

RTS2 Summary

Matthias Jung

20. April 2010

1 Aperiodic Task Scheduling

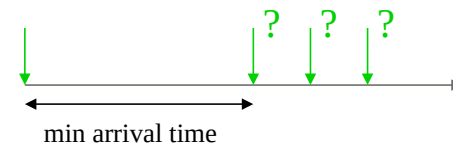
Sporadic:

Aperiodic but minimum interarrival time.

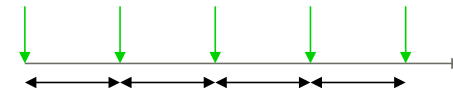
Worst case: All Sporadic Tasks arrive with highest frequency (minimum interarrival time)

Proof method: If we can schedule the worst case we can schedule all other. → Assume sporadic as periodic for scheduling tests

Example:



Worst Case:



Aperiodic:

No limitations on arrival times known

- *Soft:* no deadline → not much to do from the scheduling point of view
- *Firm:* with deadline → WECT has to be known.

1.1 Rate Monotonic with Aperiodic Tasks (Fixed Priority)

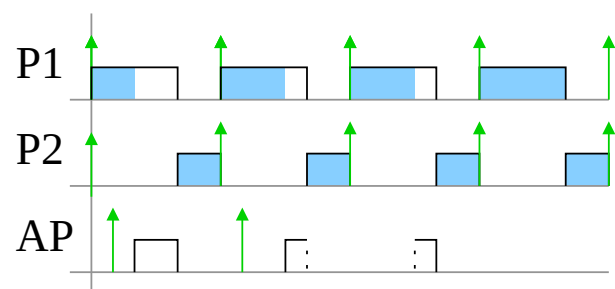
1.1.1 Background Service:

Executes aperiodic tasks when no periodic ones are executing

- Demand: no disturbance of periodic tasks and their feasibility
- Simple runtime mechanisms (queues)
- But no guarantees

Background service lives from 'left overs' of periodic tasks without any guarantee, but it is okay with enough idle time.

Example:



1.1.2 Polling Server:

Introduce a server pseudo task whose purpose it is to service aperiodic requests as soon as possible (T_S , C_S)
Algorithm:

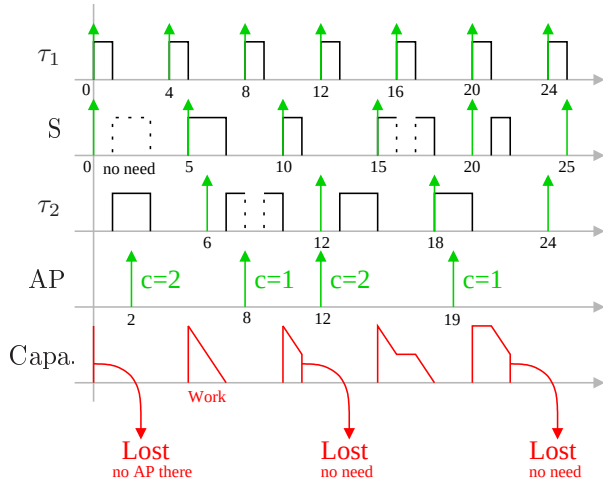
- At period T_S the server becomes active
Serves aperiodic activities
Or if no aperiodic activities → no execution, capacity is lost.
- Wait for next period

Average response time is not very good.

Example:

Two periodic tasks and one aperiodic

	C_i	T_i
τ_1	1	4
τ_2	2	6
S	2	5



Aperiodic Guarantee of Polling Server:

Worst case:

The aperiodic request misses the server task. The aperiodic task has to wait until the next server instance

Time of the server: C_A

Time of the aperiodic: C_S

If $C_A \leq C_S$ then the request is completed within two server periods (one for waiting and one for executing)

$$2 \cdot T_S \leq D_A$$

If $C_A > C_S$ then:

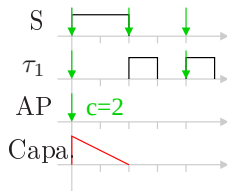
$$\underbrace{T_S}_{\text{wait}} + \underbrace{\left\lceil \frac{C_A}{C_S} \right\rceil T_s}_{\text{execution}} \leq D_A$$

1.1.3 Priority Exchange Server:

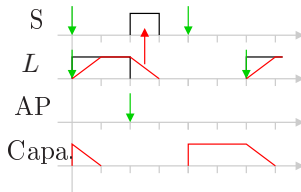
Preserve the capacity by exchanging capacity for lower priority execution time.

- For shorter response times $\rightarrow$ high priority for server task S
- At the beginning of period: the capacity of server is refilled

1: There is an aperiodic request:



2: If there is no aperiodic request a lower priority task L can be executed. Exchange capacity of server with execution of L

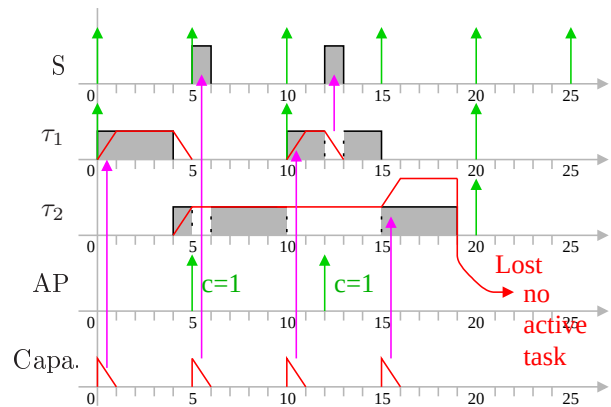


- L executes with priority of S
- S accumulates capacity at priority level of L from execution time of L
- S let L execute from its capacity and L gives its reserved capacity back to S later.

Example:

Two periodic tasks and one aperiodic

	C_i	T_i
S	1	5
τ_1	4	10
τ_2	9	20



1.1.4 Sporadic Server:

Replenish capacity only after aperiodic execution

1.1.5 Optimum Algorithm:

A optimum algorithm does not exist.

1.1.6 Slack Stealing:

Slack: difference between task's deadline and current computation time (time by which task can be delayed without missing its deadline).

- No periodic server task for AP
- Passive task (slack stealer) which tries to steal time for AP from Ps.

Summary:

	Performance	Complexity	Memory	Implementation
Background Service	☹️	😊	😊	😊
Polling Server	☹️	😊	😊	😊
Priority Exchange Server	☹️	☹️	☹️	☹️
Sporadic Server	☹️	☹️	☹️	☹️
Slack Stealer	😊	☹️	☹️	☹️

1.2 Earliest Deadline First with Aperiodic Tasks (Dynamic Priority)

1.2.1 Dynamic Priority Exchange Server

Like rate monotonic priority exchange for EDF

1.2.2 Dynamic Sporadic Server

Like rate monotonic sporadic server for EDF

1.2.3 Earliest Deadline Late Server

Execute periodic tasks ALAP and the aperiodic tasks ASAP. Free space for server which gets therefore a high priority.

1.2.4 Total Bandwidth Server

How to shorter the response times?

- Change the deadline of aperiodic task to a earlier time.
- Make sure that total load of aperiodic tasks does not exceed maximum value for the server U_s
- TBS assigns deadlines based on maximum U_s
- Uses all bandwidth of the Server (not CPU!)
- No extra server task

$$d_0 = 0$$

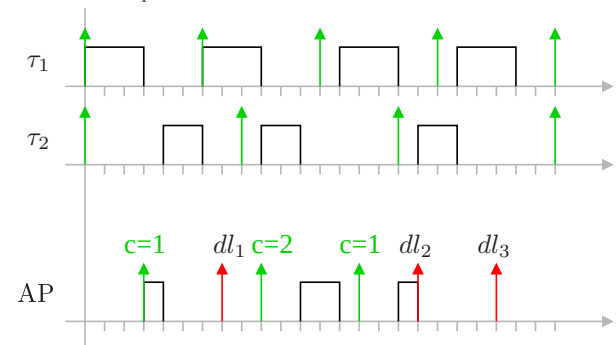
$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s}$$

$$U_p + U_s \leq 1$$

Problem is when a aperiodic task overruns.

Example:

$$U_s = 1 - U_p = 0.25$$



$$dl_1 = 3 + \frac{1}{0.25} = 7$$

$$dl_2 = 9 + \frac{2}{0.25} = 17$$

$$dl_3 = \max(14, 17) + \frac{1}{0.25} = 21$$

1.2.5 TB*

- Assigns deadline d_k first according TBS
- Then shortens as much as periodics allow
- The new d'_k is the finishing time according to EDF schedule including periodics
- Very complex due to recursion

1.2.6 Constant Bandwidth Server

The CBS approach handles the overload of aperiodic tasks. When a new job enters the system a suitable deadline is assigned. If the job tries to execute more than expected its deadline is postponed (priority is decreased)

Periodic $\leftrightarrow$ Hard task

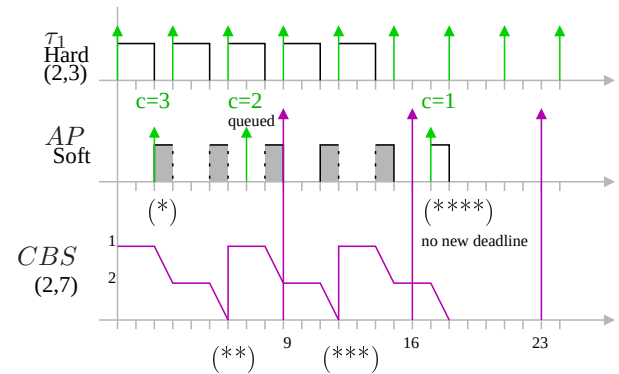
Aperiodic $\leftrightarrow$ Soft task

Cases:

- AP arrives and CBS is active:
 - put in queue, wait
- AP arrives and CBS is idle:
 - if $c_s \geq (d_{s,k} - r_{i,j}) \cdot U_s$:
 - new deadline at $d_{s,k+1} = r_{i,j} + T_s$
 - else :
 - AP served with last server deadline $d_{s,k}$
- $c_s = 0$:
 - budget recharged
 - new server deadline
$$d_{s,k+1} = d_{s,k} + T_s$$

Example:

$$U_s = \frac{2}{7} = 0.28571$$



(*) $c_s = 2 \geq (d_0 - r_1) \cdot U_s = -0.57$: $d_1 = r_1 + T_s = 9$

(**) $c_s = 0$: $d_2 = d_1 + T_s = 16$

(***) $c_s = 0$: $d_3 = d_1 + T_s = 23$

(****) $c_s = 1 \leq (d_3 - r_3) \cdot U_s$: no new deadline

2 Offline Scheduling with Aperiodic Tasks

2.1 Offline Pre-Scheduling

Offline Scheduling:

- Offline constructed Schedule
- General timing constraints
- Offline scheduler resolves constraints
- Constructs one possible solution which meet all constraints
- Fixed runtime execution - no flexibility
- Include aperiodic tasks and schedule them online
- Overconstrains schedule to give place for AP

2.1.1 Pseudo Periodic:

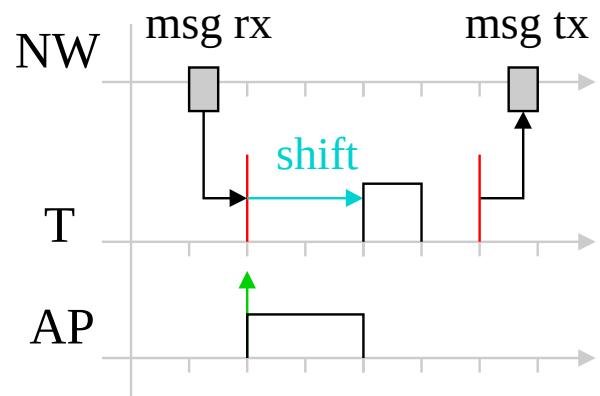
Include AP in offline scheduling with a pseudo periodic task or fit into empty slots.

→ No guarantees, we can do it better!

2.1.2 Slot Shifting:

- Execute the aperiodic task at once
- Shift the periodic
- Feasibility of tasks not violated, but how much can we shift?

Slot Shifting Idea:



Limitations for Slot Shift:

- Receipt of a message
- Sending of a message
- Earliest start time and deadline of:
 - precedence graph (task chain)
 - end-to-end constraints

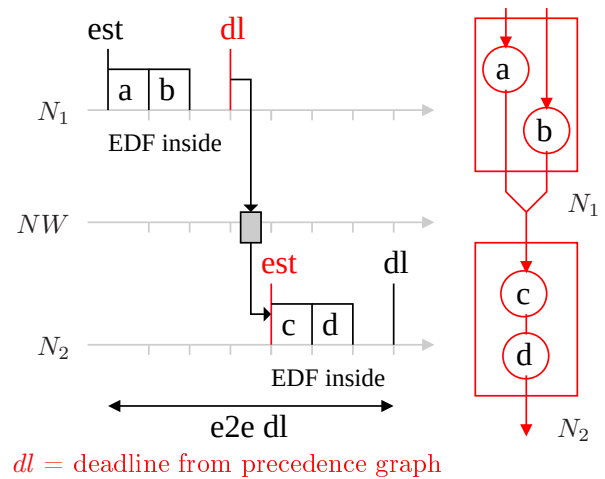
Calculation of Deadlines:

Calculate start-time - deadline pairs for the tasks:

- Offline scheduling: These tasks are assigned fixed starttimes and deadlines
- Expresses feasibility of task
- Reduces overconstraining
- Fit AP by shifting as long as the constraints are met

→ Independent tasks with est, dl on single nodes: use EDF inside the node.

Example:



Spare Capacities - How Much Can we Shift?:

The spare capacity SC is the time which is available for AP tasks. We know the amount an location from offline scheduling.

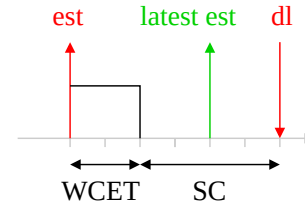
First approach to calculate SC (almost the truth):

$$SC = |dl - est| - WCET$$

Administration:

At least 3 informations are needed for each task (int):

- est
- dl
- WCET



Interval:

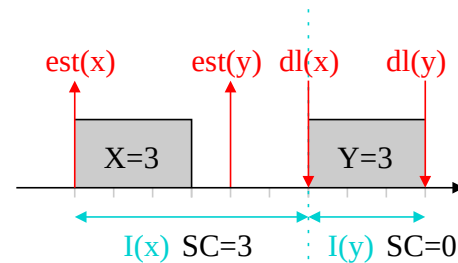
Sort the deadlines to obtain disjoint intervals! (intervals $\neq$ execution intervals):

est : $\max(\text{est of task(s)}, \text{dl of previous interval})$

dl : deadline of task(s)

SC : idle resources in that interval

Example:



Borrowing Mechanism:

If tasks in a subsequent interval need more resources than available they can borrow these from another interval.

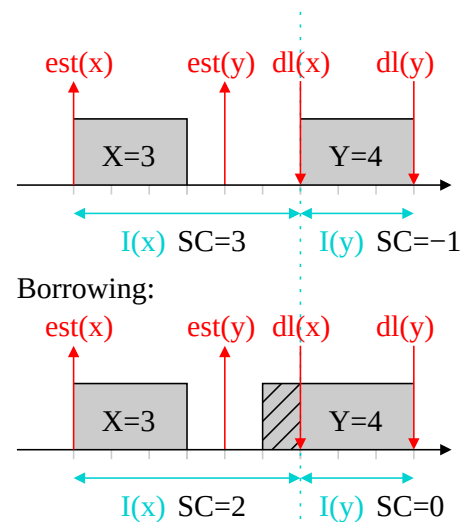
→ start time of interval $\neq$ earliest start time

- only for calculation and flexibility
- runtime mechanisms to resolve negative SC

So the spare capacity can be calculated as follows:

$$SC(I_i) = |I_i| - \sum_{\tau \in I_i} WCET(\tau) + \min[SC(I_{i+1}), 0]$$

Example:



2.2 Online Application of Offline Scheduling

Online Mechanisms:

Online scheduler is invoked at each node after each slot:

- Check for new AP tasks
- Guarantee Algorithm (*)
- Take scheduling decision
- Update SC
- Execute scheduling decision

Scheduling Decision:

- No read task $\rightarrow$ CPU Idle
- $SC(I_c) > 0, \exists$ soft AP: execute AP
- $SC(I_c) > 0, \neg \exists$ execute offline or guaranteed task (no shift)
- $SC(I_c) = 0$: an offline or previously scheduled guaranteed firm AP has to be executed or a deadline is missed

Acceptance of AP:

Soft AP (without a dl):

$SC > 0$ one slot can be given to it

Firm AP (with $WCET$ and dl):

Executed either completely or not at all

A guaranteed algorithm is needed (*)

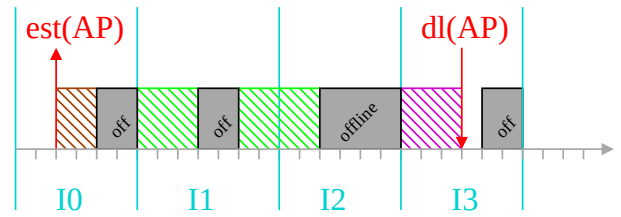
(*) Guarantee New Intervals:

We have to ensure that guarantees resources are not used otherwise.

If dl of a new AP and no other task has same dl :

- Create a new interval for AP
- Split interval where it's deadline is in
- End of new interval is deadline of guaranteed task
- Recalculated spare capacities.
- After guarantee:
 - update I_l , update I_{l-1} , ... , update I_c

How to Give Guarantee:



$SC(I_c)$ = remaining SC in current Interval

$SC(I_i) : c < i \leq l :$

$end(I_i) \leq dl(AP) \wedge end(I_{i+1}) > dl(AP) = SC$ in all full intervals between $est(AP)$ and $dl(AP)$.

$$\min [SC(I_{i+1}), dl(AP) - end(I_i)]$$

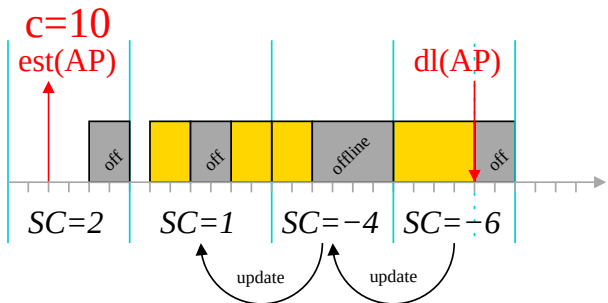
If the sum of total SC between $est(AP)$ and $dl(AP)$ are larger or equal than $WCET(AP)$ we can guarantee the execution

Update SC :

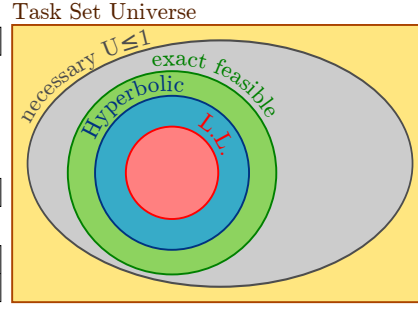
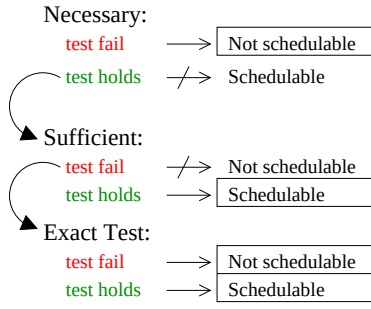
Update capacity of I_f :

- if ≥ 0 done
- if < 0 update previous in. I_{f-1}
- ...
- until $sc \geq 0$ or I_c

Example:



3 Hyperbolic Bound - Better Test for Rate Monotonic:



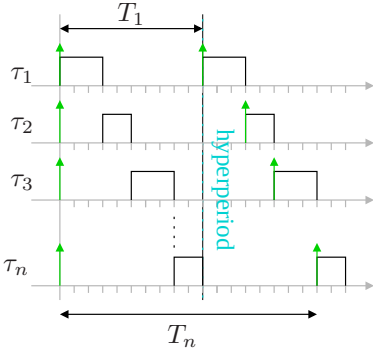
Necessary Test: $\sum_{i=1}^N \frac{C_i}{T_i} = U \leq 1$

Sufficient Tests:
Liu Layland: $U \leq n \cdot \left(2^{\frac{1}{n}} - 1\right)$

Hyperbolic Bound: $\prod_{i=1}^N (1 + U_i) \leq 2$

Exact Test:
 $R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_j^n}{T_j} \right\rceil \cdot C_j$

3.1 Derivation and Proof



Worst Case:

The worst case for Rate Monotonic is when all tasks arrive simultaneously (*critical instance*) and when the period of all tasks is shorter than two times the period of task₁: $\forall i = 2, \dots, n \quad T_1 < T_i < 2 \cdot T_i$

Fully Utilized:

The system is fully utilized when any increase of C_i will crash the system.

From picture:

$$C_1 = T_2 - T_1(*) \quad , \quad C_2 = T_3 - T_2 \quad , \quad \dots \quad , \quad C_{n-1} = T_n - T_{n-1}$$

With (*):

$$C_1 = T_2 - T_1 \quad | + T_1$$

$$T_2 = C_1 + T_1 = T_1 \cdot \left(\frac{C_1}{T_1} + 1 \right) = T_1 \cdot (1 + U_1)$$

$$T_{i+1} = T_i \cdot (1 + U_i) \quad 1 \leq i \leq n-1 \quad (1)$$

All task in the interval $[0, T_n[$ are executed twice:

$$T_n = C_n + 2 \cdot \sum_{i=1}^{n-1} C_i \quad | + C_n$$

$$T_n + C_n = 2 \cdot C_n + 2 \cdot \sum_{i=1}^{n-1} C_i = 2 \cdot \sum_{i=1}^n C_i$$

$$T_n \cdot (1 + U_n) = 2 \cdot \sum_{i=1}^n C_i \quad (2)$$

All tasks in the interval $[0, T_1[$ are executed once :

$$T_1 \geq \sum_{i=1}^n C_i \quad (3)$$

With (2) and (3):

$$T_n(1 + U_n) \leq 2 \cdot T_1$$

Now insert (1) recursively:

$$(1 + U_n) \cdot (1 + U_{n-1}) \cdot \dots \cdot (1 + U_1) \cdot T_1 \leq 2 \cdot T_1$$

$$\prod_{i=1}^N (1 + U_i) \leq 2$$

4 Time-Triggered and Event-Triggered Scheduling

4.1 Time-Triggered (Offline):

Activities are initiated at predefined points in time. Everything is planned before the system is deployed.

- Rules for runtime dispatcher expressed as scheduling table
- Assumes a periodic world

- 😊 Deterministic (known what happens when)
- 😊 Good for complex demands and dist. sys.
- 😊 Low runtime overhead and good resource util.
- 😊 Non temporal constraints are easy to integrate
- 😊 Complex algorithms but refinement!
- 😞 Inflexible can handle only what is known before
- 😞 High cost because we have to know everything
- 😞 A task with unknown WCET can not be handled, and pessimism due to WCET

4.2 Event-Triggered (Online):

An Event initiates activities in a system immediately.

- Rules are applied at runtime (RM, EDF...)
- Assumes no periodic world
(But for offline guarantees ET has to assume periodic as well)

- 😊 Flexible, not known act. can be added easily
- 😊 Widely used.
- 😞 Only simple constraints
- 😞 High runtime overhead
- 😞 Limited predictability

5 How to do a Offline Scheduling

Analyzing Scheduling for How Long?

Analyze scheduling for one Hyperperiod H :

$$H = \text{lcm}(\text{all involved periods } T)$$

(which can be large). After H the scheduling repeats.

Requirements to Offline Scheduling:

We want to find solutions (NP hard $\rightarrow$ can take very long time). If we find a possible solution we are done (solution might be a bad one!). Even maybe a solution does not exist (infeasible)

We have to find:

- Fast detection of **no** solution
- Find a feasible solution.

5.1 Different Approaches:

Online Approach:

- A online procedure is used (e.g. EDF ...)
 - But backtrack or redesign is possible.
- $\rightarrow$ Better utilization, WC is often not needed.

Search Algorithms:

- Generate search tree from precedence graph
- Easy known algorithms
- Easy to change constraints
- Problems with feedback

Genetic Algorithms:

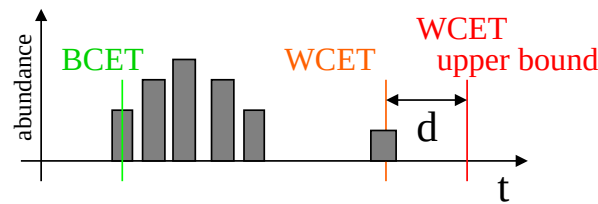
- Simple heuristics (e.g. Simulated Annealing)
- Does not get stuck easily with hard sub-problems
- Difficult with complex constraints

By Hand:

- Sometimes really fully by hand
- Resolve difficult parts
- Extend existing scheduling
- Place only some tasks by hand

6 WCET analysis

WCET Analysis:



- Execution time depends on input data.
- Difference between upper bound and actual WCET (d) = lost resources
- Try to have d small
- The according input to WCET might never happen
- Pessimistic

Simple Architecture: Count

p Program
 et Execution time
 a Action, Command
 n Execution Count

$$et(p) = \sum_i n_i \cdot t(a_i)$$

Complex Architecture: Context

c Execution Context

$$et(p) = \sum_i \sum_j n_i \cdot t(a_i, c_j)$$

Method of Measurement:

- Start measurement and execute
- Stop measurement
- $t_{\text{stop}} - t_{\text{start}}$ = execution time?
- have one execution time for one particular input and one particular situation
- measuring all different execution traces is intractable in practice
- selected test data for measurement may fail to trigger the longest execution trace
- internal processor state may have not been in its worst-case initial settings
- worst case = have to try all possible inputs

Method of Analysing Source code:

- Analyse program without running it (source code)
- Guaranteed safe WCET (safe bound tight to real WC)
- Use annotation and restrictions in program...
- **Flow Analysis** identifying execution paths in program
- **Low Level Analysis** model hardware
- then calculate

Deriving flow information:

In general, automation is impossible...

Treebased calculation:

Bottom-up traversal of syntax tree, using rules to compute timing of compound program statement.

Example:

$$T(\text{for}) = (N + 1) \cdot T(\text{test}) + N \cdot T(\text{body})$$

$$T(\text{if}) = T(\text{test}) + \max(T(\text{then}), T(\text{else}))$$

- simple method with "cheap" computation effort
- scales very good with program size
- false paths possible

Pathbased calculation

- path: instructions that can be executed during a single iteration of a loop
- calculate times for different paths in a program
- flow facts can be used to prune set of valid paths
- complex

PbC: Implicit path enumeration technique

The program structure is mapped onto a set of graph flow constraints. Methods like integer linear programming (ILP) or constraint-solving used to calculate the WCET

7 Clock Synchronization:

Why Clock Synchronization:

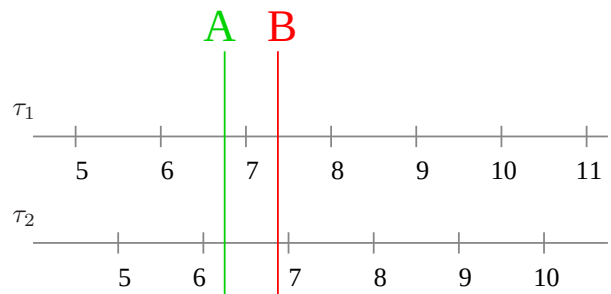
- Time basic is necessary in real-time systems.
- In distributed systems sync between nodes
- Global time base: all nodes have access to same time
- Ordering of Events? What happens before?

Problems:

- Clocks do not have the same speed (e.g. due to temperatures)
- Failures
- Communication delays

Need two ticks difference in measurement for order

Example for What Happens Before?

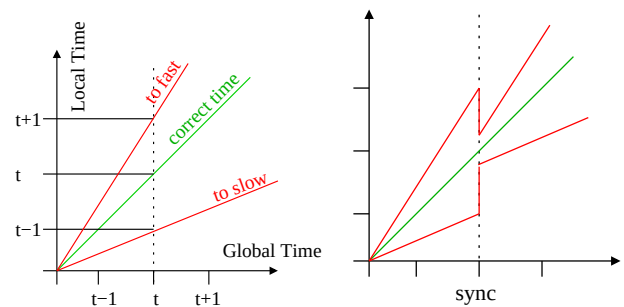


τ_1 can say that A was for B.

τ_1 cannot say that A was for B.

7.1 Master:

- ☹ Easy
- ☹ Communication (e.g. delays ...)
- ☹ Failure of the master



7.2 With Average:

for each clock:

- Send out local time
- Collect times from others
- Perform sync (average)
- Set clock to computed value

end

Average does not work with faulty clocks k (byzantine errors)

$$n = 3 \cdot k + 1$$

7.3 Fault Tolerant Average:

- To tolerate k erroneous clocks the k fastest and the k slowest reading are abandoned
- Correction Matrix
- Convergence Function

$$\Pi(\Delta, N, k, \epsilon) = \frac{k \cdot \Delta}{N - 2 \cdot k} + 2 \cdot \epsilon$$

8 Energy-Aware RT scheduling:

Portable devices:

- Battery dictates the lifetime
- Energy supply not infinite
- Battery capacity is limited by its size

Servers:

- Energy supply is infinite
- Energy is dissipated as heat
- Cooling is necessary
- Cooling needs even more energy (... expensive)

How can we save energy?

Ressources are not fully used 100% of the time (suspend...)

Portable devices:

- Wireless antenna turned off when nothing is transmitted
- Turn CPU off when idle
- Reducing CPU frequency (timing constraints !)

Servers:

- Reducing CPU frequency
- Turn CPU off when idle
- Data placement to shut down some devices
- Memory Compression

8.1 VOVO - Vary On Vary Off:

- During idle periods turn off to save energy
- Incoming work load cannot be handled → Go to lower power incative state
- Worked distribution on processors
- Wake on LAN

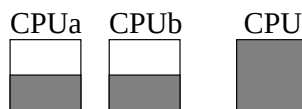
8.2 FVS - Dynamoc Voltage Scaling:

- $E \sim V^3$
- $F \sim V$
- $E \sim F^2$

Decreasing Frequency → decreasing of Voltage → decreasing of Energy

VOVO vs. DVS:

2 processors running at half speed save more energy than one processor at full speed



$$\left(\frac{1}{2}F\right)^2 + \left(\frac{1}{2}F\right)^2 \leq F^2$$

$$\frac{1}{4}F^2 + \frac{1}{4}F^2 \leq F^2$$

$$\frac{1}{2} \leq 1 \quad \checkmark$$

Time and energy overhad in VOVO is higher than DVS. VOVO saven even though more energy in real life because lower frequencies are not always more energy efficient. If the system is often idle VOVO is the best option.

Energy Saving in Real-Time Systems:

RT applications tasks must complete by their deadline

→ Task completion can be delayed as long as they meet their deadlines

8.3 SVS - Static Voltage Scaling:

Feasibility Test:

$$\begin{array}{ll} \text{EDF:} & \text{RM:} \\ \sum_{i=1}^N \frac{C_i}{T_i} \leq \alpha & \sum_{i=1}^N \frac{C_i}{T_i} \leq n \cdot \left(2^{\frac{1}{n}} - 1\right) \cdot \alpha \end{array}$$

Processor is scaled down by factor α . These tests are based on WCET so the tasks may run shorter and more energy might be saved.

8.4 Cycle conserving - EDF-RT-DVS:

- First use SVS
- On task completion:
 - Recompute U with **actual** C_c
 - determine a new factor α
 - See if system can run on lower speed
- On new instance
 - Recompute U with **WCET** $C \rightarrow$ determine a new factor α
 - See if system needs to run at higher speed

→ U is decreased....

8.5 EDF - Lookahead RT-DVS:

- Postpones execution as much as possible
- Tasks that have to execute right now can run at a lower speed
- If all tasks execute with WCET deadline can be met anyway
- If tasks run shorter more energy can be saved

8.6 Leakage Aware Scheduling:

- Complete work AFAP
- Go to dormant state
- Minimize switching by grouping work
- Delay execution of tasks so that they can execute later with other tasks

Results:

Also Frequency switching is not free (time and energy overhead) but better than VOVO.

- Cycle conserving exploits better the energy saving opportunities, but switching should be limited
- Lower speeds are not always the most energy efficient (energy leakage)

9 Wireless Sensor Networks:

Wireless Sensor Networks:

Set of small autonomous systems (sensor nodes) which cooperate to solve at least one common application. Their tasks include some kind of perception of physical parameters.

Challenges:

- Reduce size and computing power
- Save energy
- Scalability and density range
- dynamic NW morphology
- Maintainability

Application:

- Medicine
- Wearable computing
- Disaster relief
- Intelligent buildings

Architectures:

- μC (better than CPU, DSP, FPGA) 1-400 mW
- Memory: mix of rom/ram/flash often in chip
- Sensors: active, passive
- Communication RF

Network: Robustness!

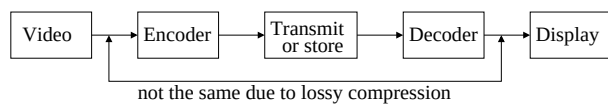
- QoS
- Energy efficiency
- Scalability

Operating System:

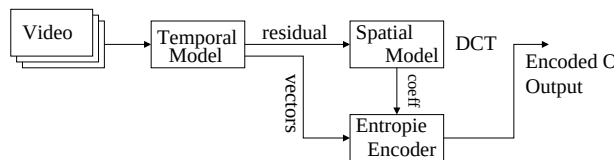
- Event driven
- Control for event and power management

10 MPEG

Video Encoding:



Video Encoder:



Stream Adaption Based on NW-Bandwidth:

Calculation of RNB (Required Network Bandwidth):

$$\text{GOP}_{\text{Size}} = \sum_{i=1}^N \text{Frame Size}(f_i)$$

$$\text{GOP}_{\text{Time}} = \frac{N}{\text{FPS}}$$

$$\text{RNB} = \frac{\text{GOP}_{\text{Size}}}{\text{GOP}_{\text{Time}}}$$

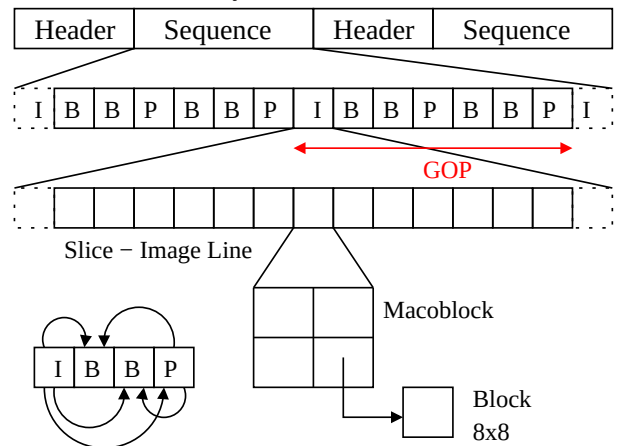
Adaption:

First Approach: Frame Skipping

Second Approach: slice, macroblock, coeff. skipping

A combination of first and second approach give good results.

MPEG Hierarchy:



Procedure:

