Real-Time Systems & Fault Tolerance

Flávia Maristela

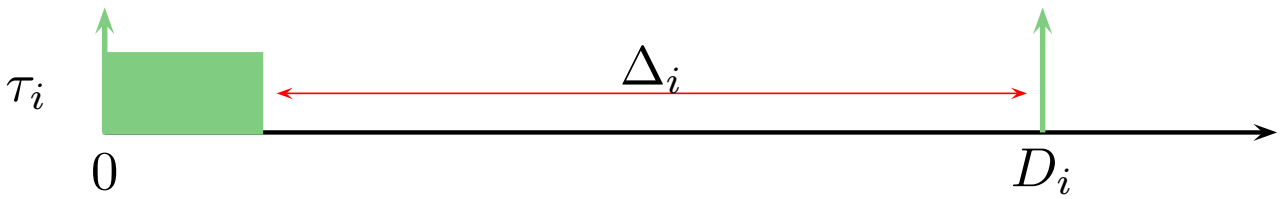
Instituto Federal da Bahia
Especialização em Computação Distribuída e Ubíqua (ECDU)

Salvador, Outubro de 2013

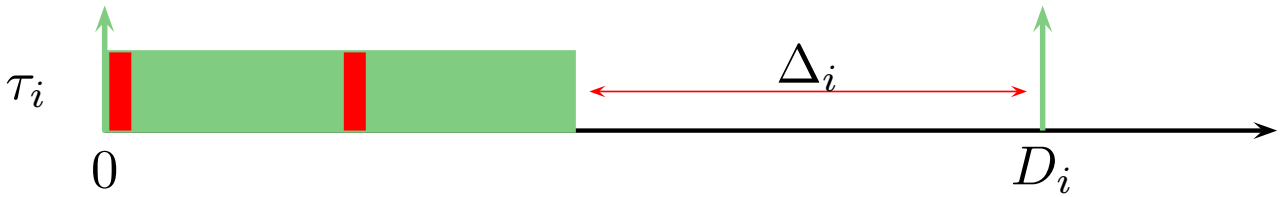
Schedule

- Slacks has been used for fault tolerance purposes:
  - Restart the task or part of it.
- The available slack in the schedule may not allow the entire task re-execution.
- Checkpoints may be used in this case to provide fault tolerance

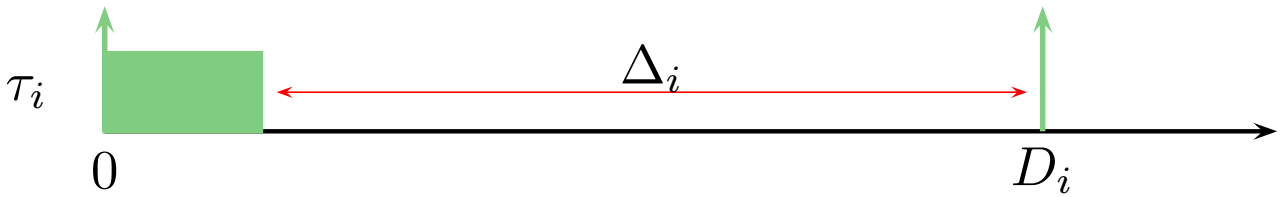
- Task with no checkpoints



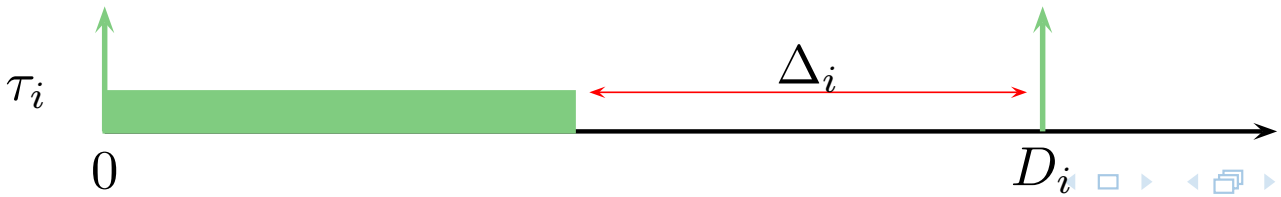
- Task considering checkpoints



- In terms of energy consumption, slacks have been used to slow down the processor such that tasks meet their deadlines.
- Task being executed with total voltage supply



- Task executed with lower voltage supply



- The idea is simple: use the slack for both reducing energy consumption and providing fault tolerance.
- In this case there are two possibilities:
  - Uniform checkpoints placement: the optimal number of checkpoints that must be inserted to minimize the energy consumption, subject to the constraints of recovering from a single failure.
  - Non-uniform checkpoints placement: places checkpoints in a manner where the frequency of checkpointing starts slowly at the beginning and increases as we approach the task deadline.

- Uniform Checkpoint Placement

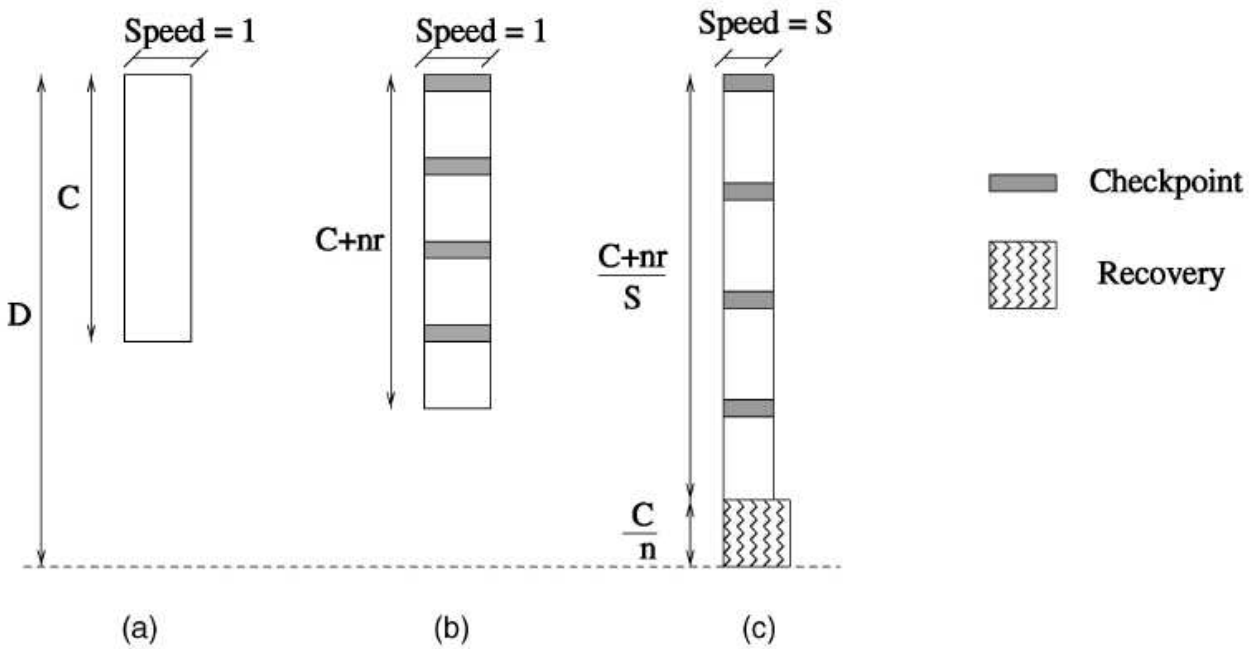


Fig. 1. Equally spaced checkpoints.

- Why recovery is executed at higher speed?
  - To allow more slack to be available for speed reduction
- The speed S allows τ_i to finish on time must also:
  - minimize energy consumption;
  - allows slack for checkpointing overhead;
  - allows time to roll back from a fault

- Some important notes on checkpoints placement:
  - The more checkpoints are taken, the less work is at risk and, therefore, the portion of the slack reserved for rollback-recovery is smaller, allowing the remaining slack to be used for further reduction of processor speed.
  - The overhead of checkpointing consumes a part of the available slack that would be used to reduce speed.
    - As a consequence, the more checkpoints are taken, the less the opportunity to reduce speed.

- **GOAL:** investigate if further speed reduction is possible through an alternative checkpoint placement policy.
- The intuition behind this exploration is based on the optimistic view that failures are not frequent.
  - if a failure occurs, the task executes at maximum speed not only during rollback but also following the rollback and recovery when the task executes the remainder of the computation.

- Non-Uniform Checkpoint Placement

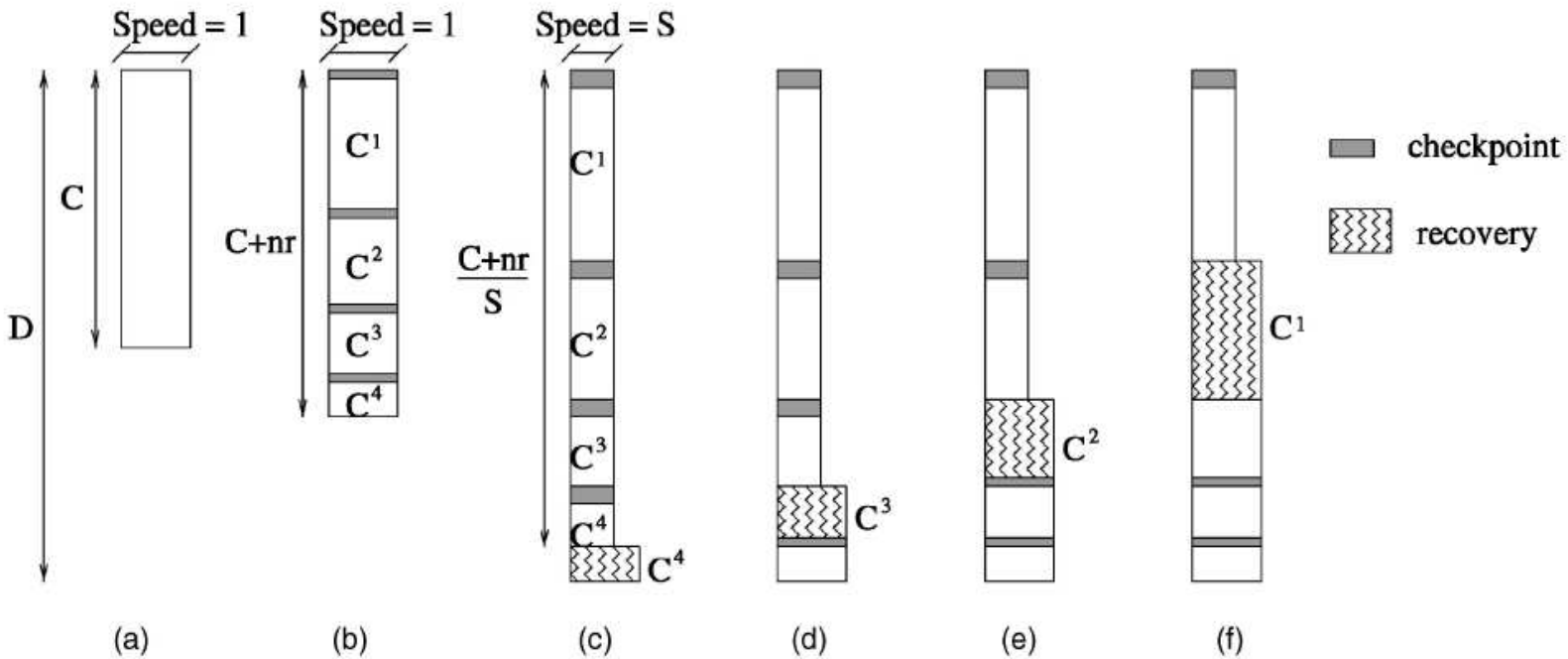
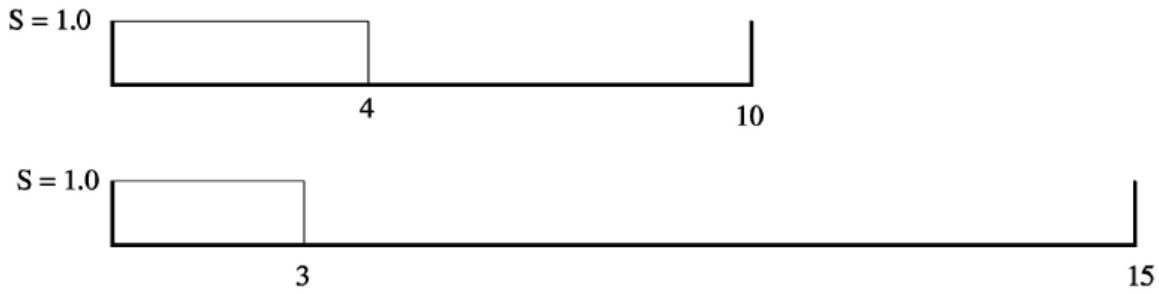


Fig. 4. Nonequally spaced checkpoints.

- Why to do so?
  - At the beginning of execution, the task has not "consumed" much of "available" slack
  - Consequently, low frequency of checkpoints is possible because most of the slack is available to accommodate a large amount of work at risk if we need to recover at maximum speed.
  - As the task continues to execute, it slowly "consumes" the available slack.
  - Gradually, the remaining slack can accommodate decreasing amounts of work at risk and, hence, there is a need for increasing the frequency of the checkpoints to accommodate the decreasing ability of the remaining slack to handle work at risk as the task approaches the deadline.

- Consider a set of two periodic tasks, τ_1 and τ_2, with $C_1 = 4$, $T_1 = 10$, $C_2 = 3$, and $T_2 = 15$



(a)

- At maximum speed, this task set has a utilization $U = 0.6$
- There is 40% slack in the system, which can be used to guarantee error recovery and/or save power consumption.
- Assume that the cost of a checkpoint is fixed at $r = 0.15$.

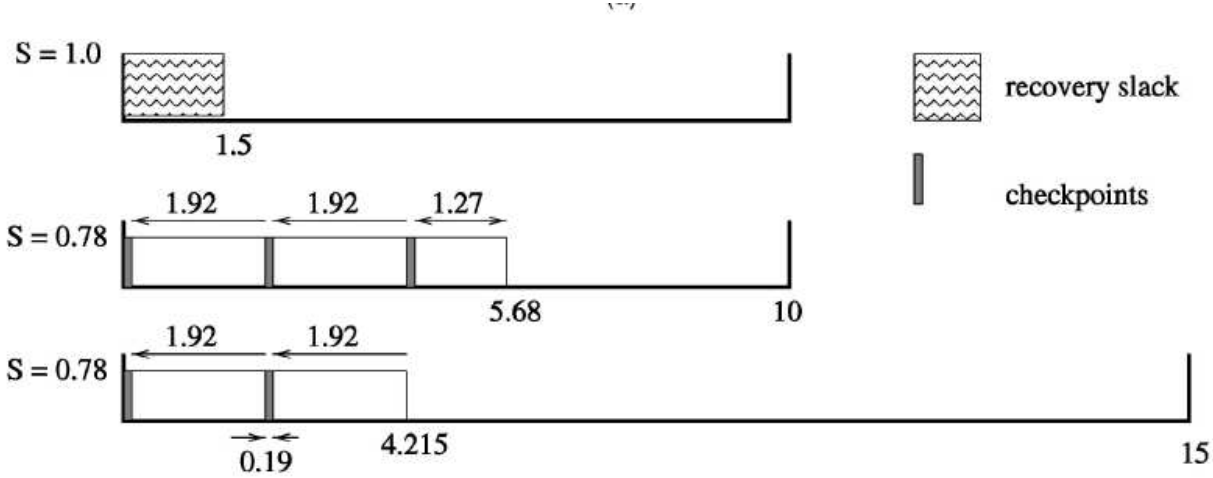
Illustrative Example

- For uniform checkpoints placement, assume $\gamma = 1.5$ is the time interval between two checkpoints. Hence,

$$\lceil \frac{C_1}{\gamma} \rceil = \lceil \frac{4}{1.5} \rceil = 3$$

$$\lceil \frac{C_2}{\gamma} \rceil = \lceil \frac{3}{1.5} \rceil = 2$$

represents the number of checkpoints in C_1 and C_2, respectively.



- Authors extend this approach for multiprocessor real-time systems with global scheduling approach:
  - Why global approach?
  - In partitioned scheduling approach tasks are statically mapped to processors and can only run in the processor it is assigned to. After such a mapping, applying the approach for uniprocessor systems is straight forward.

- System Model:
  - n frame-based independent real-time tasks that share the same deadline.
  - Tasks are scheduled on k identical processors

- Power Model:
  - Same proposed for uniprocessor real-time systems (similar to the ones used in other studies)

$$P(f) = P_s + \sum_{i=1}^{k}(P_{ind} + Pd, i) \quad (1)$$

where

- P_s: static power used to maintain basic circuits, which can only be removed by powering off the whole system
- P_{ind}: frequency independent active power (assumed constant and the same for all processors).[1]
- P_d: frequency dependent active power (depends on the supply voltage and processing frequency of each processor)

$$P_{d,i} = C_{ef} f_i^m, \forall m \geq 2 \quad (2)$$

[1]Related to memory and processor. Can be removed by putting the system to sleep. It is independent of supply voltage and frequency

- Power Model:
  - Due to the overhead of turning on/off processors, authors assume that the system is always on (P_s is always consumed)
  - The energy efficient frequency f_{ee} to minimize the energy consumption for each processor can be defined as:

$$f_{ee} = \sqrt[m]{\frac{P_{ind}}{C_{ef}(m-1)}} \tag{3}$$

  - This means that for energy efficiency the processing frequency for any task should be within $[f_{ee}, f_{max}]$
  - The time overhead for adjusting frequency is negligible (in practice they can be easily incorporated in tasks execution time)

- Fault Model
  - Transient faults, with inter-arrival rate which follows a Poisson distribution
  - When considering the negative effects of DVFS on transient faults, the average fault rate $\lambda(f)$ at a scaled frequency f ($f < f_{max}$) can be given as:

$$\lambda(f) = \lambda_0 g(f)$$

where λ_0 is the average fault rate at f_{max} and V_{max} and $g(f)$ is the fault rate, given as

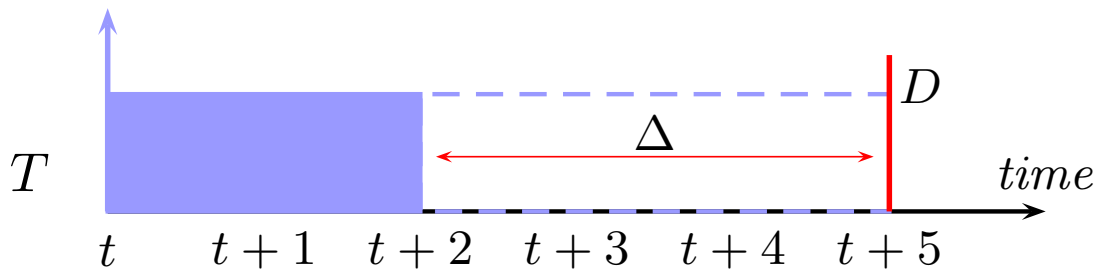
$$g(f) = 10^{\frac{d(1-f)}{1-f_{ee}}}$$

where $d(> 0)$ is a constant which represents the sensitivity of fault rates to DVFS

- Fault Model
  - Faults are detected by the end of tasks execution
  - Recovery is based on *backward recovery* (tasks are re-executed)
  - Overhead of fault detection is assumed to be incorporated into the WCET of tasks

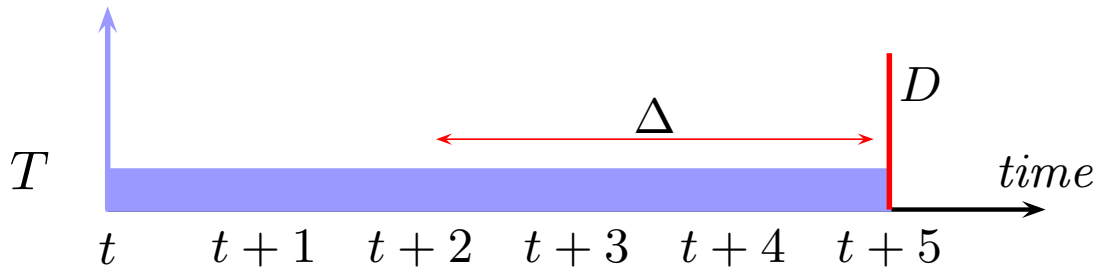
Example

Suppose a task T is dispatched at time t with WCET of 2 time units and deadline of $t + 5$ time units.



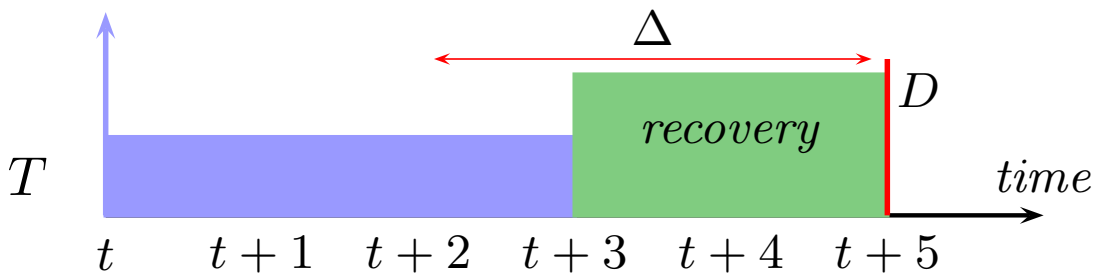
- If task T needs to finish its execution by its deadline at $t + 5$, there will be 3 time units of available slack.

- Ordinary Power Management Approach



- This approach does not pay attention to the negative effects of DVFS on task reliability

- Reliability-Aware Power Management Approach



- The overall reliability of T is the summation of the probability of T being executed correctly and the probability of having transient faults during the execution of T [2]

[2] It has been shown that this is no worse than the original reliability of T when no power management is applied- RTAS 2006

- Authors consider:
  - a set of n independent real-time tasks
  - tasks are executed on a multiprocessor system
  - k identical processors.
  - Tasks share a common deadline D and D =Period= frame period of task set
  - WCET for a task T_i at the maximum frequency f_{max} is denoted as $c_i (1 \leq i \leq n)$.
  - When T_i is executed at a lower frequency f_i, it is assumed that its execution time will scale linearly and T_i will need $t = \frac{c_i}{f_i}$ time units to complete its execution in the worst case.

- Authors consider:
  - A recovery task will be scheduled for each task whose execution will be scaled down.
  - Any faulty scaled task will be recovered sequentially on the same processor
  - A given task cannot run in parallel on multiple processors
  - Since the system is assumed to be on all the time, authors focus on managing the energy consumption related to system active power (P_{ind} and P_d).
  - At the scaled frequency f_i, the active energy consumption to execute task T_i is given as:

$$E_i(f_i) = (P_{ind} + C_{ef} f_i^m) \frac{c_i}{f_i} \tag{4}$$

- Formally, the problem addressed in this paper is:
  find the priority assignment (i.e., execution order of tasks), task selection (i.e., h_i) and the scaled frequencies of tasks (i.e., f_i) to ensure the schedulability of the tasks and at the same time to

$$minimize \sum_{i=1}^{n} E_i(f_i) \tag{5}$$

subject to

$$f_{ee} \leq f_i < f_{max};\ \text{if } h_i = 1 \tag{6}$$

$$f_i = f_{max};\ \text{if } h_i = 0 \tag{7}$$

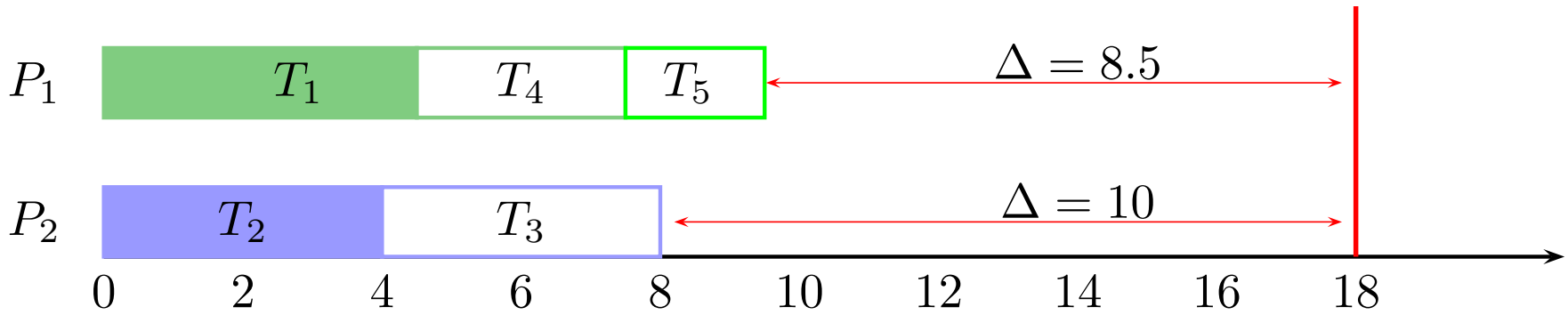
where $h_i = 1$ is used to indicate that T_i was selected for management

- Key issues in solving the static G-RAPM problem
  - priority assignment
  - slack determination
  - task selection

- Optimal priority assignment to minimize the scheduling length on multiprocessor systems under global scheduling is NP-hard
- Such priority assignment, even if it is found, may not lead to the maximum energy savings due to the runtime behaviors of tasks
- In order to get the static mapping of tasks to processors and determine the amount of available slack, authors adopt the longest-task-first (LTF) and worst-fit heuristics.
- After, existing RAPM solutions for uniprocessor systems can be applied for the tasks on each processor individually

Example

Consider a task set with five tasks $T_1(4.5)$, $T_2(4)$, $T_3(4)$, $T_4(3)$ *and* $T_5(2)$, *which will be executed on a 2-processor system with the common deadline of* 18. *Tasks are scheduled according to LTF priority assignment and worst-fit assignment to processors.*



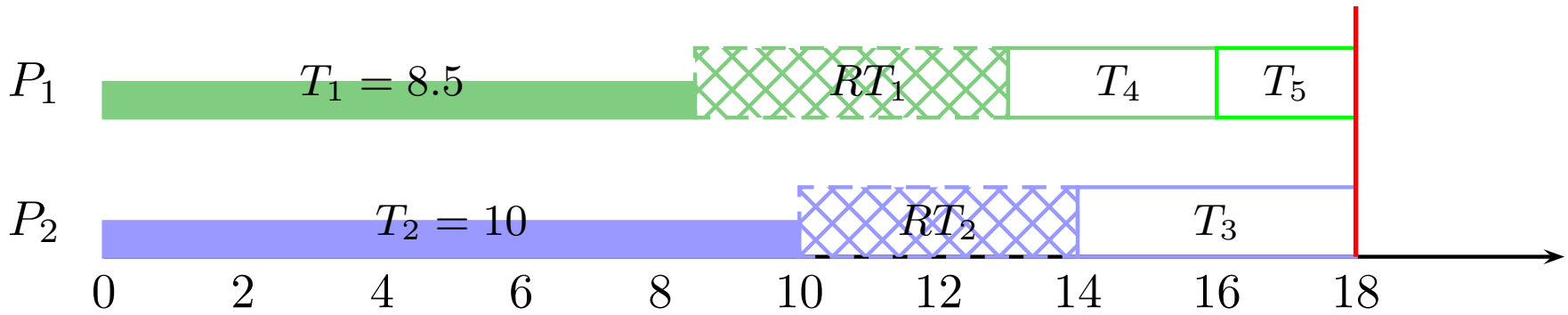
- In previous work, authors have shown that for a single processor system with slack S, to maximize energy savings under RAPM, the optimal aggregate workload for the selected tasks should be:

$$X_{opt} = S\left(\frac{P_{ind} + C_{ef}}{mC_{ef}}\right)^{\frac{1}{m-1}} \tag{8}$$

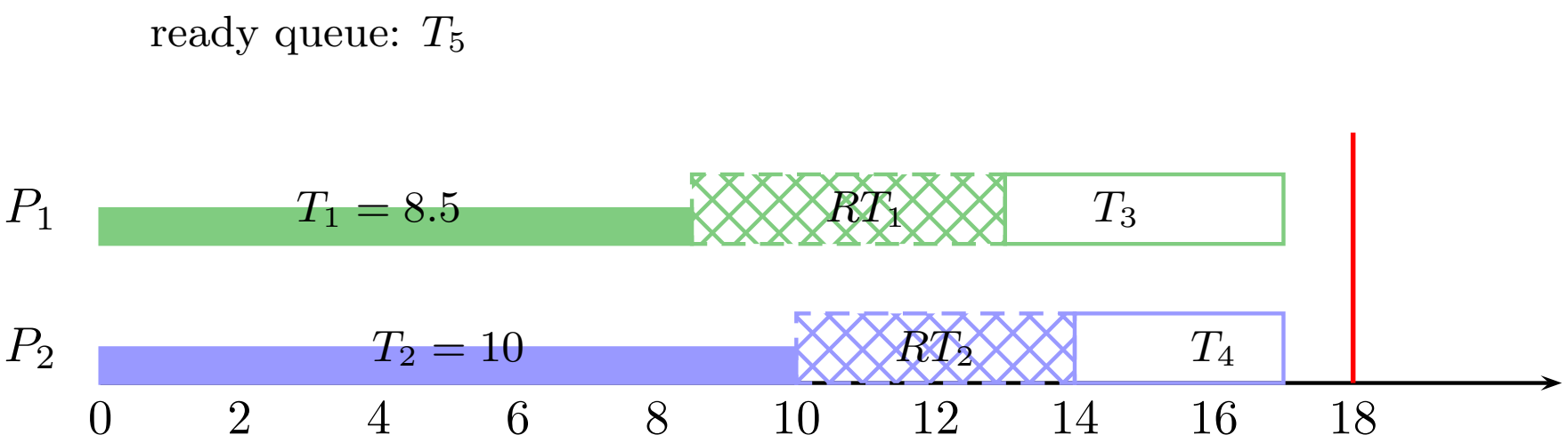
- For previous example $X_{opt,1} = 5.147$ and $X_{opt2} = 6.055$ which results in the following schedule

- Final Schedule for Global-RAPM (G-RAPM) with local task selection

ready queue: T_1, T_2, T_4, T_3, T_5

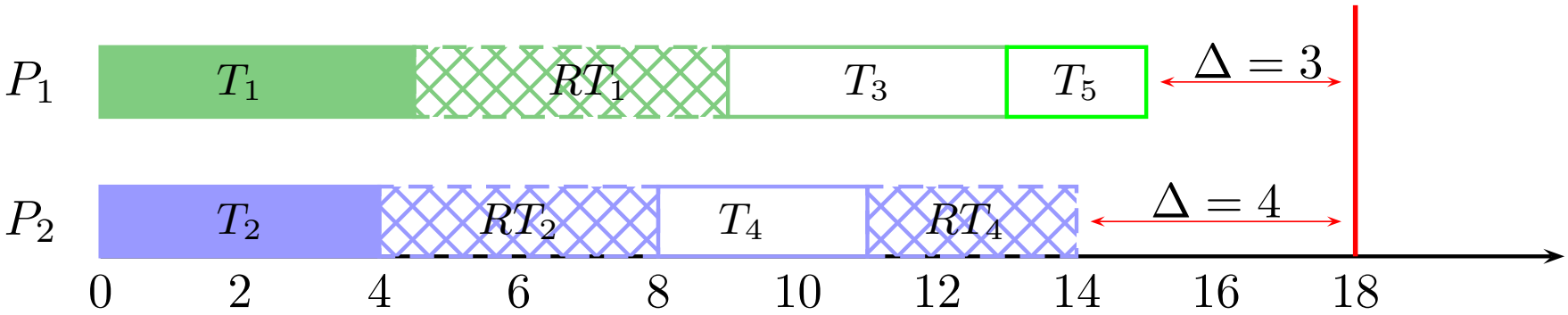


- Task T_5 misses the deadline following the original execution order

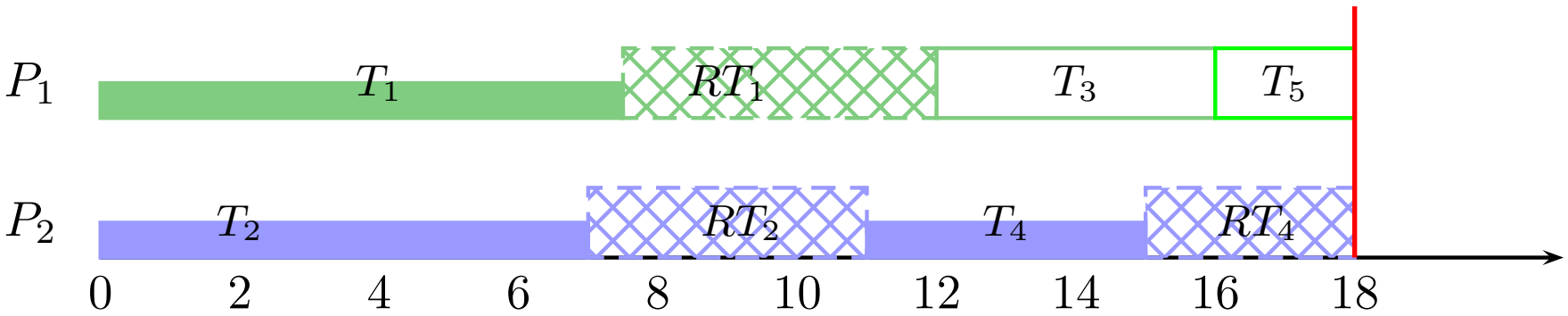


- Motivation for Global Task Selection Approach
  - Local Task Selection: after obtaining S and X_{opt} for each processor, it is not always possible to find a subset of tasks that have the exact optimal workload.
  - This can lead to a situation with less energy savings.
  - A solution could be to consider a **global approach** when (a) determining the amount of available slack and (b) selecting tasks for management.

- The overall workload of all tasks in the previous example is $W = 17.5$.
- Since $D = 18$ and $k = 2$, the total available computation time will be 2x18 = 36.
- The total amount of available slack will be $S = 36 - 17.5 = 18.5$.
- It is possible to calculate that the overall optimal workload of the selected tasks to minimize energy consumption globally. In this case, $X_{opt} = 11.2$.
- Following the same heuristic as the longest task first, three tasks (T_1, T_2 and T_3, which have the aggregated workload of 11.5) are selected to achieve the maximum energy savings.


P_1
T_1
RT_1
T_3
T_5
$\Delta = 3$
P_2
T_2
RT_2
T_4
RT_4
$\Delta = 4$
0
2
4
6
8
10
12
14
16
18

- Final Schedule for Global-RAPM (G-RAPM) with global task selection



- G-RAPM with global task selection can save 32.4% energy when compared to that of no power management;
- G-RAPM with local task selection saves only 26.6% (an improvement of 5.8%).
- By scheduling the managed tasks in the front of the schedule, the scheme with global task selection can provide more opportunities to reclaim the dynamic slack from free of the recovery tasks at runtime.

- In general, real-time tasks only take a small portion of their WCETs and dynamic slack can be expected at runtime
- When the execution of a scaled task completes successfully, the time reserved for its recovery becomes dynamic slack
- Different from using dynamic slack only for scaling down the processing frequency of tasks, "Dynamic Slack Reclamation"have a different behavior for (a) scaled tasks that have scheduled recovery tasks and (b) tasks that do not have recovery tasks yet.

- If the next task to be dispatched is a scaled task, the dynamic slack can be utilized to further scale down the processing frequency for more energy savings.
- If the next task has not been scaled down (does not have any recovery) and the reclaimable dynamic slack is larger than the task's size, a recovery task will be scheduled first and remaining slack is used to scale down the execution of the next task.
- If the slack is not enough to schedule the recovery task, no power management will be applied to the next task, which should run at the maximum frequency to preserve its reliability.