NICTA

THE UNIVERSITY OF NEW SOUTH WALES

# Real-Time Systems

Stefan M. Petters

NICTA Ltd. and CSE UNSW

smp@cse.unsw.edu.au

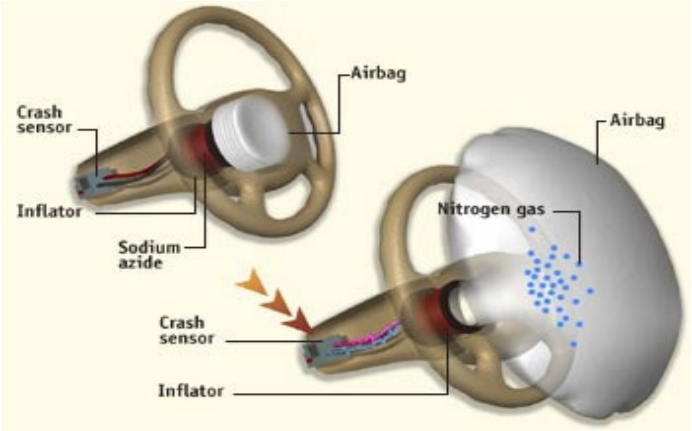
# Lecture Content

- Definition of Real-Time Systems (RTS)
- Scheduling in RTS
- Schedulability Analysis
- Worst Case Execution Time Analysis
- Time and Distributed RTS
- Rate Based Scheduling

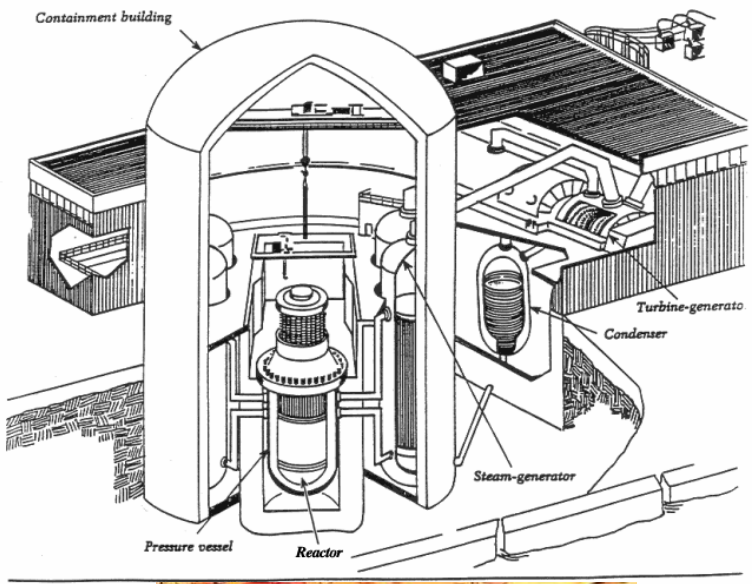
# Definition

- A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period
  - the correctness depends not only on the logical result but also the time it was delivered
  - failure to respond is as bad as the wrong response!

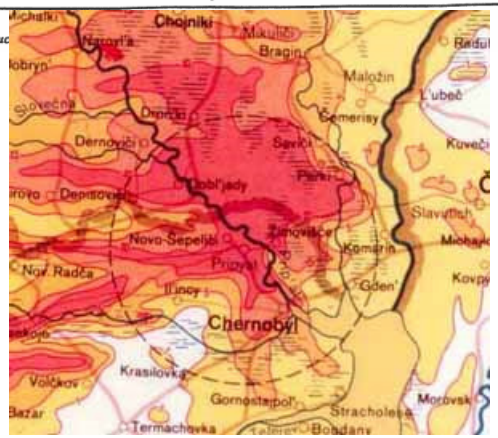
# Real-Time Systems



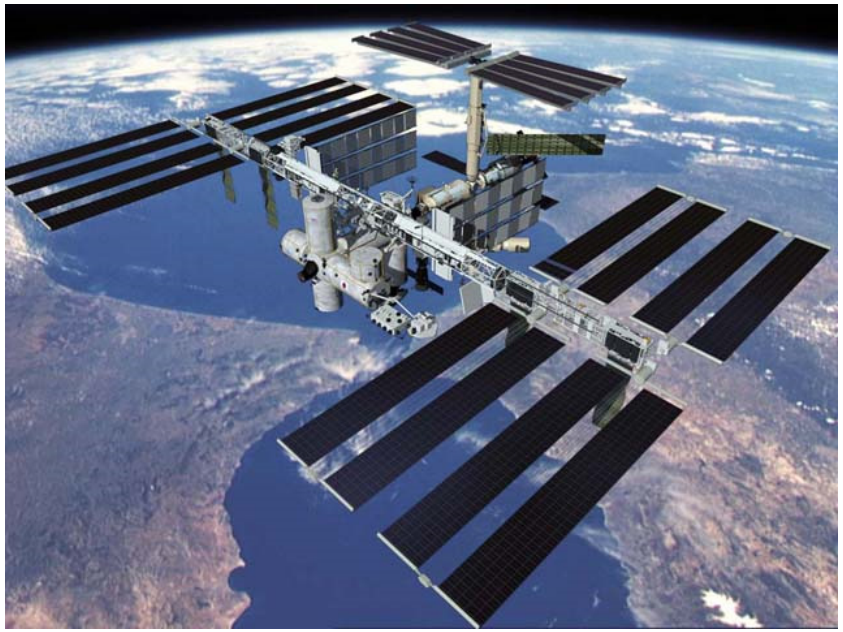
# Real-Time Systems



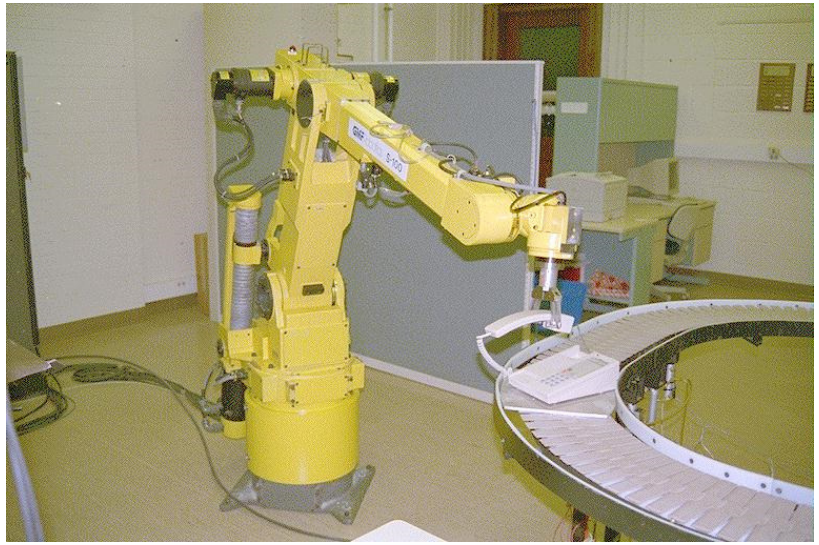

2 major parts of a nuc



# Real-Time Systems



# Real-Time Systems



# Real-Time Systems

NICTA

Is there a pattern?

- Hard real-time systems
- Soft real-time systems
- Firm teal-time systems
- Weakly hard real-time

- A deadline is a given time after a triggering event, by which a response has to be completed.
- Therac 25 example

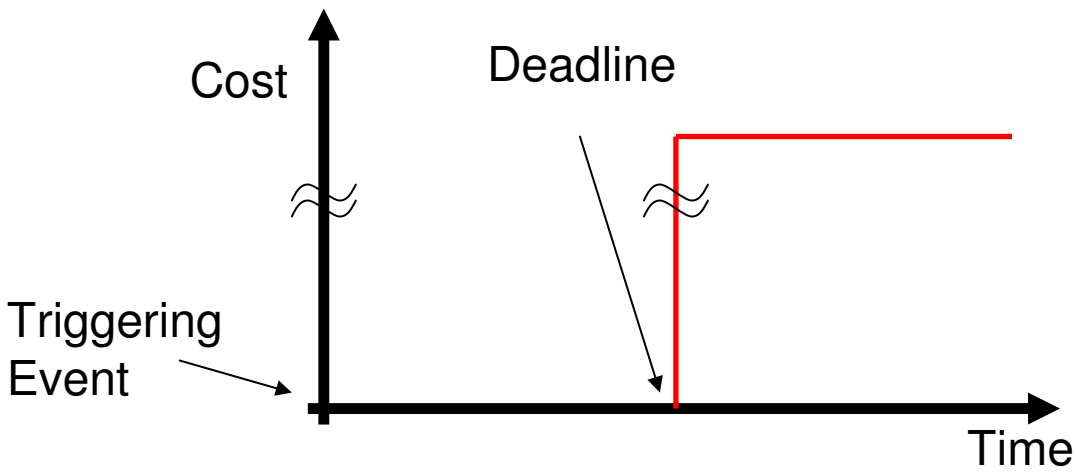
# What's needed of an RTOS

NICTA

- Fast context switches?
  - should be fast anyway
- Small size?
  - should be small anyway
- Quick response to external triggers?
  - not necessarily quick but predictable
- Multitasking?
  - often used, but not necessarily
- "Low Level" programming interfaces?
  - might be needed as with other embedded systems
- High processor utilisation?
  - desirable in any system (avoid oversized system)

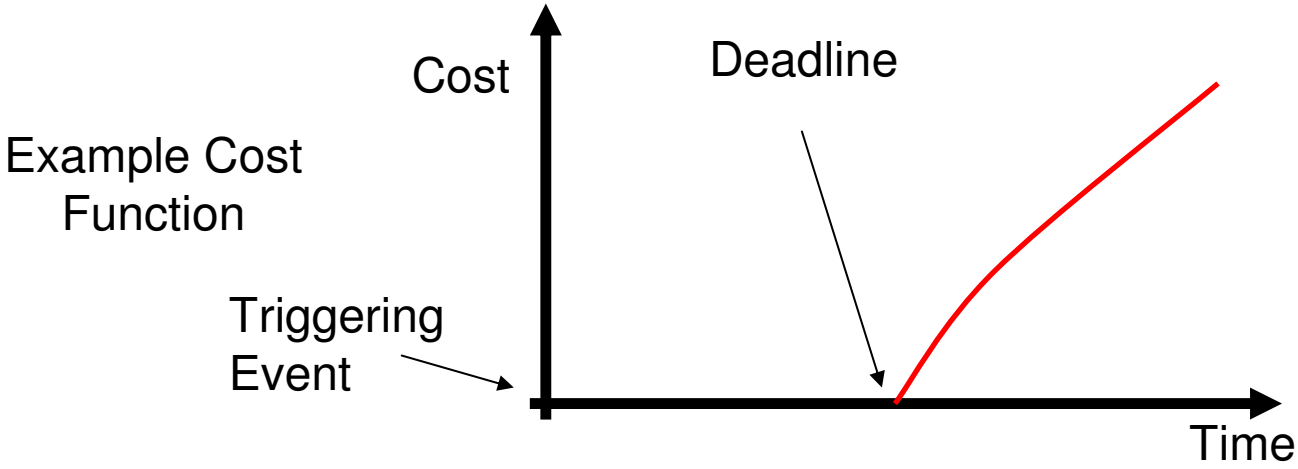
# Hard Real-Time Systems

- An overrun in response time leads to potential loss of life and/or big financial damage
- Many of these systems are considered to be safety critical.
- Sometimes they are “only” mission critical, with the mission being very expensive.
- In general there is a cost function associated with the system.



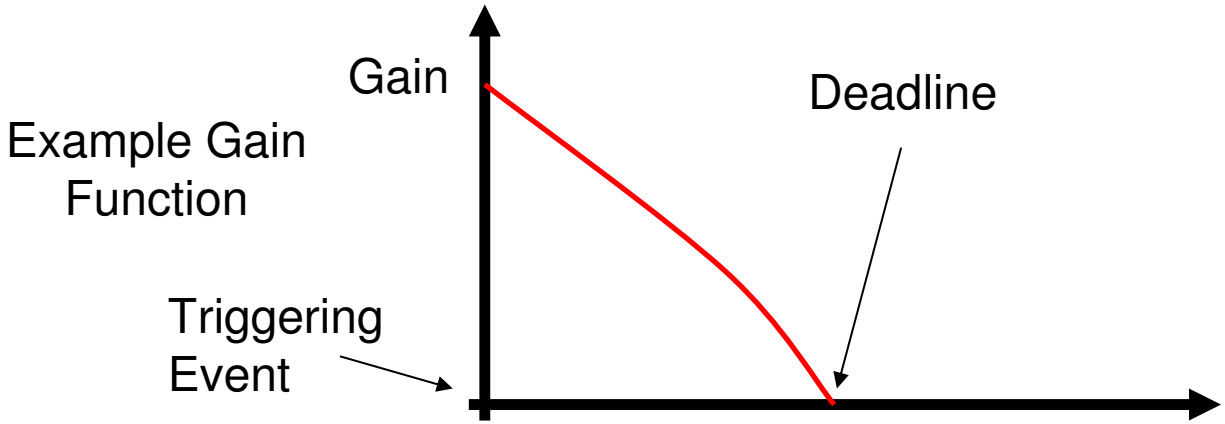
# Soft Real-Time

- Deadline overruns are tolerable, but not desired.
- There are no catastrophic consequences of missing one or more deadlines.
- There is a cost associated to overrunning, but this cost may be abstract.
- Often connected to Quality-of-Service (QoS)



# Firm Real-Time Systems

- The computation is obsolete if the job is not finished on time.
- Cost may be interpreted as loss of revenue.
- Typical example are forecast systems.



# Weakly Hard Real-Time Systems

NICTA

- Systems where *m* out of *k* deadlines have to be met.
- In most cases feedback control systems, in which the control becomes unstable with too many missed control cycles.
- Best suited if system has to deal with other failures as well (e.g. Electro Magnetic Interference EMI).
- Likely probabilistic guarantees sufficient.

# Non Real-Time Systems?

NICTA

- Yes, those exist!
- However, in most cases the (soft) real-time aspect may be constructed (e.g. acceptable response time to user input).
- Computer system is backed up by hardware (e.g. end position switches)
- Quite often simply oversized computers.

# Requirement, Specification, Verification

NICTA

- **Functional requirements**: Operation of the system and their effects.

- **Non-Functional requirements**: e.g., timing constraints.
  - F & NF requirements must be precisely defined and together used to construct the specification of the system.

- A **specification** is a mathematical statement of the properties to be exhibited by a system. It is abstracted such that
  - it can be checked for conformity against the requirement.
  - its properties can be examined independently of the way in which it will be implemented.

- The usual approaches for specifying computing system behavior entail enumerating events or actions that the system participates in and describing orders in which they can occur. It is not well understood how to extend such approaches for real-time constraints.
- F18 example

# Scheduling in Real-Time Systems

# Overview

- Specification and religious believes
- Preemptive vs. non preemptive scheduling
- Scheduling algorithms
- Message based synchronisation and communication
- Issues of mixed RT and non RT applications
- Overload situations
- Semaphores and blocking

# Requirements

- Temporal requirements of the embedded system
  - Event driven
    - Reactive sensor/actuator systems
    - No fixed temporal relation between events (apart from minimum inter arrival times)
  - Cyclic
    - Feedback control type applications
    - Fixed cycles of external triggers with minimal jitter
  - Mixed
    - Anything in between

# Specification

- Event triggered systems:
  - Passage of a certain amount of time
  - Asynchronous events
- Time triggered systems:
  - Predefined temporal relation of events
  - Events may be ignored until it's their turn to be served
- Matlab/Simulink type multi rate, single base rate systems:
  - All rates are multiples of the base rate
- Cyclic
  - feedback control loop

# Task Model

- Periodic tasks
  - Time-driven. Characteristics are known a priori
  - Task $\tau_i$ is characterized by $(T_i, C_i)$
  - E.g.: Task monitoring temperature of a patient in an ICU.
- Aperiodic tasks
  - Event-driven. Characteristics are not known a priori
  - Task $\tau_i$ is characterized by $(C_i, D_i)$ and some probabilistic profile for arrival patterns (e.g. Poisson model)
  - E.g.: Task activated upon detecting change in patient's condition.
- Sporadic Tasks
  - Aperiodic tasks with known minimum inter-arrival time $(T_i, C_i)$

# Task Model

$C_i$ = Computation time (usually Worst-Case Execution Time, WCET)

$D_i$ = Deadline

$T_i$ = Period or minimum interarrival time

$J_i$ = Release jitter

$P_i$ = Priority

$B_i$ = Worst case blocking time

$R_i$ = Worst case response time

# Task Constraints

- Deadline constraint
- Resource constraints
  - Shared access (read-read), Exclusive access (write-x)
  - Energy
- Precedence constraints
  - $\tau_1 \Rightarrow \tau_2$: Task $\tau_2$ can start executing only after $\tau_1$ finishes its execution
- Fault-tolerant requirements
  - To achieve higher reliability for task execution
  - Redundancy in execution

# Preemption

- Why preemptive scheduling is good:
  - It allows for shorter response time of high priority tasks
  - As a result it is likely to allow for a higher utilisation of the processor before the system starts missing deadlines
- Why preemptive scheduling is bad:
  - It leads to more task switches then necessary
  - The overheads of task switches are non-trivial
  - The system becomes harder to analyse whether it is able to meet all its deadlines
  - Preemption delay (cache refill etc.) becomes more expensive with modern processors
- Cooperative preemption?
  - Applications allow preemption at given points
  - Reduction of preemptions
  - Increase of latency for high priority tasks

# Event Triggered Systems

NICTA

"... The asynchronous design of the [AFTI-F16] DFCS introduced a random, unpredictable characteristic into the system. The system became untestable in that testing for each of the possible time relationships between the computers was impossible. This random time relationship was a major contributor to the flight test anomalies. Adversely affecting testability and having only postulated benefits, asynchronous operation of the DFCS demonstrated the need to avoid random, unpredictable, and uncompensated design characteristics."

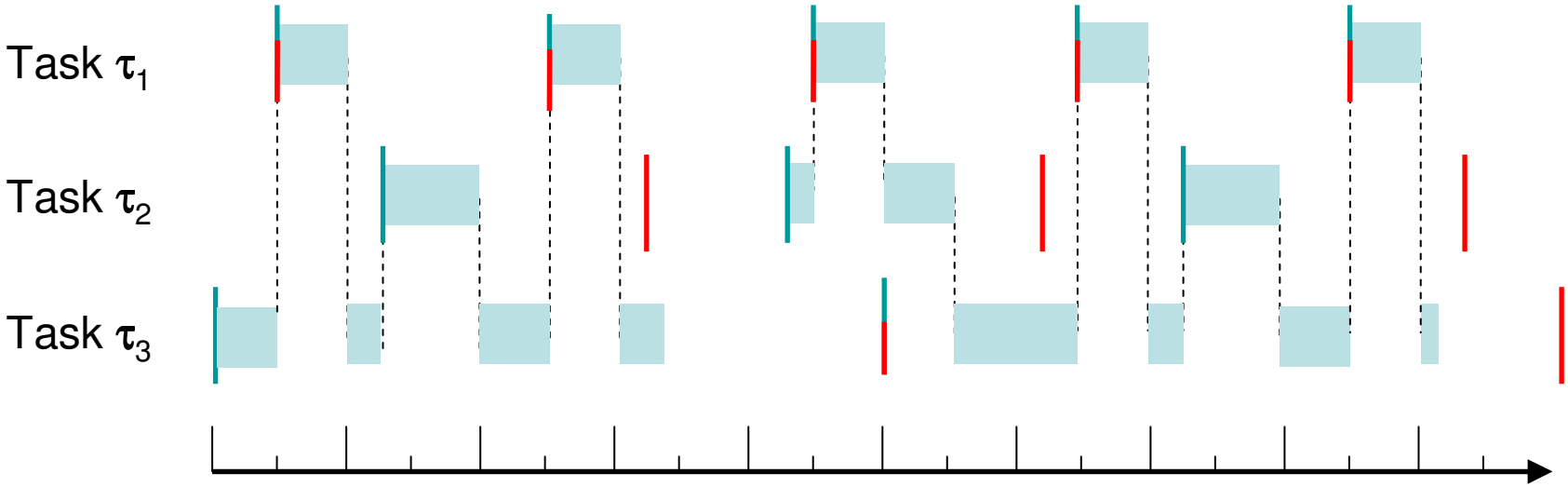
*D. Mackall, flight-test engineer AFTI-F16 AFTI-F16 flight tests*

# Fixed Priority Scheduling

NICTA

- Priorities may assigned by
  - Deadline: shortest deadline $\Rightarrow$ highest priority
  - Period: shortest period $\Rightarrow$ highest priority
  - "Importance"
- Scheduler picks from all ready tasks the one with the highest priority to be dispatched.
- Benefits:
  - Simple to implement
  - Not much overhead
  - Minimal latency for high priority tasks
- Drawbacks
  - Inflexible
  - Suboptimal (from analysis point of view)

# Fixed Priority Scheduling(FPS)



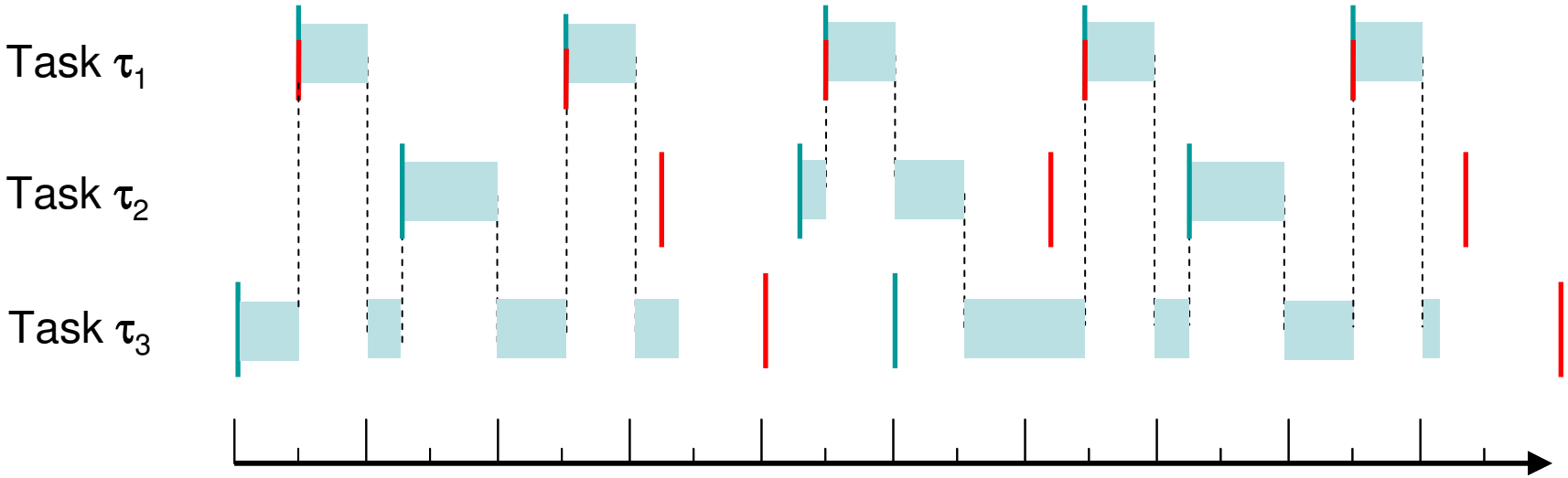

| | Priority | C | T | D |
|---|---|---|---|---|
| Task $\tau_1$ | 1 | 5 | 20 | 20 |
| Task $\tau_2$ | 2 | 8 | 30 | 20 |
| Task $\tau_3$ | 3 | 15 | 50 | 50 |

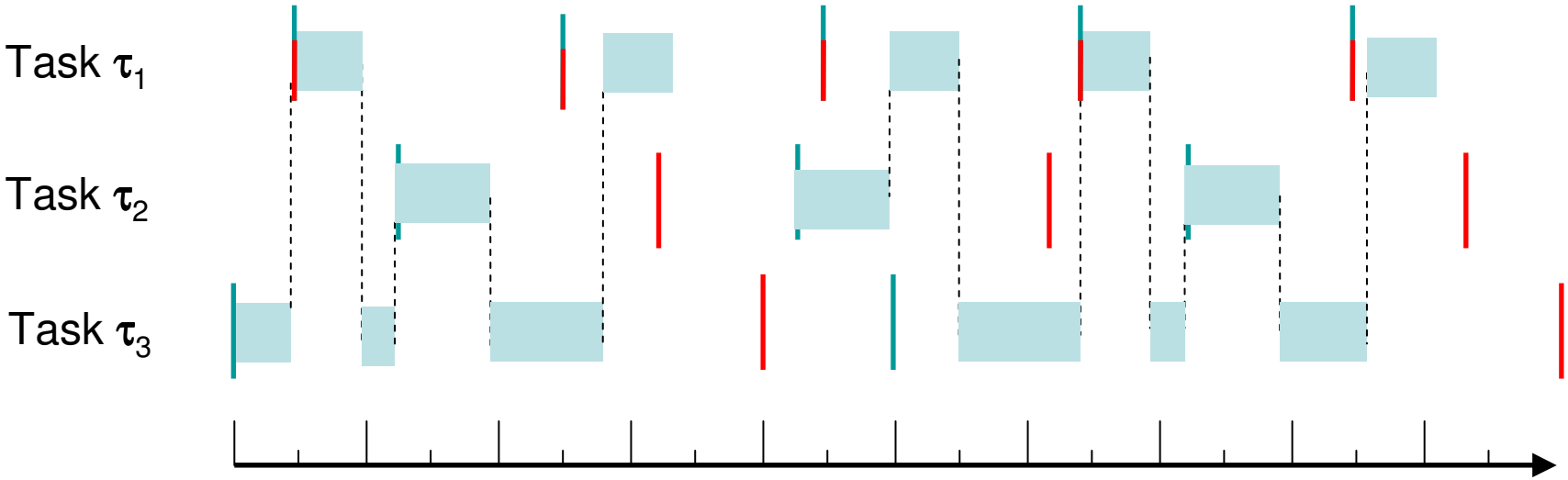
# Earliest Deadline First (EDF)

NICTA

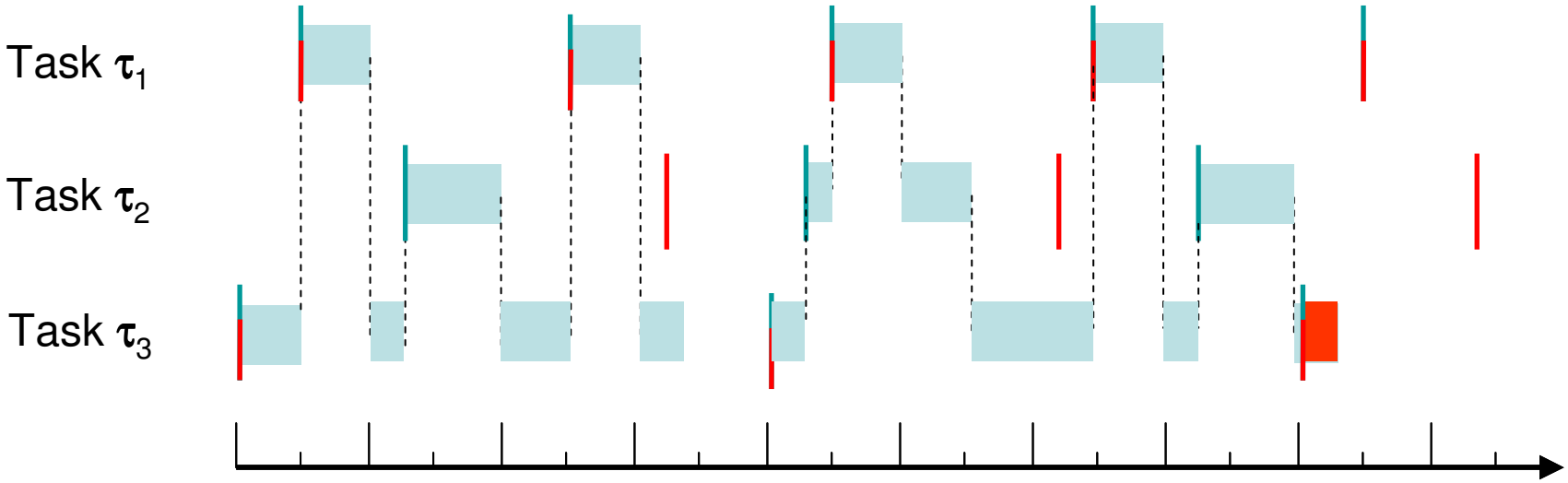
- Dynamic priorities
- Scheduler picks task, whose deadline is due next
- Advantages:
  - Optimality
  - Reduces number of task switches
  - Optimal if system is not overloaded
- Drawbacks:
  - Deteriorates badly under overload
  - Needs smarter scheduler
  - Scheduling is more expensive

# FPS vs. EDF



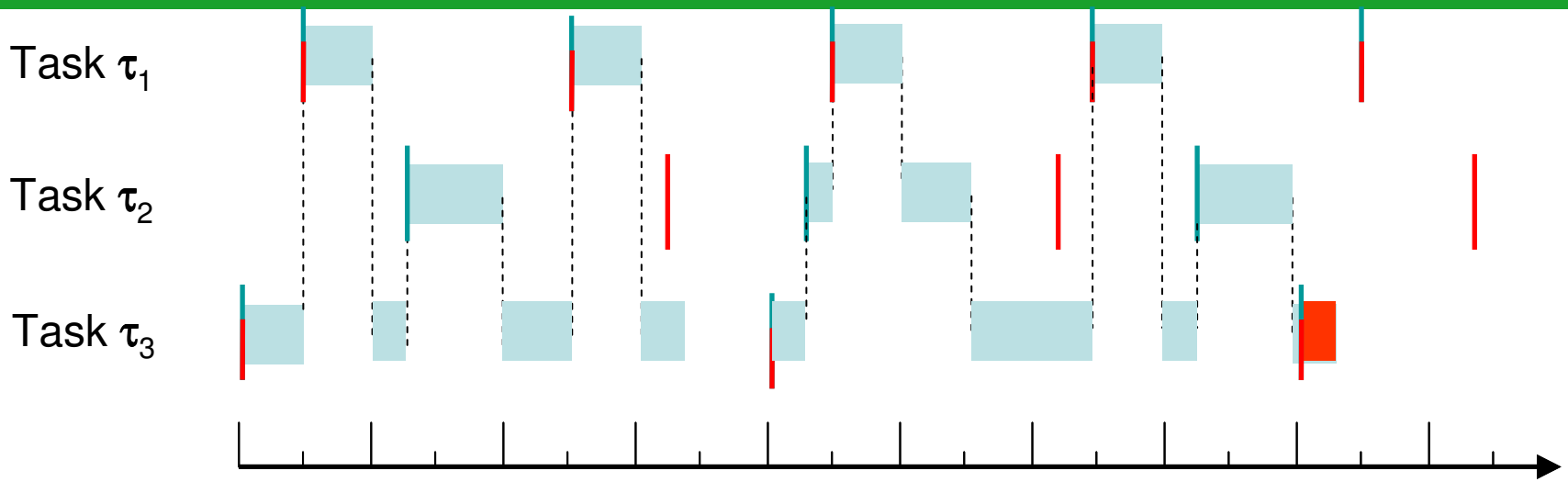



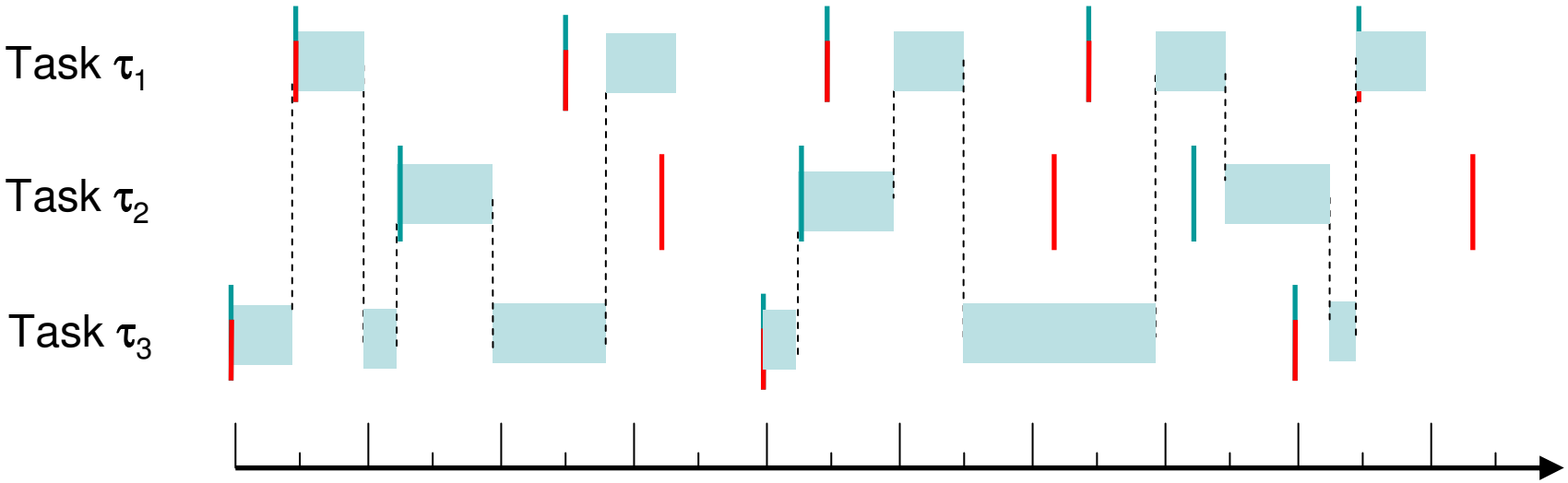
# FPS vs. EDF




| | Priority | C | T | D |
|---|---|---|---|---|
| Task $\tau_1$ | 1 | 5 | 20 | 20 |
| Task $\tau_2$ | 2 | 8 | 30 | 20 |
| Task $\tau_3$ | 3 | 15 | 40 | 40 |

# FPS vs. EDF



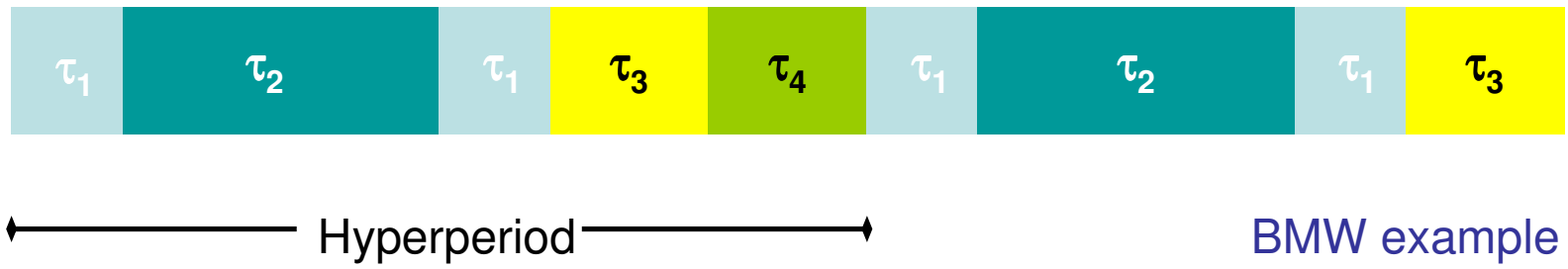
# Time Triggered/Driven Scheduling

NICTA

- Mostly static scheduling
- Time triggered scheduling allows easier reasoning and monitoring of response times
- Can be used to avoid preemption
- Can be used in event triggered systems, but increases greatly the latency
- Most often build around a base rate
- Can be implemented in big executive, using simple function calls

# Time Triggered Scheduling

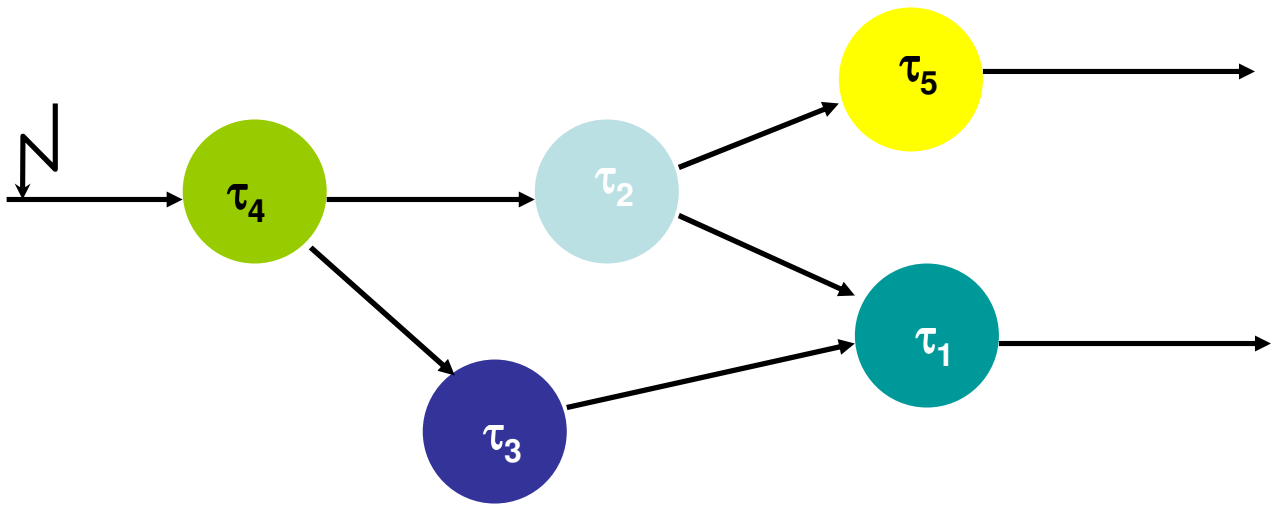
- Advantages:
  - Very simple to implement
  - Very efficient / little overhead (in suitable case)
- Disadvantages:
  - Big latency if event rate does not match base rate
  - Inflexible
  - Potentially big base rate (many scheduling decisions) or hyperperiod




BMW example

# Message Based Synchronisation

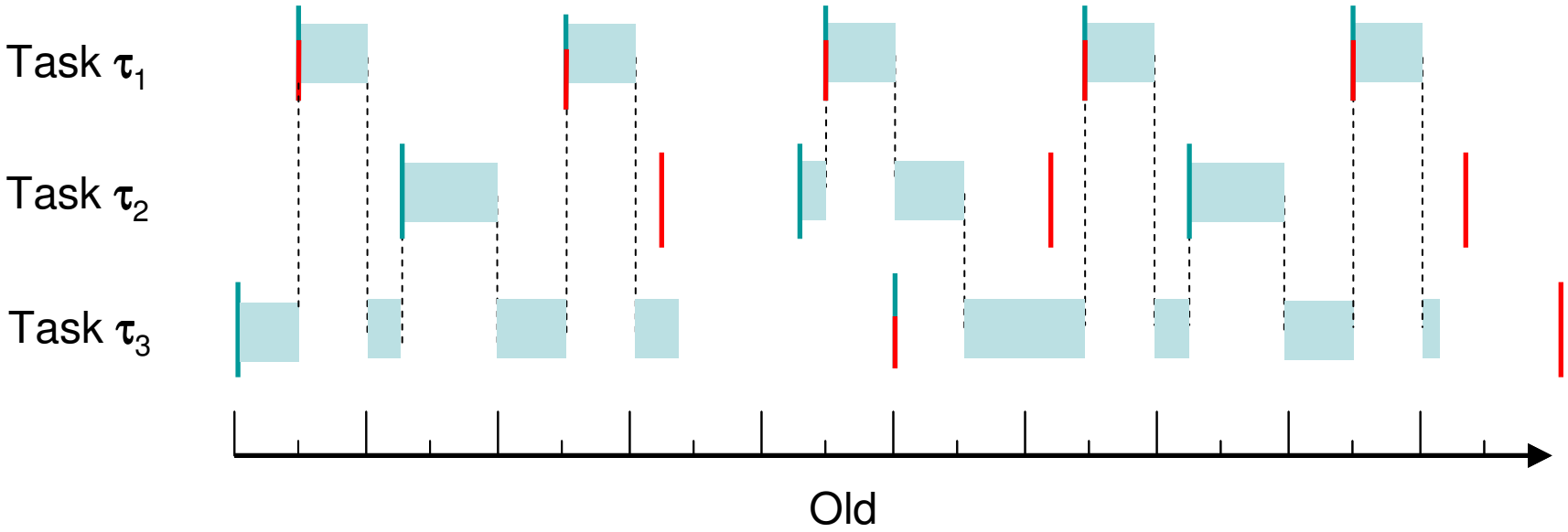
- Tasks communicate via messages
- Task wait for messages (blocked until message arrives)
- Suitable to enforce precedence relations
- Enables messages to be used to transport deadlines



# Overload Situations

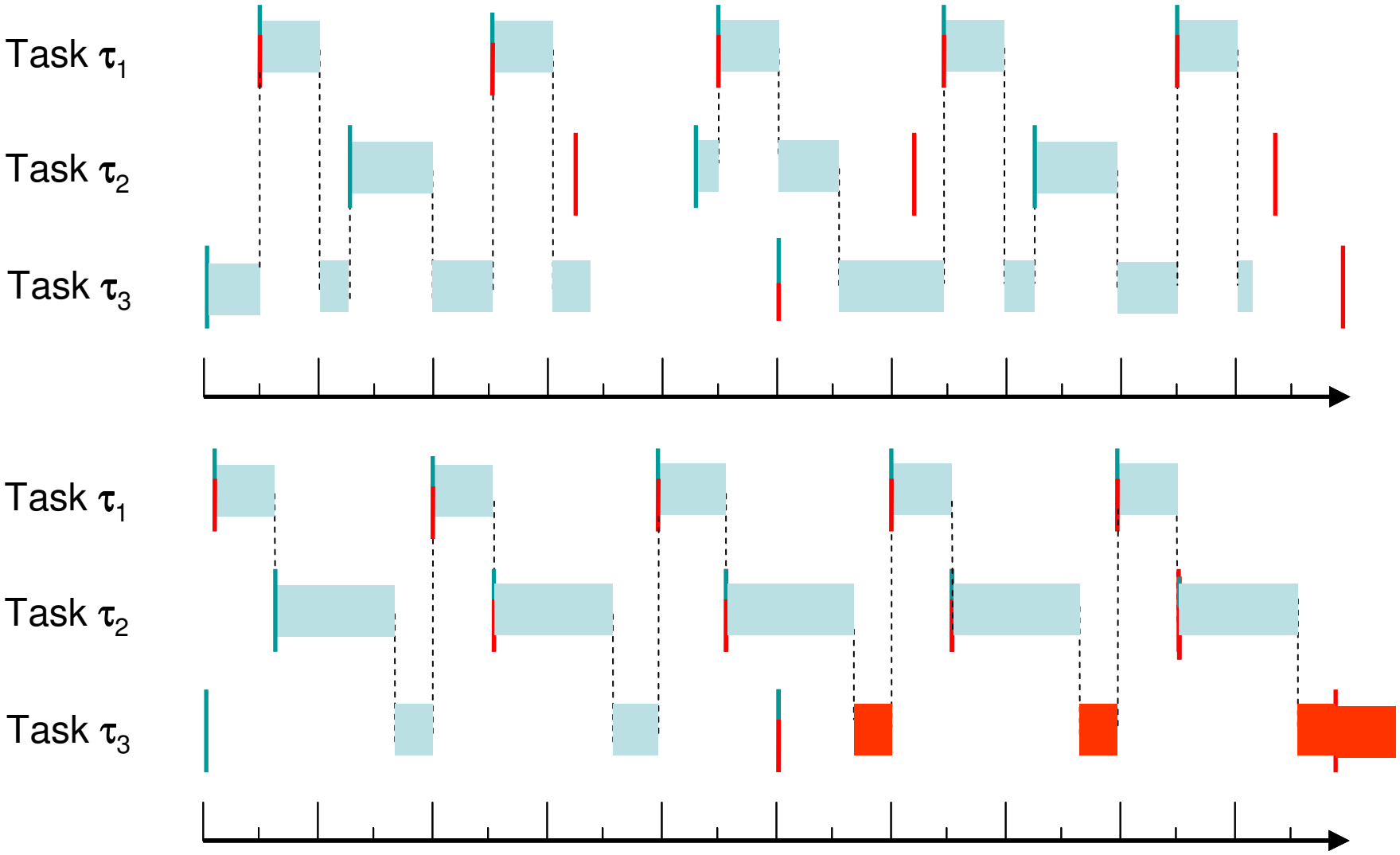
- Caused by faulty components of the system
  - Babbeling idiot or
  - A receiver part erroneously “receiving input”
  - EMI
- Or caused by wrong assumptions regarding the embedding environment
  - Basically wrong event rates or event correlation

# Overload Situations in FPS

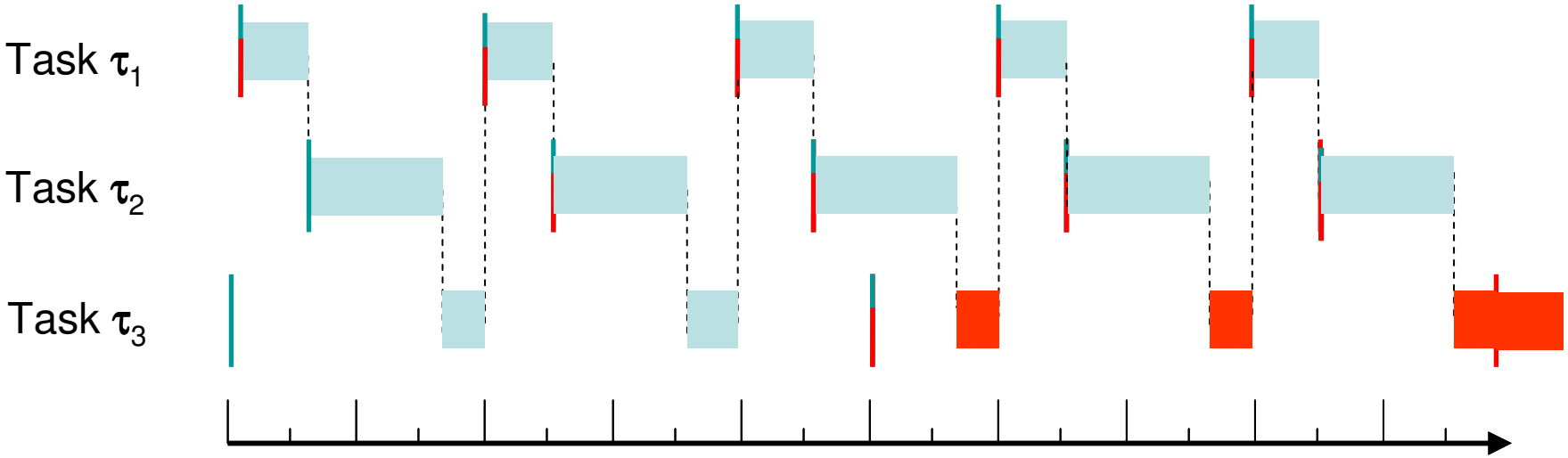


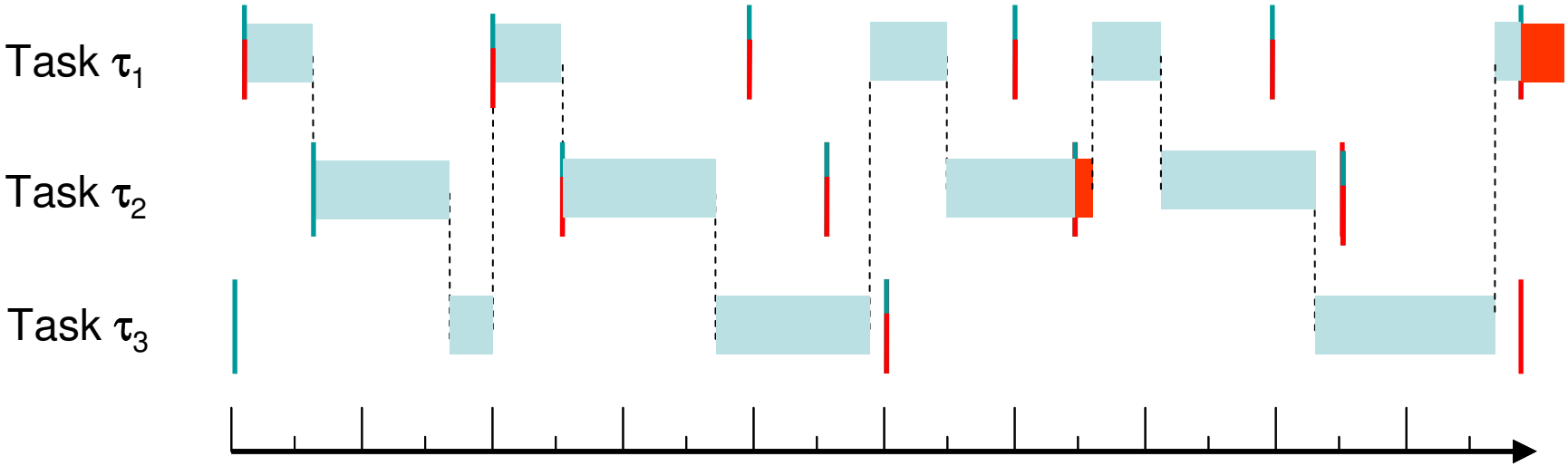

| | Priority | C | T | D |
|---|---|---|---|---|
| Task $\tau_1$ | 1 | 5 | 20 | 20 |
| Task $\tau_2$ | 2 | 12 | 20 | 20 |
| Task $\tau_3$ | 3 | 15 | 50 | 50 |

# Overload Situations in FPS

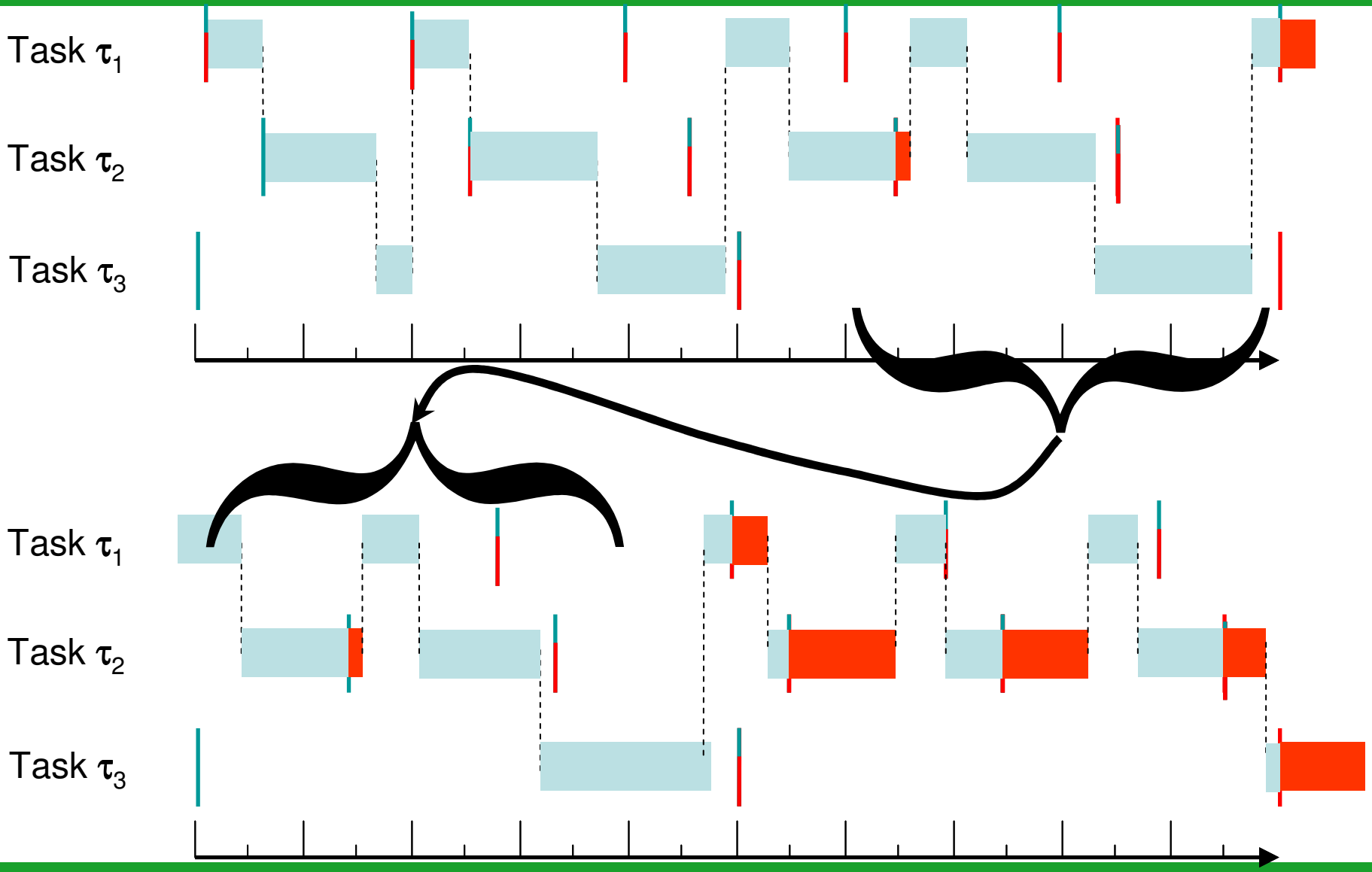


# Overload Situations in EDF



# Overload Situations in EDF



# Priority Inversion

NICTA

- Happens when task is blocked in acquiring semaphore from held by lower priority task which is preempted by medium priority task.
- Similar case for server tasks.
- Cure
  - Priority Ceiling or
  - Priority inheritance
- Pathfinder example

# Reflective/Feedback-based Scheduling

NICTA

- Adaptive systems
- By definition soft real time
- Adjusts scheduling based on information about change
- Capable of better coping with “the unknown”
- Connects quite well with adaptive applications

NICTA

THE UNIVERSITY OF
NEW SOUTH WALES

# Schedulability Analysis of Real-Time Systems

# Schedulability Analysis

NICTA

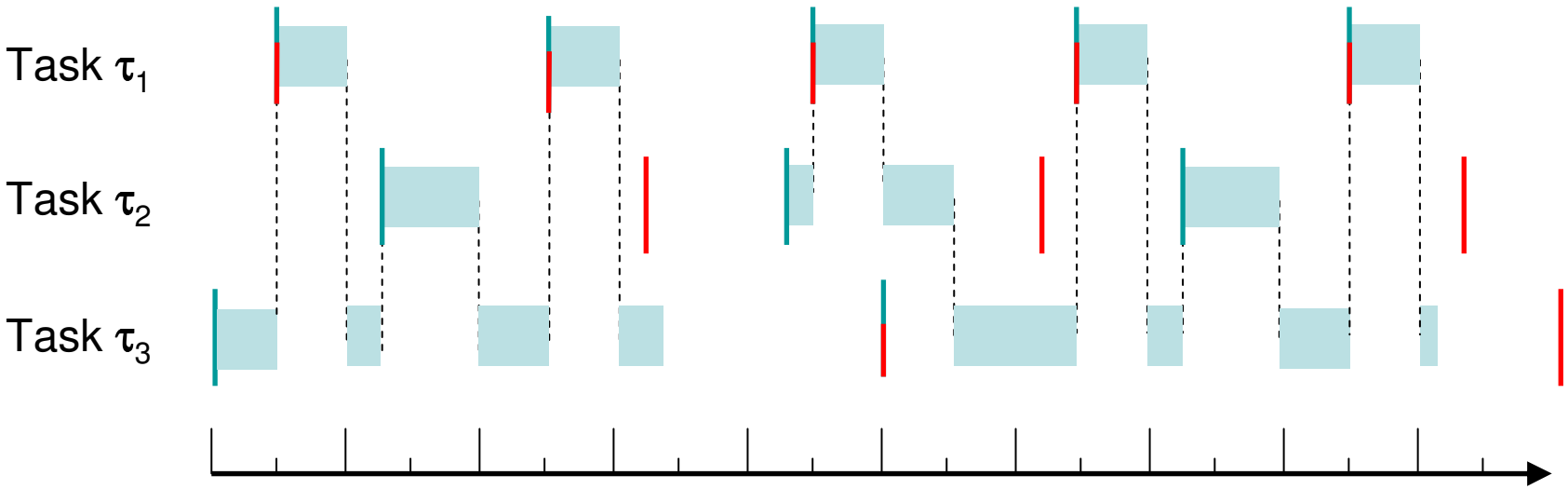
- Tries to establish, whether the task system described is actually schedulable
  - In the classical sense this is, whether all the deadlines are met under all circumstances;
  - Recent move to satisfaction of Quality-of-Service constraints;
- Relies on availability of computation time of tasks
  - WCET;
  - Execution time profiles.

# Critical Instant

- Trivial for independent tasks
  - All events happen at the same time;
  - However, implicitly consider all possible phases (take nothing for granted).
- However, get's more tricky (but tighter) having dependencies
  - What phasing of other activities produces the biggest load.
  - An activity is a string of tasks triggered by a single event.

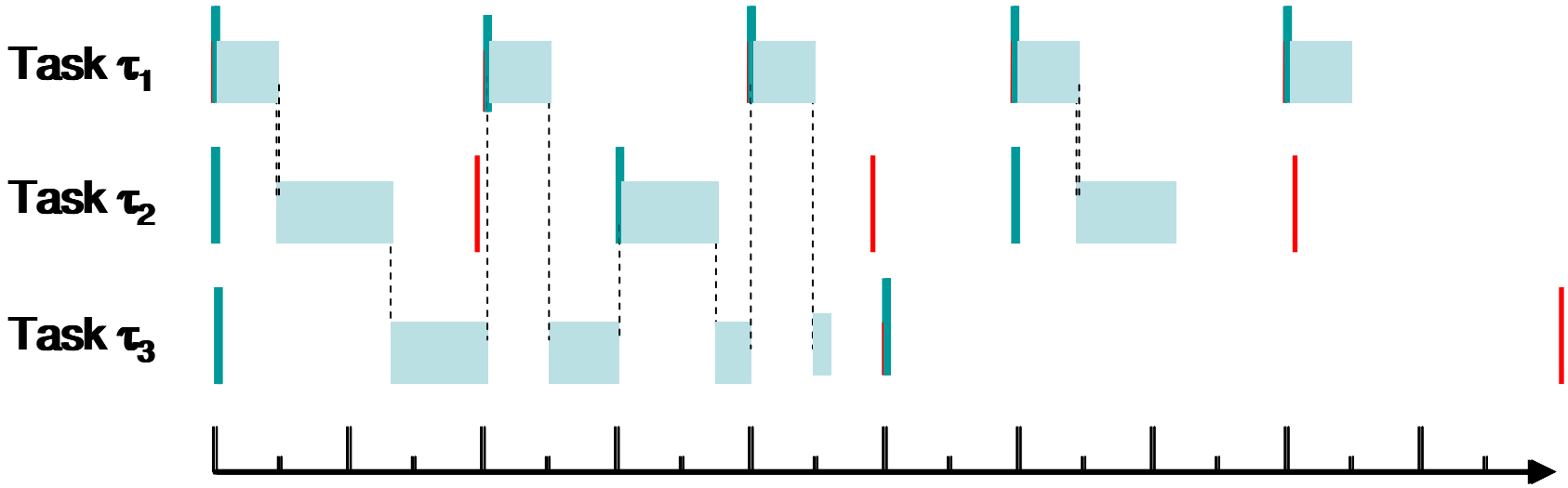
# Response Time Analysis

- Does not directly consider deadlines
- Makes the assumption of jobs being executed in order
- Usually used in fixed priority systems



# Response Time Analysis

| | Priority | C | T | D |
|---|---|---|---|---|
| Task $\tau_1$ | 1 | 5 | 20 | 20 |
| Task $\tau_2$ | 2 | 8 | 30 | 20 |
| Task $\tau_3$ | 3 | 15 | 50 | 50 |



# Formal RTA

- Assumptions j<i ⇒priority j is lower than priority i
- Critical instant
- Iterative process

$$w_i^0 = C_i$$

$$w_i^{n+1} = C_i + \sum_{\forall j<i} \left\lceil \frac{w_i^n}{T_j} \right\rceil * C_j$$

# Blocking Time and Other Nasties

- Blocking time
- Jitter
- Pre-emption delay

$$w_i^{n+1} = C_i + B_i + \sum_{\forall j<i} \left\lceil \frac{J_j + w_i^n}{T_j} \right\rceil * (C_j + \delta_{i,j})$$

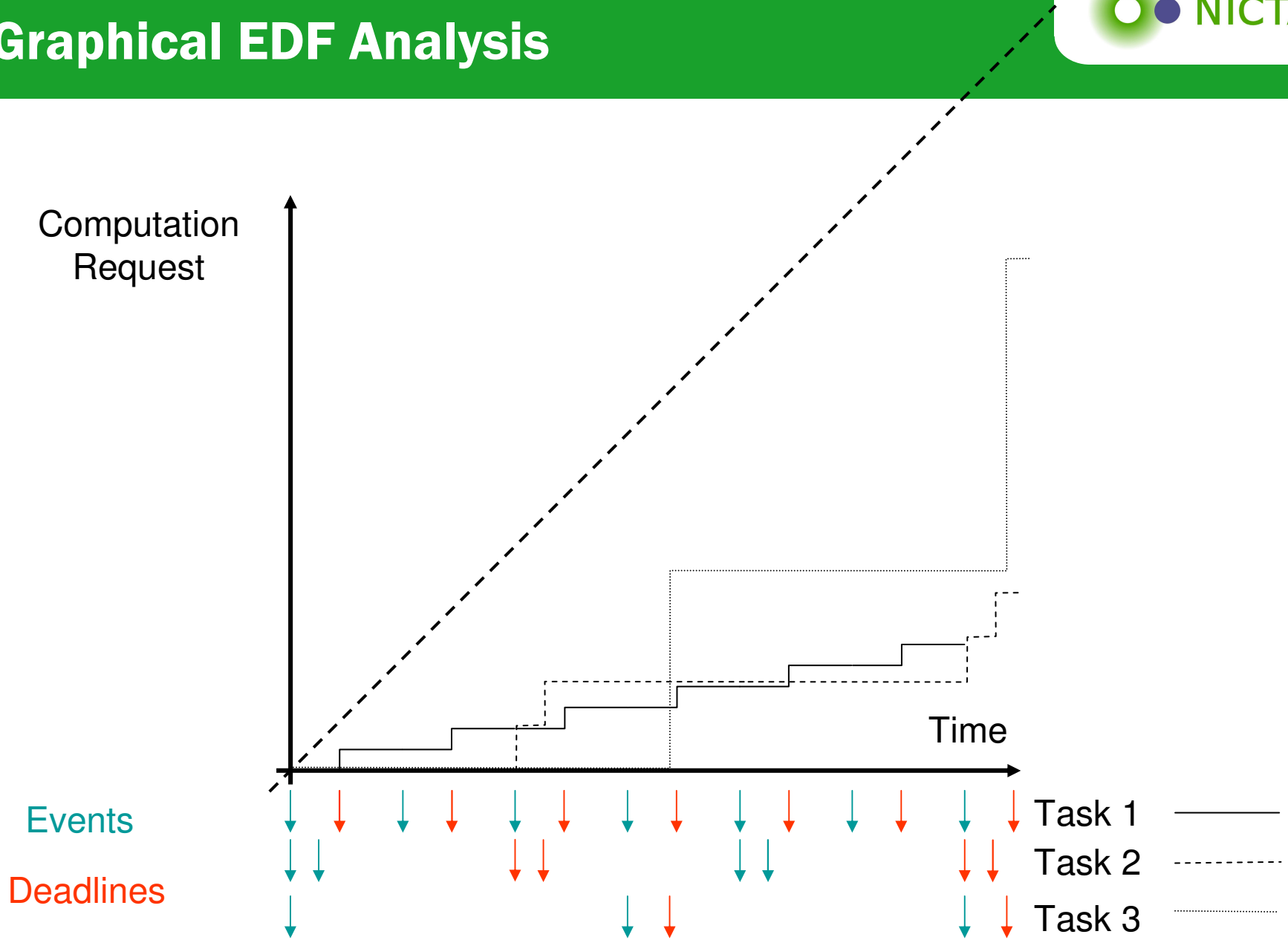
# Rate Monotonic Analysis

- Looks at *utilisation* do determine whether a task is schedulable
- Initial work had following requirements:
  - All tasks with deadlines are periodic
  - All tasks are independent of each other (there exists no precedence relation, nor mutual exclusion)
  - $T_i = D_i$
  - $C_i$ is known and constant
  - Time required for context switching is known
- Bound is given by:

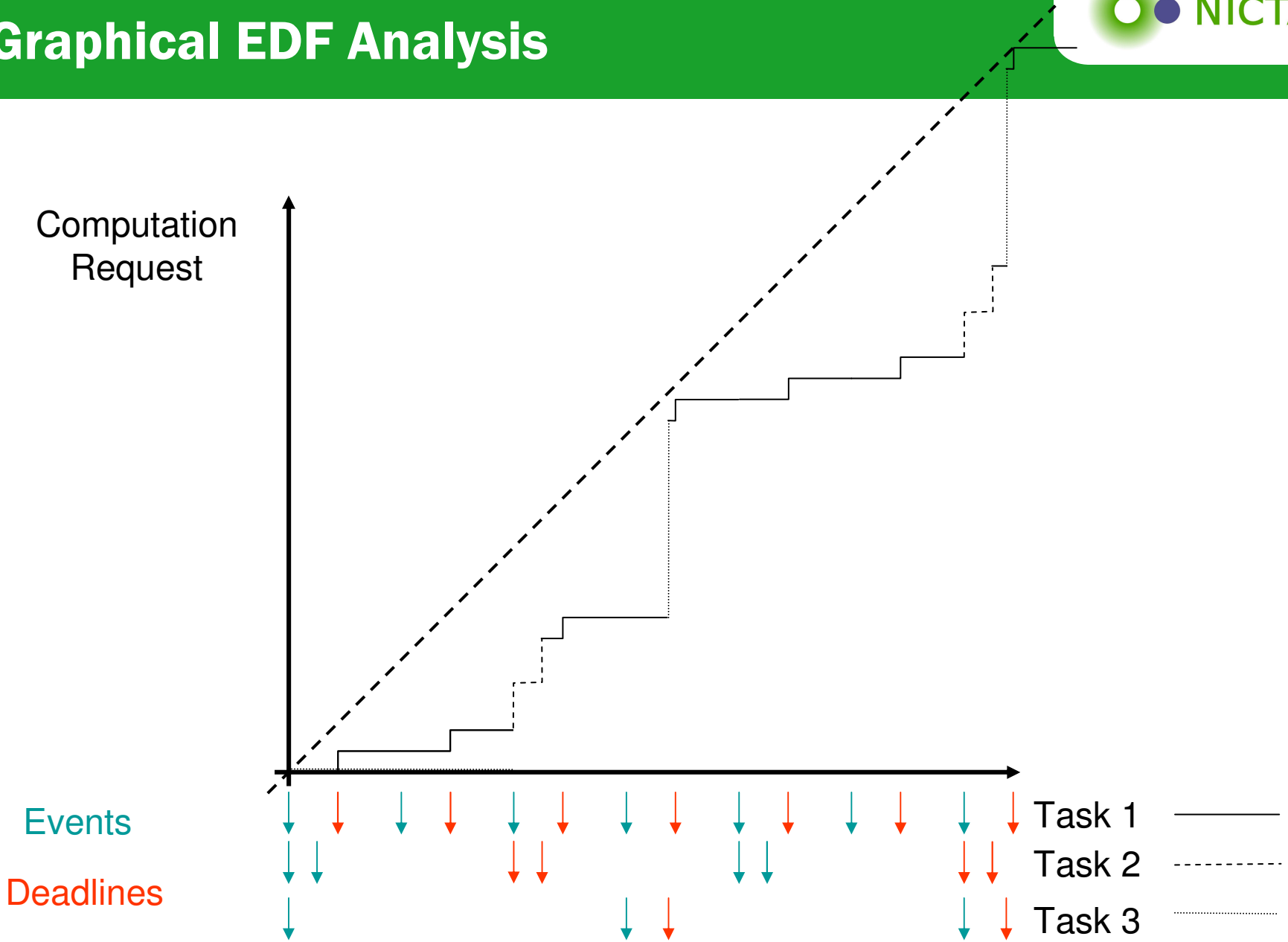
$$\mu = \sum_{\forall i} \left\lceil \frac{C_i}{T_i} \right\rceil \leq n * (2^{1/n} - 1)$$

- Has been relaxed in various ways, but still it is only an approximate technique.
- Further info can be found here:
  http://www.heidmann.com/paul/rma/PAPER.htm

# Graphical EDF Analysis



# Graphical EDF Analysis



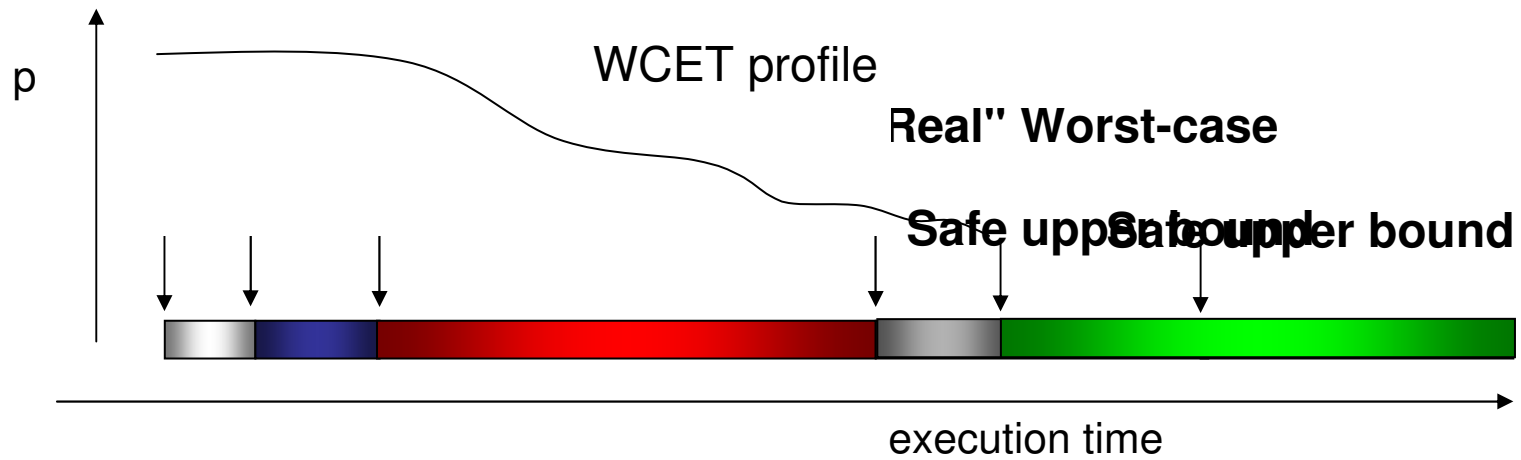
NICTA

THE UNIVERSITY OF
NEW SOUTH WALES

# Worst Case Execution Time Analysis

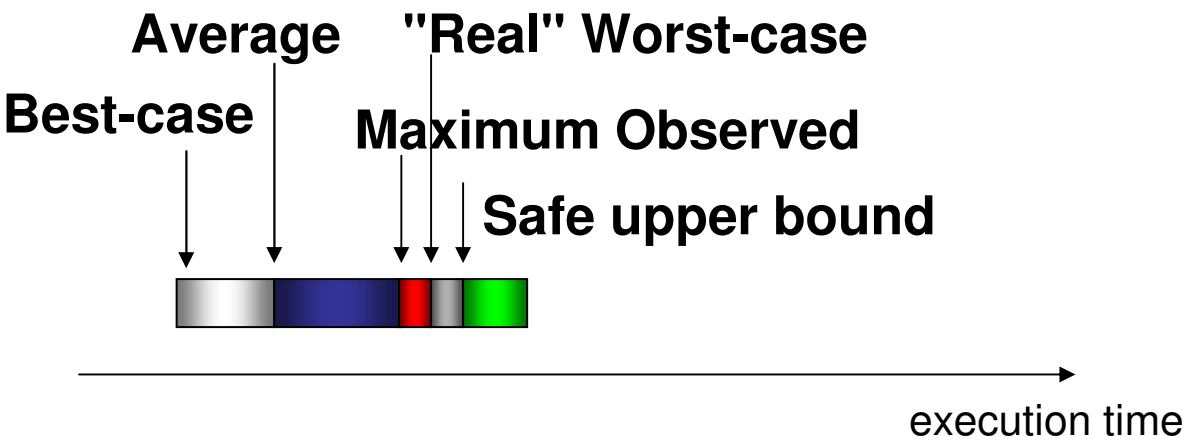
# Problem Definition

- All of the scheduling analysis presented previously requires the Worst-Case Execution time to be known
- Target is to come up with
  - a safe upper bound
  - as close as possible to the “real” worst case.
  - Ideally with more than just single number (probabilistic analysis)

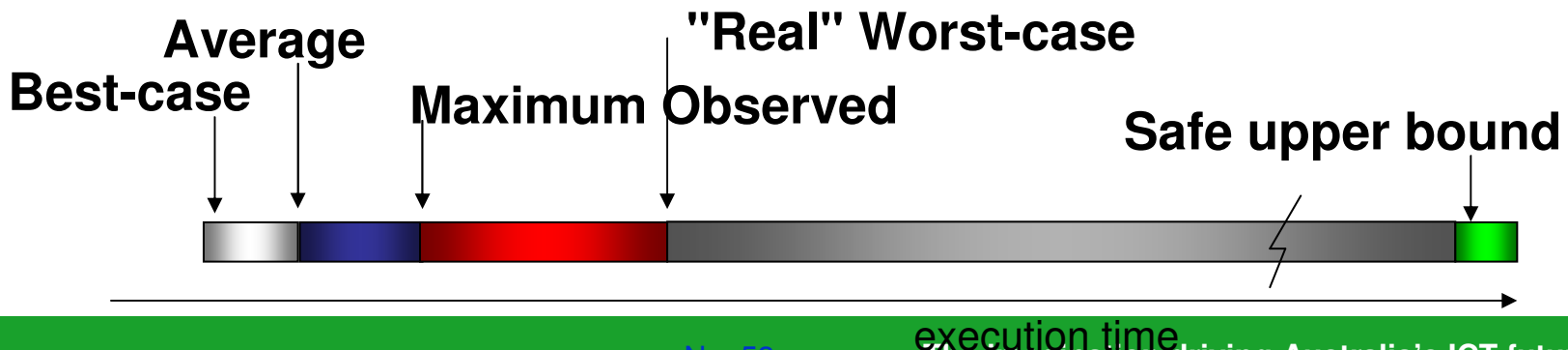


# Problem Definition contd

## Simple Code + Simple Processors




## Complex Code + Advanced processors



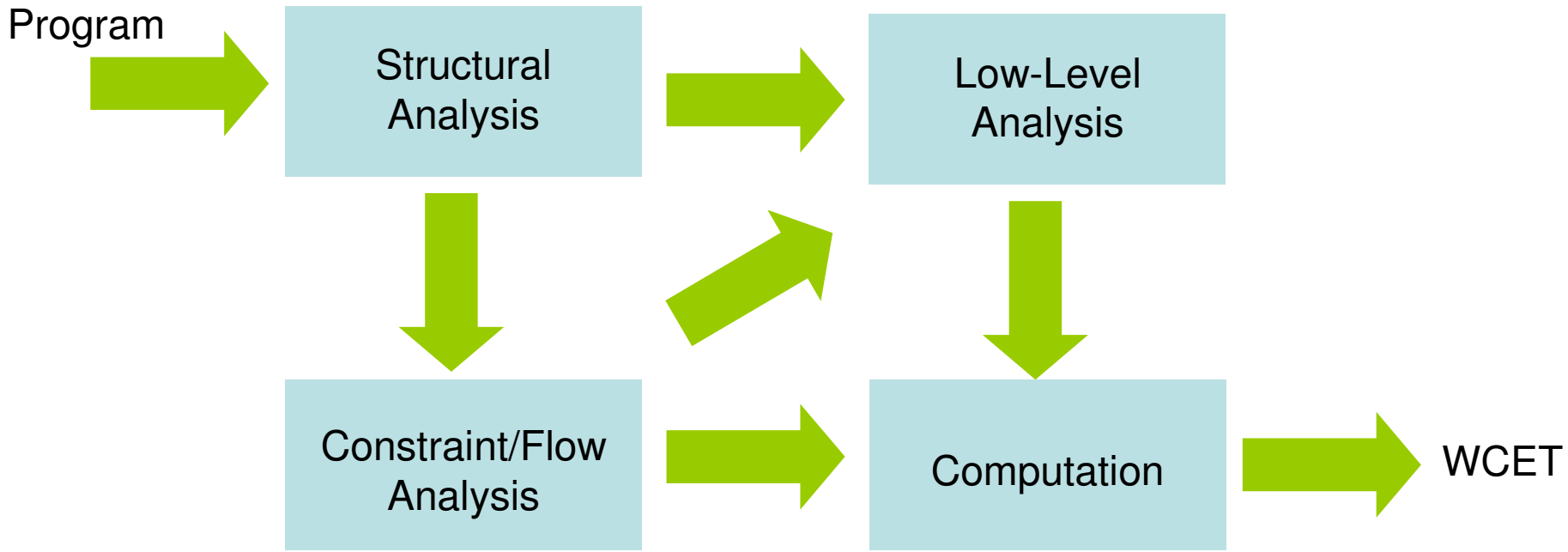
# Is it a Problem?

NICTA

- Safety critical computer systems exist and are deployed
- Yes, but …
  - Safety critical systems have been
    - highly restrictive in terms of HW/SW used
    - highly restrictive in terms of complexity
    - used a lot of manual inspection and pen and paper work
  - The above doesn't scale!
  - industry not in the safety critical arena have been using measurements with safety factors.
    - Worked fine with simple architectures, but doesn't work good with more advanced computers
    - Excessive overestimation and underestimation with same factor for different programs, parameterisation doesn't help too much
- Large body of academic work, but little industrial uptake: YET

# Generic Problem Partitioning

- Some analysis methods integrate some aspects of these, but the general requirements are the same



# Structural Analysis

NICTA

- Can work on:
  - Source code and/or
  - Object code or
  - Assembly Code
- Object Code
  - Pros:
  - All compiler optimisations are done
  - This is what really is running, no trouble with macros, preprocessors
  - Cons:
  - Needs to second guess what all the variables *meant*
  - A lot of the original information is lost
    - E.g. multiple conditions, indirect function calls, object member functions

# Structural Analysis contd

NICTA

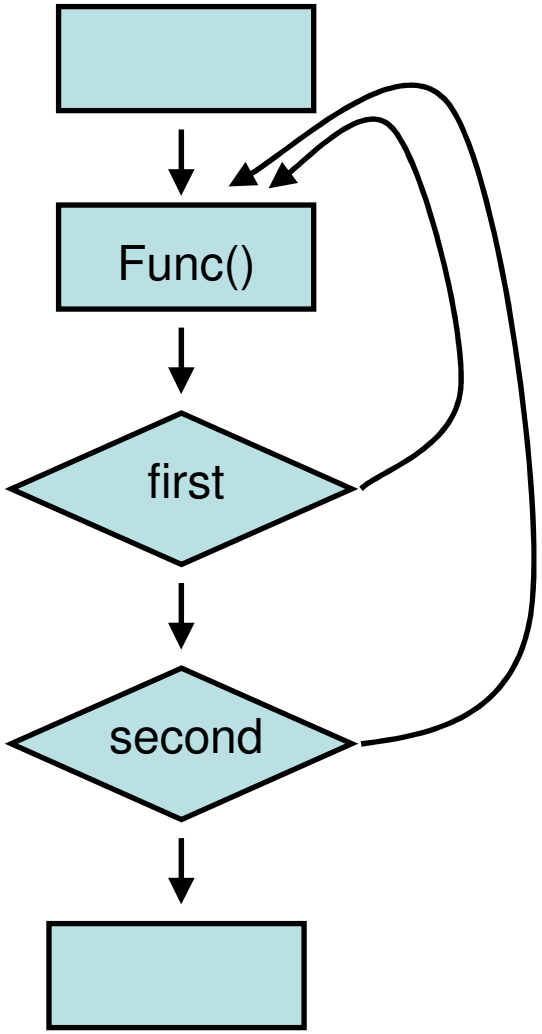
- Assembly Code
  - Pros:
  - All compiler optimisations done
  - Cons:
  - Same as Object Code +
  - Potentially still some macros
- Source Code
  - Pros:
  - All information the user has put there is there
  - Structure in pure form (e.g. multiple loop continue conditions, object member functions, indirect calls)
  - Cons:
  - Trouble with macros, preprocessors etc.
  - Needs to second guess what the compiler *will do*

# Multiple Continue conditions explained

```
for (; first condition || other cond;){
    Func();
}
```

May look like

```
for (; first condition;){
    for (; other cond;){
        Func();
    }
}
```



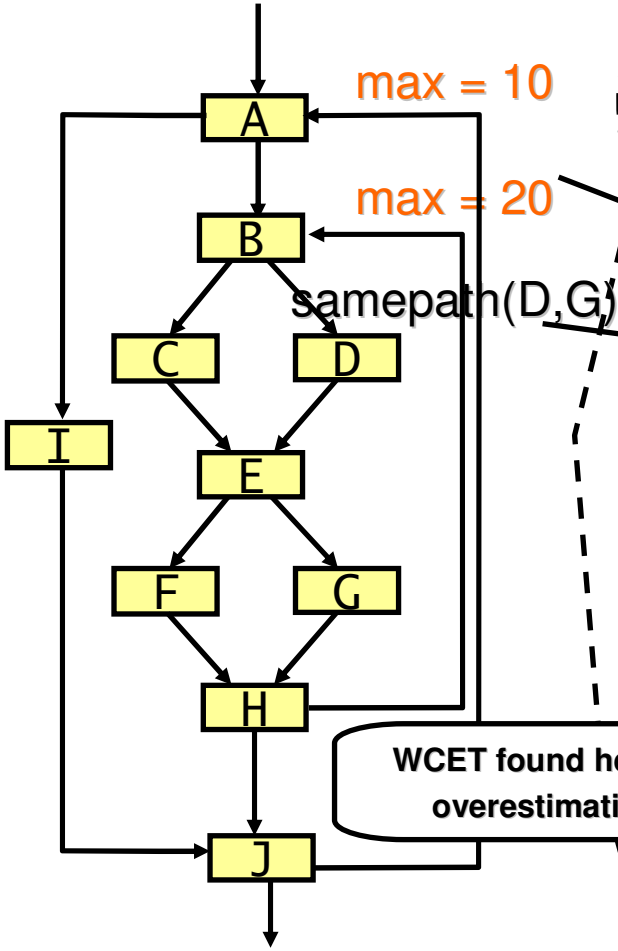
# Flow information

- For low level analysis and computation we need to restrict flow to reasonable subset.
- This information can be gained:
  - By static analysis (most importantly abstract interpretation)
  - By observation (worst case?)
  - By user annotations

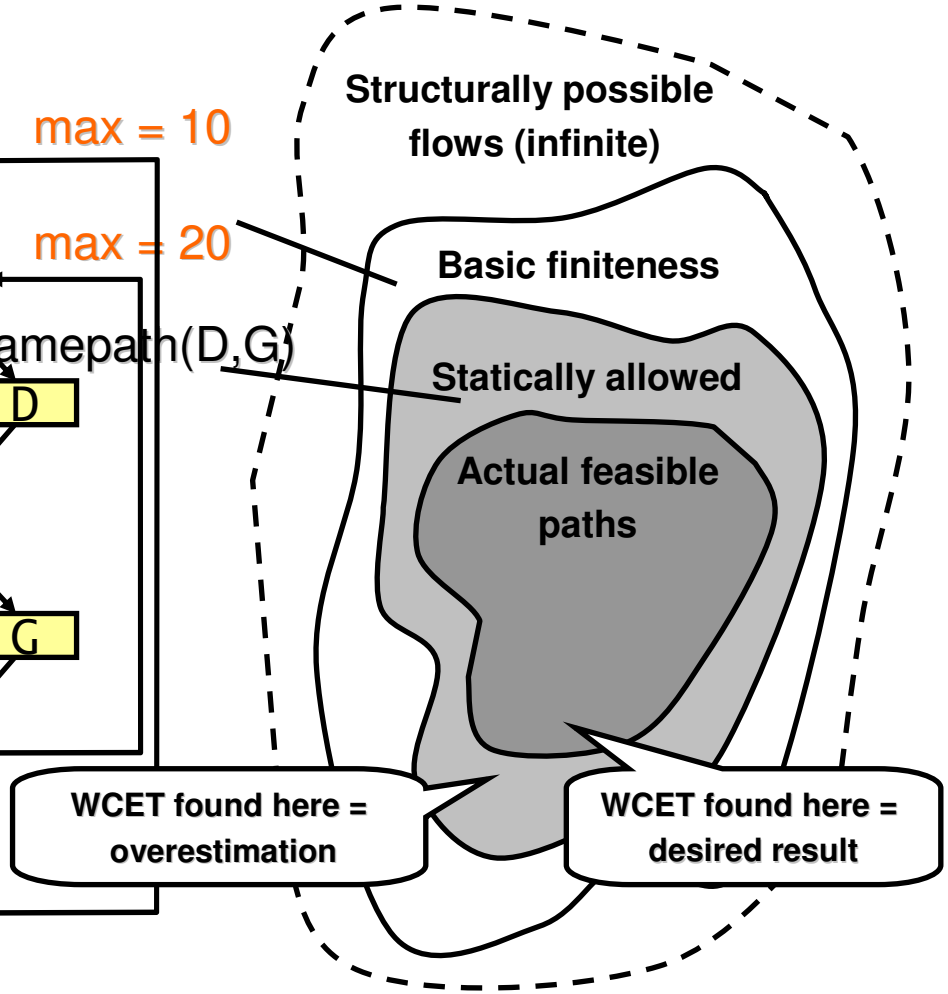
# Flow Info Characteristics

```
do
{
   if(...)        // A
      do
      {
         if(...)      // B
            ...        // C
         else
            ...     // D
         if(...)      // E
            ...     // F
         else
            ...     // G
      } while(...) // H
   else
      ...     // I
}  while(...)     // J
...
```

Example program



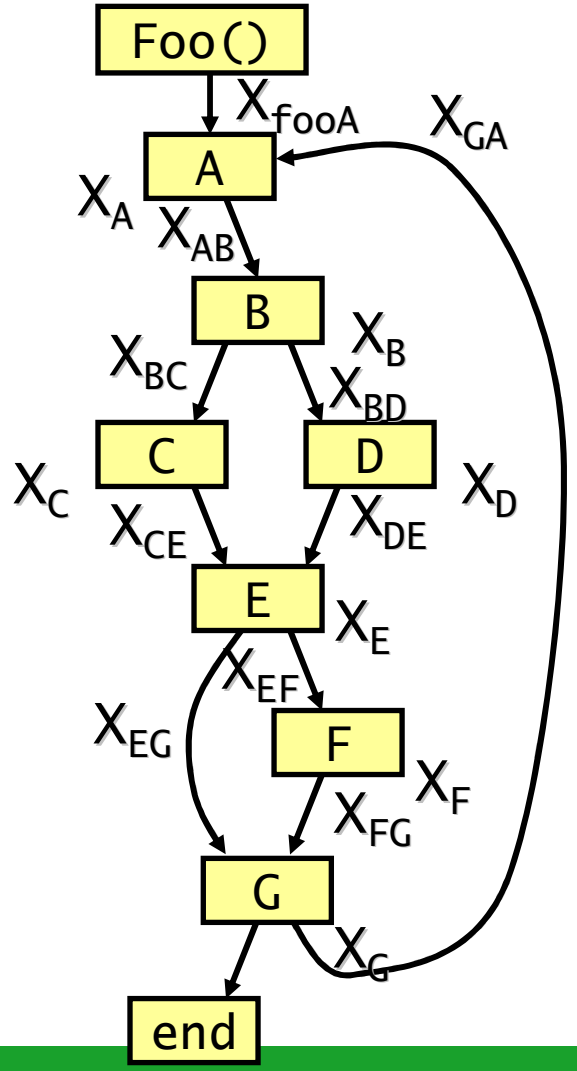

Basic block graph




Relation between possible executions and flow info

# Constraints Generated

- Constraints:
  - **Start and end condition**
    $X_{foo}=1$
    $X_{end}=1$
  - **Program structure**
    $X_A=X_{fooA}+X_{GA}$
    $X_{AB}=X_A$
    $X_{BC}+X_{BD}=X_B$
    $X_E=X_{CE}+X_{DE}$
  - **Loop bounds**
    $X_A<=100$
  - **Other flow information**
    $X_C+X_F<=X_A$



# Hardware

- WCET analysis requires a deep understanding of hardware features of processors
- Hence I consider it essential to go through some features found in processors and get a feel for what we need to deal with later

# Static Analysis

NICTA

- Looking at basic blocks in isolation (tree based, IPET based)
  - Problem of caching effects
- Path based analysis popular but very expensive
- Problem of conservative assumptions
- Hardware analysis is very expensive
  - Data caches and modern branch prediction are very hard to model right.
  - Call for simpler hardware, e.g. scratchpad memory instead of caches

# Measurement Based Analysis

NICTA

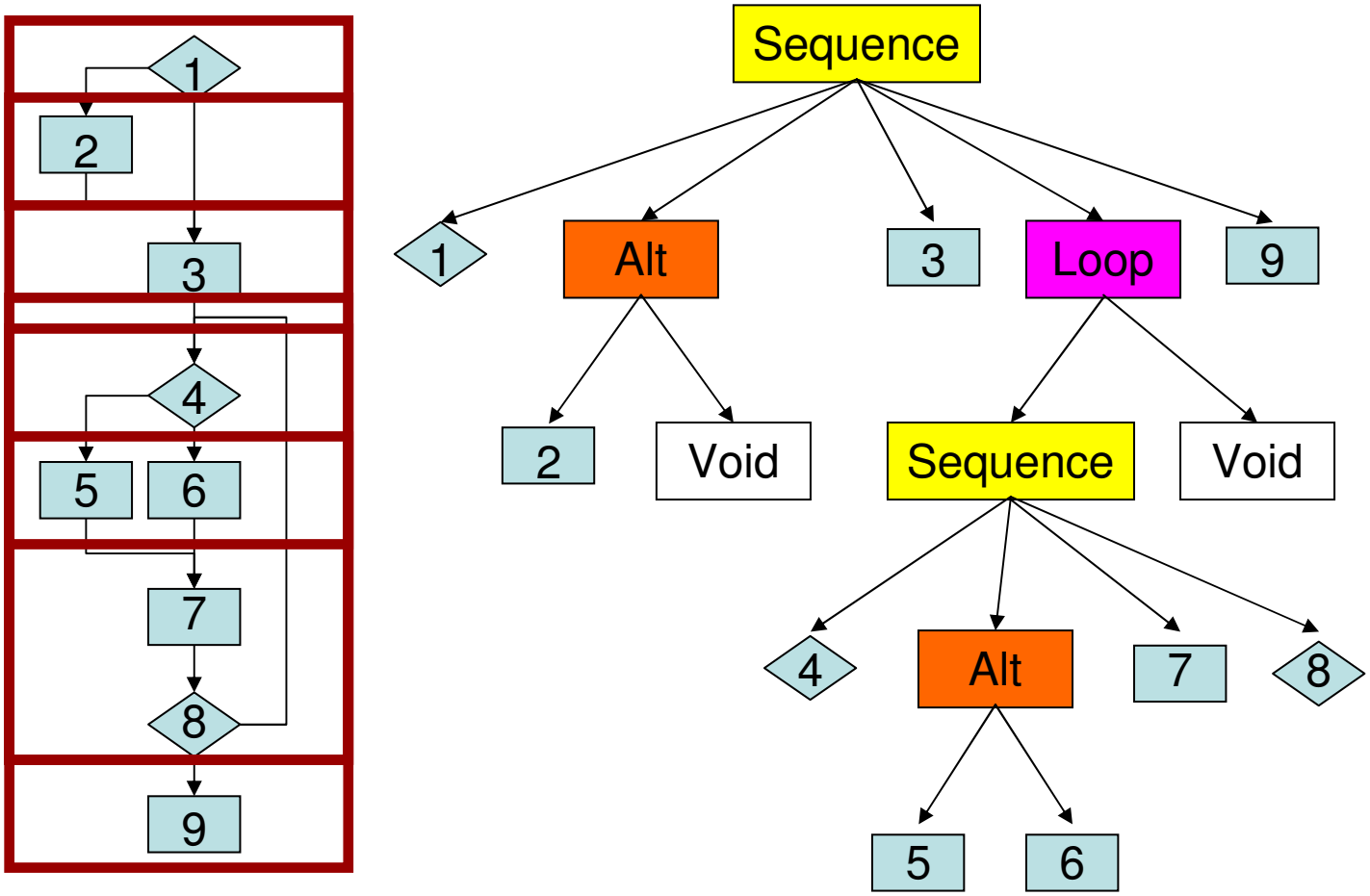
- End-to-end measurements + safety factor used for industrial soft-real time system development
  - Failed for modern processors as WC could hardly be expressed as function of AC
- Measurement on basic block level
  - Safer than end-to-end measurements but potentially very pessimistic
- What is the worst-case on HW?
- Can it be reliably produced?
- What about preemption delay?

# Path Based Computation

NICTA

- Follows each individual paths
- Becomes quickly intractable for large applications
- Altenbernd and Co have tried a simplified approach:
  - Starting out from the entry point of a function follow a path in the CFG and annotate each node with the execution time up to this node
  - Do so with any other path, but whenever a join node is reached compare new value up to this point with annotated value
  - Continue if new value is larger or not smaller than the the old value minus the smallest of the largest 5 overall execution times paths computed so far. (otherwise start next path)
  - If overall path is larger than smallest of the largest 5 overall execution times, keep (remove the compared smallest of the largest 5 overall execution time paths.
  - Check feasibility of 5 longest paths (path may actually happen)

# Tree Representation



# IPET Calculation

- WCET=

  max $\Sigma(x_{entity} * t_{entity})$

  – Where each $x_{entity}$ satisfies all constraints

$X_{foo}=1$

$X_A=X_{fooA}+X_{GA}$

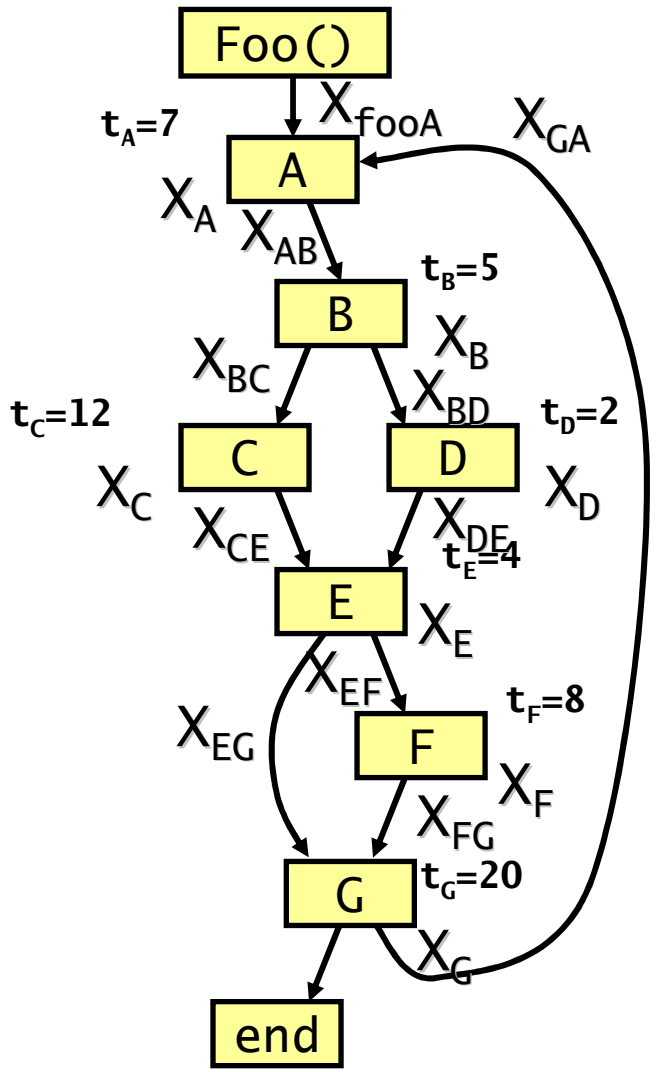
$X_{AB}=X_A$

$X_{BC}+X_{BD}=X_B$

$X_E=X_{CE}+X_{DE}$

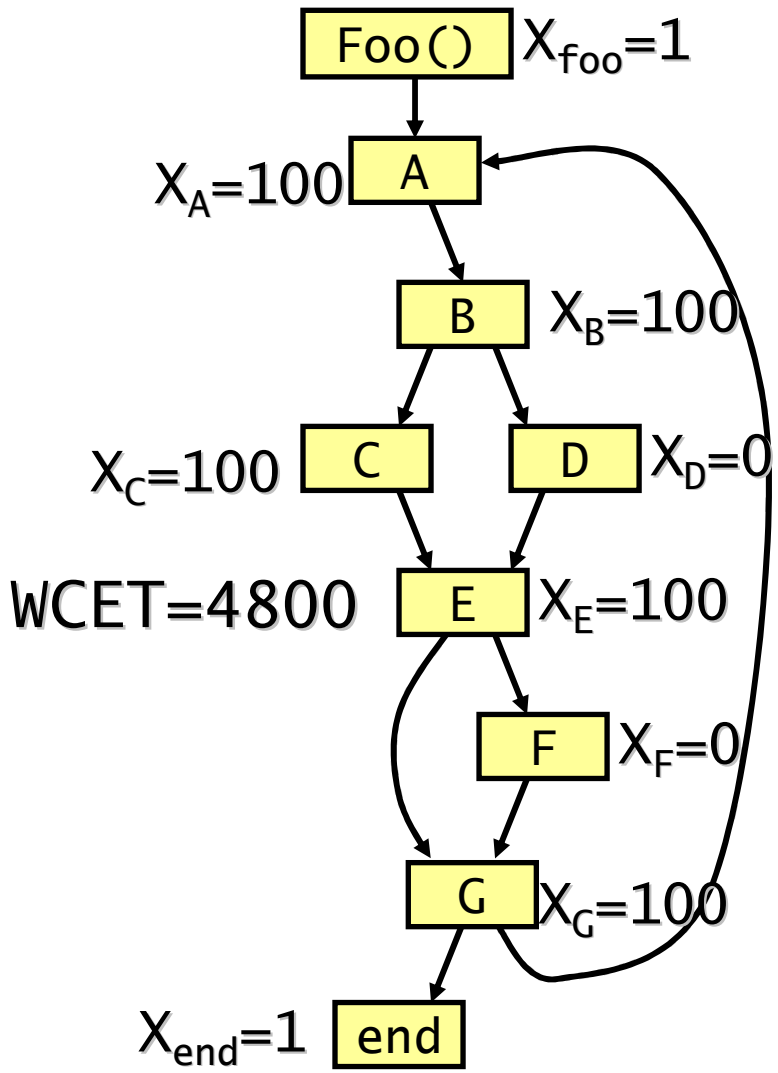
$X_C+X_F=100$

$X_A<=100$



# Calculation methods

- Solution methods:
  - Integer linear programming
  - Constraint satisfaction
- Solution:
  - Counts for each individual node and edge
  - The value of the WCET



NICTA

THE UNIVERSITY OF NEW SOUTH WALES

# Multiprocessor/Multithreaded Real-Time Systems

Stefan M. Petters
NICTA Ltd. and CSE UNSW
smp@cse.unsw.edu.au

# WHY

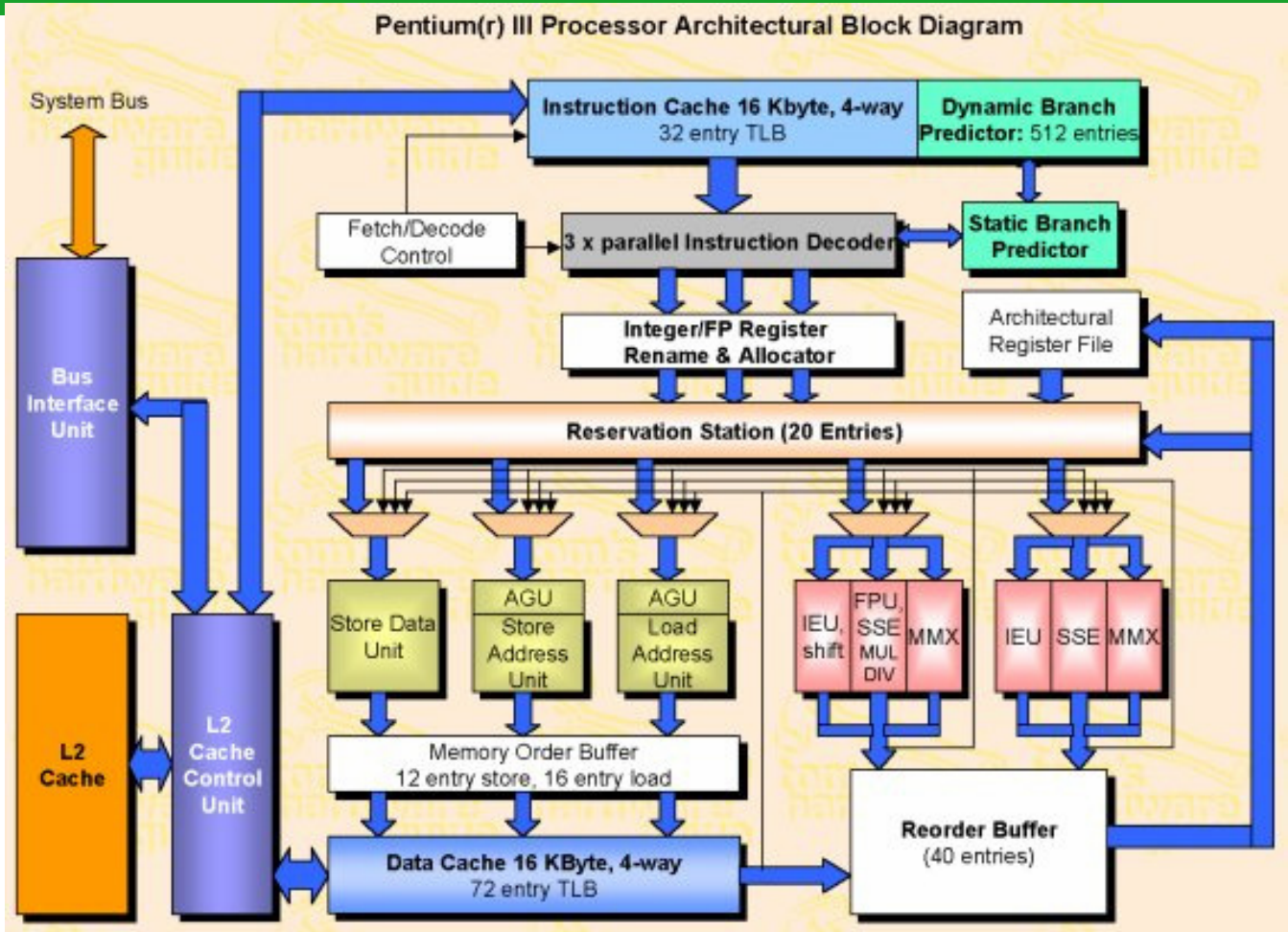
- Performance
  - Responsiveness in the presence of many external events
- Throughput
  - Managing continuous load
- Fault tolerance
  - Managing bugs, HW faults
- Reliability
  - Ensuring uptime, HW/SW upgrades …

# Hardware

- Symmetric Multithreading (SMT)
  - Contention on execution units, caches, memory
- Symmetric Multiprocessor (SMP)
  - Contention on memory, cache coherency, eg NUMA
- Asymmetric Multiprocessor
  - Specialised units, coherency

- Distributed System
  - Latency in communication, loosely coupled

# SMT




Almost an SMT:

Image taken from http://www.tommesani.com/images/P3Architecture.jpg

# SMP

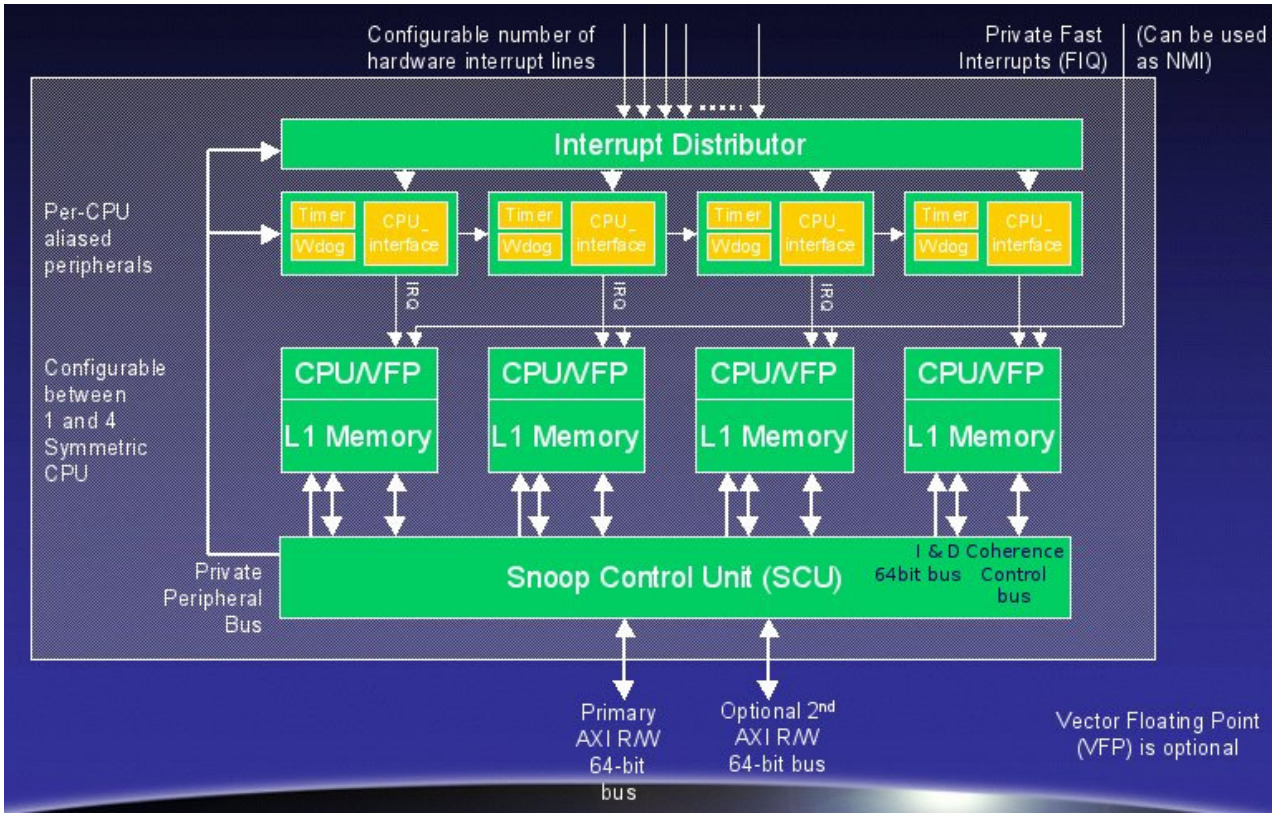
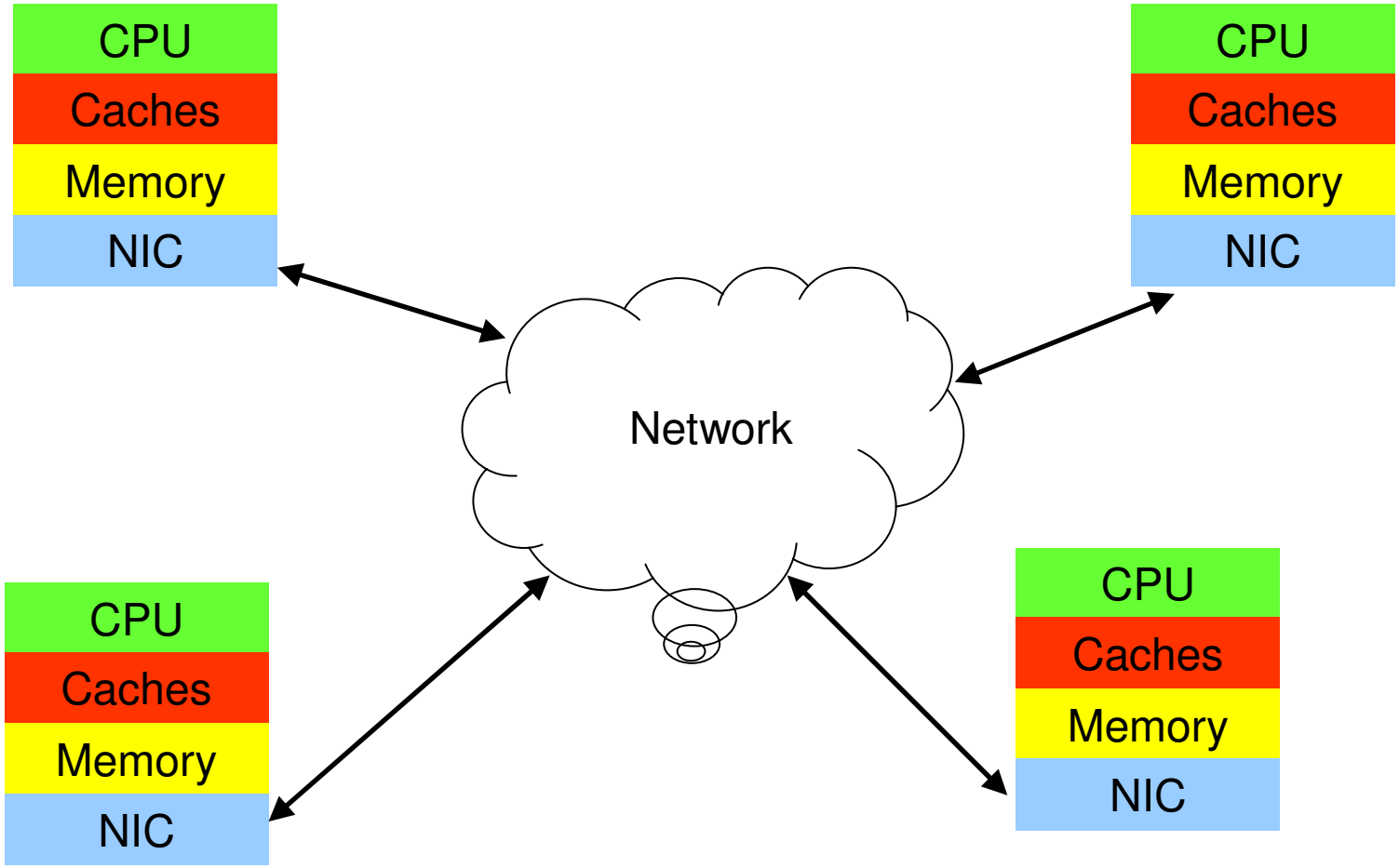



Image taken from http://linuxdevices.com/files/misc/arm-mpcore-architecture-big.jpg

# Distributed System



# Issues

- Resource contention
  - Execution units
  - Caches
  - Memory
  - Network
- Adding a CPU does not help
  - Example double the load, 2 instead of 1 CPU

# Solutions??

- Partitioning
  - Resource contention still there!
  - Assignment using heuristics
- Non partitioning
  - mostly theoretical so far
  - Assumptions:
    - Zero preemption cost
    - Zero migration cost
    - Infinite time slicing
  - Don't translate into reality
  - Acceptance test and no task migration a way to make it work

# Solutions??

- Quite often non-preemptive
  - Fewer context switches
  - Reasoning is easy
    - IEEE Computer reference to insanity
  - Testing is easier??
  - Reduce need for blocking
- But!

# Non-Preemptive

- But!!!
  - Less efficient processor use
  - Anomalies: response time can increase with
    - Changing the priority list
    - Increasing number of CPUs
    - Reducing execution times
    - Weakening the precedence constraints
  - Bin packing problem NP hard
  - Theoretically: time slicing into small quantums (PFAIR), but practically useless, as preemption and task migration overhead outweigh gains of Multiprocessors.

# And now?

NICTA

- No global solution.
- Partitioning and reducing it to single CPU problem good, but still contention of resources.
- Next step: After figuring out how to do the scheduling, what about preemption delay?
- Industry works with SMP/SMT, but most often on a very ad hoc basis.
- Active and unsolved research area
- Why does it work on non-RT?
  - Running the “wrong” task is not critical.

# Integrating Real-Time and General-Purpose Computing

Many thanks to: Scott A. Brandt
University of California, Santa Cruz

# Real-Time vs. General-Purpose OS

NICTA

- Real-time and general-purpose operating systems implement many of the same basic operations
  - Process mgmt., memory mgmt, I/O mgmt, etc.
- They aim for fundamentally different goals
  - Real-time: Guaranteed performance, timeliness, reliability
  - General-purpose: Responsiveness, fairness, flexibility, graceful degradation, rich feature set
- They have largely evolved separately
  - Real-time system design lags general-purpose system design by decades
- They need to merge

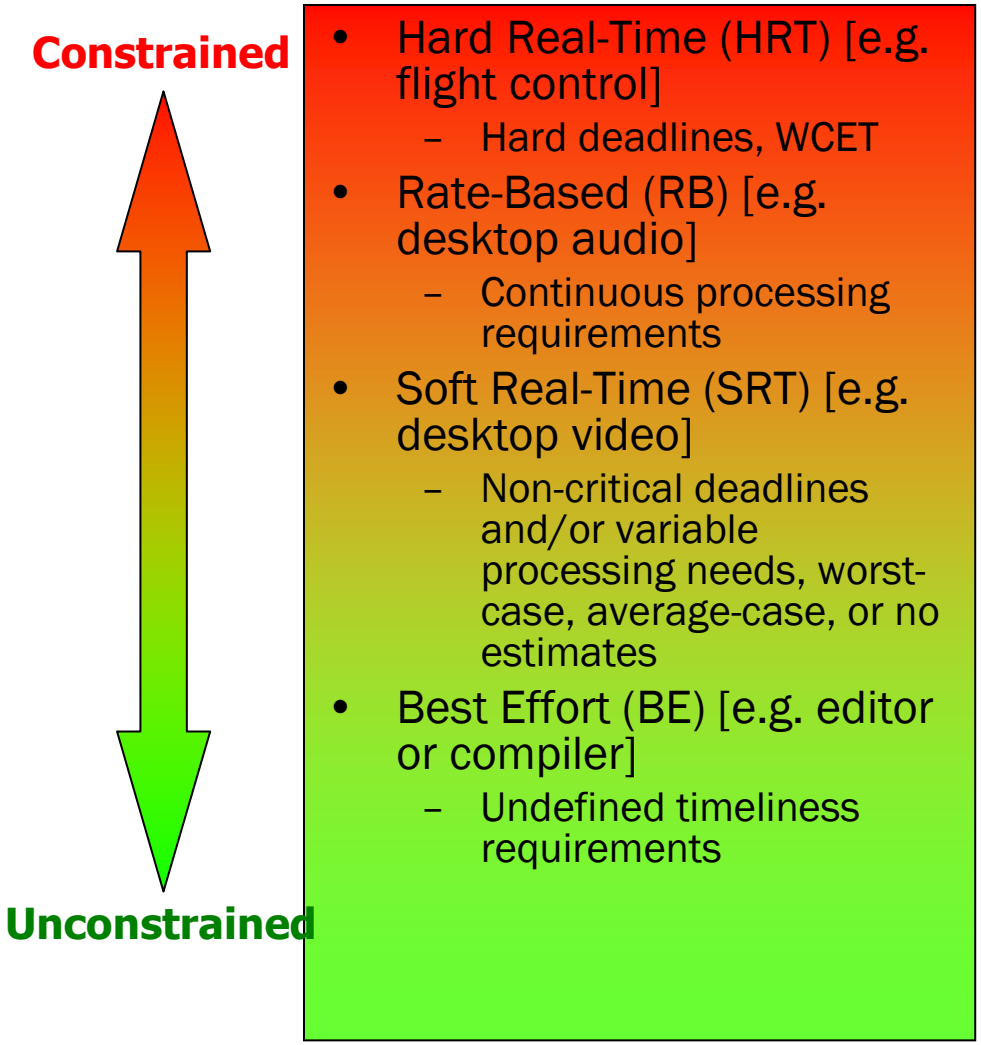
# Why?

- We want both flexible general-purpose processing and robust real-time processing
  - Multimedia is ubiquitous in general-purpose systems
  - Real-time systems are growing in size and complexity
  - Look at the popularity of RTLinux
- Such systems are possible
  - GP hardware has grown powerful enough to support traditional hard real-time tasks (multimedia, soft modems, etc.)
  - Windows, MacOS, etc., are already headed in this direction
- Existing solutions are *ad hoc*
  - RTLinux, MacOS, Windows?
- The world is already headed that way
  - Microsoft, HP, Intel, Dell all want to develop integrated home systems
  - Complex distributed real-time systems do more than hard real-time
- We need to get out in front and lead the way

# How?

- We need integrated solutions for each type of resource
  - CPU, storage, memory, network, …
- They must be hard real-time at their core
  - This is the only way to guarantee the hardest constraints
- They must provide native hard real-time, soft real-time, and best-effort support
  - SRT and BE support cannot be added as an afterthought
  - Neither can HRT
- We need an overall model for managing the separate resources
  - Each process must be able to specify it's per-resource constraints
  - Defaults should be reasonable, and helpful

# Kinds of Timeliness Requirements




- We want to run processes with different timeliness requirements in the same system
  - HRT, RB, SRT, and BE
- Existing schedulers largely provide point solutions:
  - HRT **or** RB **or** one flavor of SRT **or** BE
- Hierarchical scheduling is a partial solution
  - Allows apps with a variety of timeliness requirements, BUT
  - Static, inflexible hierarchies
- Goal: Uniform, fully dynamic integrated real-time scheduling
  - Same scheduler for all types of applications

# Separate Resource Allocation and Dispatching

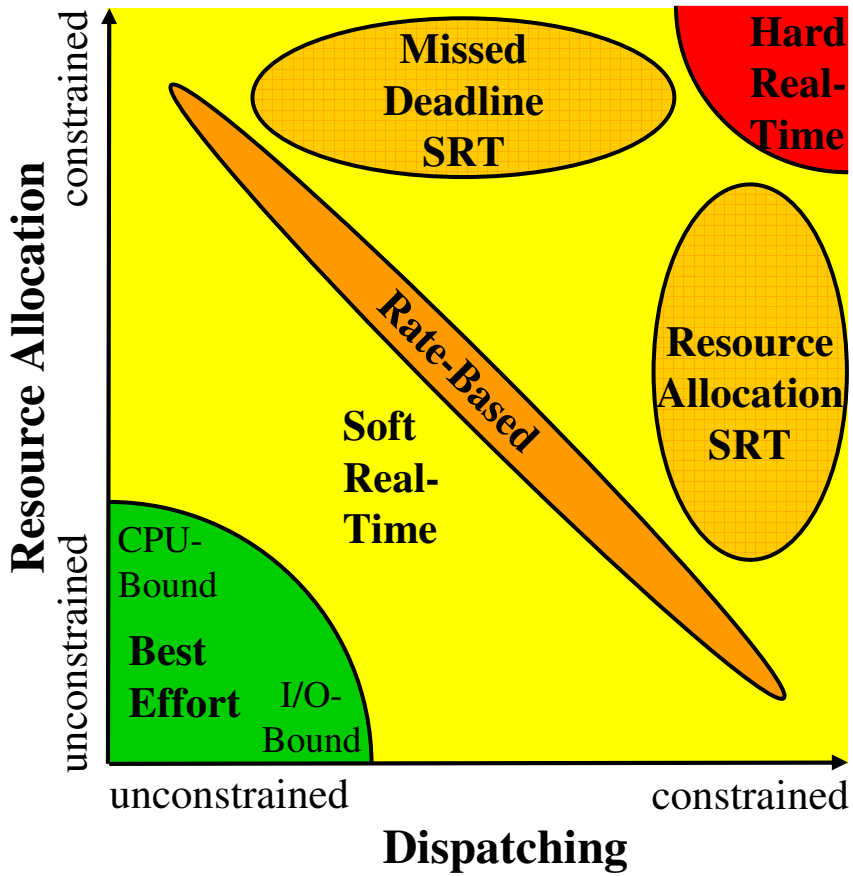
- Observation: Scheduling consists of two distinct questions:

  Resource allocation

  - *How much* resources to allocate to each process

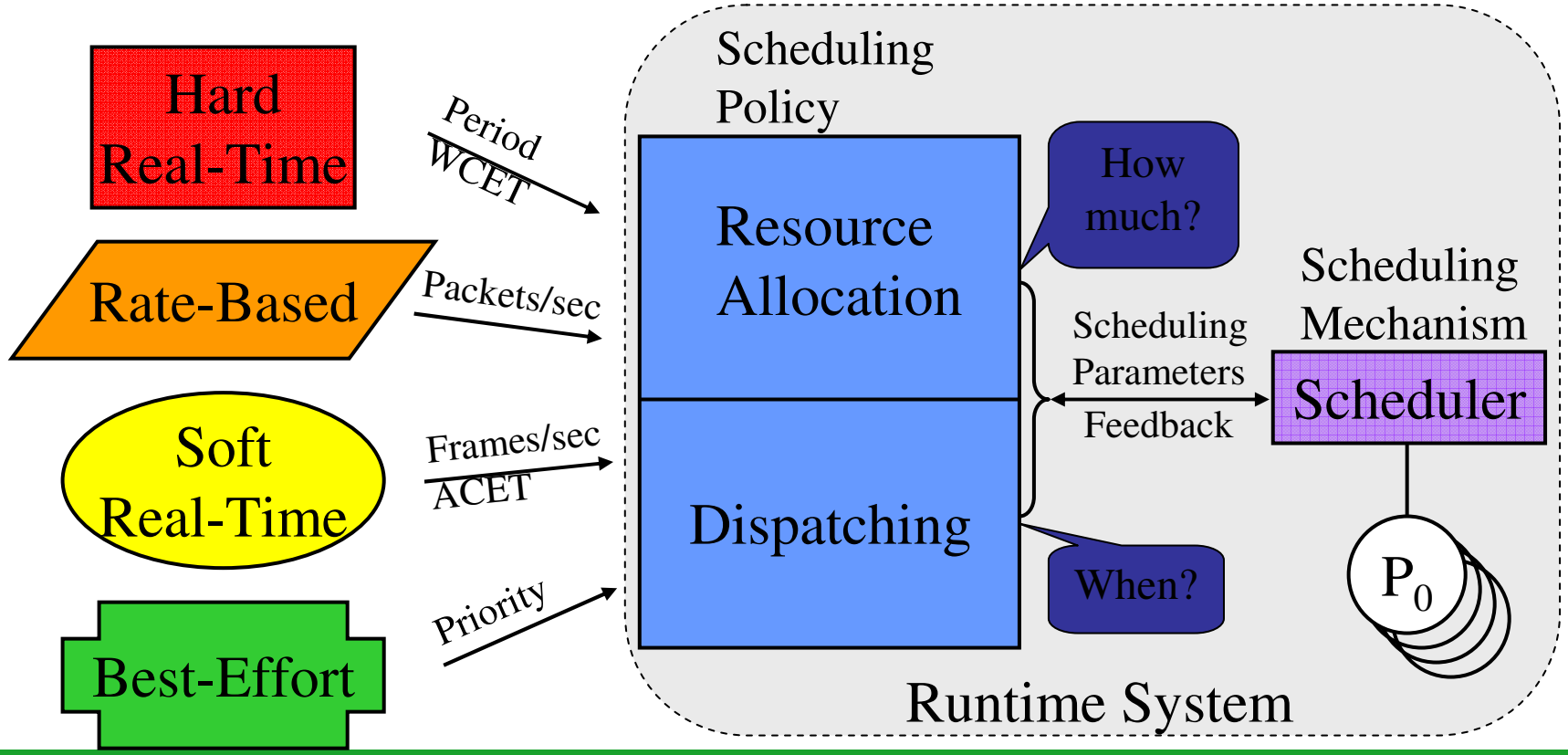
  Dispatching

  - *When* to give each process the resources it has been allocated
- Existing schedulers integrate their management
  - Real-time schedulers implicitly separate them somewhat via job admission



# The (RAD) Scheduling Model

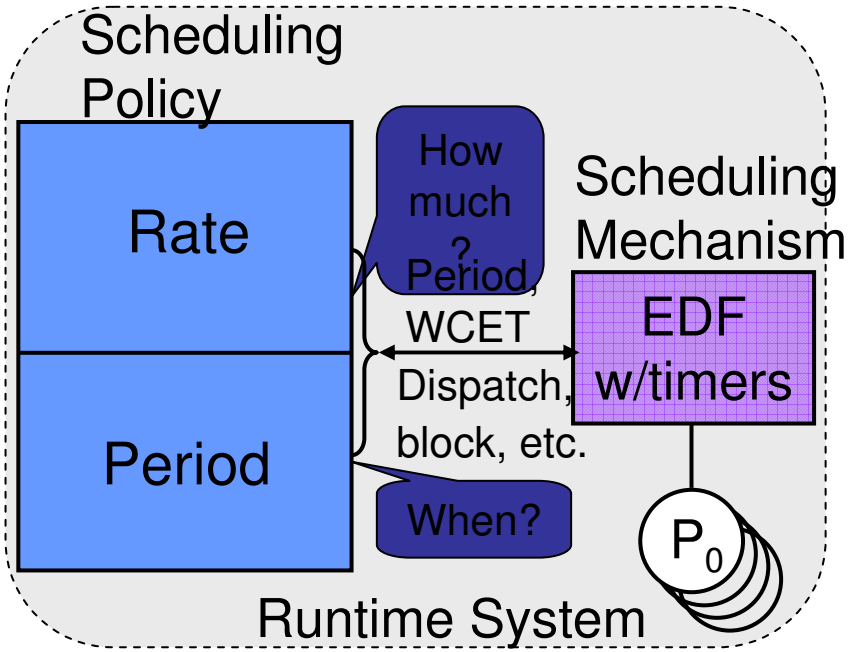
- Separate management of Resource Allocation and Dispatching
  - and separate policy and mechanism



# Rate-Based Earliest Deadline Scheduler

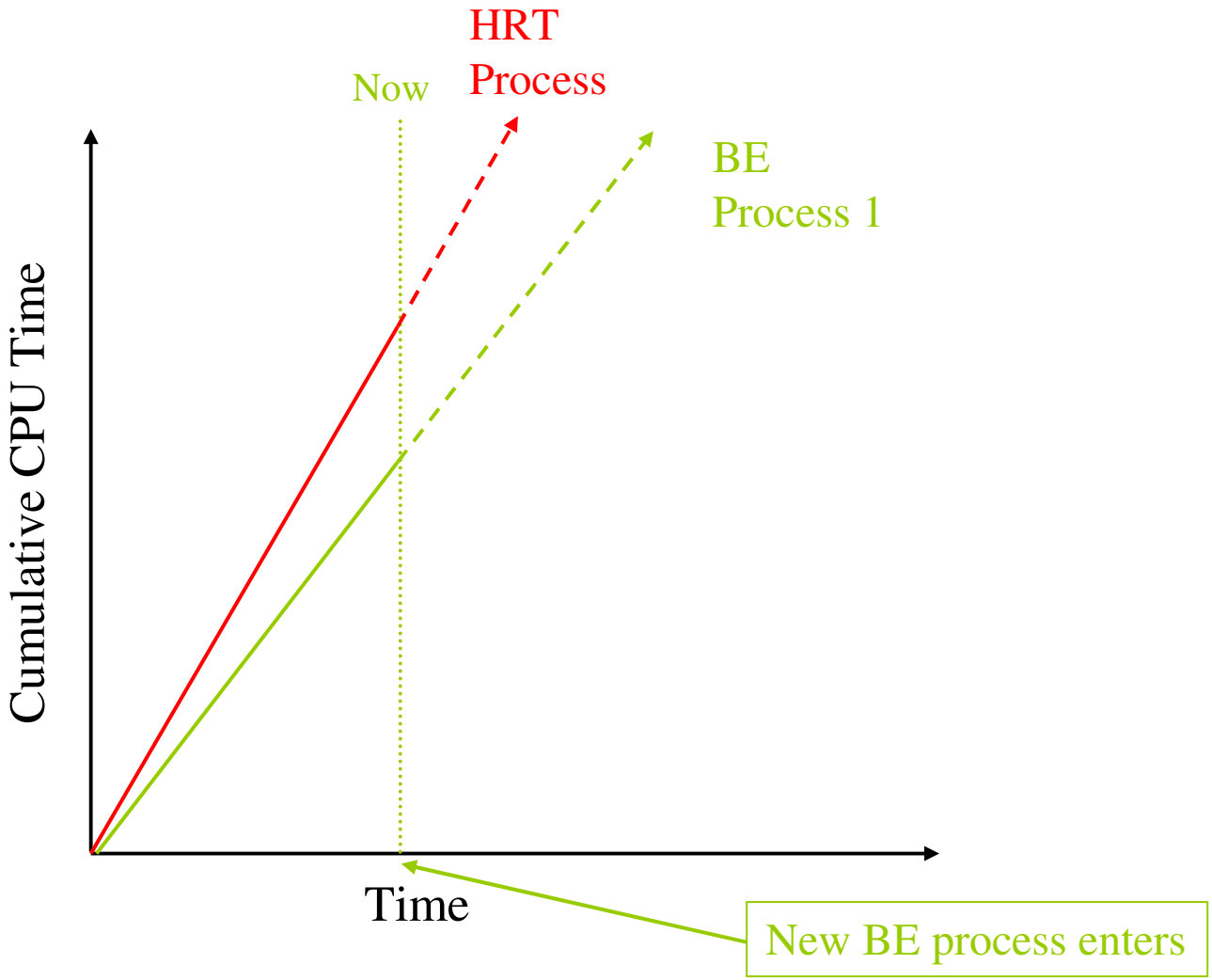
- Basic Idea
  - EDF provides hard guarantees
  - Varying rates and periods provide flexibility
  - Programmable timer interrupts guarantee isolation between processes
- RBED policy
  - Resource allocation: Target rate-of-progress for each process ($\Sigma \leq 100\%$)
  - Dispatching: Period based on process timeliness needs
- RBED mechanism
  - Rate-Enforcing EDF: EDF + programmable timer interrupts

RBED: RAD Scheduler using *rate* and *period* to control resource allocation and dispatching

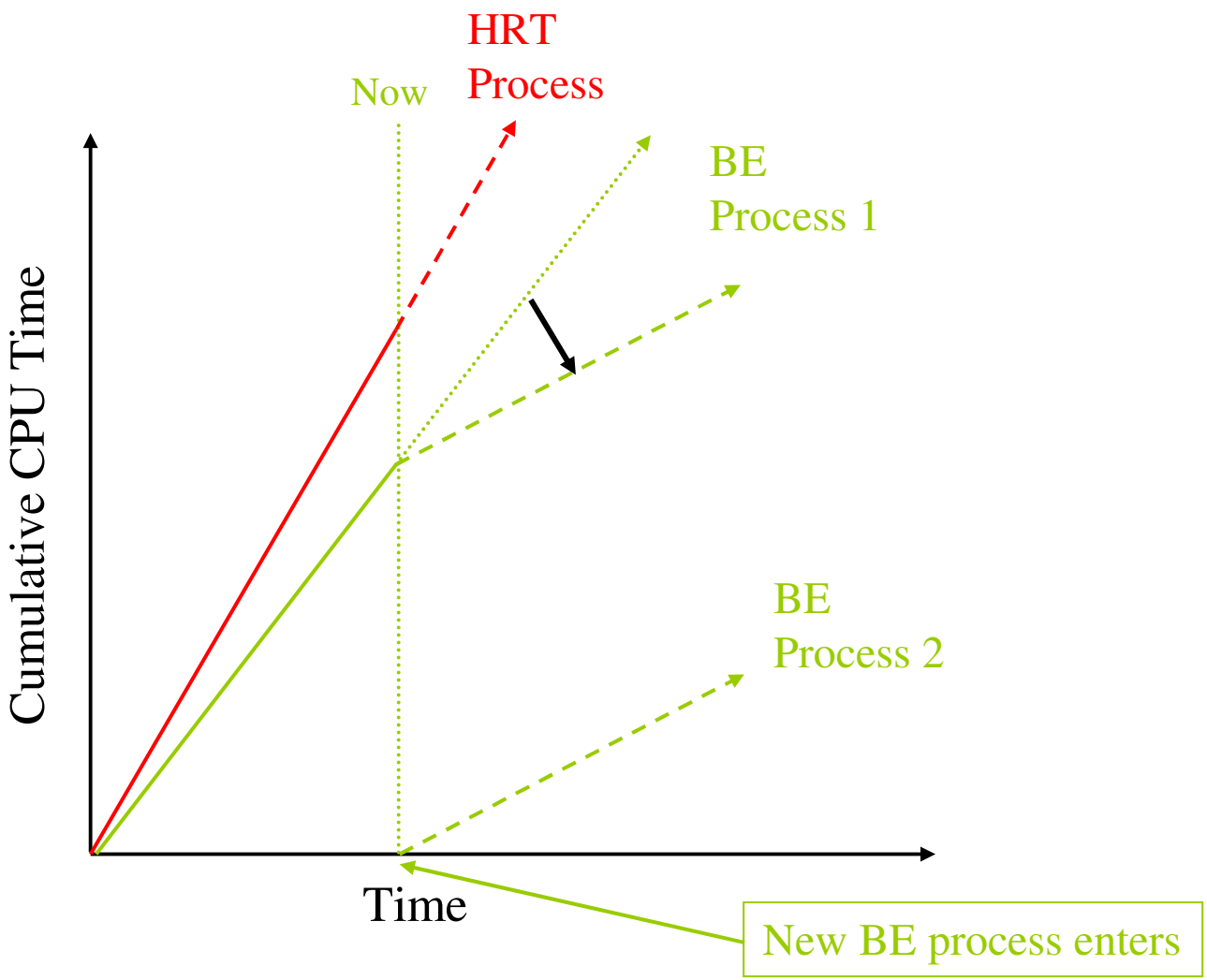



rate = utilization
WCET = rate*period

# Adjusting Rates at Runtime



# Adjusting Rates at Runtime



# RBED Periodic Task Model

EDF

- Period and WCET are specified per task
  - $T_i$ has sequential jobs $J_{i,k}$
  - $J_{i,k}$ has release time $r_{i,k}$, period $p_i$, deadline $d_{i,k}$
  - $r_{i,k} = d_{i,k-1}$, and $d_{i,k} = r_{i,k} + p_i$
  - $u_i = e_i / p_i$ and $U = \Sigma u_i$

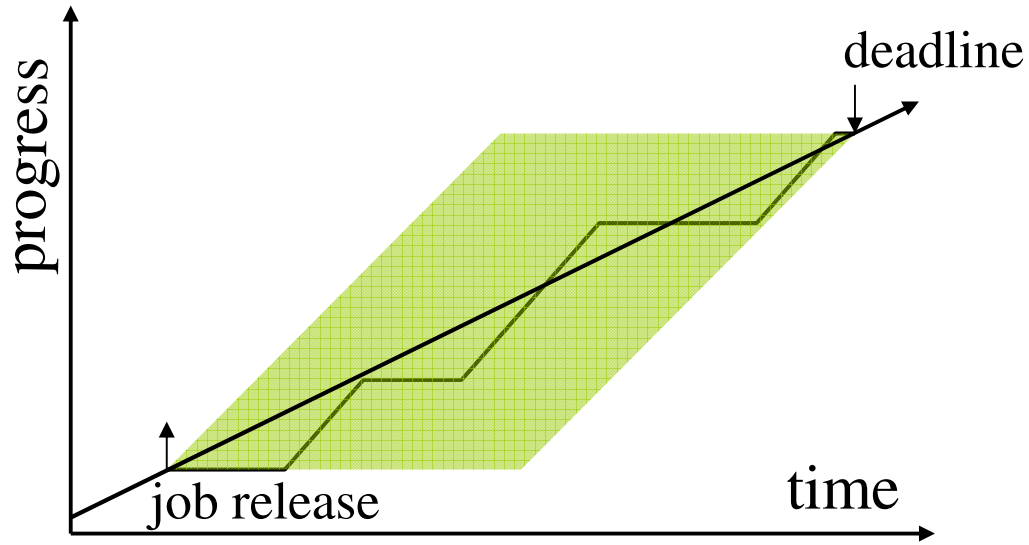
RBED

- Period and WCET are specified per job
  - $T_i$ has sequential jobs $J_{i,k}$
  - $J_{i,k}$ has release time $r_{i,k}$, period $p_{i,k}$, deadline $d_{i,k}$
  - $r_{i,k} = d_{i,k-1}$, and $d_{i,k} = r_{i,k} + p_{i,k}$
  - $u_{i,k} = e_{i,k} / p_{i,k}$ and $U = \Sigma u_{i,k}$

- Theorem 1: *EDF is optimal under the new task model*
  - Corollary: *A new task may enter the system at any time, as long as resources are available for it*

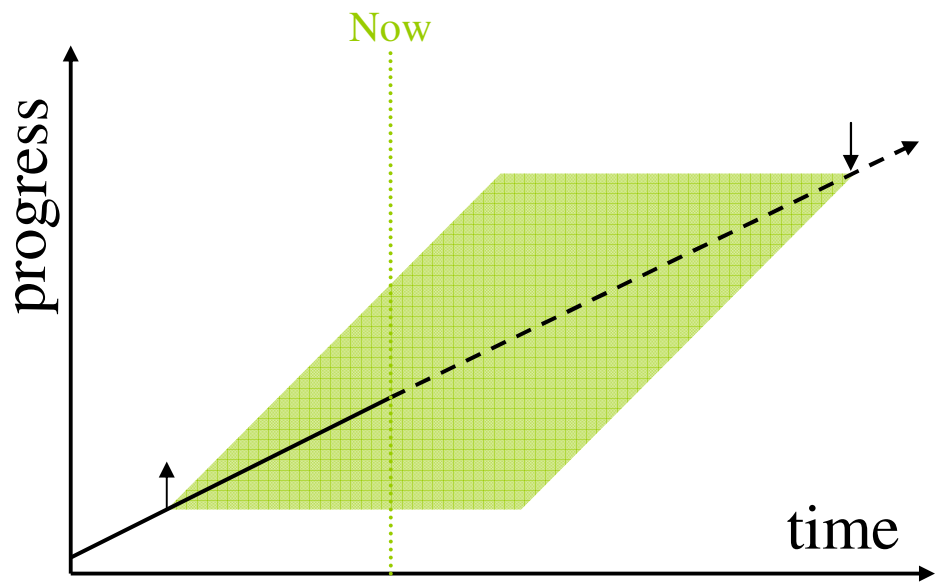
# Two Observations

- At deadlines, a task's actual resource allocation is equal to its target resource allocation
- Actual resource allocation is bounded to the *feasible region*



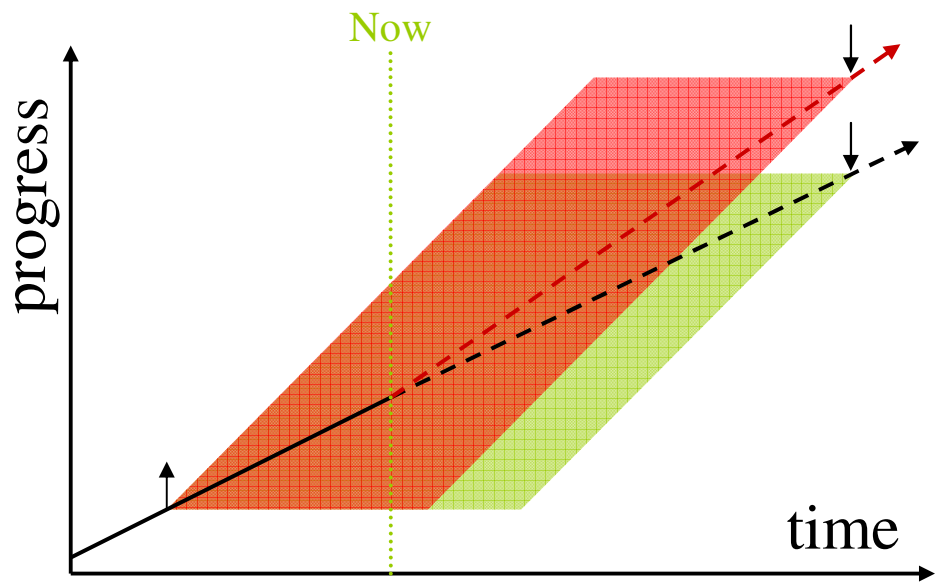
# Increasing Rate (= increasing WCET)

- Theorem 2: *The resource usage of any task can be increased at any time, within the available resources*
  - Given a feasible EDF schedule, at any time task $T_i$ may increase utilization by any amount up to $1-U$ without causing any task to miss deadlines in the resulting EDF schedule



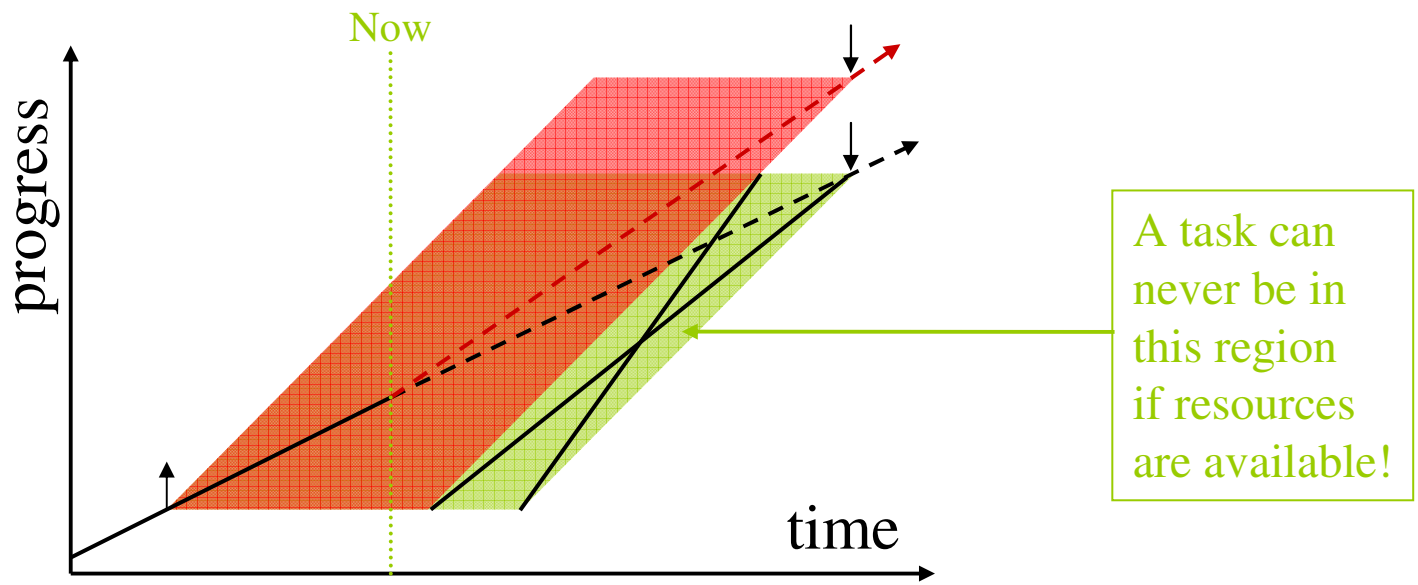
# Increasing Rate (= increasing WCET)

- Theorem 2: *The resource usage of any task can be increased at any time, within the available resources*
  - Given a feasible EDF schedule, at any time task $T_i$ may increase utilization by any amount up to $1-U$ without causing any task to miss deadlines in the resulting EDF schedule



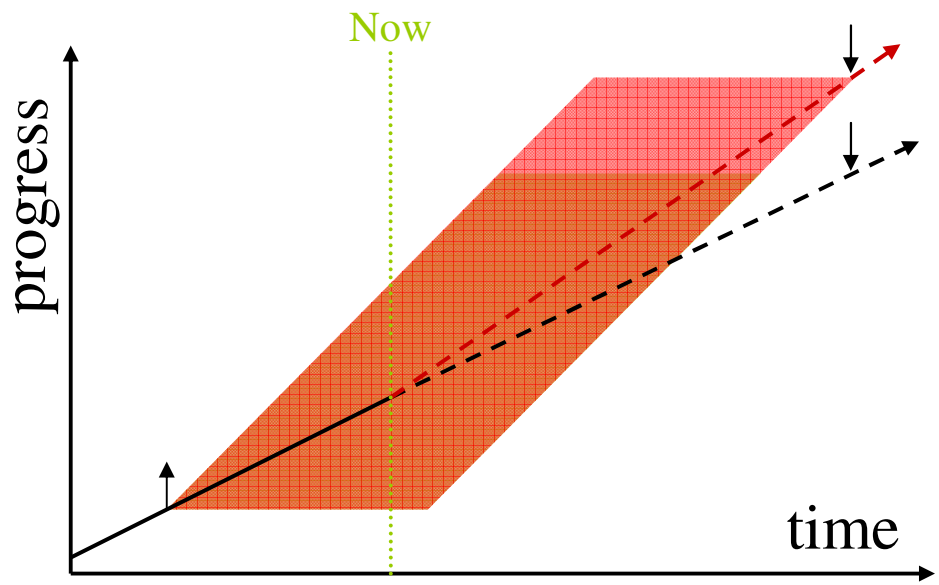
# Increasing Rate (= increasing WCET)

- Theorem 2: *The resource usage of any task can be increased at any time, within the available resources*
  - Given a feasible EDF schedule, at any time task $T_i$ may increase utilization by any amount up to $1-U$ without causing any task to miss deadlines in the resulting EDF schedule



# Increasing Rate (= increasing WCET)

- Theorem 2: *The resource usage of any task can be increased at any time, within the available resources*
  - Given a feasible EDF schedule, at any time task *Ti* may increase utilization by any amount up to *1–U* without causing any task to miss deadlines in the resulting EDF schedule



# RBED EDF Mode Change Theory

- Theorem 1: *EDF is optimal under this task model*
- Corollary: *A new task may enter at any time, within available resources*
- Theorem 2: *The rate of any task can be increased at any time, within available resources*
- Theorem 3: *The period of any task can be increased at any time*
- Theorem 4: *The rate of any task can be lowered at any time, down to what it has already used in the current period*
- Theorem 5: *The period of any task can be reduced at any time, down to the time corresponding to the current period's resource usage*
- Corollary: *The period of any task can be increased at any time (without changing WCET)*
- Corollary: *The period of a job which is ahead of its target allocation can be reduced at any time, down to the time corresponding to its current resource usage (without changing WCET) as long as the resources are available for the rate change*

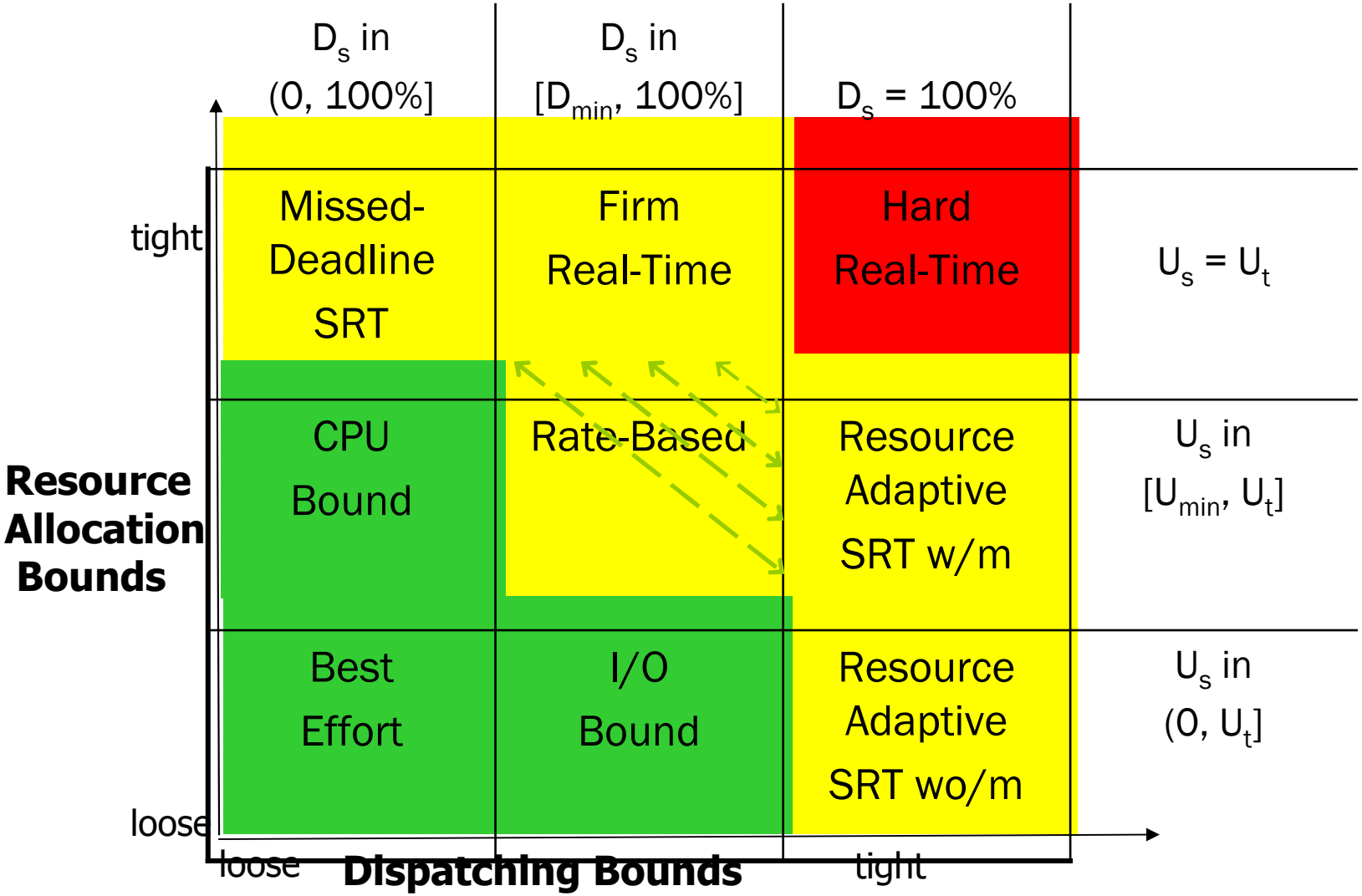
# RBED Theory Summary

NICTA

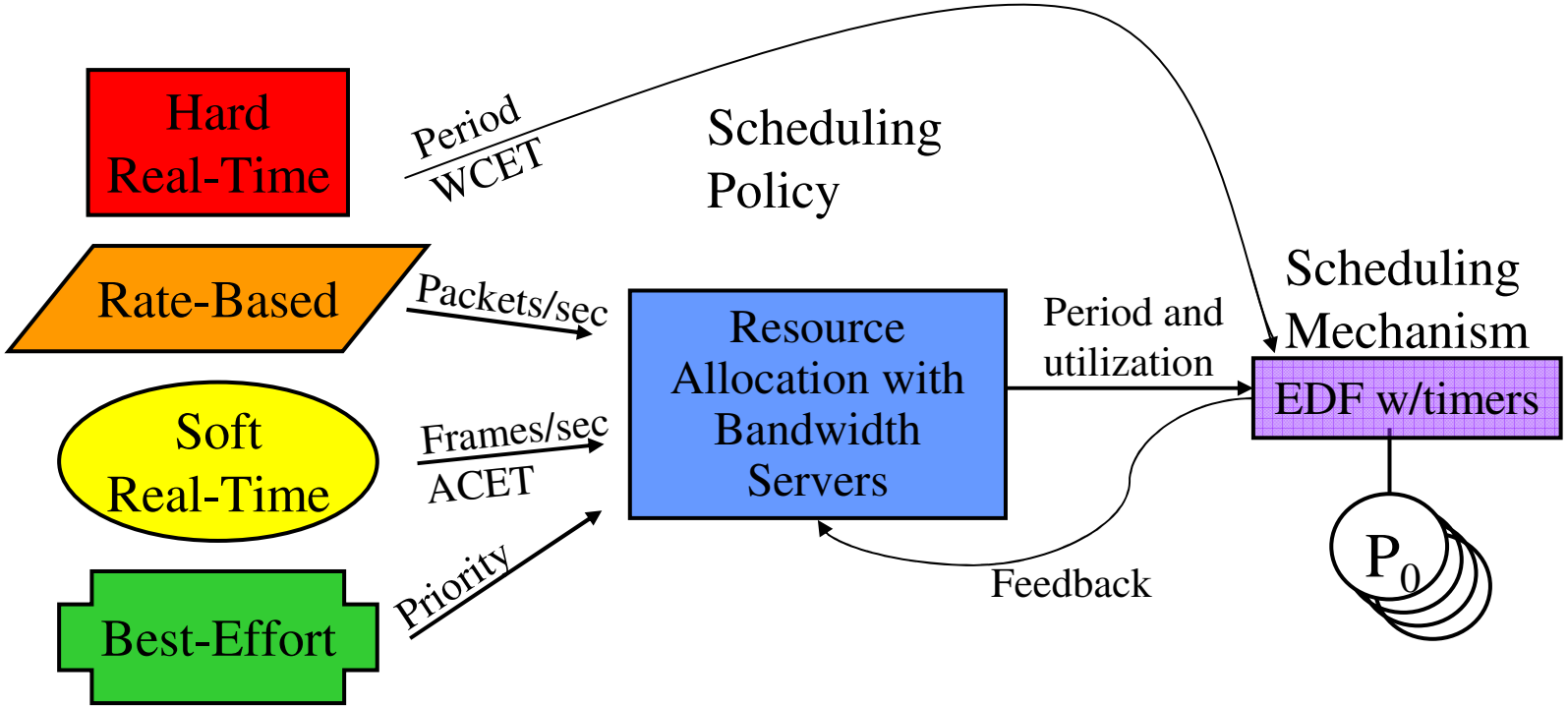
- Rate and period can be changed without causing missed deadlines
  - At deadlines, rate and period changes are unconstrained (except by available resources)
  - In between, decreases are constrained by resource usage in the current period
  - The changes may be combined

- Isolation between processes is guaranteed

# Extension 1: Rich SRT Support



# Extension 2: Automated SRT Support



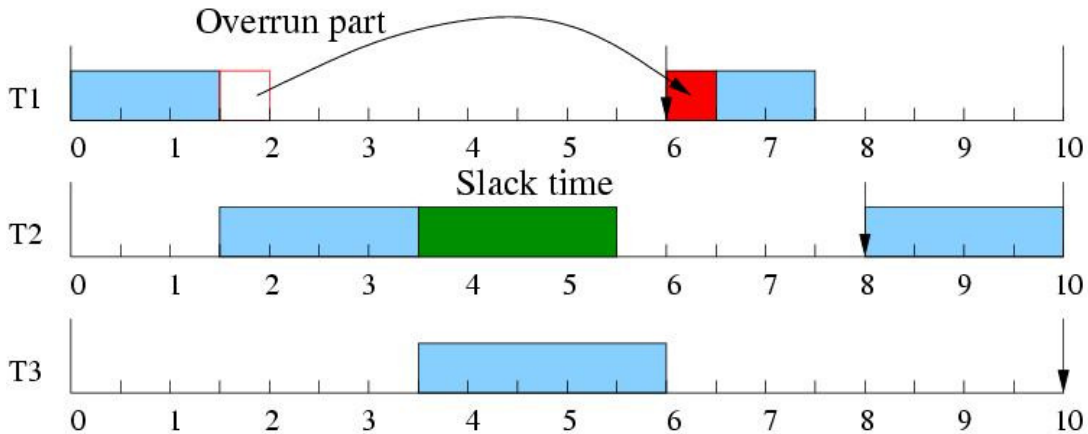

- A third type of BE process: CPU-bound, interactive, and **periodic**
- Aperiodic server for **each** BE/SRT/RB process
- **Dynamically determined rate and period** for each bandwidth server
- Improved slack reclamation

# Better Slack Management: BACKSLASH

NICTA

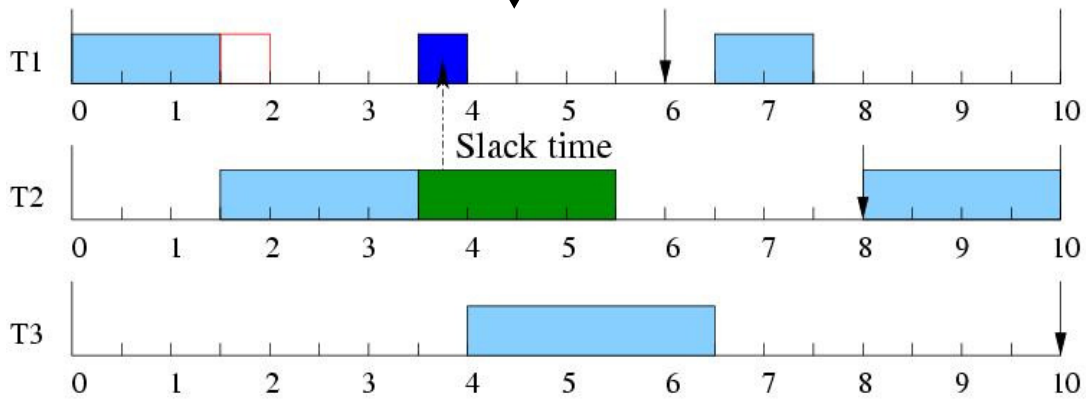
- Existing algorithms tend to ignore the needs of "background" tasks
  - Slack provided when everything else is idle
  - Aim for "fair" allocation and 100% utilization
- Slack reclamation is critical in an integrated real-time system
  - Utilization is important for best-effort systems
  - Soft real-time and best effort performance depends on the effective use of slack
- BACKSLASH improves performance via slack scheduling
  - Focuses on when slack is allocated, and to which process

# When To Allocate Slack?



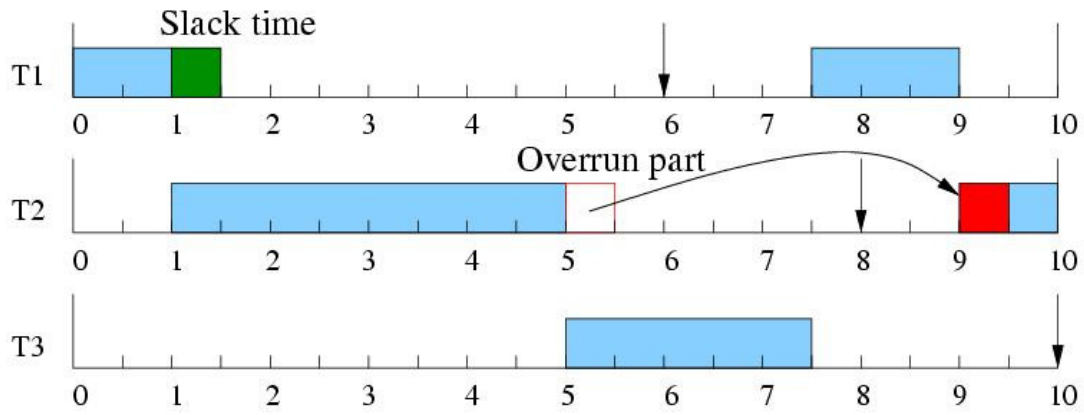

| Task | Reservation | Period |
|---|---|---|
| T1 | 1.5 | 6.0 |
| T2 | 4.0 | 8.0 |
| T3 | 2.5 | 10 |

Solution



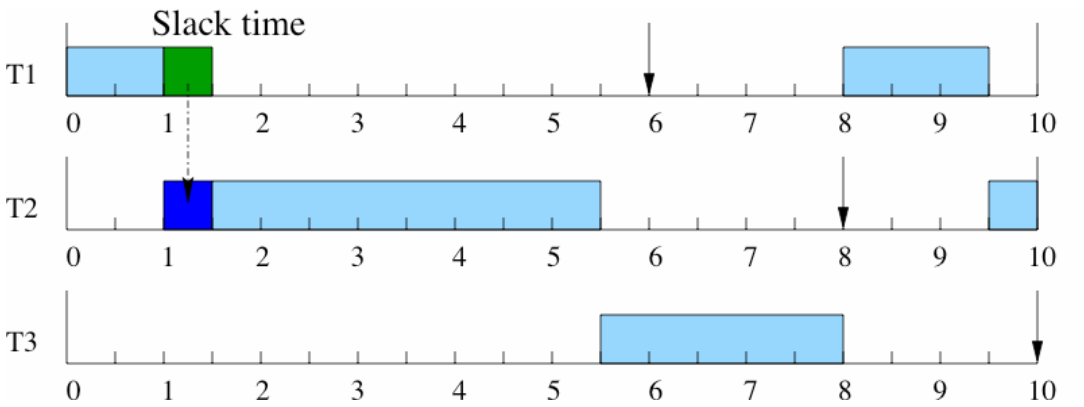

Answer: Allocate slack as early as possible

# Who To Allocate Slack To?



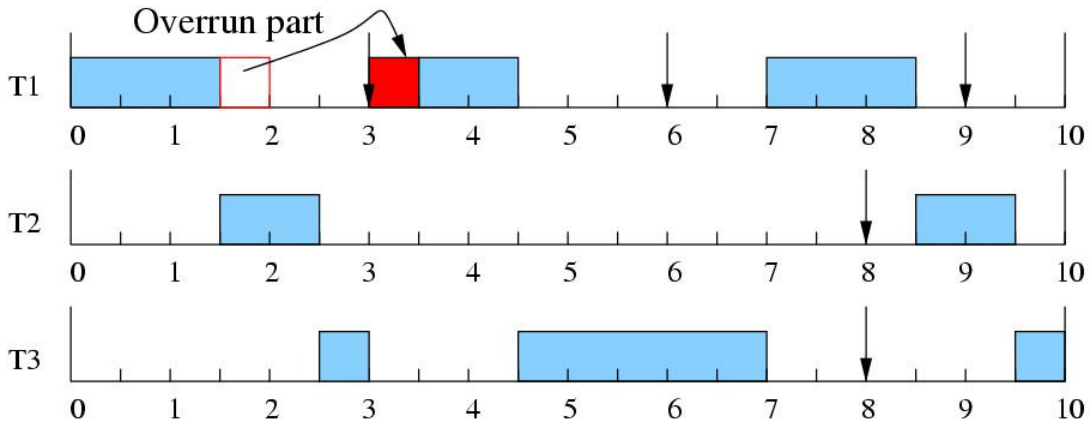

| Task | Reservation | Period |
|---|---|---|
| T1 | 1.5 | 6.0 |
| T2 | 4.0 | 8.0 |
| T3 | 2.5 | 10 |

Solution



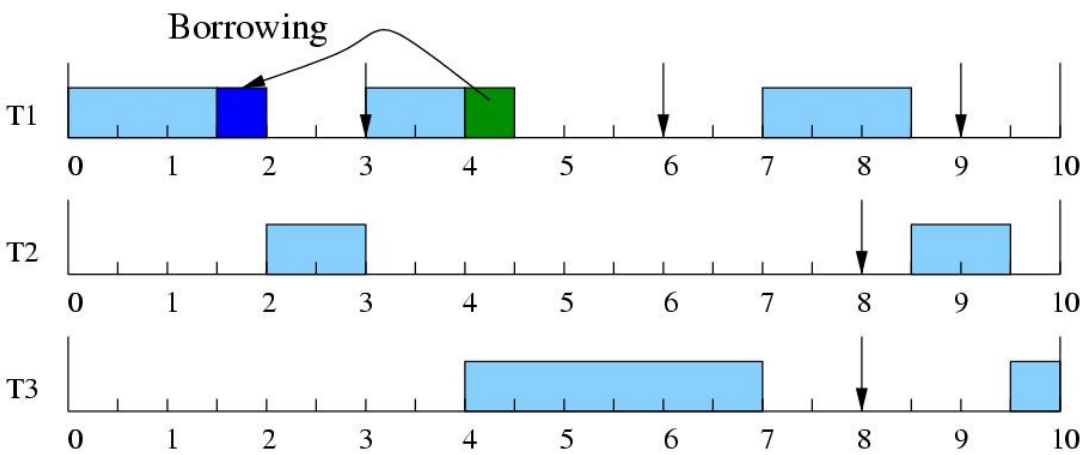

Answer: Allocate slack to the task with the earliest deadline

# How To Use Future Slack?



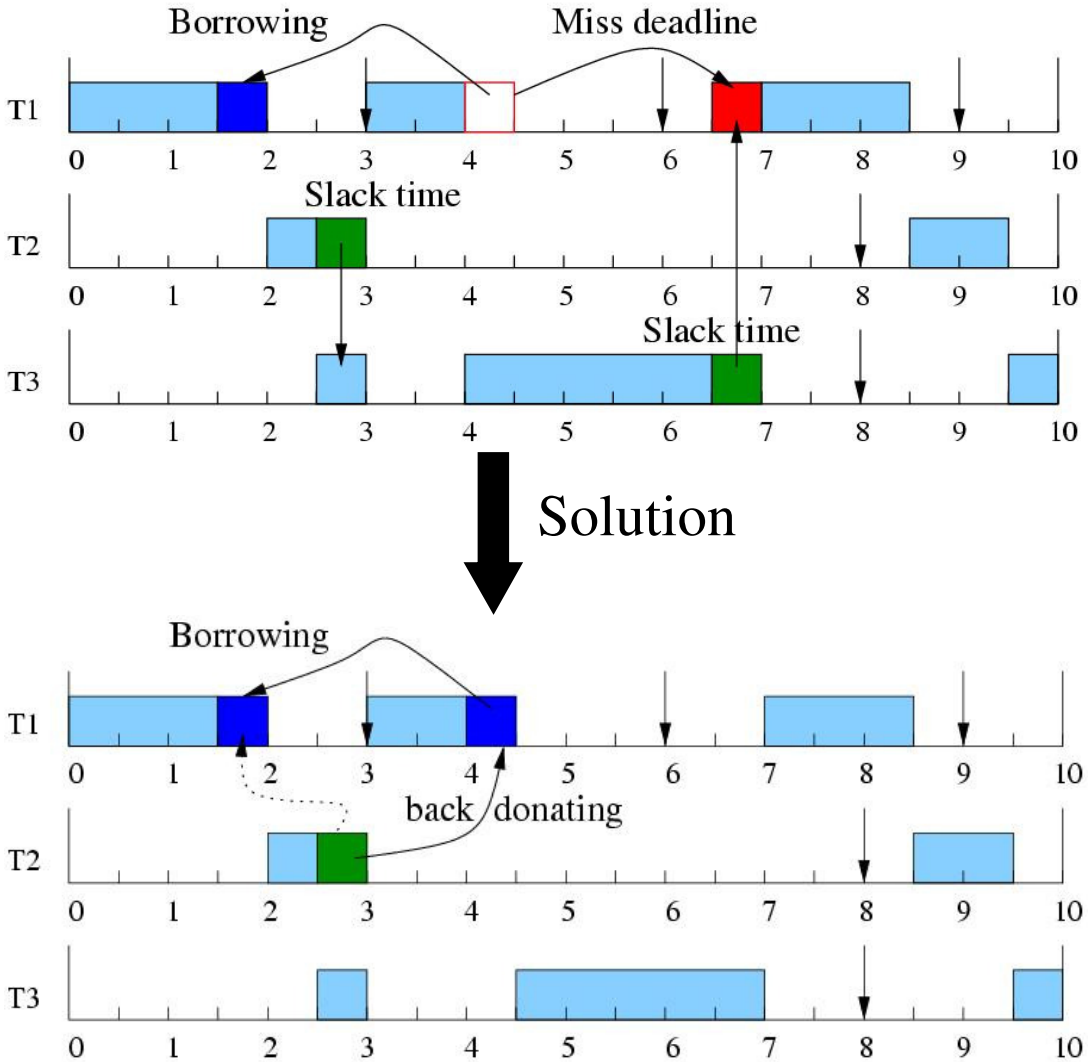

| Task | Reservation | Period |
|---|---|---|
| T1 | 1.5 | 3.0 |
| T2 | 1.0 | 8.0 |
| T3 | 3 | 8 |

Solution




Answer: Borrow resources (potential slack) from the next job to meet the current deadline

# How to Allocate Slack to Past Overruns?




| Task | Reservation | Period |
|---|---|---|
| T1 | 1.5 | 3.0 |
| T2 | 1.0 | 8.0 |
| T3 | 3 | 8 |

Answer: Back-donate slack to tasks that borrowed from the future

# Principles

1. Allocate slack as early as possible
   - With the priority of the donating task
2. Allocate slack to the task with highest priority (earliest original deadline)
   - *Task* deadline, not *server* deadline
3. Allow tasks to borrow against their own future resource reservations to complete their current job
   - With the priority of the donating job
4. Retroactively allocate slack to tasks that have borrowed from their current budget to complete a previous job

SRAND

+

SLAD

+

SLASH

+

BACK SLASH

# BACKSLASH Conclusions

NICTA

- In an integrated system supporting HRT, SRT and BE, the performance of SRT (and BE) depends on the effective reclamation and distribution of slack
- Four principles for effective slack reclamation and distribution:
  1. Distribute slack as early as possible
  2. Give slack to the ready task with the highest priority
  3. Allow tasks to borrow against future reservations
  4. Retroactively give slack to tasks that needed it
  5. SMASH: Conserve slack across idle times!
- Our results show that these principles are effective: BACKSLASH significantly outperforms the other algorithms and improves SRT (and/or BE) performance

# Books and other Info

Burns, Alan & Wellings, Andrew: Real-Time Systems and Programming Languages (3rd ed), Addison Wesley, 2001

Kopetz, Hermann: Real-time Systems : Design Principles for Distributed Embedded Applications, Kluwer Academic Publishers, 1997

Joseph, Mathai: Real-time Systems: Specification, Verification and Analysis, 2001

http://www.tcs.com/techbytes/htdocs/RTSbook.zip

Dale A. Mackall: *Development and flight test experiences with a flight-crucial digital control system*. NASA Technical Paper 2857, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1988.