Real-Time on General Purpose Systems

Real-Time and Embedded Systems (M)

Lecture 12

UNIVERSITY
of
GLASGOW

Lecture Outline

- Real-time on general purpose systems
- Need for flexible applications
- Implementation strategies
- Scheduling

Material corresponds to parts of chapters 10 and 12 of Liu's book

Real-Time on General Purpose Systems

- Many real-time systems built using a general purpose operating system, not an RTOS
  - Internet telephony; streaming audio and video; set-top boxes running Linux
  - DVD player software
- Operating system may provide limited real-time support, but not engineered for robust real-time operation, with many sources of unpredictability
  - Virtual memory and/or disk activity
  - Limited timer resolution
  - Limited scheduler granularity
- Need to engineer applications around these constraints
  - Consider how to make your application flexible

Flexible Computation

- Some real-time applications must tolerate fluctuation in available resources or workload
  - A real-time network server may receive more traffic than expected
  - A failure may divert load onto a backup system
  - Real-time performance may degrade due to load from non-real-time tasks sharing the processor
- A real-time system has two degrees of flexibility when it becomes impossible to meet all deadlines
  - Graceful degradation in timeliness
  - Graceful degradation in quality

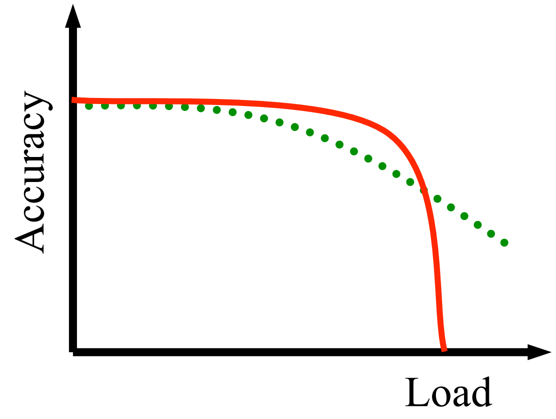
Flexible Computation: Timeliness

- A task has an (l, L) deadline if at least $l \geq 0$ jobs among any consecutive set $L \geq l$ must complete before their deadline
  - The parameter L is the *failure window* of the task; clearly a spectrum of requirements
  - A hard real-time task has (1, 1) deadlines
  - A soft real-time task has $(0, L)$ deadlines

- Depending on the application, systems may degrade by relaxing their deadlines, allowing some tasks to complete late
  - Not generally desirable, but suitable for applications with fixed resource demands and flexible timing requirements
    - Example: a DVD player running on a general purpose operating system might pause if the system is overloaded, rather than dropping frames
  - Often requires statistical analysis of performance, to estimate probability of missing deadline

Flexible Computation: Quality

- Some applications can trade-off, at run time, quality of results for the amount of time and resources used to produce those results
- As a system moves into overload, it gracefully degrades rather than suddenly failing
- Assumption: a timely result of poor quality is better than a high quality, but late, result



- Examples:
  - A telephony application might prefer a brief glitch in output, rather than a pause that leaves the other party wondering what's happening
  - An air traffic control system should deliver a timely collision warning with estimated location, rather than an exact warning, delivered too late

Implementing Flexible Computation

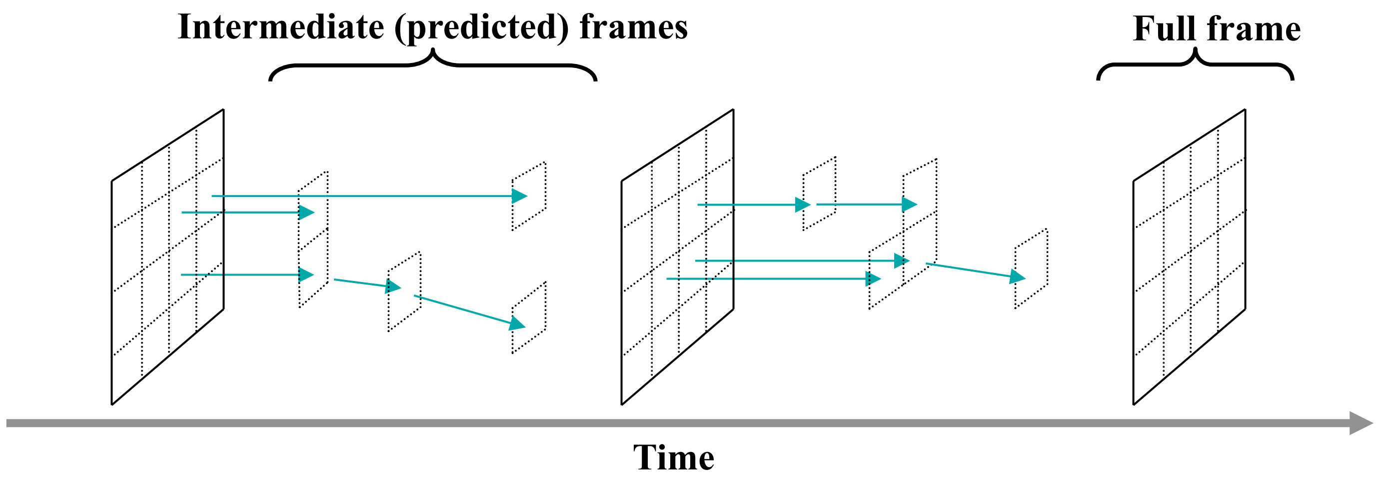
- Jobs are divided into an optional part and a mandatory part
  - With sufficient resources, both mandatory and optional parts complete; a *precise* result
  - With limited resources, the optional component is discarded, giving an *imprecise* result

- Assumption: possible to subdivide a job, produce meaningful approximate answers

- How to implement?
  - Sieve method
  - Milestone method
  - Multiple version method

Sieve Method

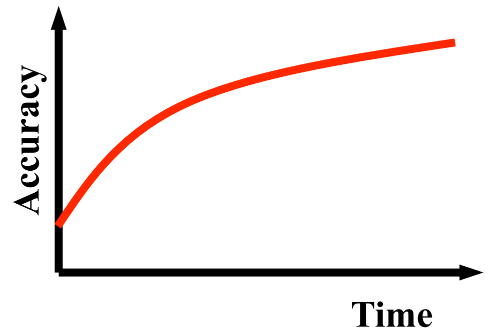
- A flexible task has a mixture of mandatory and optional jobs
- When overloaded, some optional jobs discarded
  - If they were optional, why include them in the system?
  - Useful for applications which periodically refresh state
- Example: video compression
  - Predicted frames can be discarded on overload



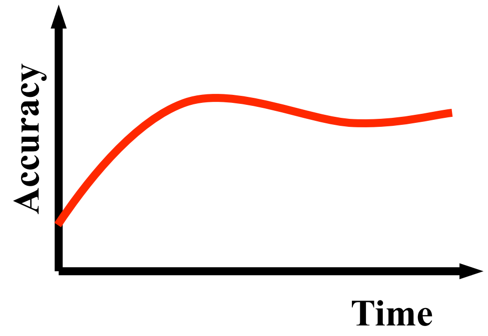
Milestone Method

- The system regularly checkpoints the result of the optional job as a set of *milestones*; when deadline reached, job terminates and latest milestone retrieved

- A *monotone* is a job with optional component that can be stopped any time; quality of result always increases with longer execution
  - Iterative numerical computation
  - Iterative statistical computation
  - Layered video encoding

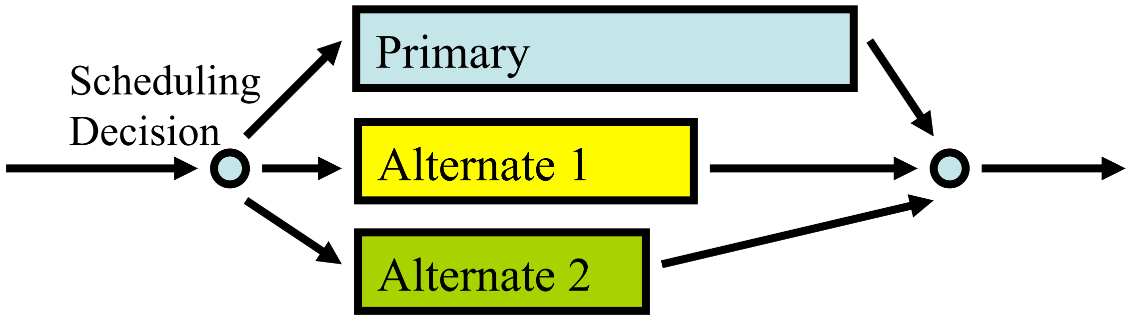


- Longer execution of a non-monotonic job may not improve results
  - E.g. approximation algorithms that don't always converge



Multiple Versions

- The flexible job can be implemented as multiple versions:
  - Primary is high quality, but has a larger execution time and resource usage
  - Alternates are lower quality, but execute quicker or use fewer resources
    - […or provide fault tolerance]



- The scheduler must make an a priori decision on which version to execute, based on load at the start of the job
  - Requires more intelligence in the scheduler than sieve or milestone methods
- Little gain from having more than one alternate

Implementing Flexible Computation

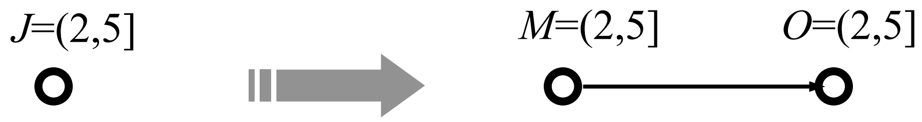
- Which is best?
  - Sieve method
  - Milestone method
  - Multiple version method

- It depends… sieve and multiple versions easiest to implement, milestones likely gives best results

- But: *highly* application dependent – what is the problem domain? What algorithm?

Workload Model

- To schedule flexible computations, need a workload model
- Definitions:
  - As usual a task, T, is comprised of a series of jobs J_i
  - Each flexible job, J_i, is logically decomposed into a chain of two jobs, M_i and O_i which are the mandatory and optional components
  - The release times and deadlines of M_i and O_i are the same as J_i but O_i is dependent on M_i
  - Execution time $e = e_m + e_o$



- A generalisation of the model used previously:
  - non-flexible jobs scheduled as-if e_o is zero

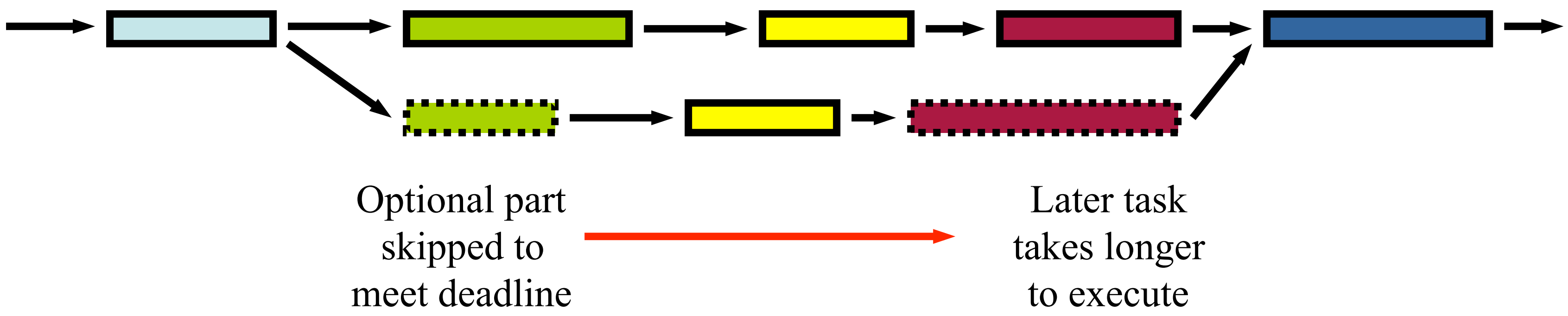
Workload Model

- Jobs are scheduled so mandatory tasks meet their deadline:
  - A schedule for a flexible application is *valid* if J_i is allocated processor time at least equal to e_m and at most equal to e
  - The schedule is *feasible* if each job is allocated at least e_m units of processor time before its deadline
  - Exactly the same definitions we saw in lecture 2 for non-flexible tasks, adapted to allow for e_o

- Optional components of each job execute if there is time before the deadline
  - An optional job completes it if receives e_o before the deadline
  - An optional job shouldn't execute beyond its deadline
    - May be terminated, and revert to the last milestone
    - May be pre-empted, and continue to execute at low priority if killing the job would leave the system inconsistent

Dependent Jobs

- Assumption: the execution time of a job is independent of the previous jobs
- In some systems, saving time in an early job – by skipping its optional component – makes a later job in the task take longer
  - Often occurs if errors are cumulative: eventually need to run the full computation periodically, to bring the error back to an acceptable level
- Need to take this into account when building the schedule, by modelling both branches of the task graph

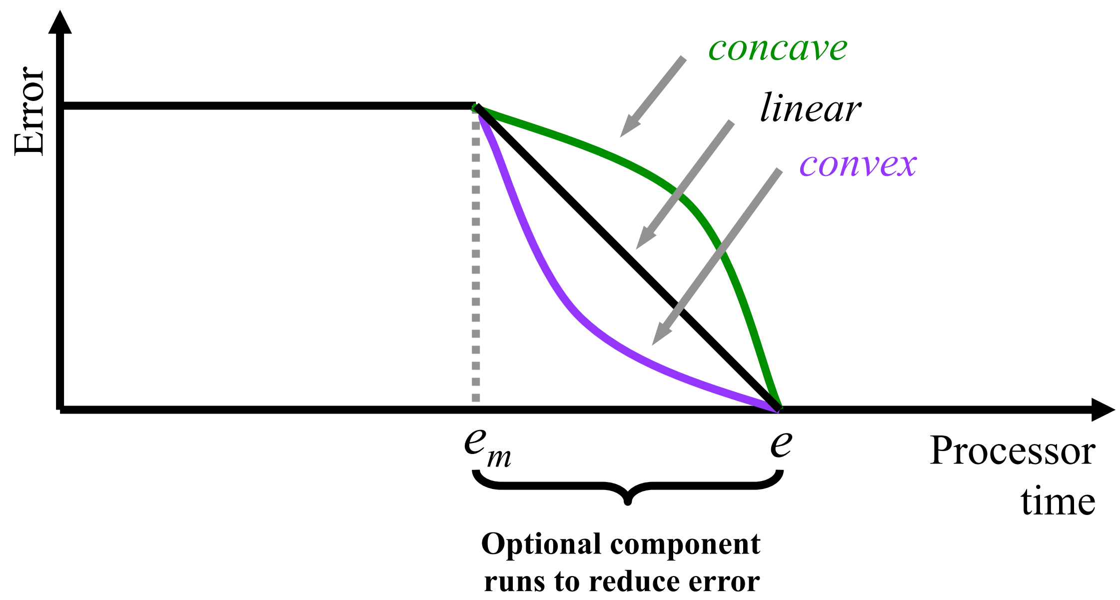


Jobs with 0/1 Constraints

- If the sieve or alternate methods used, no point running part of an optional component
  - The optional component has a *0/1 constraint*; either runs to completion, or not at all
  - For optional jobs according to the sieve method:
    - When the optional jobs becomes eligible to run, make a choice to run the job based on available execution time
  - For optional jobs according to the alternate method:
    - Model the alternates as mandatory and optional parts
    - Let e_m be execution time of the alternate, e_o be the difference in execution time between primary and alternate
    - After scheduling the mandatory part for e_m, the optional part is scheduled. If e_o available before its deadline, this corresponds to the primary version being scheduled. Otherwise, only the alternate can be scheduled

Criteria of Optimality

- Correctness: find a feasible schedule that ensures all mandatory jobs complete
- Quality of result: fit in as many optional jobs as possible, reduce error in the result
  - Measure the error according to some domain specific metric
  - Clearly desirable if the error function is convex; may influence choice of algorithm



Criteria of Optimality

Try to reduce the error in the result… which error:

- The sum of the total errors for all jobs?
- The maximum error for an individual job?
- The average error for all jobs?

Heavily application/domain dependent… no general guidelines

Scheduling Flexible Applications

- How to schedule flexible applications?
- Two approaches:
  - On-line
  - Off-line scheduling and/or heuristics

Off-line Scheduling

- Given a set of mandatory and optional tasks, an *off-line* algorithm aims to derive a static schedule that minimises some particular error metric
  - Can be executed during design, with hard coded schedule
  - Can be executed at run-time, as a result of a significant mode change that causes more tasks to run
- Generally reduces to linear programming/constraint optimisation problem
- Exponential time complexity, unrealistic for typical error functions
  - 0/1 constraints
  - non-linear error functions

On-line Heuristic Scheduling

- All useful scheduling algorithms for flexible applications use *heuristics* or are otherwise imprecise
- Two general approaches: mandatory first and slack stealing
  - *Mandatory first* algorithms schedule the mandatory parts of the system with higher priority than the optional parts
    - Use fixed priority algorithm, like rate monotonic, to schedule mandatory parts
    - Then schedule optional parts to minimise error:
      - dynamic least-attained-time suitable if error functions are convex, since diminishing returns for tasks that have attained most time
      - dynamic best-incremental-return suitable if knowledge of error functions, since run the task which will most reduce the error
    - If don't know error functions (common case):
      - Rate monotonic or earliest deadline schedule of optional parts
      - Earliest deadline always achieves zero average error, if possible
  - *Slack stealing* run optional tasks in slack time of mandatory tasks, dynamically according to EDF
  - Both seek to schedule mandatory parts as normal, fit in optional parts

Summary

- Flexible applications useful if system can be overloaded
- Typically only useful on soft real time systems, generally running on a general purpose operating system
  - Otherwise, engineer the system to avoid overload
  - Implication: don't have good scheduling support
    - Given knowledge of current time/deadline, application decides to shed work
      - sieve, incremental with milestones, alternate algorithm
    - Very much heuristic driven, rather than explicitly scheduled
    - Inherently imprecise, and difficult to reason about

- If you're building these systems:
  - program defensively
  - measure behaviour
  - adapt accordingly, based on domain specific heuristics and error functions