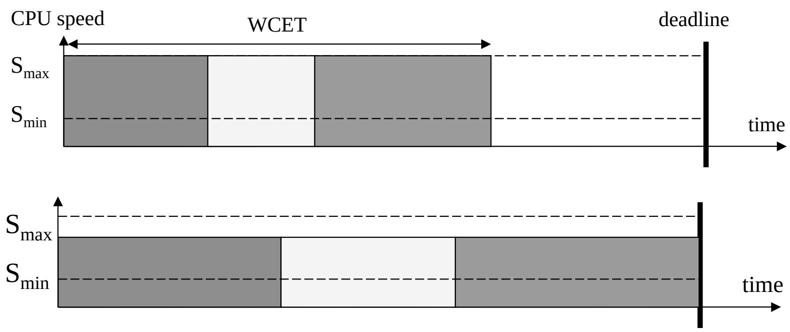
speed adjustment in frame-based systems

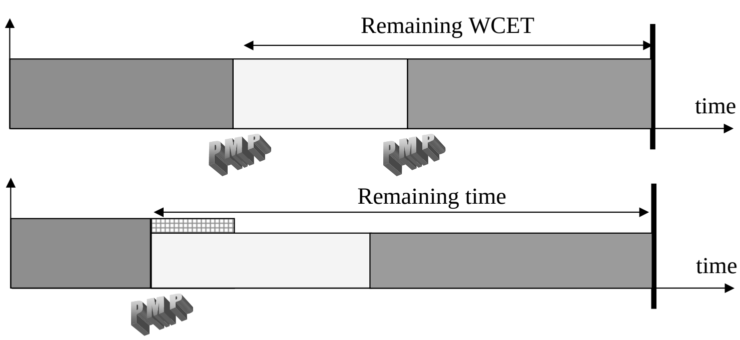
Static speed adjustment

Assumption: all tasks have the same deadline.



Select the speed based on worst-case execution time,WCET, and deadline

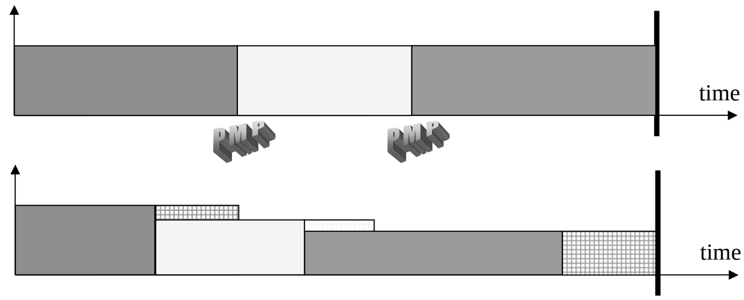
Dynamic Speed adjustment techniques for linear code



Speed adjustment based on remaining WCET

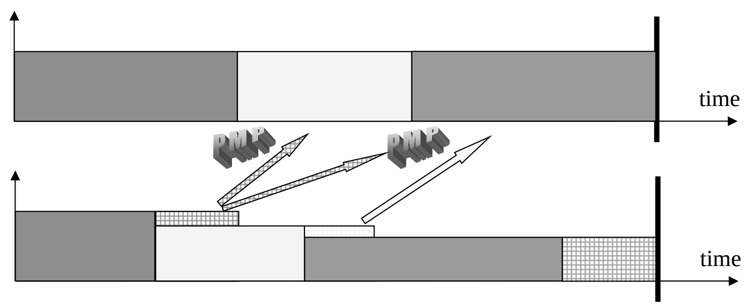
Note: a task very rarely consumes its estimated worst case execution time.

Dynamic Speed adjustment techniques for linear code

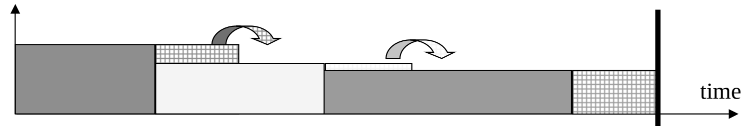


Speed adjustment based on remaining WCET

Dynamic Speed adjustment techniques for linear code



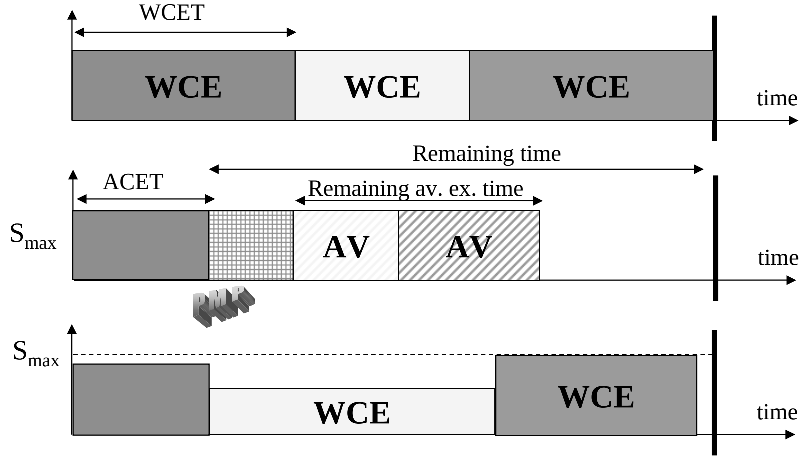
Speed adjustment based on remaining WCET



Greedy speed adjustment:

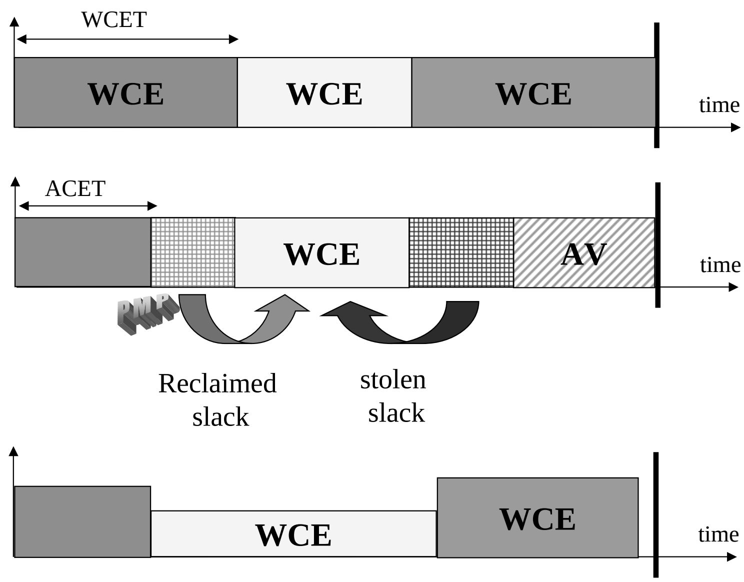
Give reclaimed slack to next task rather than all future tasks

Dynamic Speed adjustment techniques for linear code

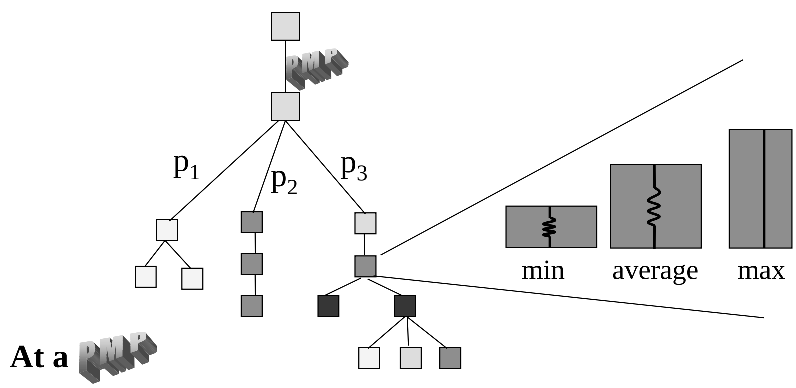


Speed adjustment based on remaining average execution time

An alternate point of view



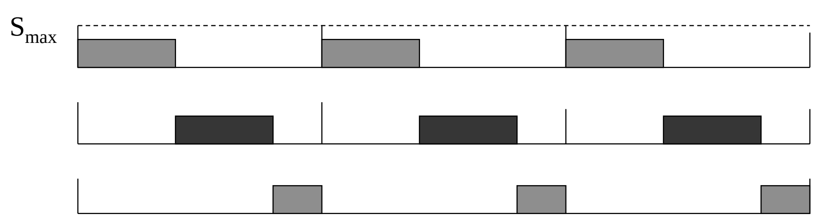
Dynamic Speed adjustment techniques for non-linear code



- Remaining WCET is based on the longest path
- Remaining average case execution time is based on the branching probabilities (from trace information).

2. Periodic, non-frame-based systems

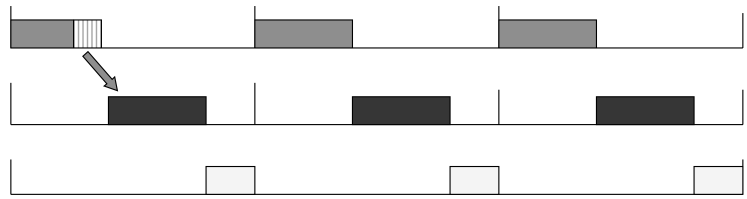
- Each task has a WCET, C_i and a period T_i
- Earliest Deadline First (EDF) scheduling
- **Static speed adjustment**: If utilization $U = \sum \frac{C_i}{T_i} < 1$, then we can reduce the speed by a factor of U, and still guarantee that deadlines are met.



Note: Average utilization, U_{av} can be much less than U

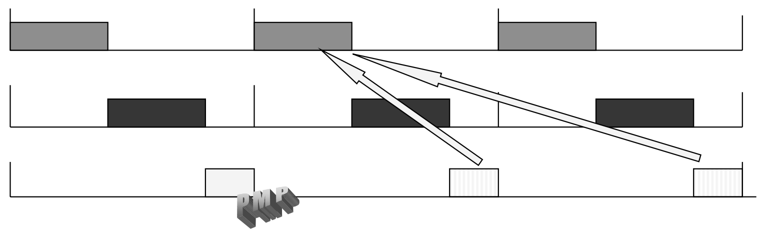
Greedy dynamic speed adjustment

correct



- Giving reclaimed slack to the next ready task is not always a correct scheme

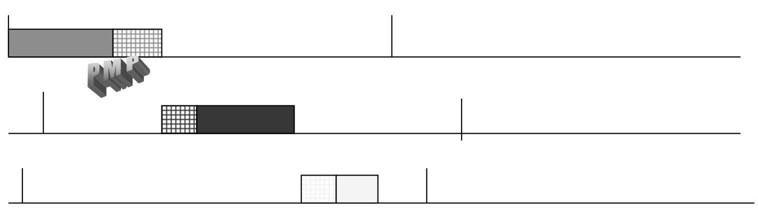
incorrect



- **Theorem:** A reclaimed slack has to be associated with a deadline and can be given safely to a task with an earlier or equal deadline.

aggressive dynamic speed adjustment

- **Theorem:** if tasks 1, …, k are ready and will complete before the next task arrival, then we can swap the time allocation of the k tasks. That is we can add stolen slack to the reclaimed slack

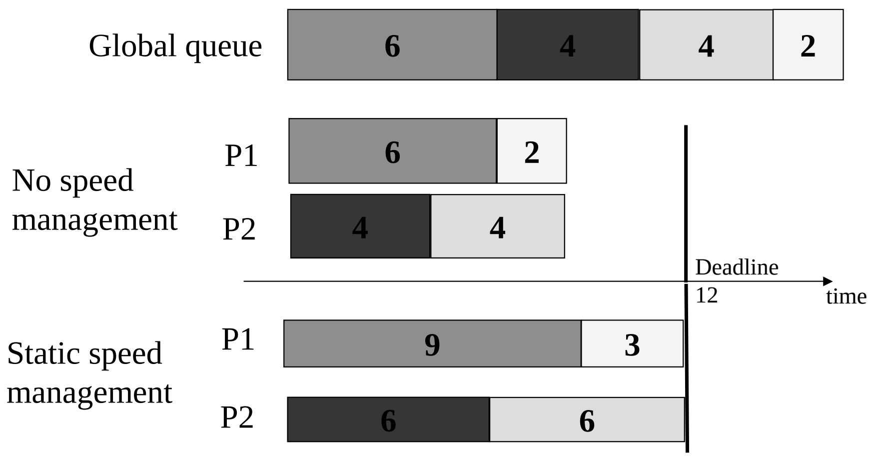


- **Experimental rule:** Do not be very aggressive and reduce the speed of a task below a certain speed (the optimal speed determined by U_{av})

Speed adjustment in Multi-processors

1. the case of independent tasks on two processors

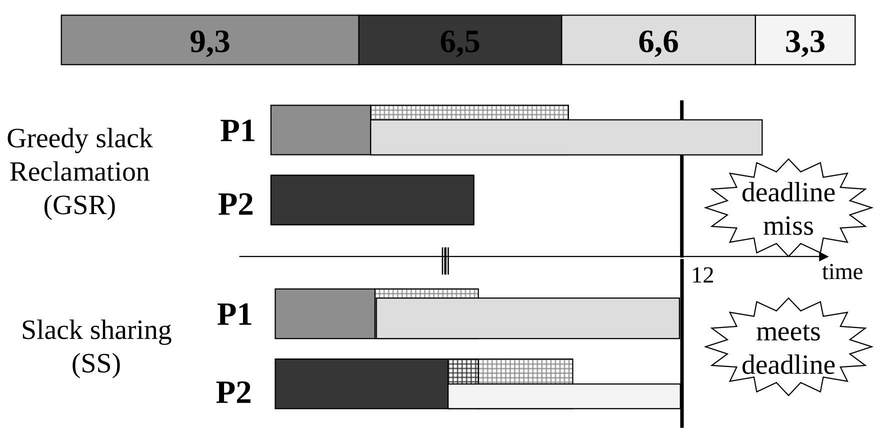
Canonical execution ==> all tasks consume WCET



Dynamic speed adjustment

Non canonical execution ==> tasks consume ACET

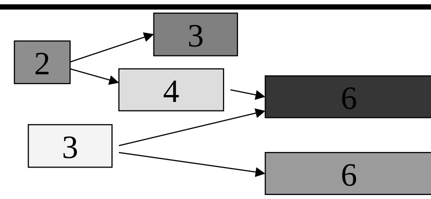
If we select the initial speed based on WCET, can we do dynamic speed adjustment and still meet the deadline?

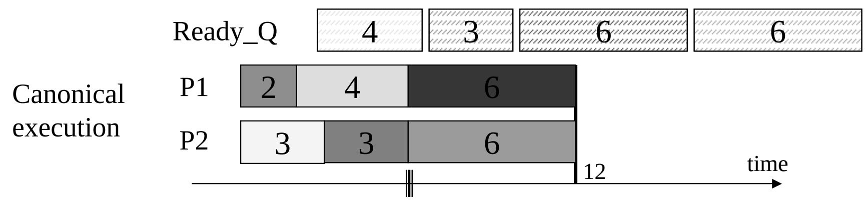


Dynamic speed adjustment (2 processors)

2. dependent tasks

Use list scheduling



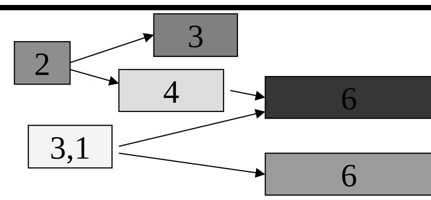


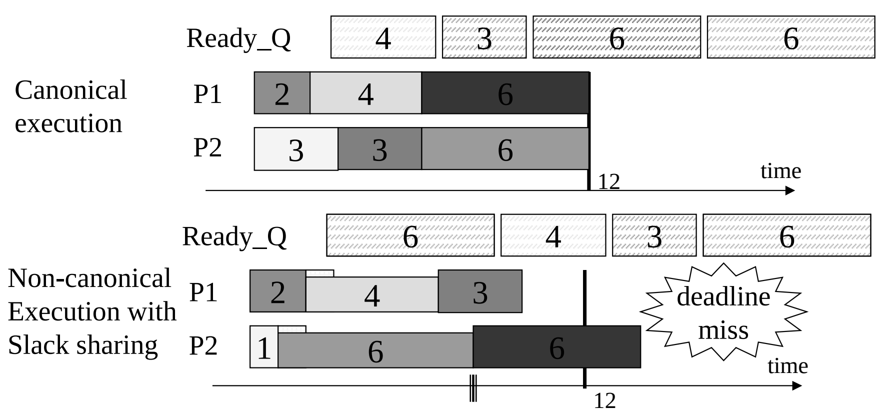
- Assuming that we adjust the speed statically such that canonical execution meets the deadline.
- Can we reclaim unused slack dynamically and still meet the deadline?

Dynamic speed adjustment (2 processors)

2. dependent tasks

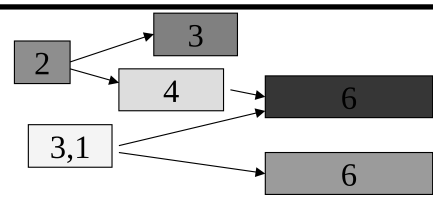
Use list scheduling

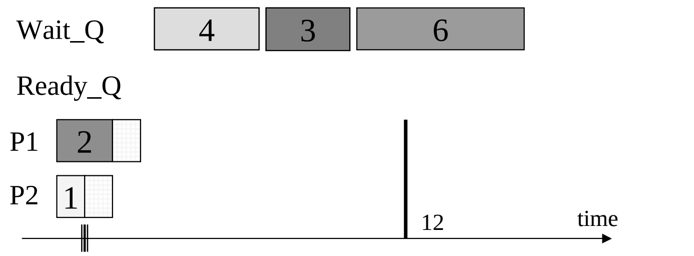




Dynamic speed adjustment (2 processors)

<u>Solution:</u> Use a wait_Q to enforce canonical order in Ready_Q

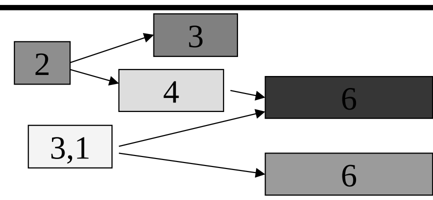


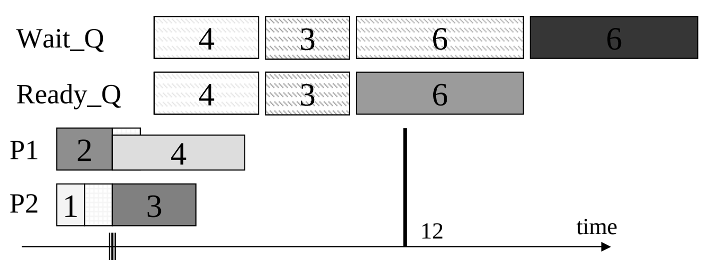


- A task is put in *Wait_Q* when its last predecessor starts execution
- Tasks are ordered in *Wait_Q* by their expected start time under WCET
- Only the head of the *Wait_Q* can move to the *Ready_Q*

Dynamic speed adjustment (2 processors)

<u>Solution:</u> Use a wait_Q to enforce canonical order in Ready_Q

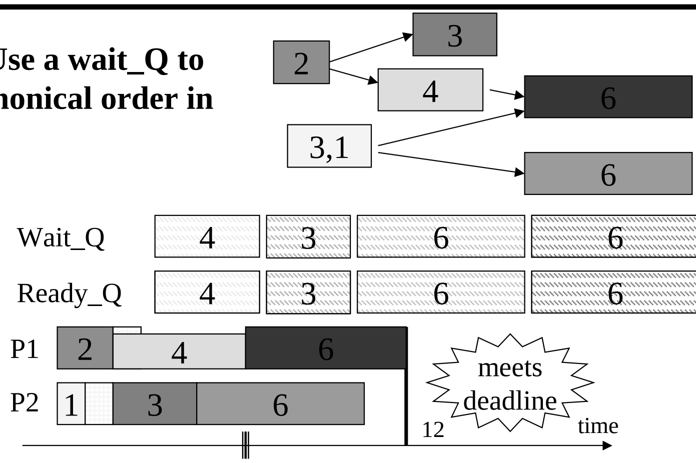




- A task is put in *Wait_Q* when its last predecessor starts execution
- Tasks are ordered in *Wait_Q* by their expected start time under WCET
- Only the head of the *Wait_Q* can move to the *Ready_Q*

Dynamic speed adjustment (2 processors)

<u>Solution:</u> Use a wait_Q to enforce canonical order in Ready_Q



- A task is put in *Wait_Q* when its last predecessor starts execution
- Tasks are ordered in *Wait_Q* by their expected start time under WCET
- Only the head of the *Wait_Q* can move to the *Ready_Q*

Theoretical results

For ***independent tasks***, if canonical execution finishes at time *T*, then non-canonical execution with ***slack sharing*** finishes at or before time *T*.

For ***dependent tasks***, if canonical execution finishes at time *T*, then, non-canonical execution with ***slack sharing and a wait queue*** finishes at or before time *T*.

<u>Implication:</u>

- Can optimize energy based on WCET (static speed adjustment)
- At run time, can use reclaimed slack to further reduce energy (dynamic speed adjustment), while still guaranteeing deadlines.

4. Static optimization when different tasks consume different power

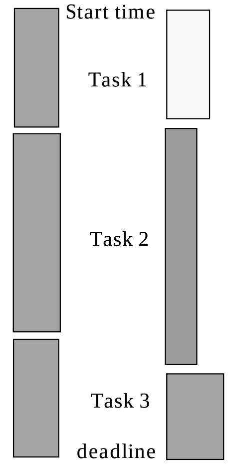
Assuming that the power consumption functions are identical for all tasks.

Then to minimize the total energy, all tasks have to execute at the same speed.

If, however, the power functions, $P_i(S)$, are different for tasks $i = 1, \dots , n,$

Then using the same speed for all tasks does not minimize energy consumption.

Let C_i = number of cycles needed to complete task i



Minimizing energy consumption

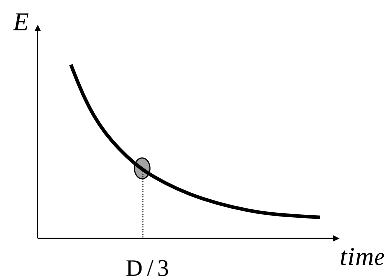
Example: Three tasks with $C_1 = C_2 = C_3$

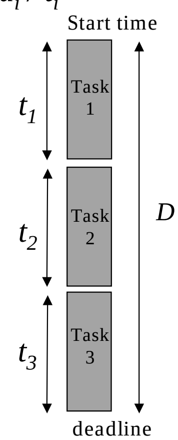
If $P_i(S) = a_i\, S^2$, for task i ,

then, energy consumed by task i is $E_i = a_i / t_i$

If $a_1 = a_2 = a_3$,

then $t_1 = t_2 = t_3$ minimizes total energy





Minimizing energy consumption

Example: Three tasks with $C_1 = C_2 = C_3$

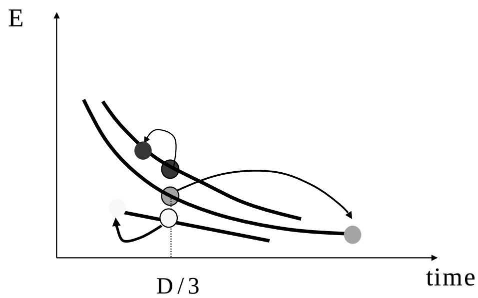
If $P_i(S) = a_i S^2$, for task i ,

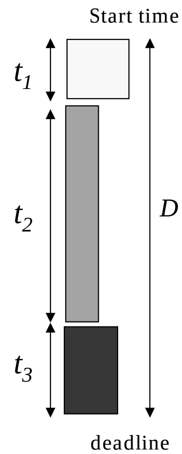
then, energy consumed by task i is $E_i = a_i / t_i$

.

If a_1 , a_2 and a_3 , are different

then $t_1 = t_2 = t_3$ does not minimize total energy





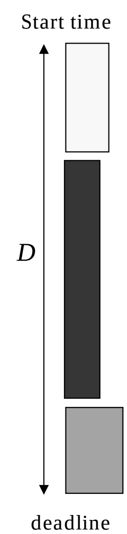
Minimizing energy consumption

The problem is to find S_i , $i=1, \dots , n$, such that to

$$\boxed{\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n} t_i \; P_i(S_i) \\ \text{subject to} & \sum_{i=1}^{n} t_i \le D \\ \text{and} & S_{\min} \le S_i \le S_{\max} \end{array}}$$

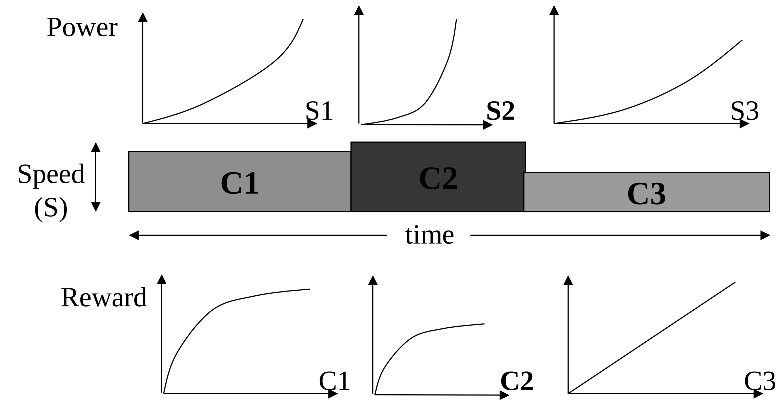
Note that $t_i = \dfrac{C_i}{S_i}$

- We solved this optimization problem, consequently developing a solution for arbitrary convex power functions.
- Algorithm complexity: $O(n^2 \log n)$



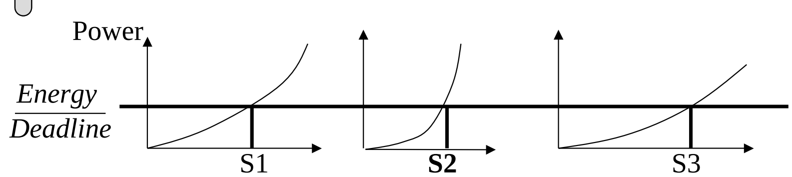
Maximizing the system’s reward

General problem assumptions:
- tasks have different power/speed functions
- tasks have different rewards as functions of number of executed cycles

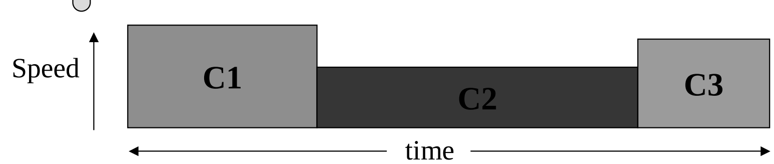


Maximizing the system’s reward

Theorem: If power functions are all of the form $\alpha_i S^p$, then reward is maximum when power is the same for all tasks



Given the speeds, we know how to maximize the total reward while meeting the deadline.



Maximizing reward for a given energy budget

The problem:

find the speeds S_i ,and execution times, t_i , $i=1, \dots , n,$ such as to

maximize $\sum_{i=1}^{n} R_i(C_i)$ — Obtain best reward

subject to $\sum_{i=1}^{n} t_i \leq D$ — Time budget

$\sum_{i=1}^{n} t_i P_i(t_i) \leq E$ — Energy budget

$S_{\min} \leq S_i \leq S_{\max}$ — Limits on speed

$l_i \leq C_i \leq u_i$ — Limits on computation

Note that $t_i = \dfrac{C_i}{S_i}$

We solved this optimization problem analytically for specific forms of the power functions. For arbitrary power functions, we use heuristics.

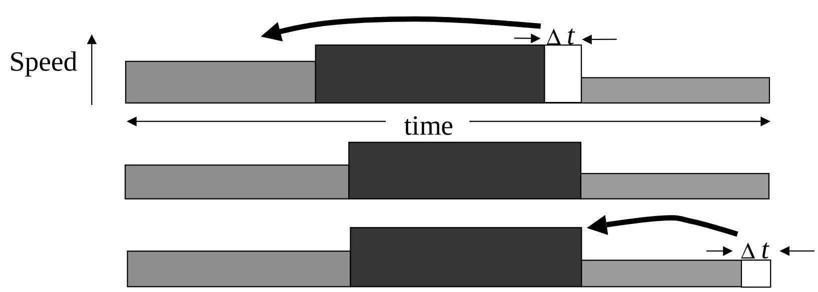
Maximizing reward for a given energy budget

1) Ignore energy and maximize reward, R, within the deadline

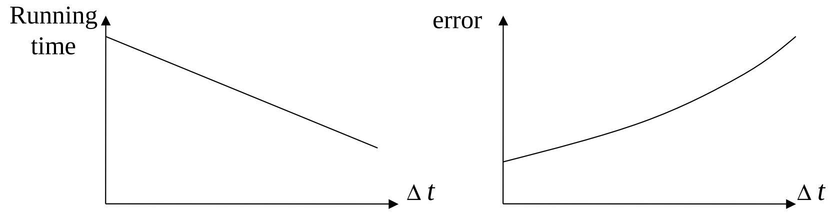
2) If exceed available energy;

– remove Δt from a task such that decrease in R is minimal
– use Δt to decrease the speed of a task, such as to maximize the decrease in energy consumption

3) Repeat step 2 until the energy constraints are satisfied

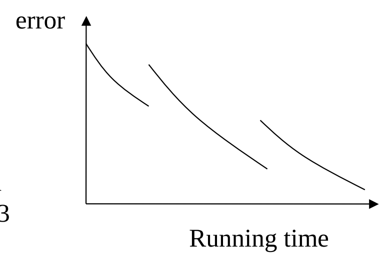


Maximizing reward for a given energy budget



What value should be used for Δt ?

An iterative refinement algorithm:
1) Define a required error, ε
2) solve the problem with an initial Δt
3) set $\Delta t = \Delta t / 2$
4) solve the problem for the new Δt
5) find the improvement in the solution
6) if improvement larger than ε, go to 3



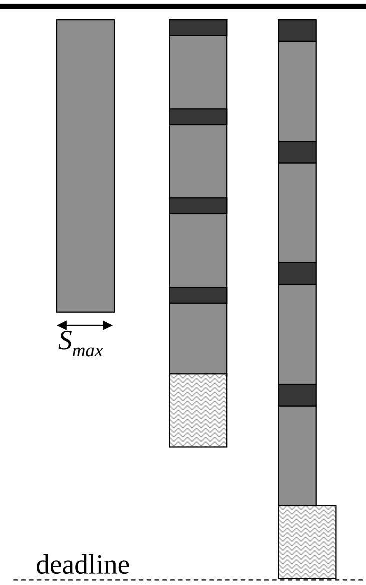
Dual use of time slack

Slack can be used for

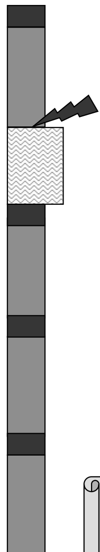
1) fault tolerance

- add checkpoints
- reserve recovery time

2) reduce processing speed



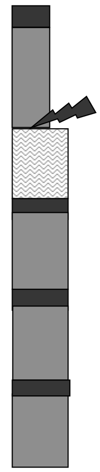
Dual use of time redundancy



Observation: May continue executing at S_{max} after recovery.

Disadvantage: consumes more energy when a fault occurs (rare)

Advantage: recovery in an early section can use slack created by execution of later sections at S_{max}

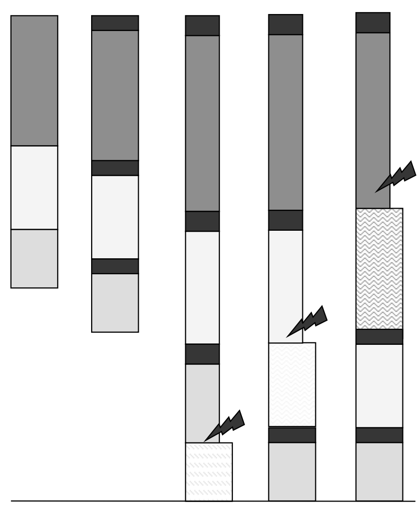


Motivates non-uniform checkpoints.

Non-uniform check-pointing

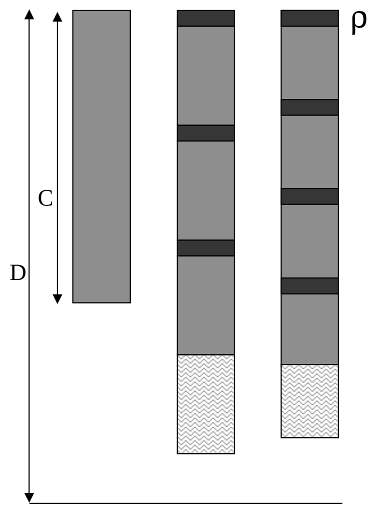
Can save more power by using non-uniformly spaced checkpoints.

Placement of checkpoints is different from placement of the PMPs



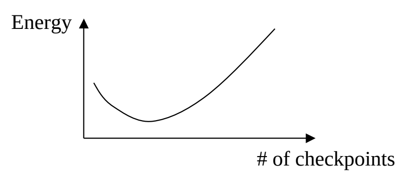
Optimal number of checkpoints

More checkpoints = more overhead + less recovery slack



For a given
- slack (C/D) and
- checkpoint overhead (ρ),

we can find the number of checkpoints that
- minimizes energy consumption, and
- guarantee recovery and timeliness.



Triple Modular Redundancy Vs. Duplex

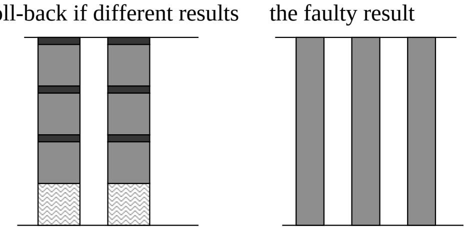
ρ : overhead of checkpoint

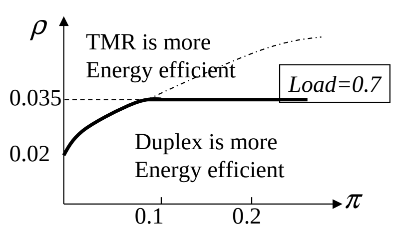
Load : slack in the system

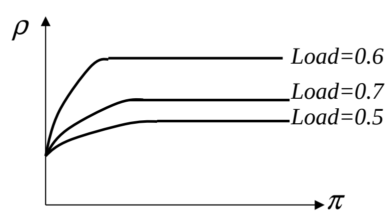
π : ratio of static/dynamic power

Duplex: Compare and roll-back if different results

TMR: vote and exclude the faulty result





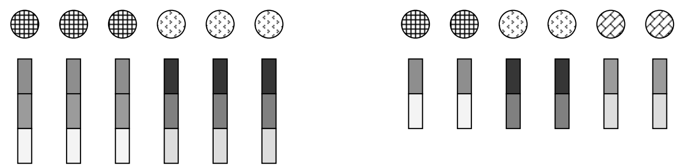


Energy efficiency of TMR Vs. Duplex depends on ρ, π, and *load*

Including the hardware cost

Assuming the ***same number of processors*** and a given task set, to obtain the ***same reliability***, will it be more energy efficient to use TMR or Duplex?

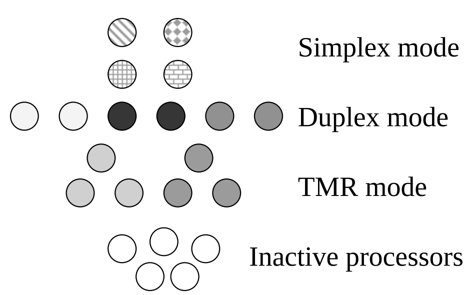
Example: 6 processors and 6 identical tasks.



Need to look at
- the fault model,
- the energy model and
- the overheads.
- the communication cost

Reconfigurable clusters of servers

Reconfigure your cluster to
- achieve required reliability
- minimize energy consumption
- optimize performance
- meet quality of service



Example:
- At a given server’s speed, it is more energy efficient to activate an additional server than to increase the speed of the active servers.
- At a given server’s speed, it is more energy efficient to power down one of the servers than to reduce the speed of the active servers