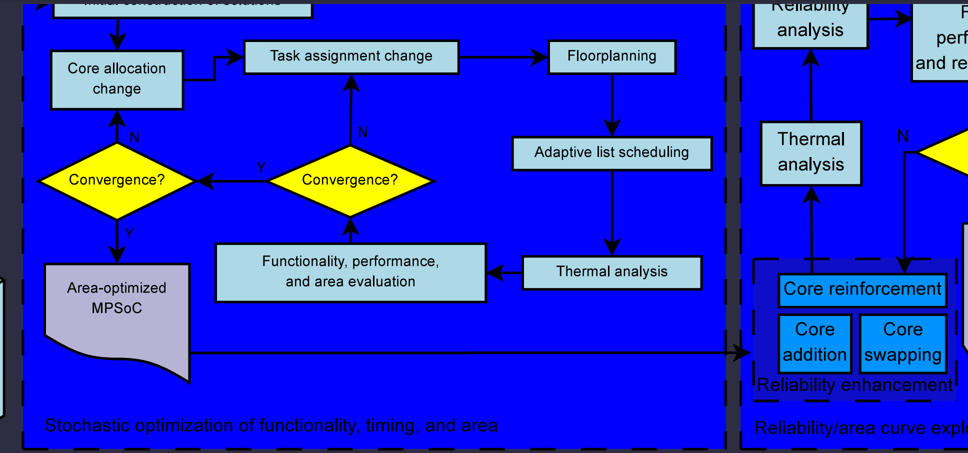
Reliability optimization flow



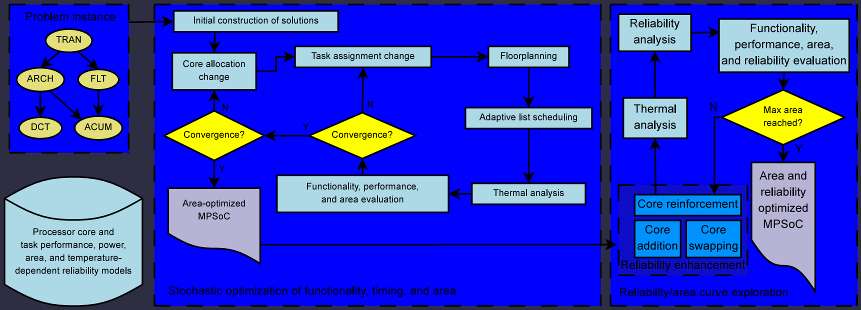
Reliability optimization flow



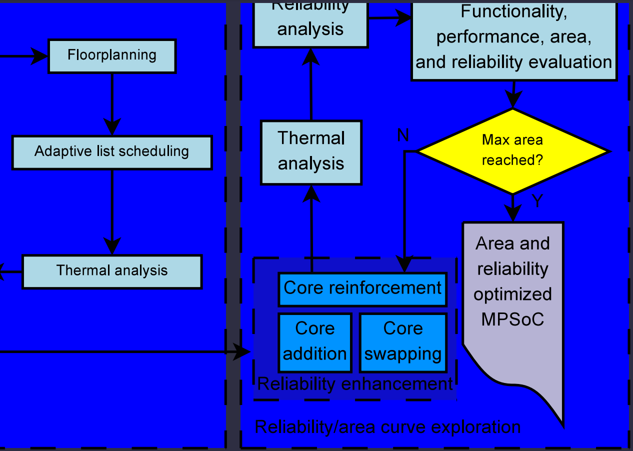
Reliability optimization flow



Reliability optimization flow



Reliability optimization flow



Lifetime reliability optimization challenges

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

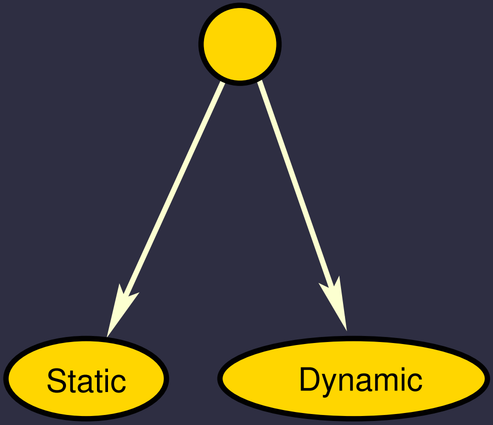
Importance of understanding fault class

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

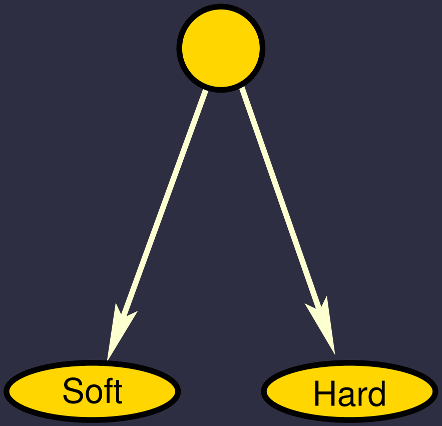
Outline

Section outline

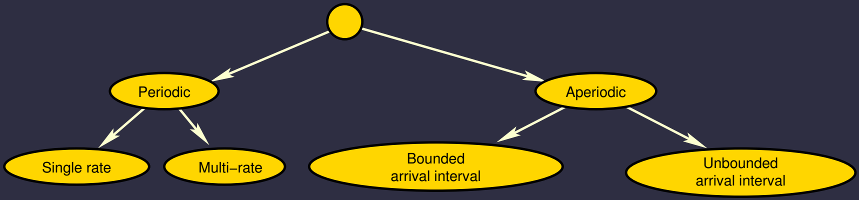
Taxonomy of real-time systems



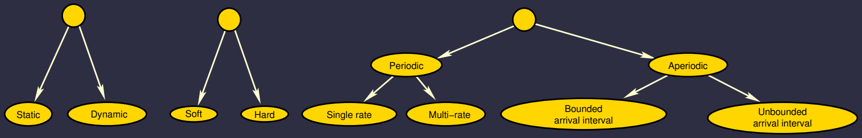
Taxonomy of real-time systems



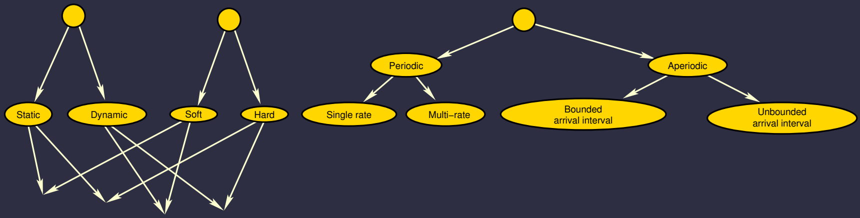
Taxonomy of real-time systems



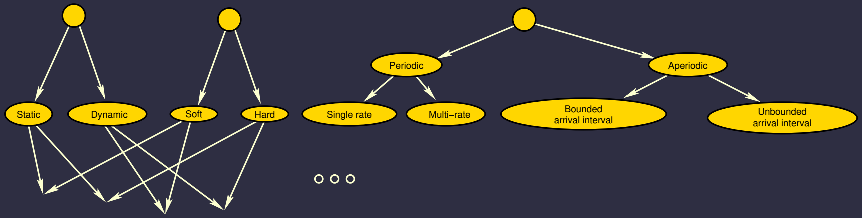
Taxonomy of real-time systems



Taxonomy of real-time systems



Taxonomy of real-time systems



Static

- 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.

Dynamic

- 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.

Soft real-time

- 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

Hard real-time

- Difficult problem. Some timing constraints inflexible.
- Simplifies problem formulation.

Periodic

- 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. Branch-and-bound methods: A survey. *Operations Research*, pages 699–719, July 1966.
- 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.

Aperiodic

- Also called sporadic, asynchronous, or reactive
- Implies dynamic
- Bounded arrival time interval permits resource reservation
- Unbounded arrival time interval impossible to deal with for any resource-constrained system

Section outline

2. Realtime systems
    - Taxonomy
    - Definitions
    - Central areas of real-time study

Definitions

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

Task

- Some operation that needs to be carried out
- Atomic completion: A task is all done or it isn't
- Non-atomic execution: A task may be interrupted and resumed

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)

	IBM PowerPC 405GP 266 MHz	IDT79RC32364 100 MHz	Imsys Cjip 40 MHz
Tooth	7.7E–6	···	
Road	330E–9	...	
FIR	4.1E–6	···	
Matrix	310E–3	···	

Relationship between tasks, processors, and costs

E.g., power consumption or worst-case execution time

Cost functions

- Mapping of real-time system design problem solution instance to cost value
- I.e., allows price, or hard deadline violation, of a particular multi-processor implementation to be determined

Back to real-time problem taxonomy: Jagged edges

- Some things dramatically complicate real-time scheduling
- These are horrific, especially when combined
  - Data dependencies
  - Unpredictability
  - Distributed systems
- These are irksome
  - Heterogeneous processors
  - Preemption

Section outline

Central areas of real-time study

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

Allocation, assignment, and scheduling

How does one best

- Analyze problem instance specifications
  - E.g., worst-case task execution time
- Select (and build) hardware components
- Select and produce software
- Decide which processor will be used for each task
- Determine the time(s) at which all tasks will execute

Allocation, assignment, and scheduling

- In order to efficiently and (when possible) optimally minimize
  - Price, power consumption, soft deadline violations
- Under hard timing constraints
- 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

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?

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.

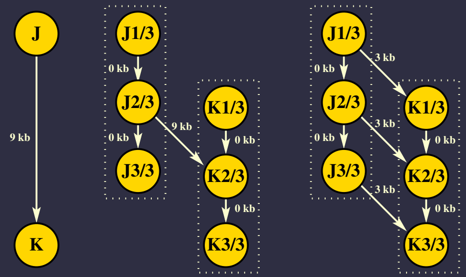
Know whether your design changes are taking you in a good direction

Outline

1. Reliable embedded system design and synthesis
2. Realtime systems
3. Scheduling
4. Overview of real-time and embedded operating systems
5. Embedded application/OS time, power, and energy estimation
6. Homework

Section outline

Graph extensions

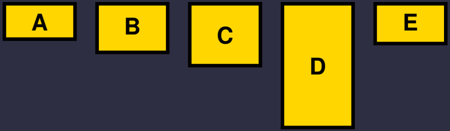


a) conventional b) pre– and post–computation c) streaming

Allows pipelining and pre/post-computation

In contrast with book, not difficult to use if conversion automated

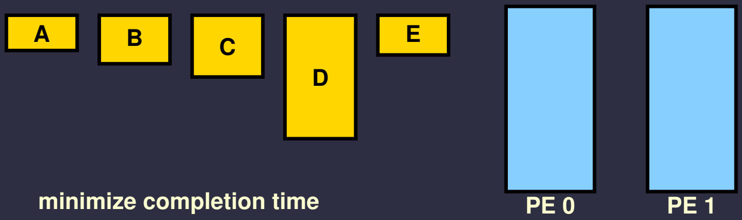
Problem definition



minimize completion time

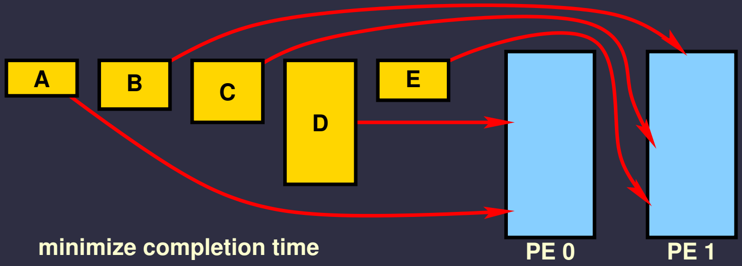
- Given a set of tasks,

Problem definition



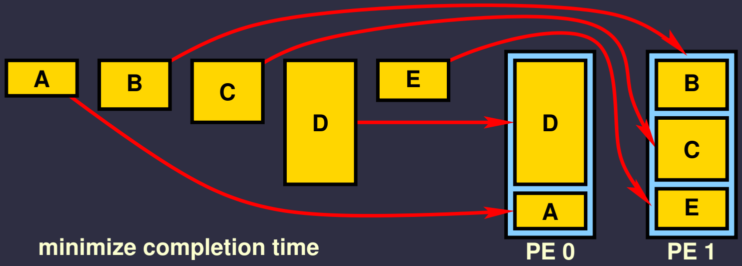
- Given a set of tasks,
- a cost function,

Problem definition



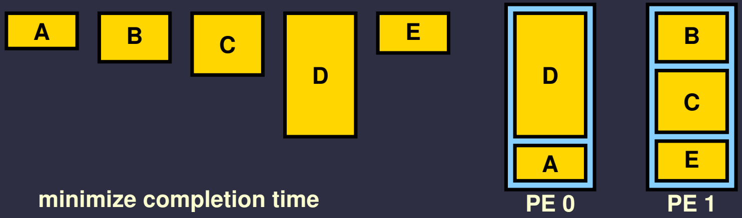
- Given a set of tasks,
- a cost function,
- and a set of resources,

Problem definition



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

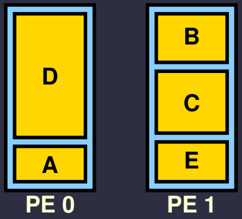
Problem definition



- 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

minimize completion time



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

Types of scheduling problems

- Discrete time – Continuous time
- Hard deadline – Soft deadline
- Unconstrained resources – Constrained resources
- Uni-processor – Multi-processor
- Homogeneous processors – Heterogeneous processors
- Free communication – Expensive communication
- Independent tasks – Precedence constraints
- Homogeneous tasks – Heterogeneous tasks
- One-shot – Periodic
- Single rate – Multirate
- Non-preemptive – Preemptive
- Off-line – On-line

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?

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

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

Unconstrained – Constrained resources

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

Uni-processor – Multi-processor

- Uni-processor
  - All tasks execute on the same resource
  - This can still be somewhat challenging
  - However, sometimes in $\mathcal{P}$
- Multi-processor
  - There are multiple resources to which tasks may be scheduled
- Usually $\mathcal{NP}$-complete

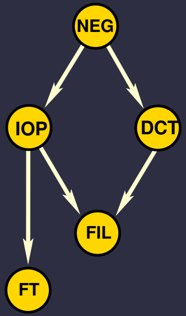
Homogeneous – Heterogeneous processors

- Homogeneous processors
  - All processors are the same type
- Heterogeneous processors
  - There are different types of processors
  - Usually $\mathcal{NP}$-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 $\mathcal{NP}$-complete

Independent tasks – Precedence constraints

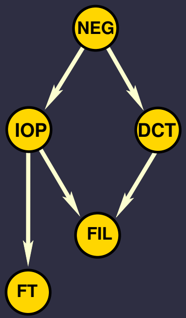


- Independent tasks: No previous execution sequence imposed

Independent tasks – Precedence constraints

- Independent tasks: No previous execution sequence imposed
- Precedence constraints: Weak order on task execution order

Homogeneous – Heterogeneous tasks

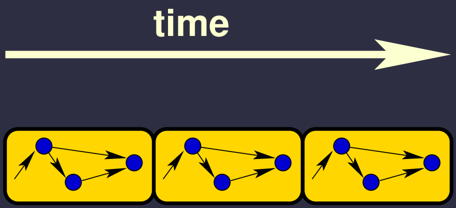


- Homogeneous tasks: All tasks are identical

Homogeneous – Heterogeneous tasks

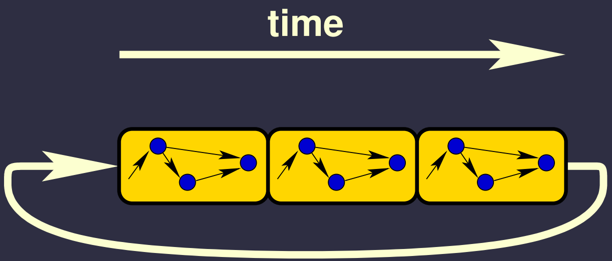
- Homogeneous tasks: All tasks are identical
- Heterogeneous tasks: Tasks differ

One-shot – Periodic



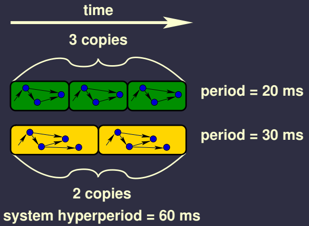
- One-shot: Assume that the task set executes once

One-shot – Periodic



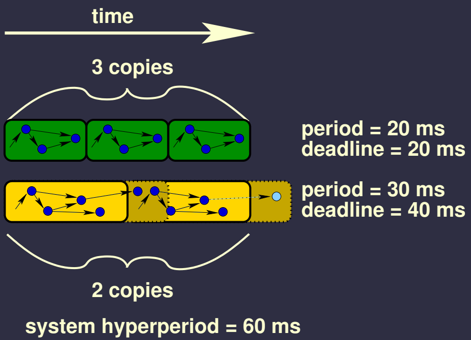
- One-shot: Assume that the task set executes once
- Periodic: Ensure that the task set can repeatedly execute at some period

Single rate – Multirate

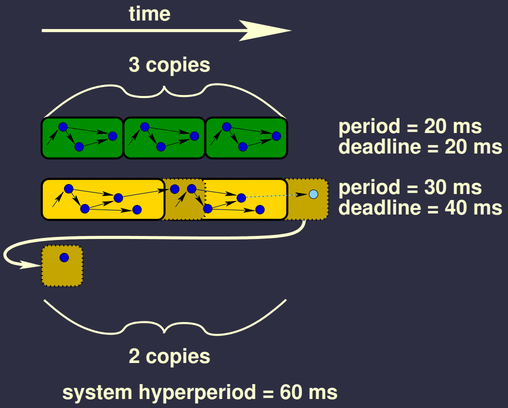


- Single rate: All tasks have the same period
- Multirate: Different tasks have different periods
  - Complicates scheduling
  - Can copy out to the least common multiple of the periods (hyperperiod)

Periodic graphs



Periodic graphs



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

Periodic vs. aperiodic

Periodic applications

- Power electronics
- Transportation applications
  - Engine controllers
  - Brake controllers
- Many multimedia applications
  - Video frame rate
  - Audio sample rate
- Many digital signal processing (DSP) applications

However, devices which react to unpredictable external stimuli have aperiodic behavior

Many applications contain periodic and aperiodic components

Aperiodic to periodic

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

- Some resources will be wasted

Example:

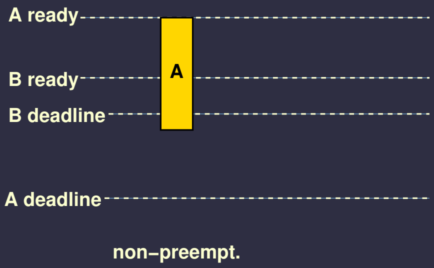
- At most one aperiodic task can arrive every 50 ms
- It must complete execution within 100 ms of its arrival time

Aperiodic to periodic

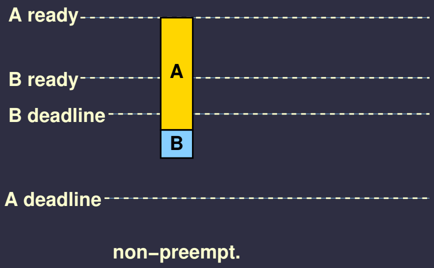
- Can easily build a periodic representation with a deadline and period of 50 ms
  - Problem, requires a 50 ms execution time when 100 ms should be sufficient
- Can use overlapping graphs to allow an increase in execution time
  - Parallelism required

The main problem with representing aperiodic problems with periodic representations is that the tradeoff between deadline and period must be made at design/synthesis time

Non-preemptive – Preemptive

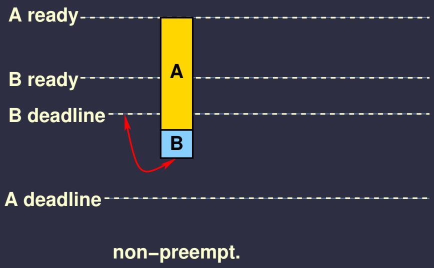


Non-preemptive – Preemptive

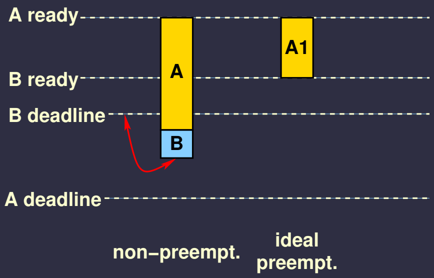


- Non-preemptive: Tasks must run to completion

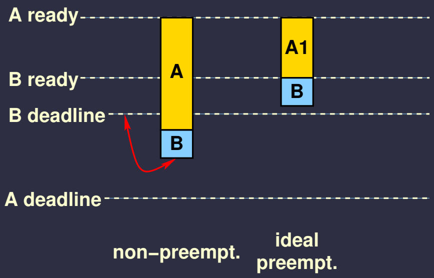
Non-preemptive – Preemptive



Non-preemptive – Preemptive

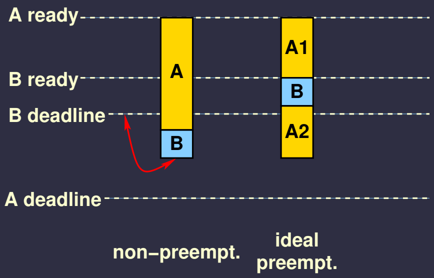


Non-preemptive – Preemptive

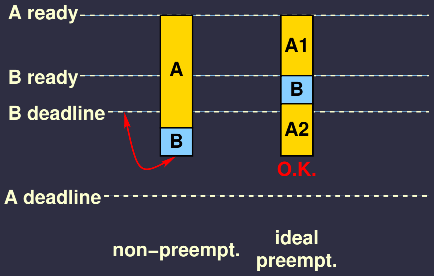


- Ideal preemptive: Tasks can be interrupted without cost

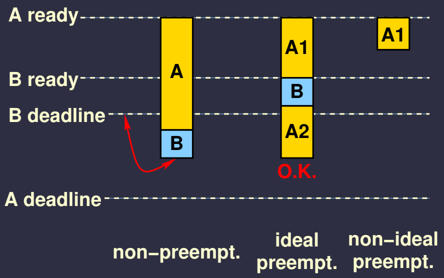
Non-preemptive – Preemptive



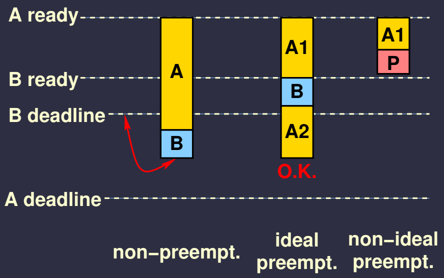
Non-preemptive – Preemptive



Non-preemptive – Preemptive

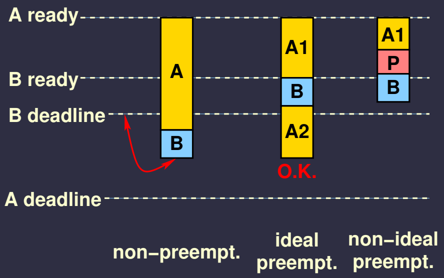


Non-preemptive – Preemptive

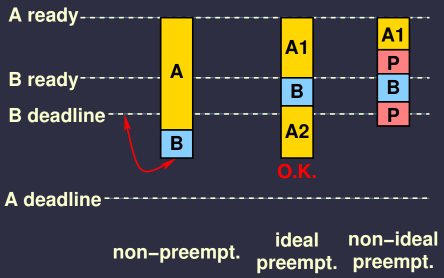


- Non-ideal preemptive: Tasks can be interrupted with cost

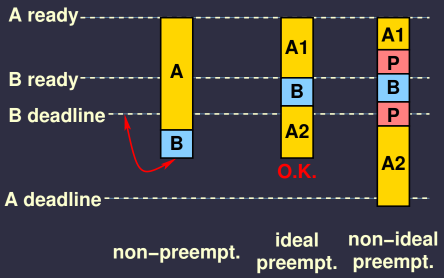
Non-preemptive – Preemptive



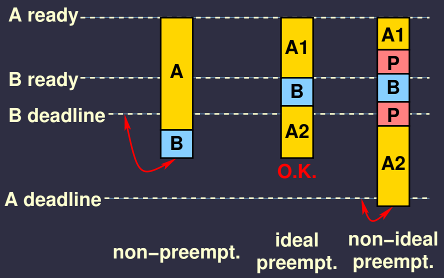
Non-preemptive – Preemptive



Non-preemptive – Preemptive



Non-preemptive – Preemptive



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

Hardware-software co-synthesis scheduling

- Expensive communication
  - Complicated set of communication resources
- Precedence constraints
- Periodic
- Multirate
- Strong interaction between $\mathcal{NP}$-complete allocation-assignment and $\mathcal{NP}$-complete scheduling problems
- Will revisit problem later in course if time permits

Behavioral synthesis scheduling

- 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?

Section outline

Scheduling methods

- Clock
- Weighted round-robbin
- List scheduling
- Priority
  - EDF, LST
  - Slack
  - Multiple costs

Scheduling methods

- MILP
- Force-directed
- Frame-based
- PSGA
- RMS

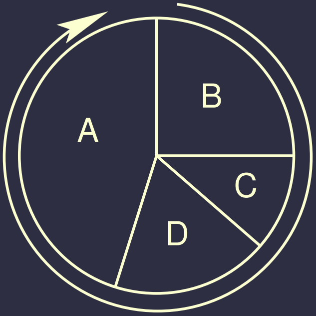
Clock-driven scheduling

Clock-driven: Pre-schedule, repeat schedule

Music box:

- Periodic
- Multi-rate
- Heterogeneous
- Off-line
- Clock-driven

Weighted round robbin



Weighted round-robbin: Time-sliced with variable time slots

List scheduling

- Pseudo-code:
  - Keep a list of ready jobs
  - Order by priority metric
  - Schedule
  - Repeat
- Simple to implement
- Can be made very fast
- Difficult to beat quality

Priority-driven

- Impose linear order based on priority metric
- Possible metrics
  - 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)

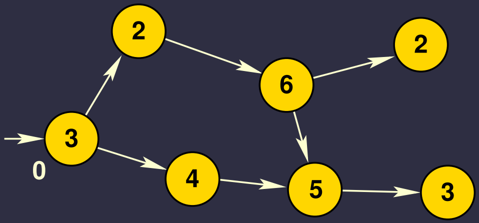
List scheduling

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

Prioritization

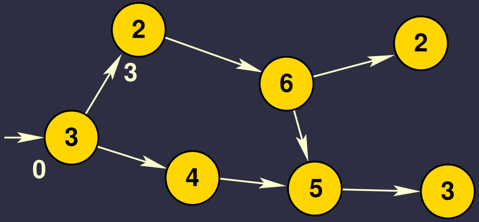
- As soon as possible (ASAP)
- As late as possible (ALAP)
- Slack-based
- Dynamic slack-based
- Multiple considerations

As soon as possible (ASAP)



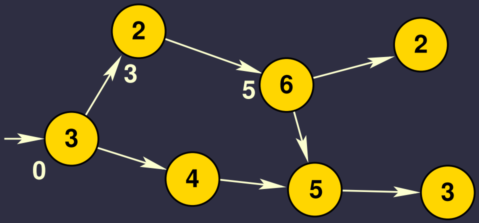
- From root, topological sort on the precedence graph
- Propagate execution times, taking the max at reconverging paths
- Schedule in order of increasing earliest start time (EST)

As soon as possible (ASAP)



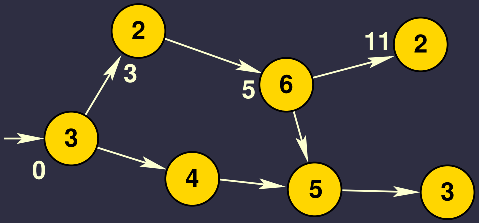
- From root, topological sort on the precedence graph
- Propagate execution times, taking the max at reconverging paths
- Schedule in order of increasing earliest start time (EST)

As soon as possible (ASAP)



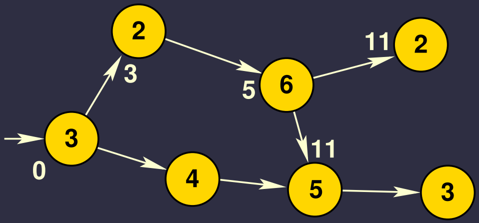
- From root, topological sort on the precedence graph
- Propagate execution times, taking the max at reconverging paths
- Schedule in order of increasing earliest start time (EST)

As soon as possible (ASAP)



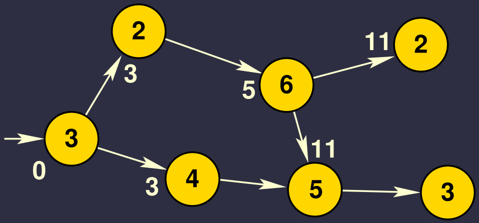
- From root, topological sort on the precedence graph
- Propagate execution times, taking the max at reconverging paths
- Schedule in order of increasing earliest start time (EST)

As soon as possible (ASAP)



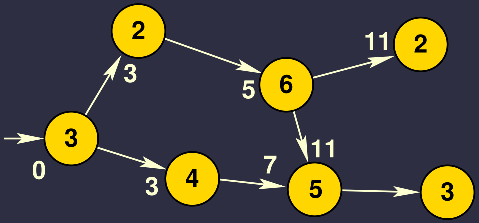
- From root, topological sort on the precedence graph
- Propagate execution times, taking the max at reconverging paths
- Schedule in order of increasing earliest start time (EST)

As soon as possible (ASAP)



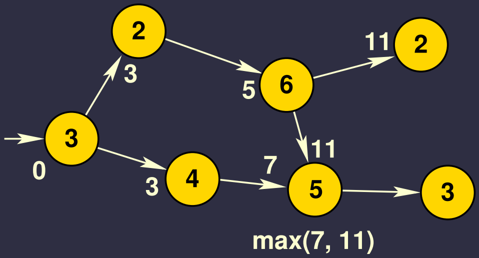
- From root, topological sort on the precedence graph
- Propagate execution times, taking the max at reconverging paths
- Schedule in order of increasing earliest start time (EST)

As soon as possible (ASAP)



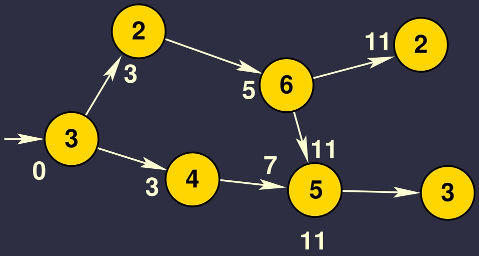
- From root, topological sort on the precedence graph
- Propagate execution times, taking the max at reconverging paths
- Schedule in order of increasing earliest start time (EST)

As soon as possible (ASAP)



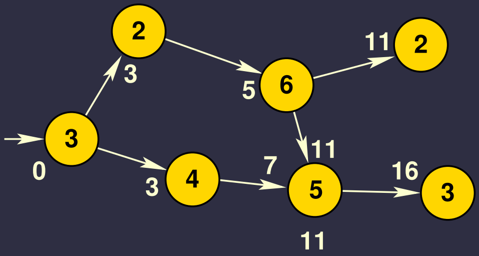
- From root, topological sort on the precedence graph
- Propagate execution times, taking the max at reconverging paths
- Schedule in order of increasing earliest start time (EST)

As soon as possible (ASAP)



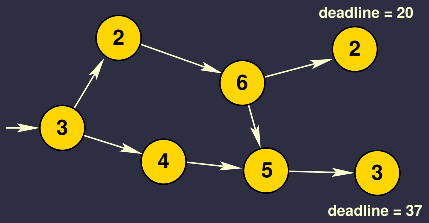
- From root, topological sort on the precedence graph
- Propagate execution times, taking the max at reconverging paths
- Schedule in order of increasing earliest start time (EST)

As soon as possible (ASAP)



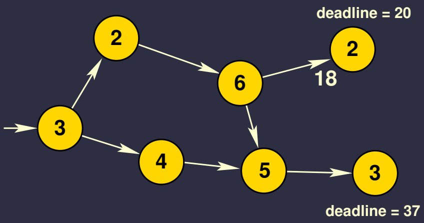
- From root, topological sort on the precedence graph
- Propagate execution times, taking the max at reconverging paths
- Schedule in order of increasing earliest start time (EST)

As late as possible (ALAP)



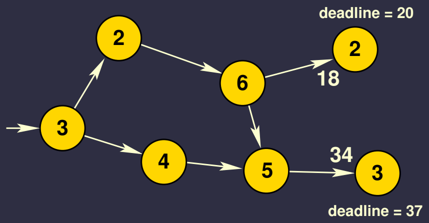
- From deadlines, topological sort on the precedence graph
- Propagate execution times, taking the min at reconverging paths
- Consider precedence-constraint satisfied tasks
  - Schedule in order of increasing latest start time (LST)

As late as possible (ALAP)



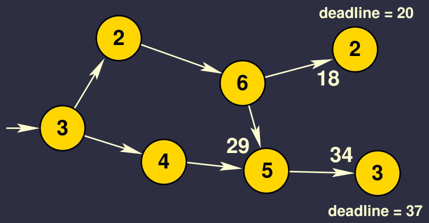
- From deadlines, topological sort on the precedence graph
- Propagate execution times, taking the min at reconverging paths
- Consider precedence-constraint satisfied tasks
  - Schedule in order of increasing latest start time (LST)

As late as possible (ALAP)



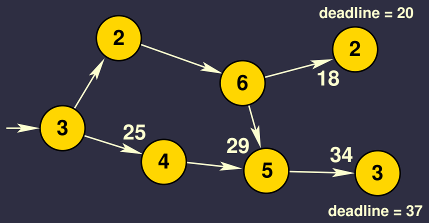
- From deadlines, topological sort on the precedence graph
- Propagate execution times, taking the min at reconverging paths
- Consider precedence-constraint satisfied tasks
  - Schedule in order of increasing latest start time (LST)

As late as possible (ALAP)



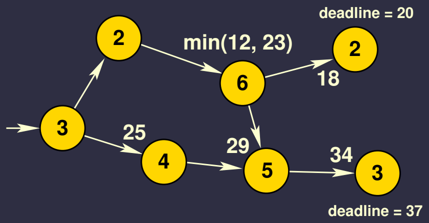
- From deadlines, topological sort on the precedence graph
- Propagate execution times, taking the min at reconverging paths
- Consider precedence-constraint satisfied tasks
    - Schedule in order of increasing latest start time (LST)

As late as possible (ALAP)



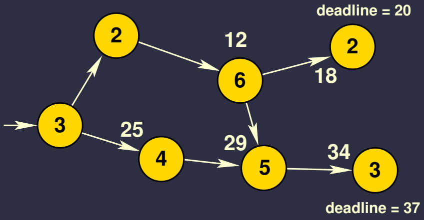
- From deadlines, topological sort on the precedence graph
- Propagate execution times, taking the min at reconverging paths
- Consider precedence-constraint satisfied tasks
  - Schedule in order of increasing latest start time (LST)

As late as possible (ALAP)



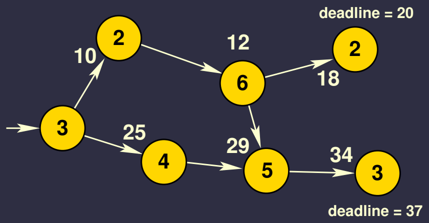
- From deadlines, topological sort on the precedence graph
- Propagate execution times, taking the min at reconverging paths
- Consider precedence-constraint satisfied tasks
  - Schedule in order of increasing latest start time (LST)

As late as possible (ALAP)



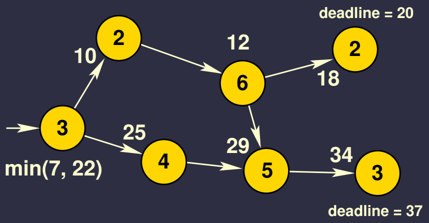
- From deadlines, topological sort on the precedence graph
- Propagate execution times, taking the min at reconverging paths
- Consider precedence-constraint satisfied tasks
  - Schedule in order of increasing latest start time (LST)

As late as possible (ALAP)



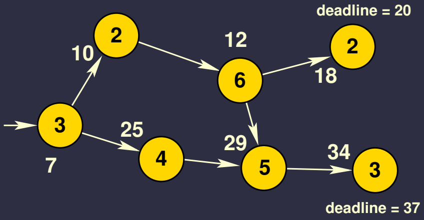
- From deadlines, topological sort on the precedence graph
- Propagate execution times, taking the min at reconverging paths
- Consider precedence-constraint satisfied tasks
  - Schedule in order of increasing latest start time (LST)

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)

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)

Slack-based

- Compute EFT, LFT
- For all tasks, find the difference, LFT − EFT
- This is the *slack*
- Schedule precedence-constraint satisfied tasks in order of increasing slack
- Can recompute slack each step, expensive but higher-quality result
  - Dynamic critical path scheduling

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

Effective release times

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

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

Breaking EDF, LST optimality

- Non-zero preemption cost
- Multiprocessor
- Why?

Multi-rate tricks

- Contract deadline
  - Usually safe
- Contract period
  - Sometimes safe
- Consequences?

Linear programming

- Minimize a linear equation subject to linear constraints
  - In $\mathcal{P}$
- Mixed integer linear programming: One or more variables discrete
  - $\mathcal{NP}$-complete
- Many good solvers exist
- Don't rebuild the wheel

MILP scheduling

P the set of tasks

t_{max} maximum time

$start(p, t)$ 1 if task p starts at time t, 0 otherwise

D the set of execution delays

E the set of precedence constraints

$$t_{start}(p) = \sum_{t=0}^{t_{max}} t \cdot start(p, t)$$ the start time of p

MILP scheduling

Each task has a unique start time

$$\forall_{p \in P}, \sum_{t=0}^{t_{max}} start(p, t) = 1$$

Each task must satisfy its precedence constraints and timing delays

$$\forall \{p_i, p_j\} \in E, \sum_{t=0}^{t_{max}} t_{start}(p_i) \geq t_{start}(p_j) + d_j$$

Other constraints may exist

- Resource constraints
- Communication delay constraints

MILP scheduling

- Too slow for large instances of $\mathcal{NP}$-complete scheduling problems
- Numerous optimization algorithms may be used for scheduling
- List scheduling is one popular solution
- Integrated solution to allocation/assignment/scheduling problem possible
- Performance problems exist for this technique

Force directed scheduling

- P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, June 1989
- 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

Self force

F_i all slots in time frame for i

F_i' all slots in new time frame for i

D_t probability density (sum) for slot t

δD_t change in density (sum) for slot t resulting from scheduling

self force

$$A = \sum_{t \in F_a} D_t \cdot \delta D_t$$

Predecessor and successor forces

pred all predecessors of node under consideration

succ all successors of node under consideration

predecessor force

$$B = \sum_{b \in \textbf{pred}} \sum_{t \in F_b} D_t \cdot \delta D_t$$

successor force

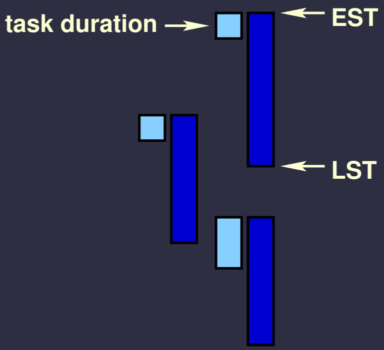
$$C = \sum_{c \in \textbf{succ}} \sum_{t \in F_c} D_t \cdot \delta D_t$$

Intuition

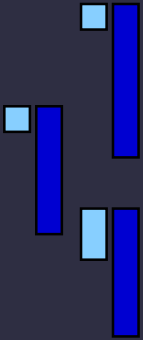
total force: $A + B + C$

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

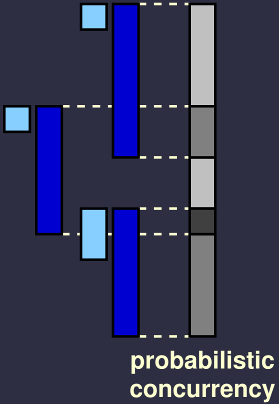
Force directed scheduling



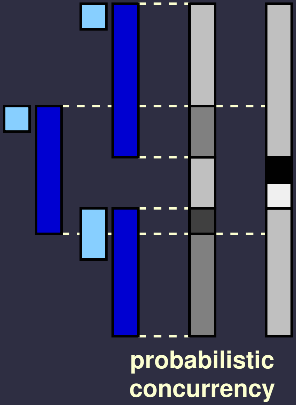
Force directed scheduling



Force directed scheduling



Force directed scheduling



Force directed scheduling

- Limitations?
- What classes of problems may this be used on?

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?

Problem space genetic algorithm

- Let's finish off-line scheduling algorithm examples on a bizarre example
- Use conventional scheduling algorithm
- Transform problem instance
- Solve
- Validate
- Evolve transformations

Rate mononotic 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

Rate monotonic scheduling

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 $n = 2$
- 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

Optimality and utilization for limited case

- Simply periodic: All task periods are integer multiples of all lesser task periods
- In this case, RMS/DMS optimal with utilization 1
- However, this case rare in practice
- Remains feasible, with decreased utilization bound, for in-phase tasks with arbitrary periods

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

Definitions

- Period: T.
- Execution time: C.
- Process: i.
- Utilization: $U = \sum_{i=1}^{m} \frac{C_i}{T_i}$.
- Assume Task 1 is higher priority than Task 2, and thus $T_1 < T_2$.

Case 1 I

All instances of higher-priority tasks released before end of lower-priority task period complete before end of lower-priority task period.

1. $C_1 \leq T_2 - T_1 \left\lfloor \frac{T_2}{T_1} \right\rfloor$.
2. I.e., the execution time of Task 1 is less than or equal to the period of Task 2 minus the total time spent within the periods of instances of Task 1 finishing within Task 2's period.
3. Now, let's determine the maximum execution time of Task 2 as a function of all other variables.
4. $C_{2,max} = T_2 - C_1 \left\lceil \frac{T_2}{T_1} \right\rceil$.

Case 1 II

5. I.e., the maximum execution time of Task 2 is the period of Task 2 minus the total execution time of instances of Task 1 released within Task 2's period.

Case 1 III

6. In this case,

$$
\begin{aligned}
U &= U_1 + U_2 \\
&= \frac{C_1}{T_1} + \frac{C_{2,max}}{T_2} \\
&= \frac{C_1}{T_1} + \frac{T_2 - C_1 \left\lceil \frac{T_2}{T_1} \right\rceil}{T_2} \\
&= \frac{C_1}{T_1} + 1 - \frac{C_1 \left\lceil \frac{T_2}{T_1} \right\rceil}{T_2} \\
&= 1 + C_1 \left(\frac{1}{T_1} - \frac{1}{T_2} \left\lceil \frac{T_2}{T_1} \right\rceil \right)
\end{aligned}
$$

7. Is $\frac{1}{T_1} - \frac{1}{T_2} \left\lceil \frac{T_2}{T_1} \right\rceil < 0$?

Case 1 IV

8. Thus, U is monotonically decreasing in C_1.

Case 2 I

Instances of higher-priority tasks released before end of lower-priority task period complete after end of lower-priority task period.

1. $C_1 \geq T_2 - T_1 \left\lfloor \frac{T_2}{T_1} \right\rfloor$.
2. $C_{2,max} = -C_1 \left\lfloor \frac{T_2}{T_1} \right\rfloor + T_1 \left\lfloor \frac{T_2}{T_1} \right\rfloor$.
3. $U = \frac{T_1}{T_2} \left\lfloor \frac{T_2}{T_1} \right\rfloor + C_1 \left(\frac{1}{T_1} - \frac{1}{T_2} \left\lfloor \frac{T_2}{T_1} \right\rfloor \right)$.

Minimal U

1. $C_1 = T_2 - T_1 \left\lfloor \frac{T_2}{T_1} \right\rfloor$.
2. $U = 1 - \frac{T_1}{T_2} \left(\left\lceil \frac{T_2}{T_1} \right\rceil - \frac{T_2}{T_1} \right) \left(\frac{T_2}{T_1} - \left\lfloor \frac{T_2}{T_1} \right\rfloor \right)$.
3. Let $I = \left\lfloor \frac{T_2}{T_1} \right\rfloor$ and
4. $f = \frac{T_2}{T_1}$.
5. Then, $U = 1 - \frac{f(1-f)}{I+f}$.
6. To maximize U, minimize I, which can be no smaller than 1.
7. $U = 1 - \frac{f(1-f)}{1+f}$.
8. Differentiate to find mimima, at $f = \sqrt{2} - 1$.
9. Thus, $U_{min} = 2\left(\sqrt{2} - 1\right) \approx 0.83$.
10. Is this the minimal U? Are we done?

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?

Scheduling summary

- Scheduling is a huge area
- This lecture only introduced the problem and potential solutions
- Some scheduling problems are easy
- Most useful scheduling problems are hard
  - Committing to decisions makes problems hard: Lookahead required
  - Interdependence between tasks and processors makes problems hard

Section outline

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

Mixing on-line and off-line

- Slack stealing
- Processor-based partitioning

Bizarre scheduling idea

- Scheduling and validity checking algorithms considered so far operate in time domain
- This is a 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?

Problem: Uniprocessor independent task scheduling

- Problem
  - Independent tasks
  - Each has a period = hard deadline
  - Zero-cost preemption
- How to solve?

Outline

1. Reliable embedded system design and synthesis
2. Realtime systems
3. Scheduling
4. Overview of real-time and embedded operating systems
5. Embedded application/OS time, power, and energy estimation
6. Homework

Essential features of RTOSs

- Provides real-time scheduling algorithms or primatives
- Bounded execution time for OS services
  - Usually implies preemptive kernel
  - E.g., Linux can spend milliseconds handling interrupts, especially disk access

Threads

- Threads vs. processes: Shared vs. unshared resources
- OS impact: Windows vs. Linux
- Hardware impact: MMU

Threads vs. processes

- Threads: Low context switch overhead
- Threads: Sometimes the only real option, depending on hardware
- Processes: Safer, when hardware provides support
- Processes: Can have better performance when IPC limited

Software implementation of schedulers

- TinyOS
- Light-weight threading executive
- μC/OS-II
- Linux
- Static list scheduler

TinyOS

- Most behavior event-driven
- High rate $\rightarrow$ Livelock
- Research schedulers exist

BD threads

- Brian Dean: Microcontroller hacker
- Simple priority-based thread scheduling executive
- Tiny footprint (fine for AVR)
- Low overhead
- No MMU requirements

μC/OS-II

- Similar to BD threads
- More flexible
- Bigger footprint

Old Linux scheduler

- Single run queue
- $\mathcal{O}(n)$ scheduling operation
- Allows dynamic goodness function

$\mathcal{O}(1)$ scheduler in Linux 2.6

- Written by Ingo Molnar
- Splits run queue into two queues prioritized by goodness
- Requires static goodness function
  - No reliance on running process
- Compatible with preemptible kernel

Real-time Linux

- Run Linux as process under real-time executive
- Complicated programming model
- RTAI (Real-Time Application Interface) attempts to simplify
  - Colleagues still have problems at $>$ 18 kHz control period

Real-time operating systems

- Embedded vs. real-time
- Dynamic memory allocation
- Schedulers: General-purpose vs. real-time
- Timers and clocks: Relationship with HW

Outline

1. Reliable embedded system design and synthesis
2. Realtime systems
3. Scheduling
4. Overview of real-time and embedded operating systems
5. Embedded application/OS time, power, and energy estimation
6. Homework

Collaborators on project

Princeton
Niraj K. Jha

NEC Labs America
Ganesh Lakshminarayana
Anand Raghunathan

Section outline

5. Embedded application/OS time, power, and energy estimation
   - Introduction, motivation, and past work
   - Examples of energy optimization
   - Simulation infrastructure
   - Results

Introduction

- Real-Time Operating Systems are often used in embedded systems
- They simplify use of hardware, ease management of multiple tasks, and adhere to real-time constraints
- Power is important in many embedded systems with RTOSs
- RTOSs can consume significant amount of power
- They are re-used in many embedded systems
- They impact power consumed by application software
- RTOS power effects influence system-level design

Real-time operating systems (RTOS)

- Interaction between HW and SW
  - Rapid response to interrupts
  - HW interface abstraction
- Interaction between different tasks
  - Communication
  - Synchronization
- Multitasking
  - Ideally fully preemptive
  - Priority-based scheduling
  - Fast context switching
  - Support for real-time clock

General-purpose OS stress

- Good average-case behavior
- Providing many services
- Support for a large number of hardware devices

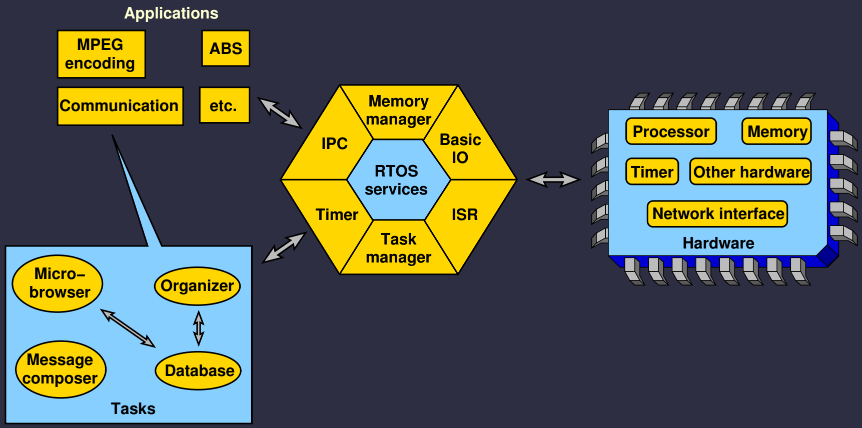
RTOSs stress

- Predictable service execution times
- Predictable scheduling
- Good worst-case behavior
- Low memory usage
- Speed
- Simplicity

Predictability

- General-purpose computer architecture focuses on average-case
  - Caches
  - Prefetching
  - Speculative execution
- Real-time embedded systems need predictability
  - Disabling or locking caches is common
  - Careful evaluation of worst-case is essential
  - Specialized or static memory management common

RTOS overview



RTOS power consumption

- Used in several low-power embedded systems
- Need for RTOS power analysis
  - Significant power consumption
  - Impacts application software power
  - Re-used across several applications

RTOS and real-time references

- K. Ramamritham and J. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proc. IEEE*, 82(1):55–67, January 1994
- Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, Boston, 2000

Prior work

- Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Compilation techniques for low energy: An overview. In *Proc. Int. Symp. Low-Power Electronics*, pages 38–39, October 1994
- Y. Li and J. Henkel. A framework for estimating and minimizing energy dissipation of embedded HW/SW systems. In *Proc. Design Automation Conf.*, pages 188–193, June 1998
- J. J. Labrosse. *MicroC/OS-II*. R & D Books, KS, 1998

RTOS power references

Journal version Design Automation Conference 2000 work in the area of RTOS power consumption analysis

- Robert P. Dick, G. Lakshminarayana, A. Raghunathan, and Niraj K. Jha. Analysis of power dissipation in real-time operating systems. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 22(5):615–627, May 2003

RTOS power references

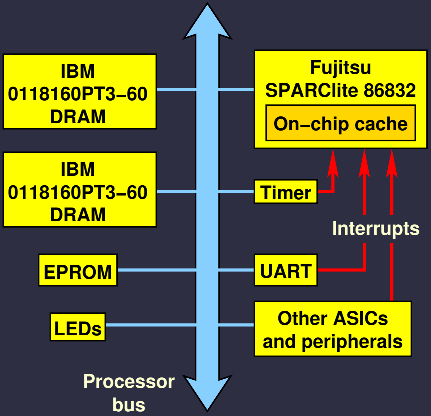
- K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. The performance and energy consumption of three embedded real-time operating systems. In *Proc. Int. Conf. Compilers, Architecture & Synthesis for Embedded Systems*, pages 203–210, November 2001
- T.-K. Tan, A. Raghunathan, and Niraj K. Jha. EMSIM: An energy simulation framework for an embedded operating system. In *Proc. Int. Symp. Circuits & Systems*, pages 464–467, May 2002

Contributions

- First detailed power analysis of RTOS
  - Proof of concept later used by others
- Applications
  - Low-power RTOS
  - Energy-efficient software architecture
  - Incorporate RTOS effects in system design

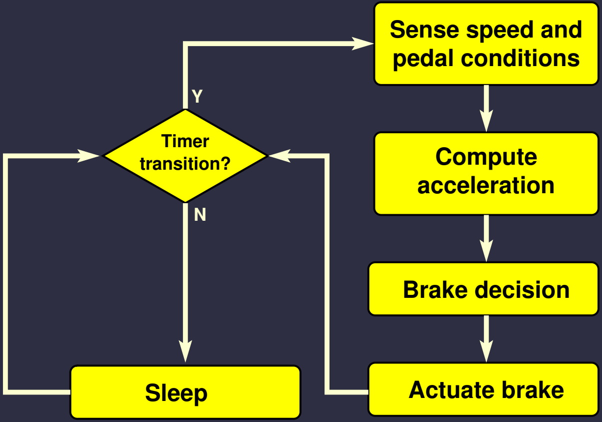
Section outline

Simulated embedded system

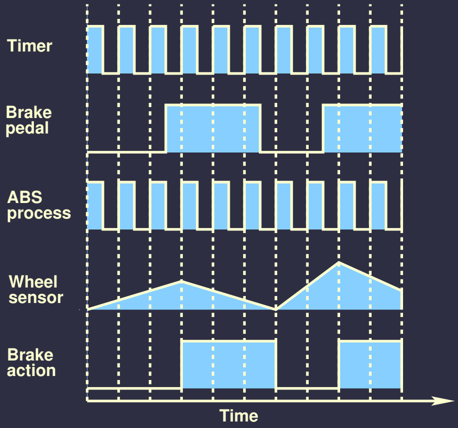


- Easy to add new devices
- Cycle-accurate model
- Fujitsu board support library used in model
- μC/OS-II RTOS used

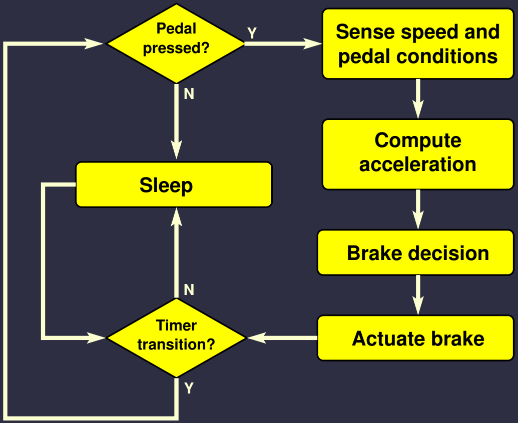
Periodically triggered ABS



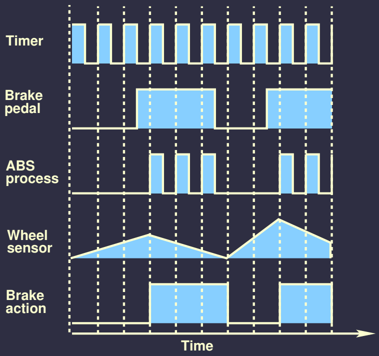
Periodically triggered ABS timing



Selectively triggered ABS

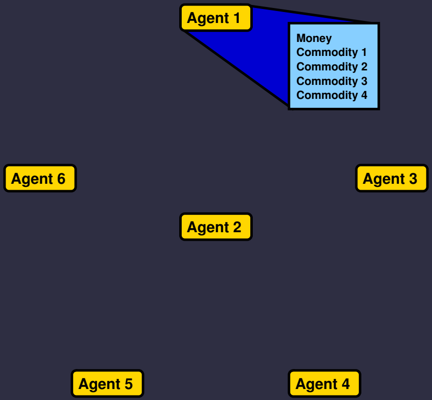


Selectively triggered ABS timing

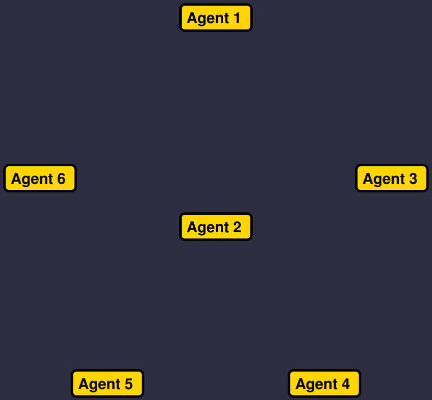


63% reduction in energy and power consumption

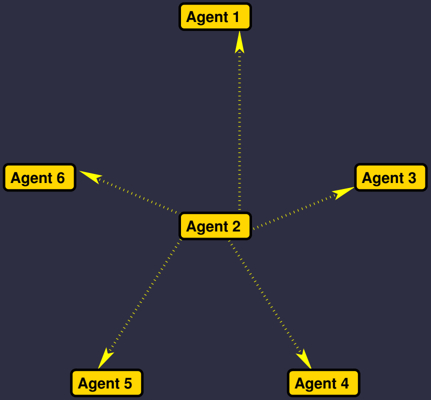
Agent example



Agent example

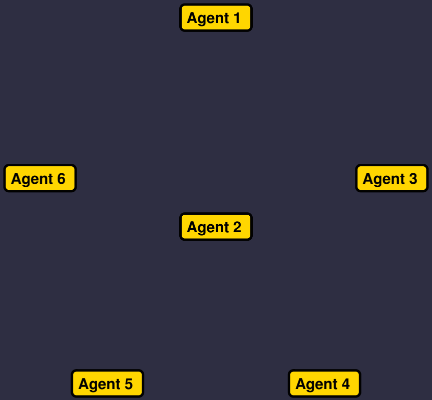


Agent example



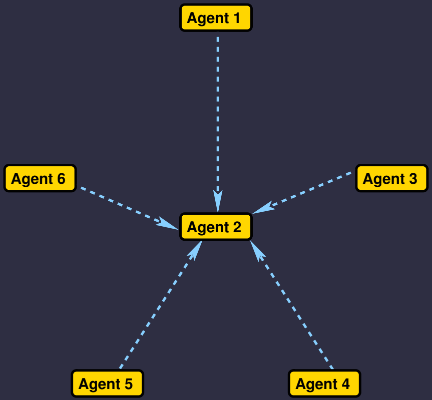
- Advertise

Agent example



- Advertise
- Bid

Agent example

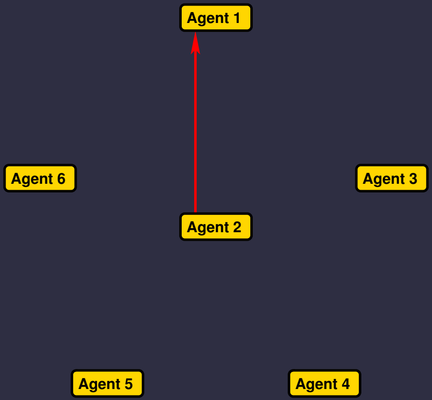


- Advertise
- Bid
- Offer

Agent example

- Advertise
- Bid
- Offer
- Transfer results

Agent example

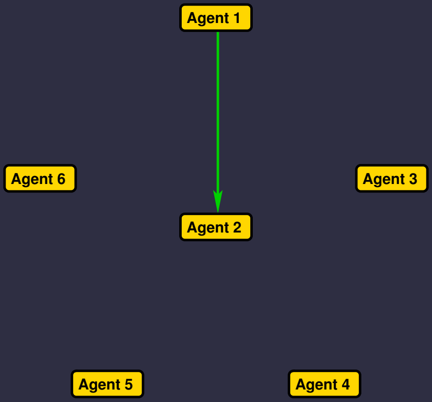


- Advertise
- Bid
- Offer
- Transfer results

Agent example

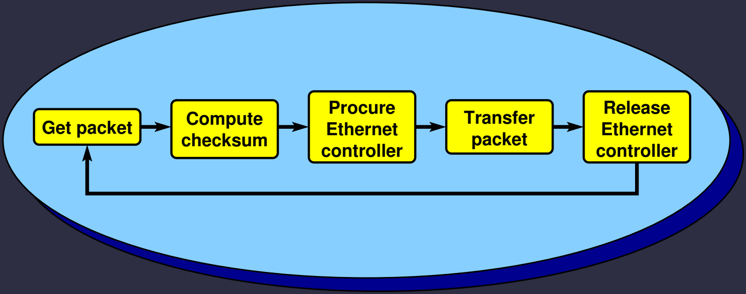
- Advertise
- Bid
- Offer
- Transfer results

Agent example



- Advertise
- Bid
- Offer
- Transfer results

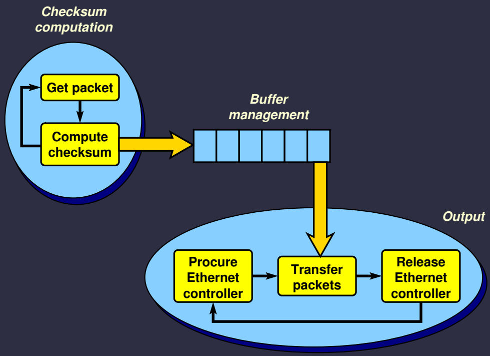
Single task network interface



Checksum computation and output

Procuring Ethernet controller has high energy cost

Multi-tasking network interface

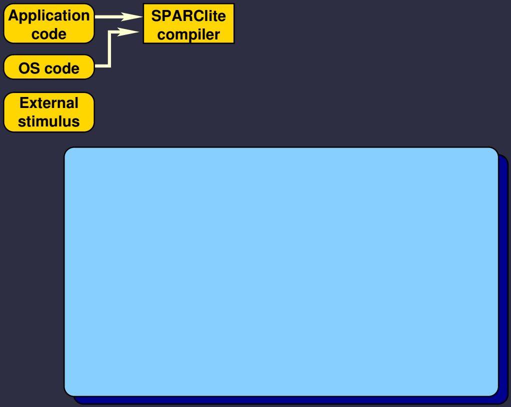


RTOS power analysis suggests process re-organization.
21% reduction in energy consumption. Similar power consumption.

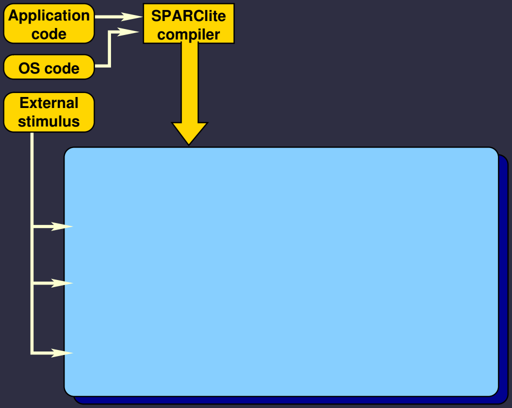
Section outline

5. Embedded application/OS time, power, and energy estimation
    - Introduction, motivation, and past work
    - Examples of energy optimization
    - Simulation infrastructure
    - Results

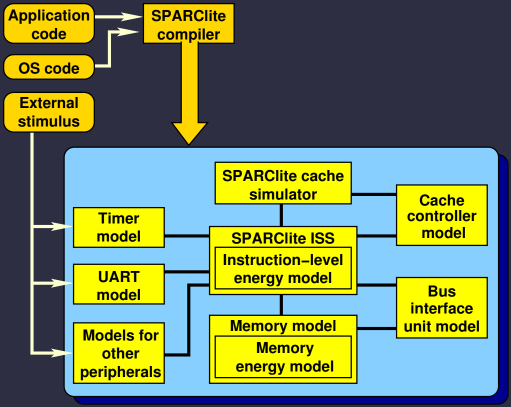
Infrastructure



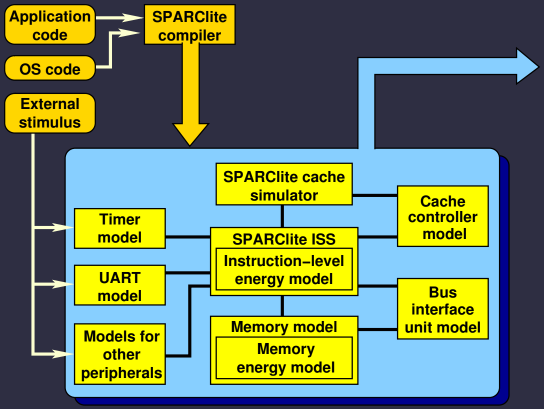
Infrastructure



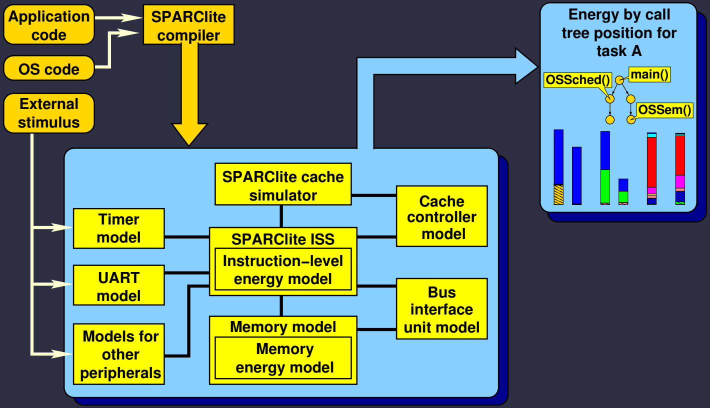
Infrastructure



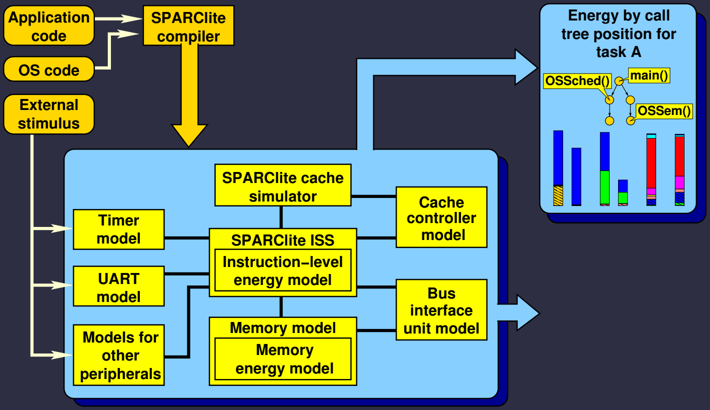
Infrastructure



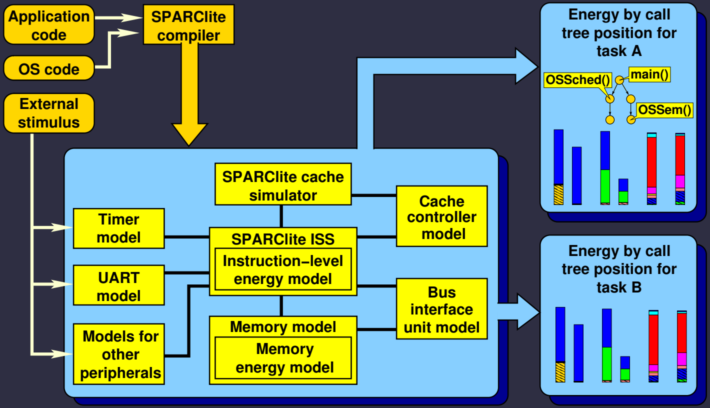
Infrastructure



Infrastructure



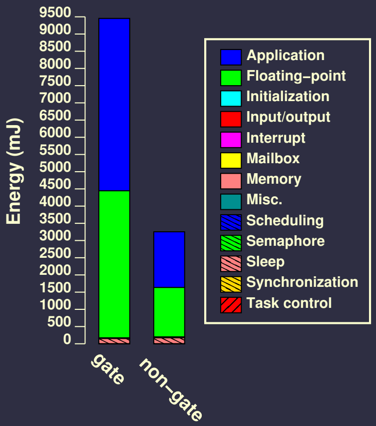
Infrastructure



Section outline

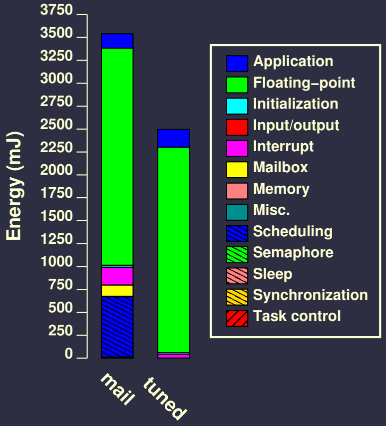
5. Embedded application/OS time, power, and energy estimation
   - Introduction, motivation, and past work
   - Examples of energy optimization
   - Simulation infrastructure
   - Results

ABS optimization effects



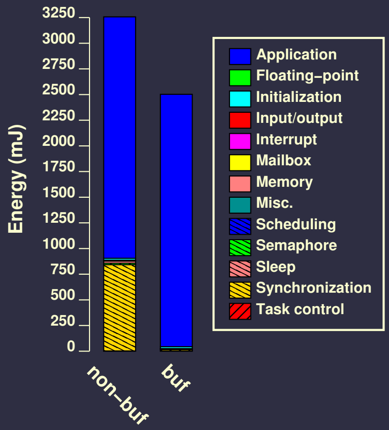
- Redesigned application after using simulator to locate areas where power was wasted
- 63% energy reduction
- 63% power reduction
- RTOS directly accounted for 50% of system energy

Agent optimization effects



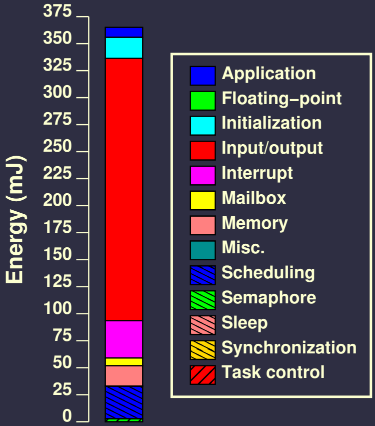
- Mail version used RTOS mailboxes for information transmission
- Tuned version carefully hand-tuned to used shared memory
- Power can be reduced at a cost
  - Increased application software complexity
  - Decreased flexibility

Ethernet optimization effects



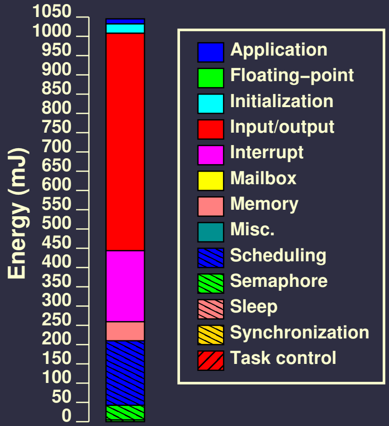
- Determined that synchronization routine cost was high
  - Used RTOS buffering to amortize synchronization costs
- 20.5% energy reduction
- 0.2% power reduction
- RTOS directly accounted for 1% of system energy
  - Energy savings due to improved RTOS use, not reduced RTOS energy

Mailbox example



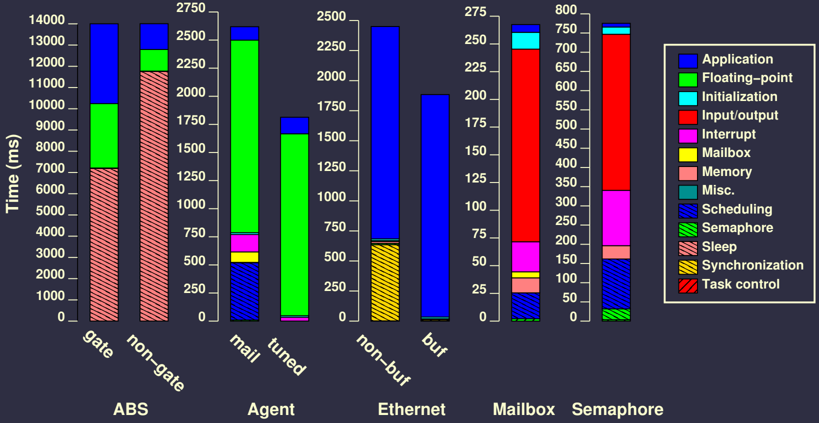
- Rapid mailbox communication between tasks
- RTOS directly accounted for 99% of system energy

Semaphore example



- Semaphores used for task synchronization
- RTOS directly accounted for 98.7% of system energy

Time results



Energy bounds

Service	Minimum energy (µJ)	Maximum energy (µJ)
AgentTask	3.41	4727.88
fptodp	17.46	49.72
BSPInit	3.52	3.52
fstat	16.34	16.34
CPUInit	287.15	287.15
fstat_r	31.26	31.26
GetPsr	0.38	0.55
init_bss	2.86	3.07
GetTbr	0.40	0.53
init_data	4.23	4.37
InitTimer	2.53	2.53
init_timer	18012.10	20347.00
OSCtxSw	46.63	65.65
init_tvecs	1.31	1.31
OSDisableInt	0.84	1.31
...	...	...

Semaphore example hierarchical call tree

		Function	$\frac{\text{Energy}(\mu J)}{\text{invocation}}$	Energy (%)	Time (ms)	Calls
realstart 25.40 mJ total 2.43 %	init_tvecs		1.31	0.00	0.00	1
	init_timer 18.01 mJ total 1.72 %	liteled	4.26	0.00	0.00	1
	startup 7.39 mJ total 0.71 %	do_main	7363.11	0.70	5.57	1
		save_data	5.08	0.00	0.00	1
		init_data	4.23	0.00	0.00	1
		init_bss	2.86	0.00	0.00	1
		cache_on	8.82	0.00	0.01	1
Task1 508.88 mJ total 48.69 %	win_unf_trap		6.09	1.16	9.43	1999
	OSDisableInt		0.98	0.09	0.82	1000
	OSEnableInt		1.07	0.10	0.92	1000
	OSSemPend 104.59 mJ total 10.01 %	win_unf_trap	6.00	0.57	4.56	999
		OSDisableInt	0.94	0.18	1.56	1999
		OSEnableInt	0.94	0.18	1.56	1999
		OSEventTaskWait	13.07	1.25	9.89	999
		OSSched	66.44	6.35	51.95	999
	OSSemPost 9.82 mJ total 0.94 %	OSDisableInt	0.96	0.09	0.78	1000
		OSEnableInt	0.98	0.09	0.81	1000
	OSTimeGet 4.62 mJ total 0.44 %	OSDisableInt	0.84	0.08	0.66	1000
		OSEnableInt	0.98	0.09	0.81	1000
	CPUInit 0.29 mJ total 0.03 %	BSPInit	3.52	0.00	0.00	1
		exceptionHandler	15.51	0.02	0.17	15
	printf 368.07 mJ total 35.22 %	win_unf_trap	6.18	0.59	4.87	1000
		vfprintf	355.04	33.97	257.55	1000

Example power-efficient change to RTOS

- Small changes can greatly improve RTOS power consumption
- µC/OS-II tracks processor loading by incrementing a counter when idle
- However, this is not a good low-power design decision
- NOPs have lower power than add or increment instructions
- Sleep mode has *much* lower power
- Can disable loading counter and use NOPs or sleep mode

Example power-efficient change to RTOS

- Alternatively, can use timer-based sampling
  - Normally NOP or sleep when idle
  - Wake up on timer ticks
  - Sample highest non-timer ISR task
  - If it's the idle task, increment a counter
  - Can dramatically reduce power consumption without losing functionality

RTOS Conclusions

- Demonstrated that RTOS significantly impacts power
- RTOS power analysis can improve application software design
- Applications
  - Low-power RTOS design
  - Energy-efficient software architecture
  - Consider RTOS effects during system design

Reference

Kaushik Ghosh, Bodhisattwa Mukherjee, and Karsten Schwan. A survey of real-time operating systems. Technical report, College of Computing, Georgia Institute of Technology, February 1994

Outline

1. Reliable embedded system design and synthesis
2. Realtime systems
3. Scheduling
4. Overview of real-time and embedded operating systems
5. Embedded application/OS time, power, and energy estimation
6. Homework

Scheduling and reliability reading

- Due 27 September: C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of the ACM*, 20(1):46–61, January 1973.
- Due 29 September: Robert P. Dick. Reliability, thermal, and power modeling and optimization. In *Proc. Int. Conf. Computer-Aided Design*, pages 181–184, November 2010.
- Due 4 October: Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *J. of Parallel and Distributed Computing*, 59(3):381–422, 1999.
- Due 6 October: L. Yang, Robert P. Dick, Haris Lekatsas, and Srimat Chakradhar. High-performance operating system controlled on-line memory compression. *ACM Trans. Embedded Computing Systems*, 9(4):30:1–30:28, March 2010.

Upcoming topic

Embedded system memory hierarchies.