Efficient TDM-based Arbitration for Mixed-Criticality Systems on Multi-Cores

Florian Brandner[1]
with Farouk Hebbache,[2] Mathieu Jan,[2] Laurent Pautet[1]

[1] LTCI, Télécom ParisTech, Université Paris-Saclay
[2] CEA List, L3S

Real-Time Systems

- Traditionally consist of
  - A set of tasks $\tau_1, \ldots, \tau_n$
  - Perform computations within a given time budget (Worst-Case Execution Time aka. WCET aka. C_i)
  - Need to respect deadlines: D_i
  - Often need to execute periodically: T_i

- Observation:

  Tasks rarely use their full time budget

  $\Longrightarrow$ Resources (CPU, memory, ...) are under-utilized

Mixed-Criticality Systems

- Basic idea:
  - Divide tasks into critical and non-critical tasks
  - Introduce two execution modes: LO and HI
  - Introduce two time budgets: $C_i(\texttt{LO})$ and $C_i(\texttt{HI})$
  - Strict timing guarantees only for critical tasks

- Objective:

  Improve resource utilization by executing non-critical tasks as long as critical tasks meet their deadlines

Mixed-Criticality Systems (2)

Monitor time budgets at runtime:

- Execute all tasks assuming their $\texttt{LO}$ budgets
- If some task exceeds its $C_i(\texttt{LO})$:
  - Drop non-critical tasks (or extend their period)
  - Continue critical tasks with their larger budget $C_i(\texttt{HI})$
  - This is called a Timing Failure Event (TFE)
    (ideally this should never happen)
- Nice features:
  - Improved resource utilization before a TFE
- Issues:
  - Only consider CPU-time at the granularity of tasks
  - Complex interactions between critical and non-critical tasks

Motivation

Improve resource utilization at a finer level of granularity while providing guarantees on the tasks' interactions.

Here: accesses to shared memory in multi-core architectures

System Model

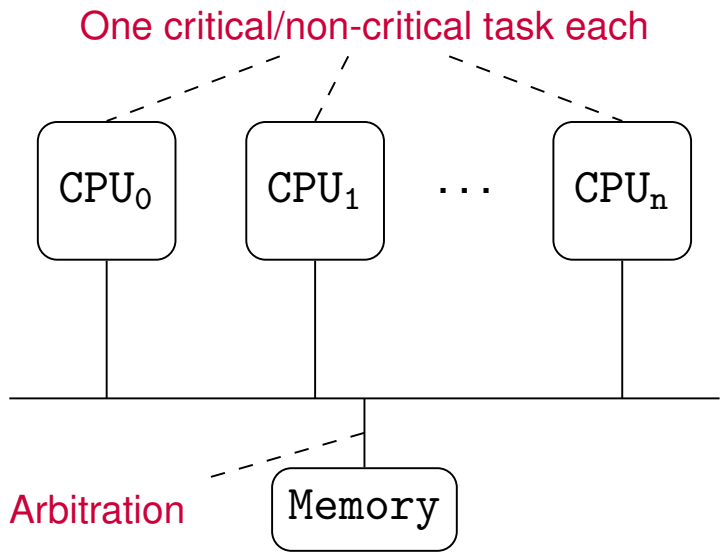
Somewhat simplified model (for now):

- Assume a multi-core architecture with shared memory
- Each core executes a single task (critical or non-critical)

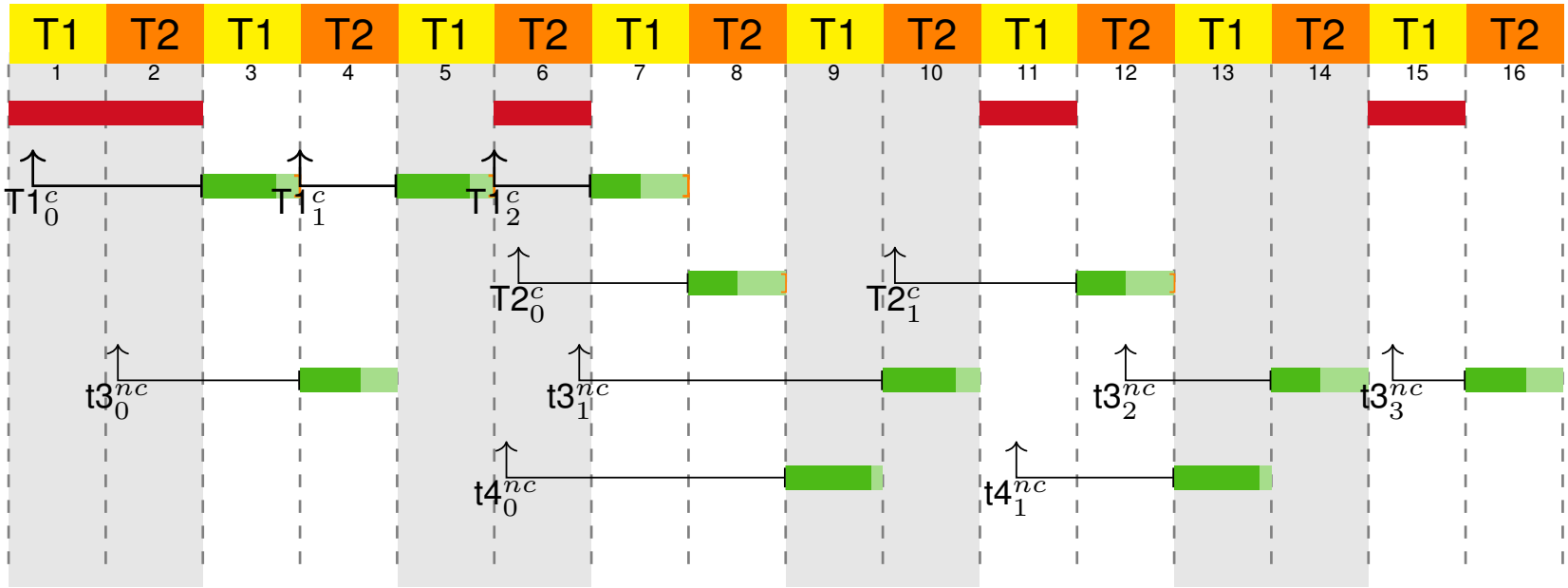
  $\Longrightarrow$ Cores are critical/non-critical

  $\Longrightarrow$ Cores emit critical/non-critical memory requests

- Memory arbitration
  - Time-Division Multiplexing (TDM)
  - Dedicated slot for critical cores
  - No slots for non-critical cores

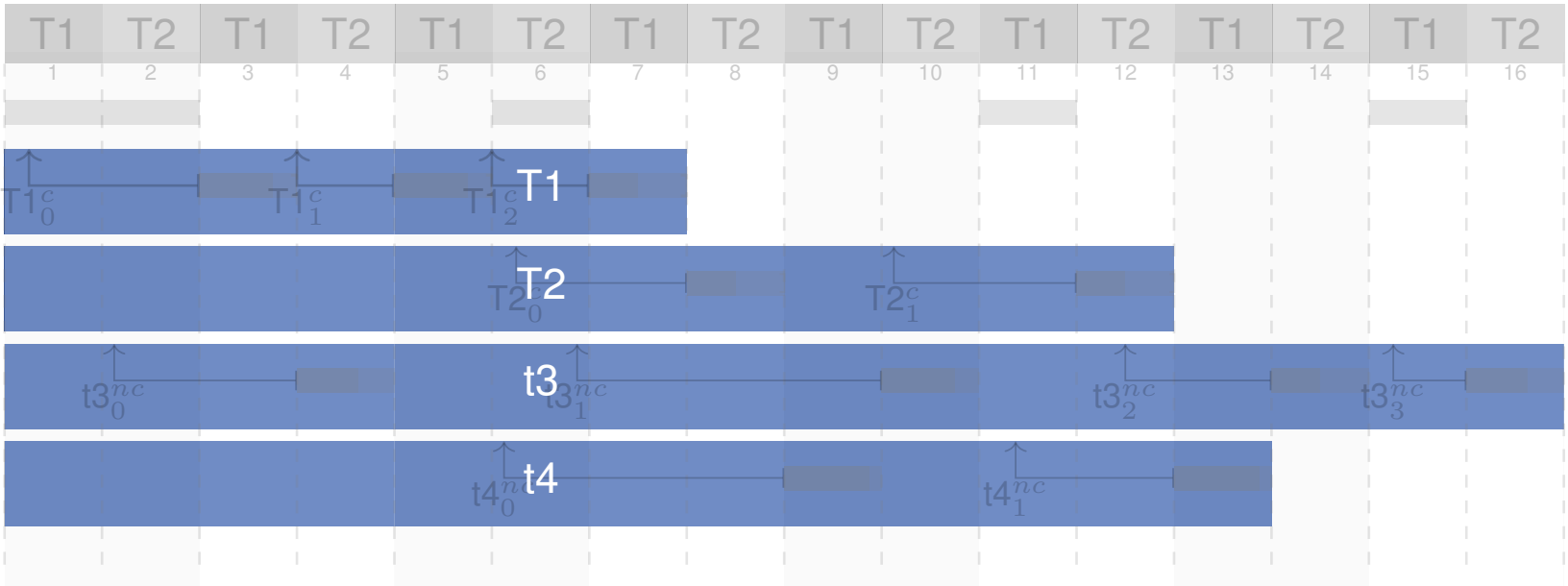


Example: Strict Time-Division Multiplexing



Critical tasks T1 and T2 with TDM slots A and B respectively as well as non-critical tasks t1 and t2.

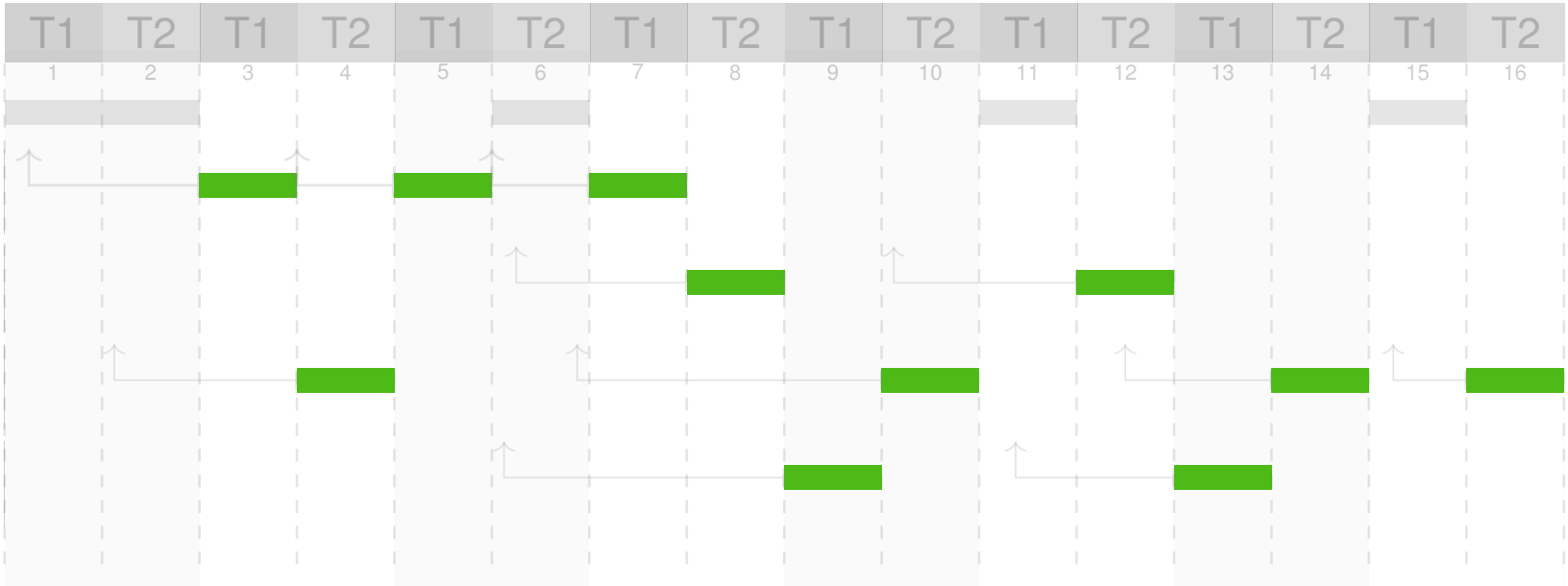
Example: Strict Time-Division Multiplexing



Critical tasks T1 and T2 with TDM slots A and B respectively as well as non-critical tasks t1 and t2.

Rows: different tasks as they execute

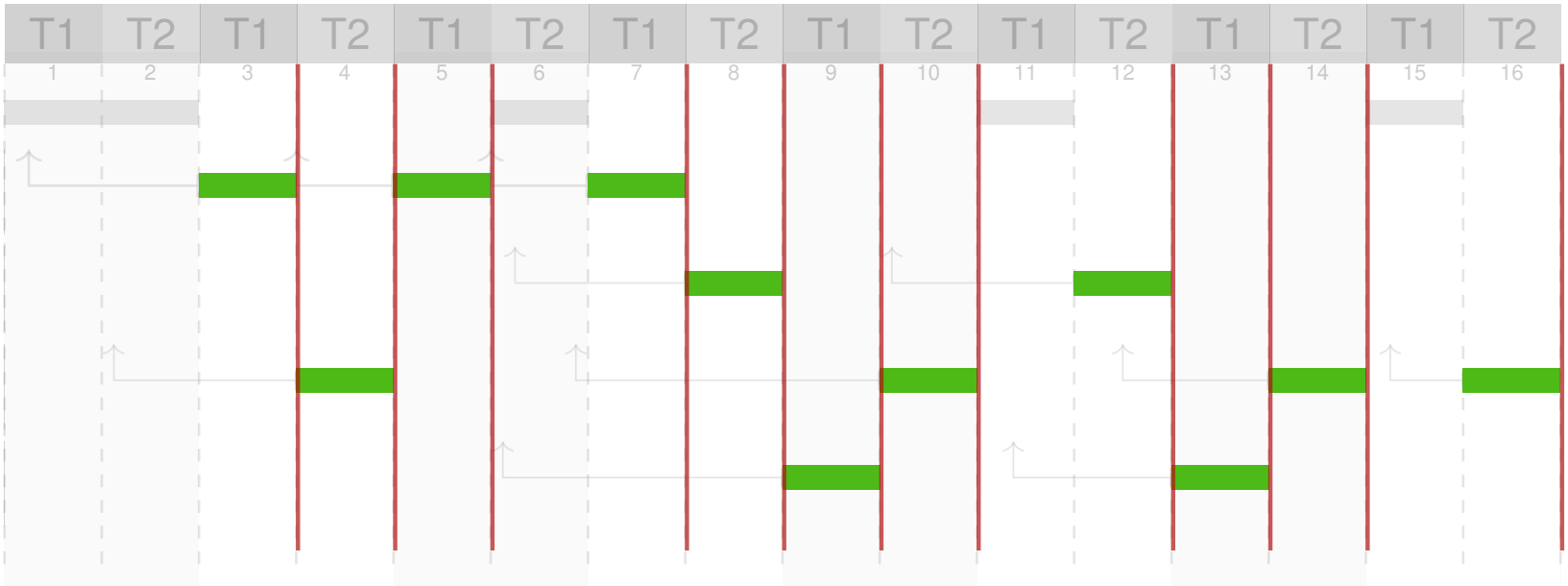
Example: Strict Time-Division Multiplexing



Critical tasks T1 and T2 with TDM slots A and B respectively
as well as non-critical tasks t1 and t2.

Green bars: requests being processed by the memory
(only a single active request at a time)

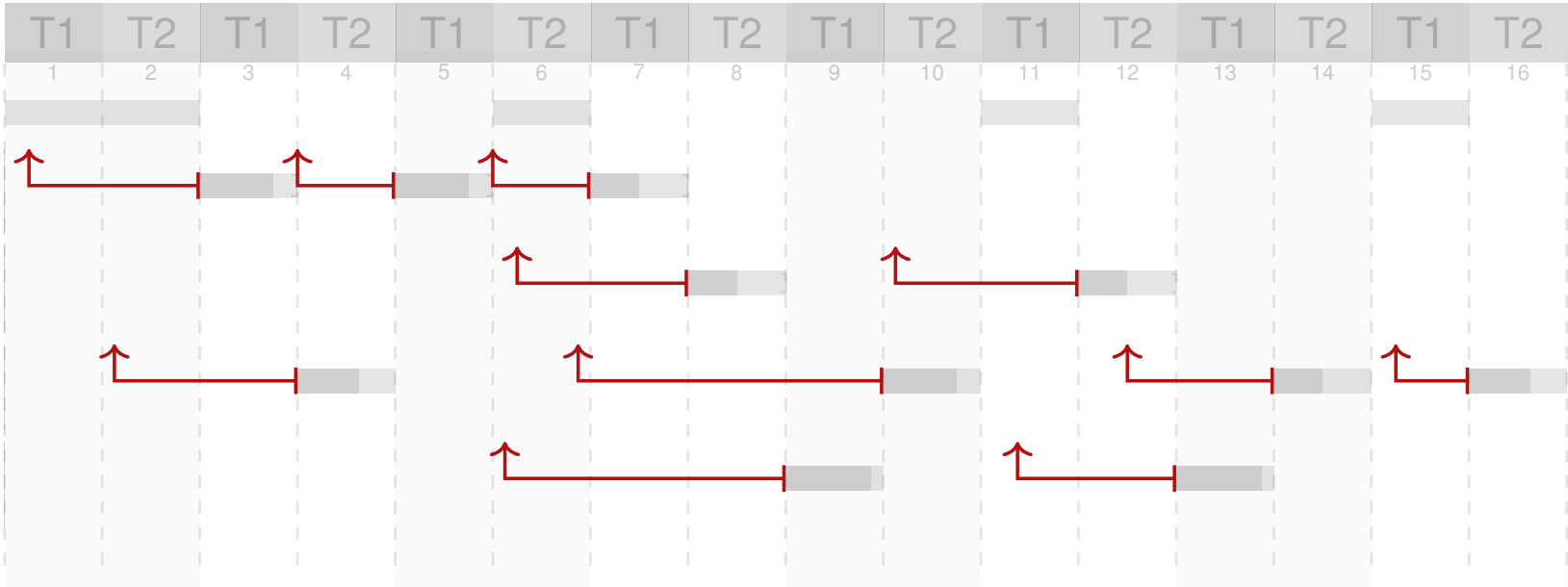
Example: Strict Time-Division Multiplexing



Critical tasks T1 and T2 with TDM slots A and B respectively
as well as non-critical tasks t1 and t2.

Requests always complete at the end of a TDM slot

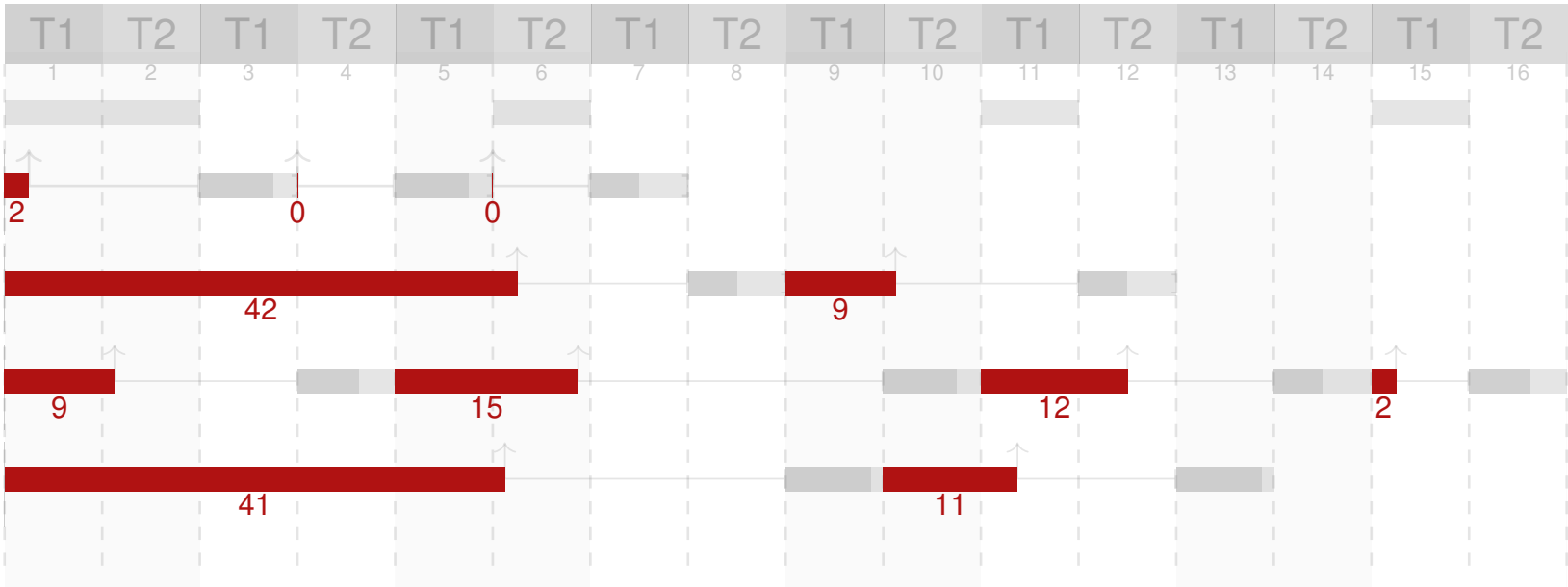
Example: Strict Time-Division Multiplexing



Critical tasks T1 and T2 with TDM slots A and B respectively
as well as non-critical tasks t1 and t2.

Arcs: memory wait time
(from issuing to start of processing)

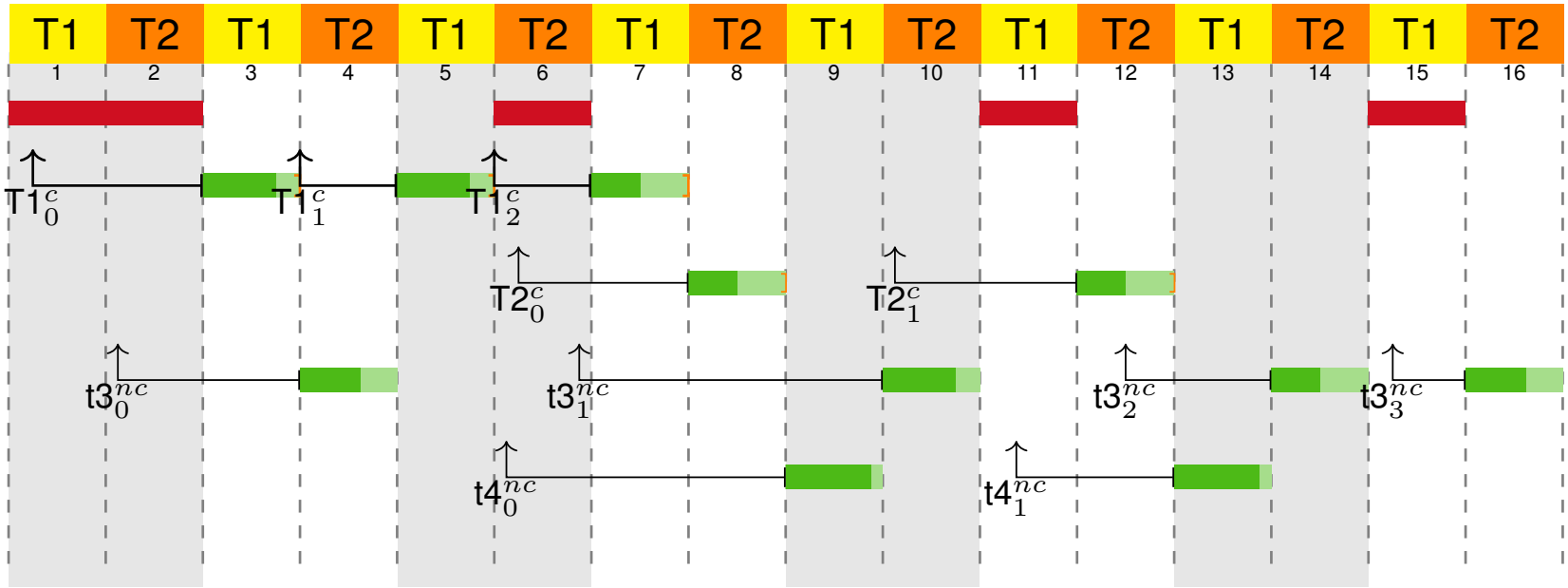
Example: Strict Time-Division Multiplexing



Critical tasks T1 and T2 with TDM slots A and B respectively as well as non-critical tasks t1 and t2.

Gaps: computation time between requests (independent from arbitration)

Example: Strict Time-Division Multiplexing

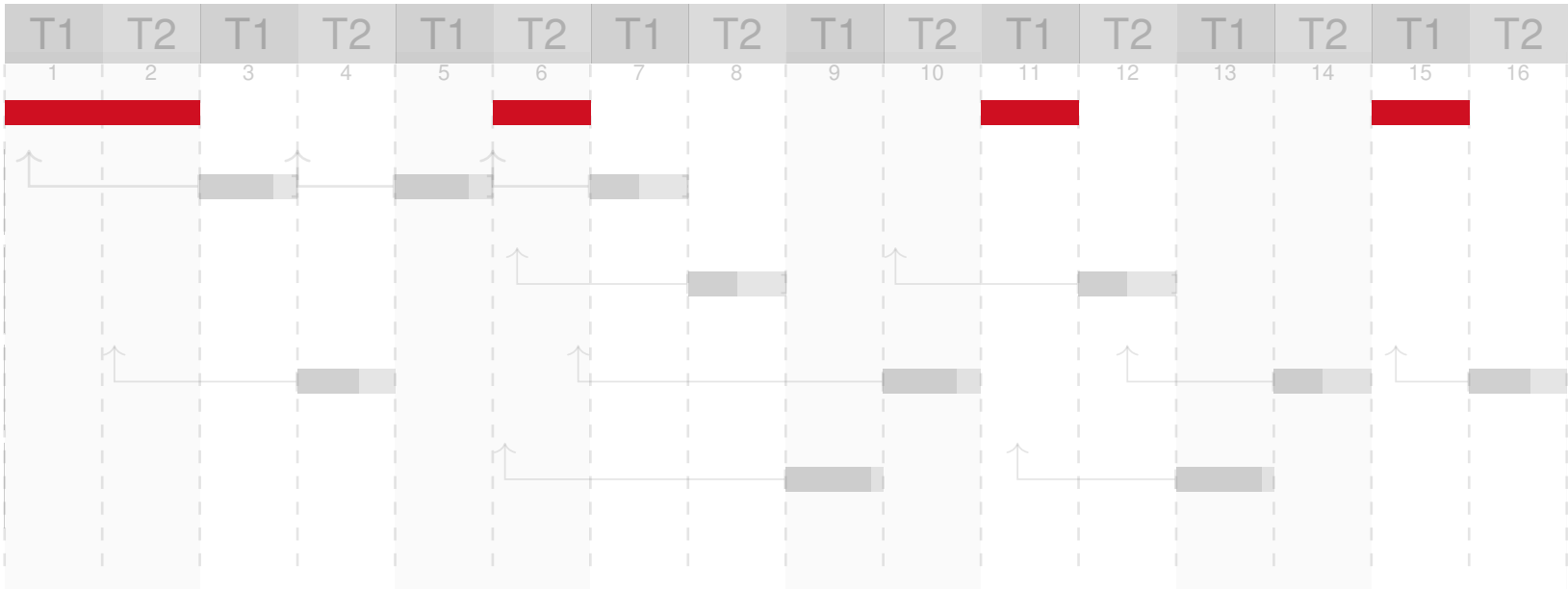


Critical tasks T1 and T2 with TDM slots A and B respectively as well as non-critical tasks t1 and t2.

Non-critical requests can only reclaim unused TDM slots. Still, the approach is non-workconserving.

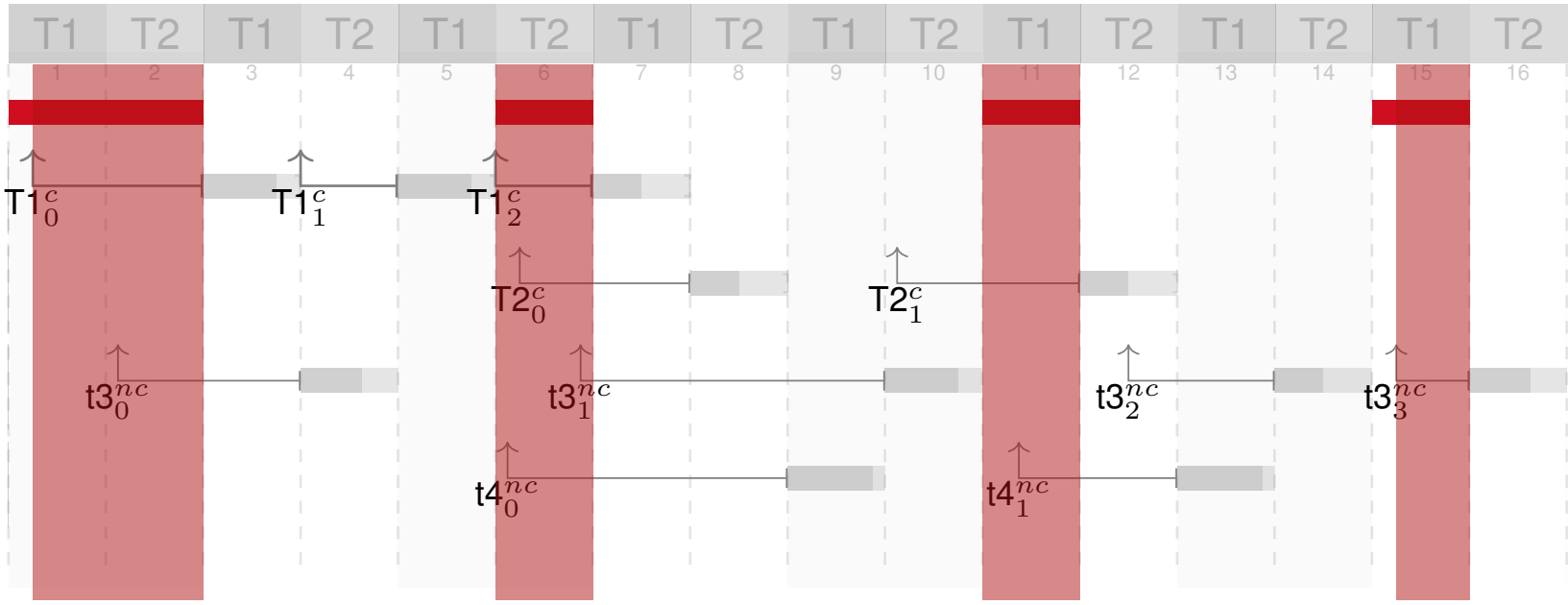
Issues with TDM

Unused TDM Slots



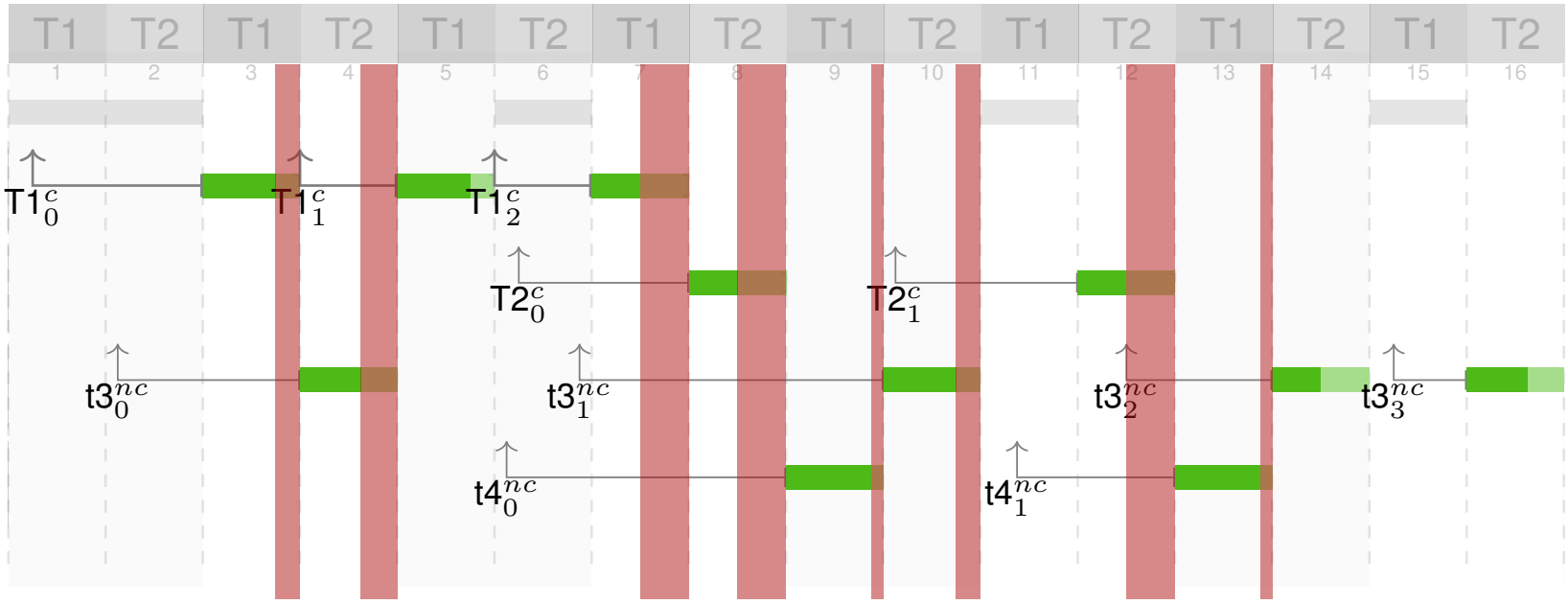
Unused TDM slots — why is the memory idle?

Issue Delay



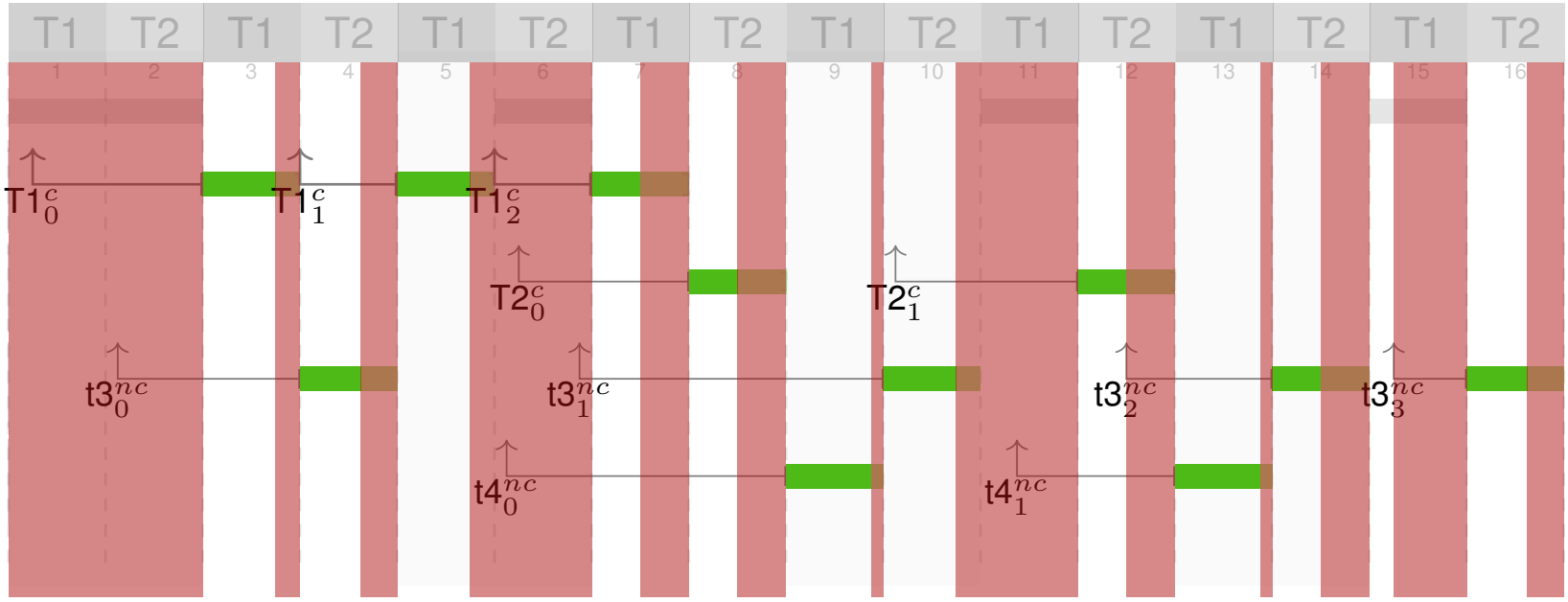
Issue Delay: Number of cycles during which the memory is idle, despite pending requests at the arbiter.

Release Delay



Release Delay: Number of cycles that memory requests finish earlier than the TDM slot length.

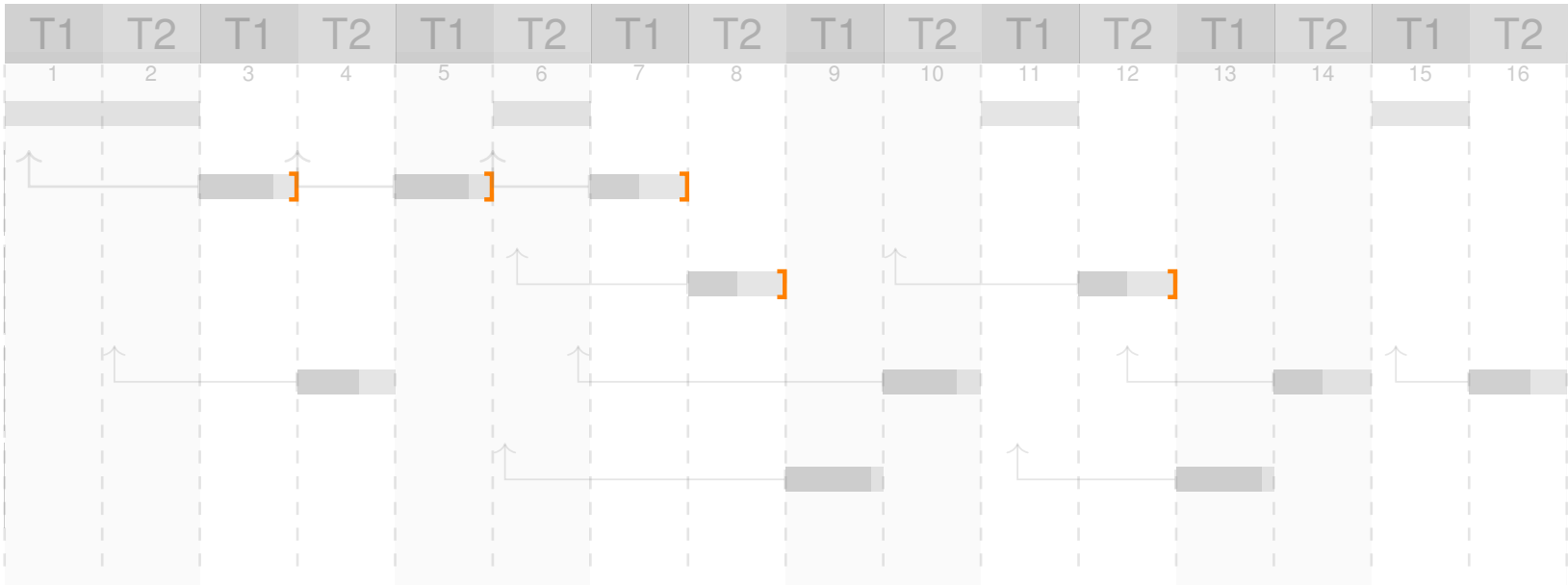
Total Memory Idling



Total Memory Idling: Number of cycles where the memory is not doing anything useful.

Dynamic TDM-based Arbitration

Basic Insight: Deadlines



Interpret the completion date of critical requests under strict TDM as a deadlines.

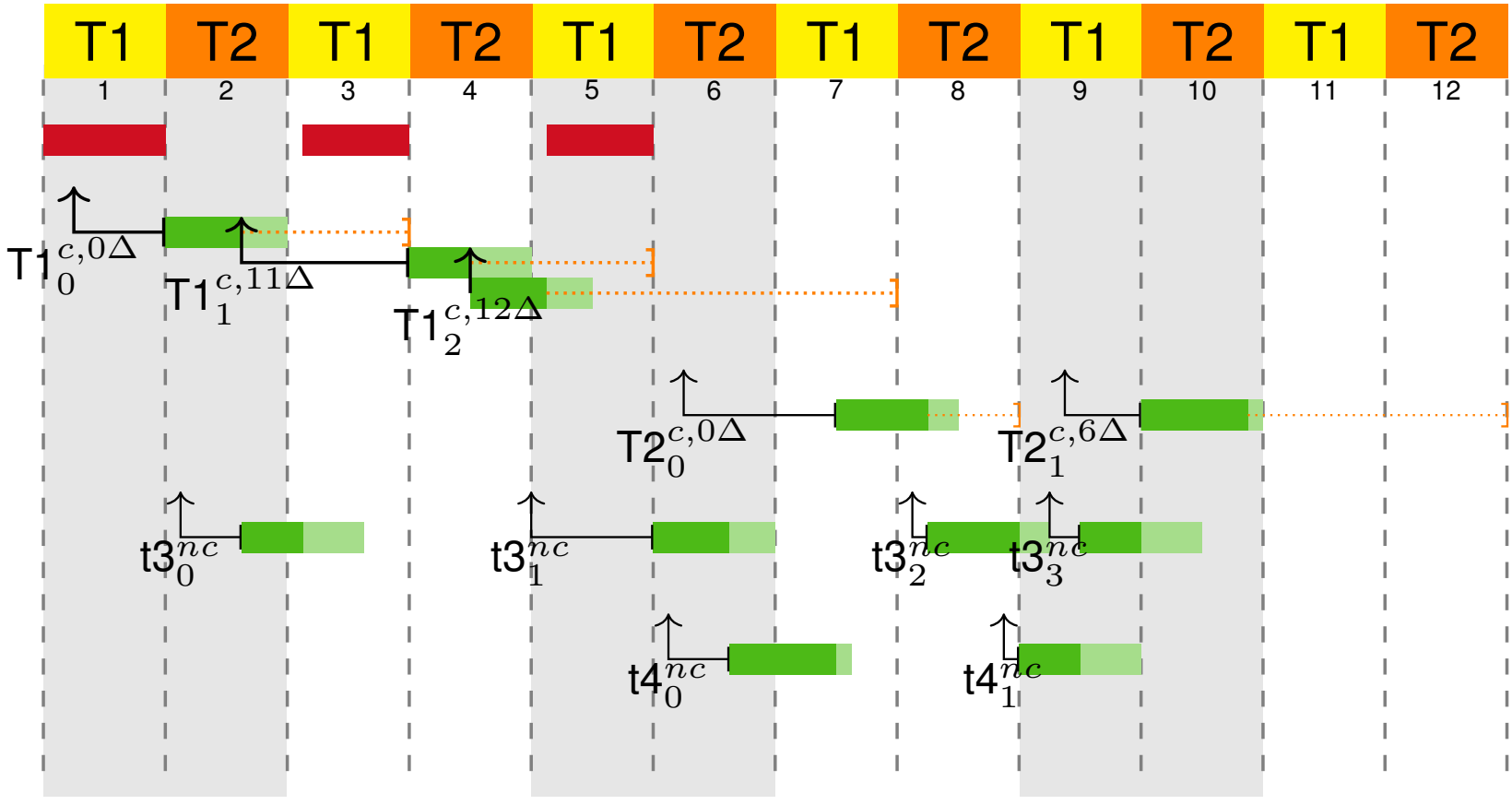
Orange brackets: deadlines of critical requests

Dynamic TDM-based Arbitration

Extend TDM arbiter such that

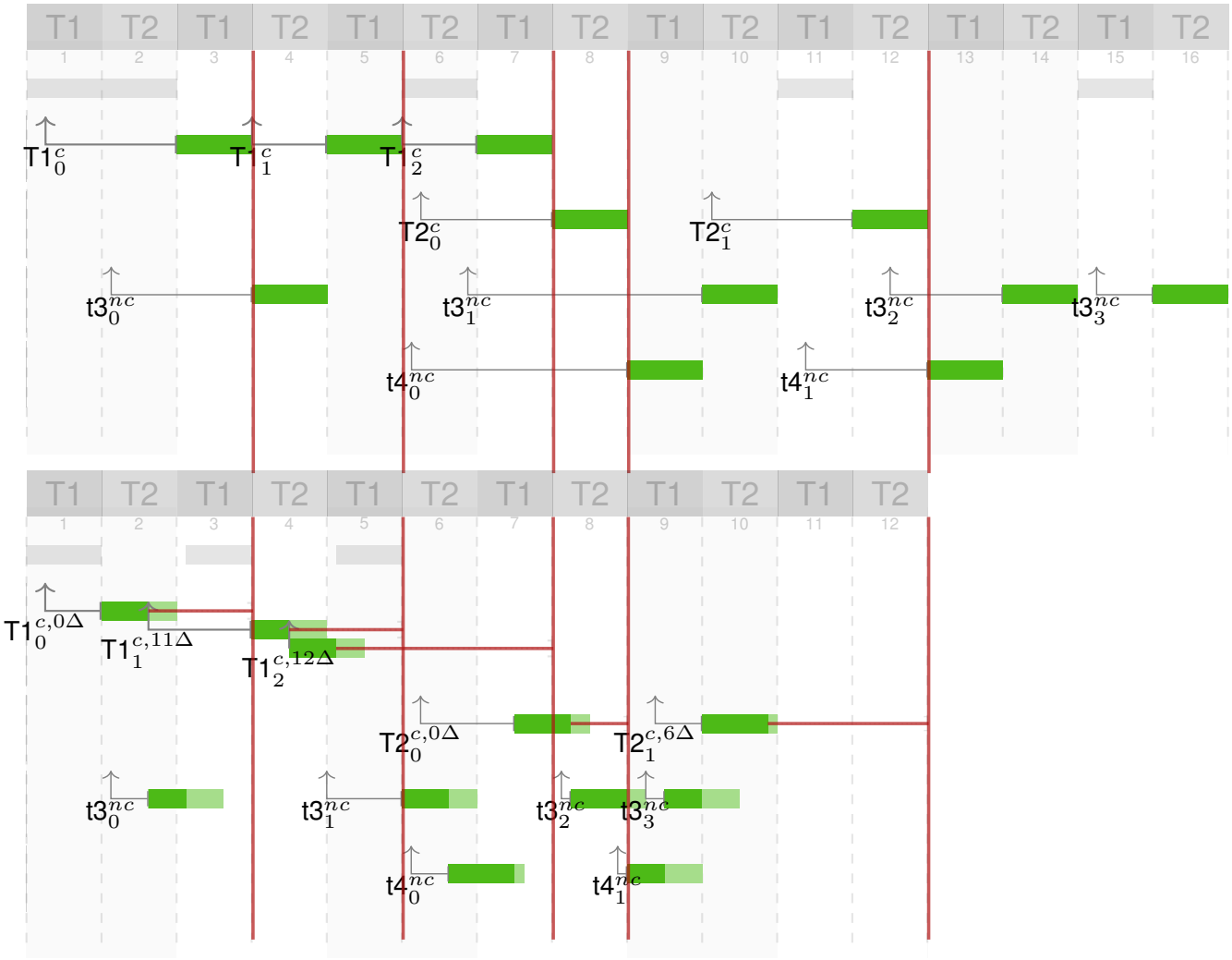
- Each critical request is associated with a deadline
  - Computed when a new request is issued
  - Deadline is at end of a TDM slot of the request owner
  - Scheduled using *Earliest-Deadline-First* strategy (EDF)
- Best-effort for non-critical requests
  - Can profit from the memory's idle time and are ...
  - ... prioritized over critical requests whose deadlines are far
  - Scheduled using *First-In First-Out* strategy (FIFO)
    (other alternatives possible, e.g., fixed priorities)
- Schedule request at any moment
  - Requests handled independently from TDM slots
  - Track slack when critical requests complete before deadline

Example: Dynamic TDM



Same task set using dynamic TDM-based arbitration (`TDMer`).

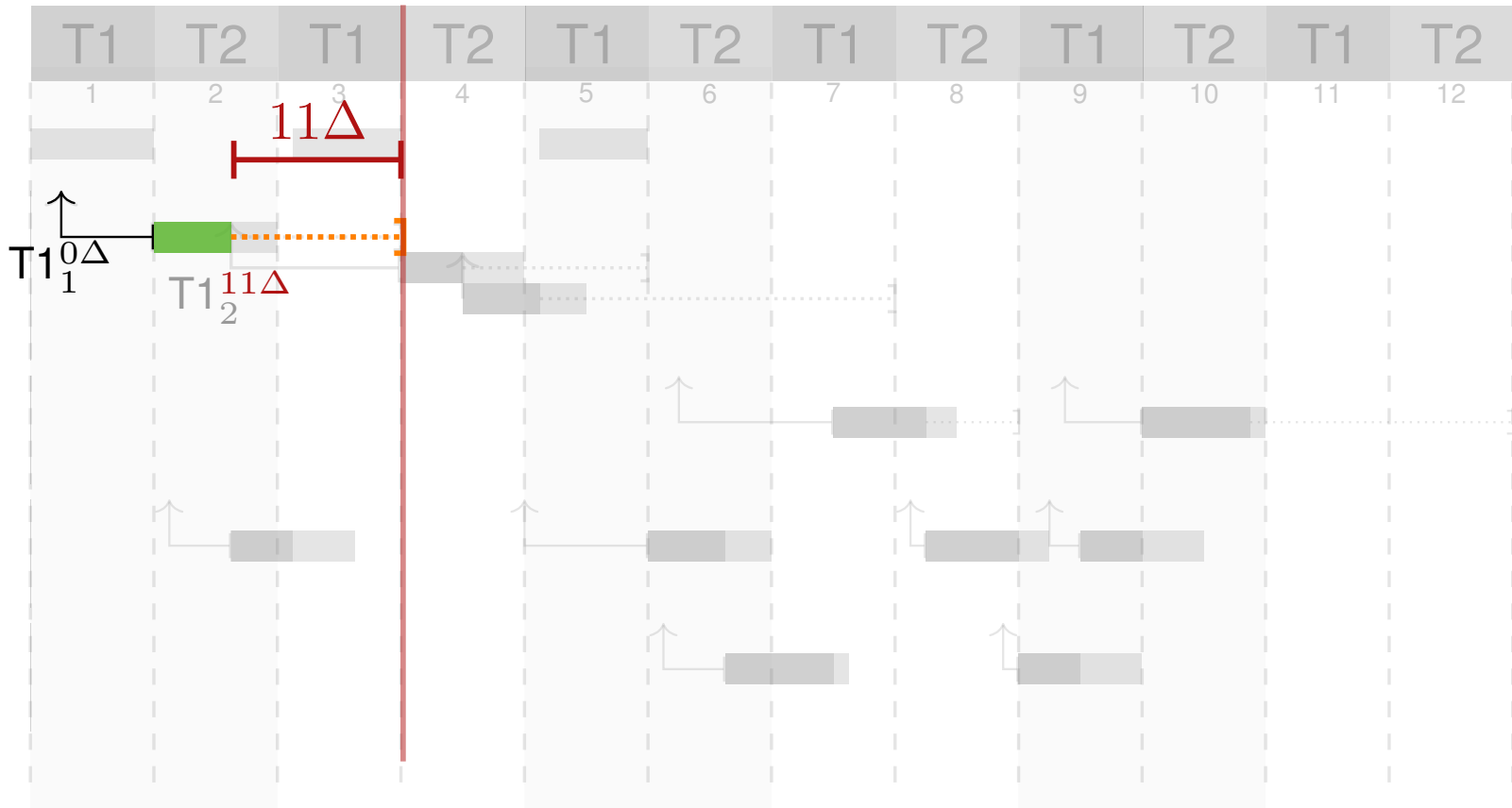
Strict TDM vs. TDMer



Critical requests complete earlier than under strict TDM. Yay!

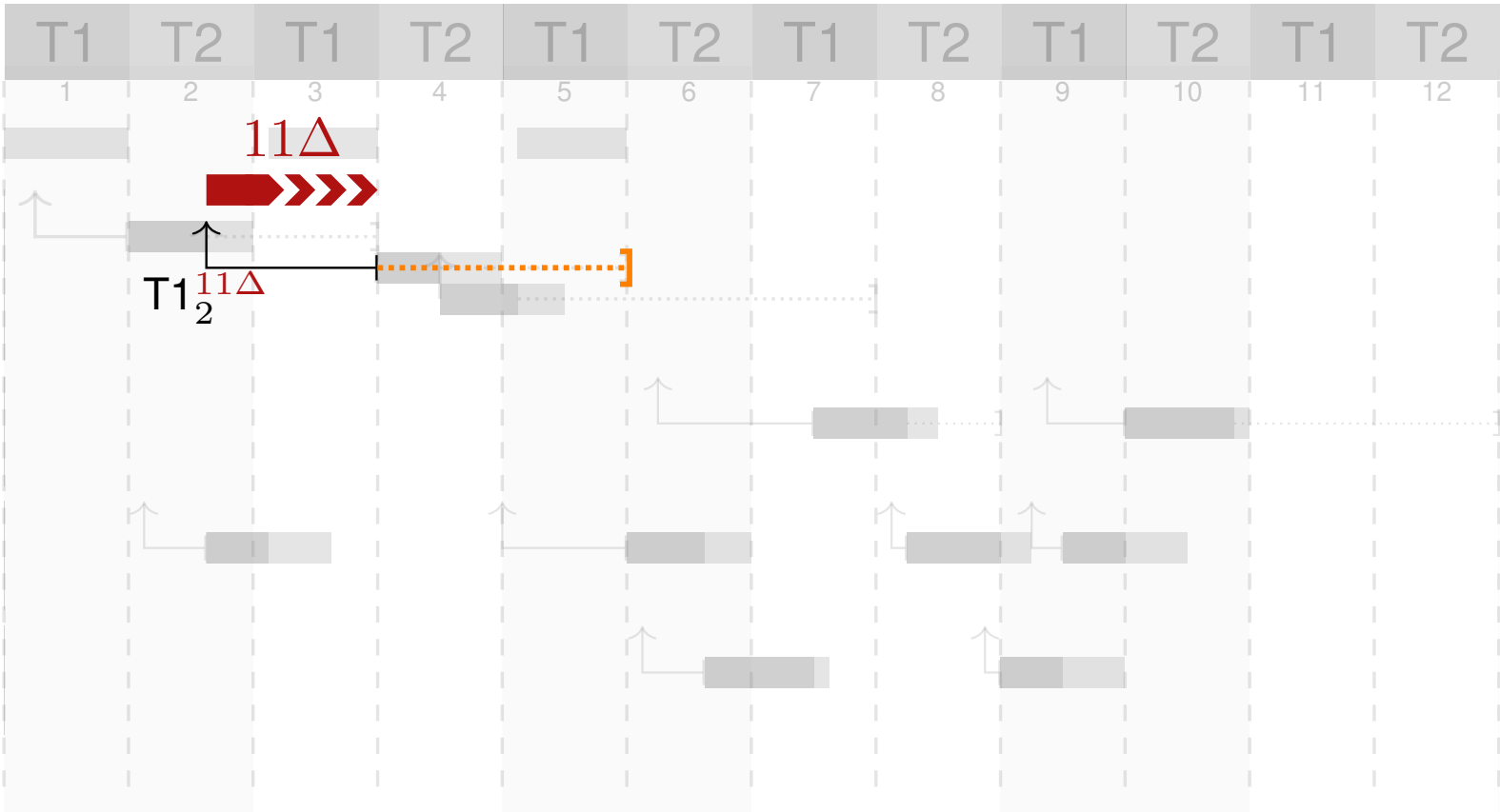
TELECOM ParisTech

Slack Counters



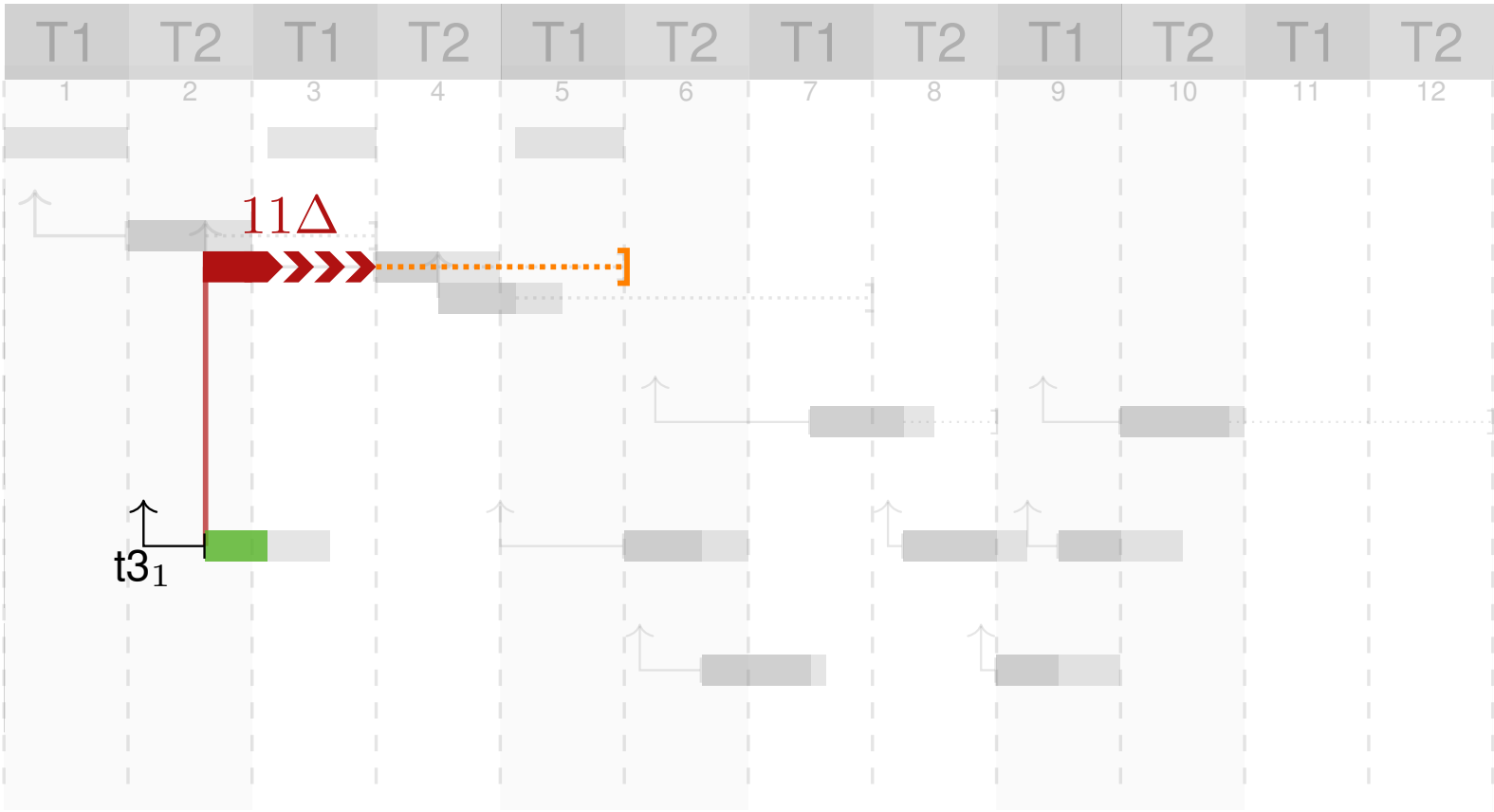
Critical tasks accumulate slack whenever a request finishes before its deadline (i.e., earlier than strict TDM).

Deadlines



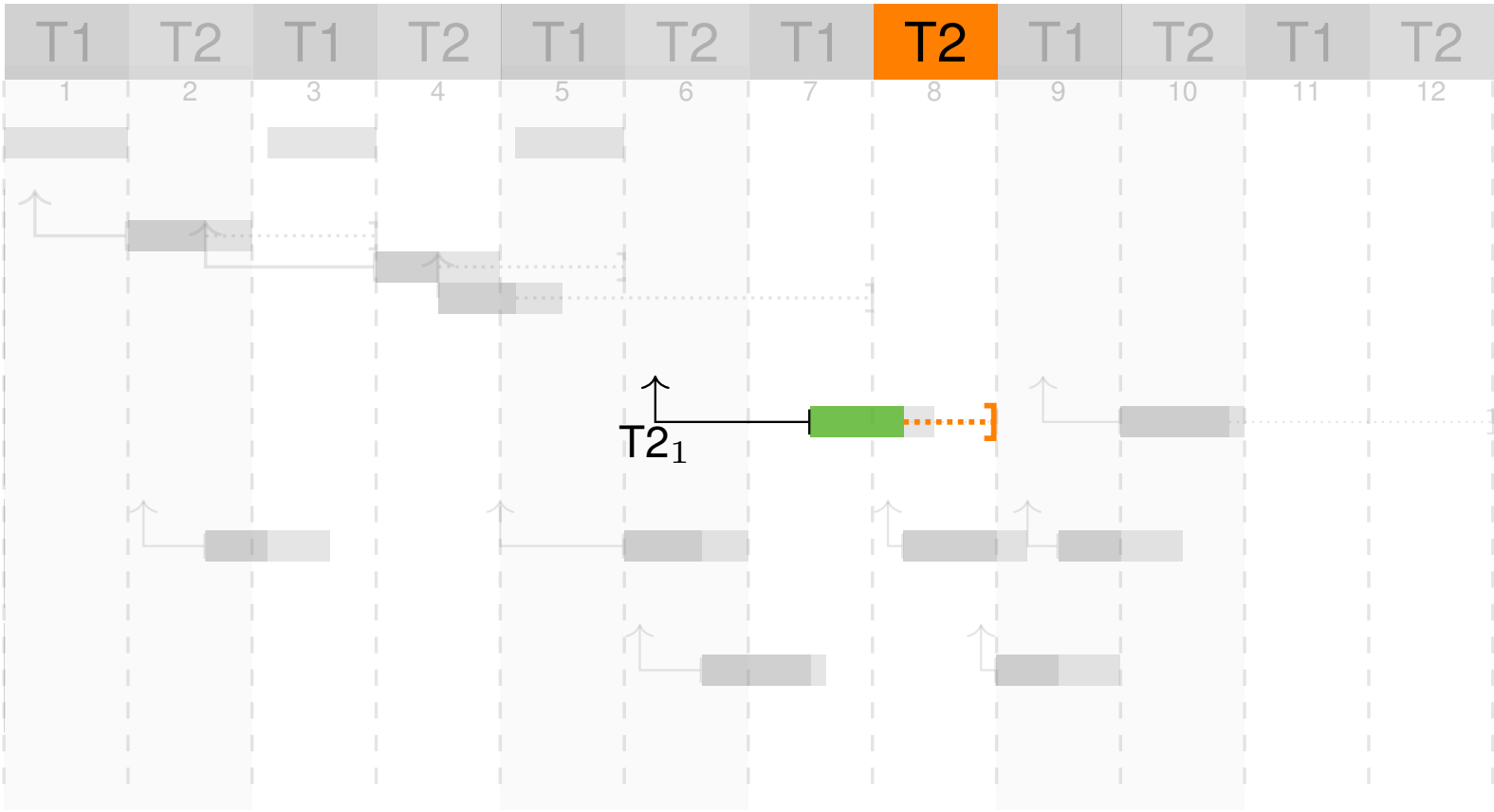
Deadlines are derived by finding the next TDM slot after the **delayed issue date** (*now* + slack).

Independence from TDM Slots (1)



Requests are processed any time, independent from TDM slots, iff critical tasks have enough slack (here T1).

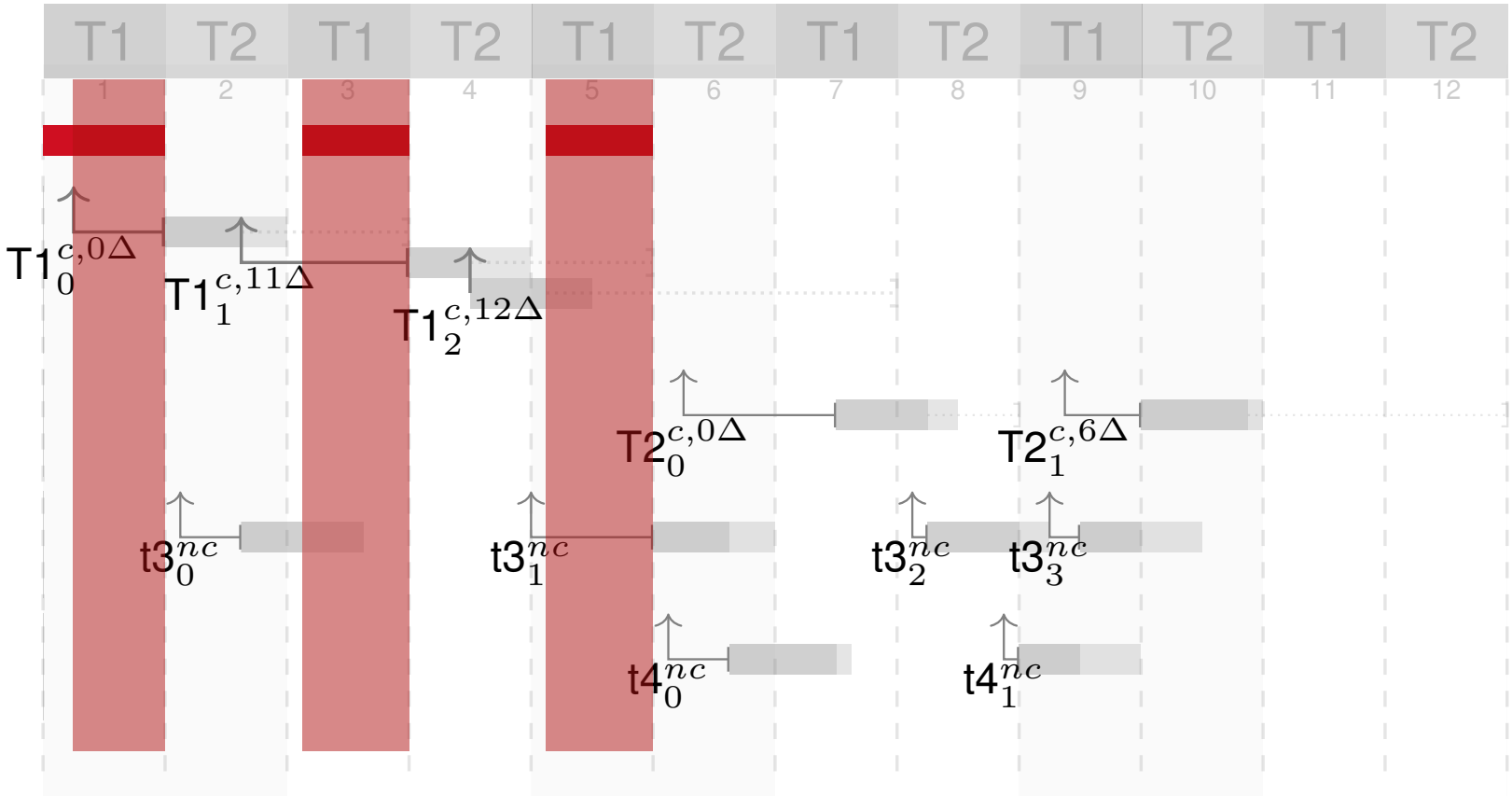
Independence from TDM Slots (2)



Critical requests may also be processed when the upcoming slot belongs to the request owner.

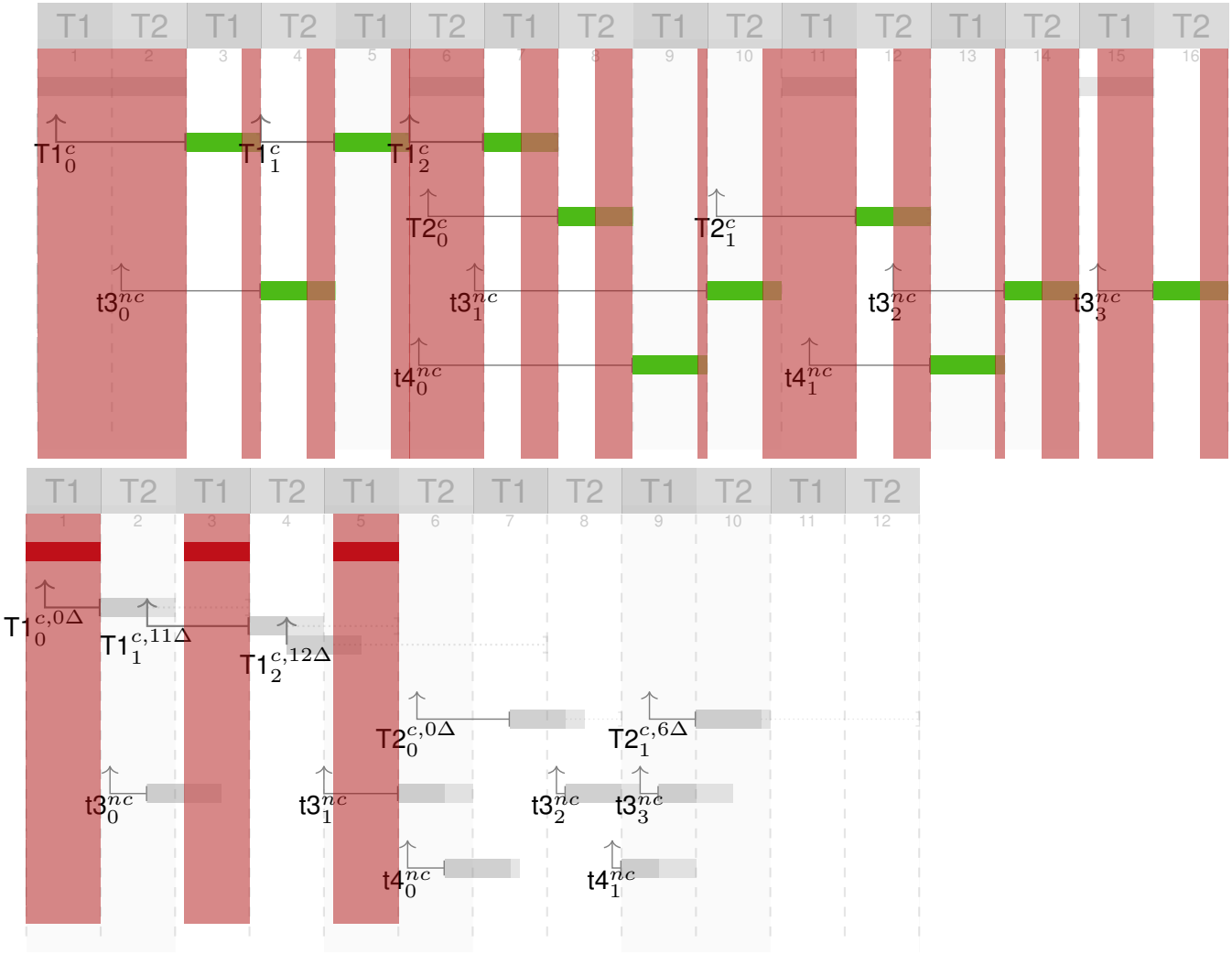
What did we win?

Issue Delay



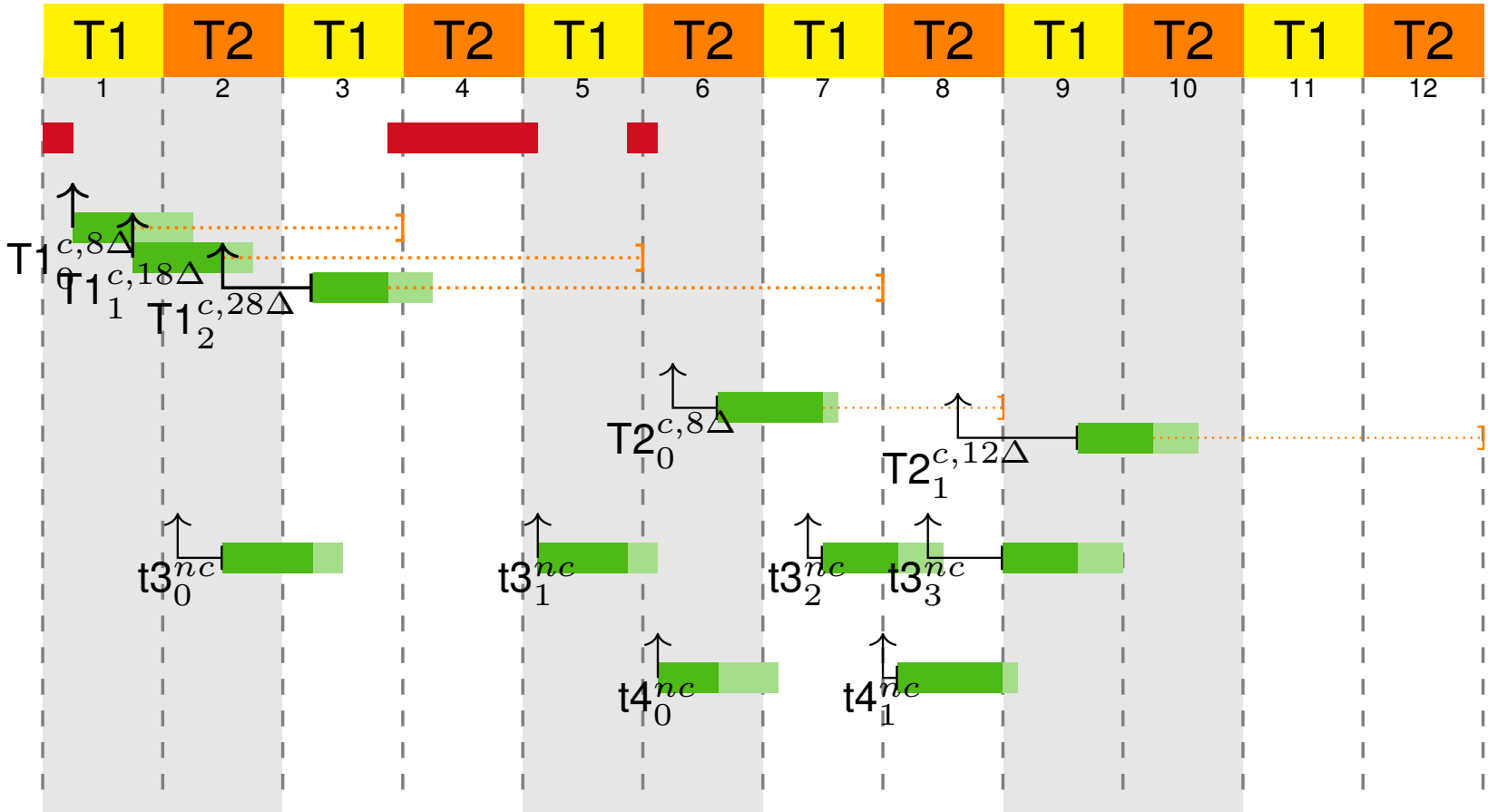
Somme issue delays remain (for now) —
all release delays transformed into issue delays.

Total Memory Idling



The total memory idling is considerably improved — but not (yet) work-conserving . . .

Providing Initial Slack (`TDMeri`)



Providing initial slack of a single TDM slot (8Δ) eliminates (almost) all release delays . . .

Experiments

Hardware Setup

- Multiple Patmos cores (http://patmos.compute.dtu.dk)
  - 256 B stack cache for stack data
  - 32 KB method cache for code (LRU)
  - 32 KB data cache (2-way set-asso., LRU, write-through)
- Shared main memory
  - Random access latencies between $[21, 40]$ cycles
    $\Longrightarrow$ TDM slot length of 40 cycles
  - Arbitration various variants of based on TDM
    (aka. `TDM`, `TDMds`, `TDMes`, `TDMer`, and `TDMeri`)

Task Set Simulation

Randomly generated task sets:

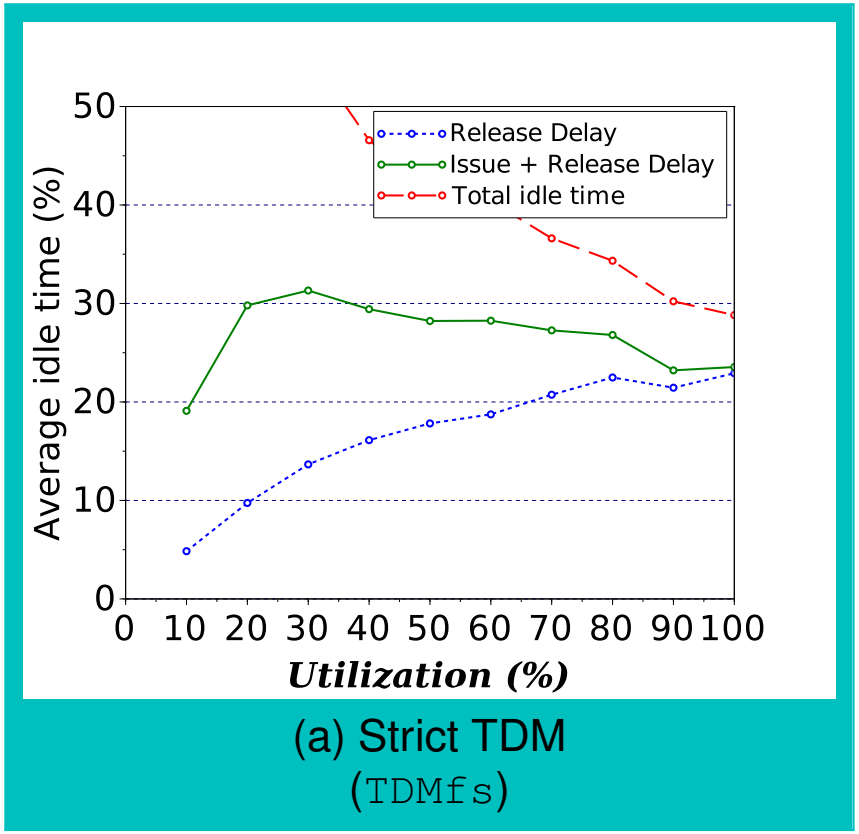
- 25%/75% or 50%/50% critical/non-critical tasks
  - 4, 8, 12, 16, 20, 24 cores/tasks
  - Overall 4320 simulation runs
- Based on randomized memory traces

  (calibrated from actual traces of MiBench on Patmos)

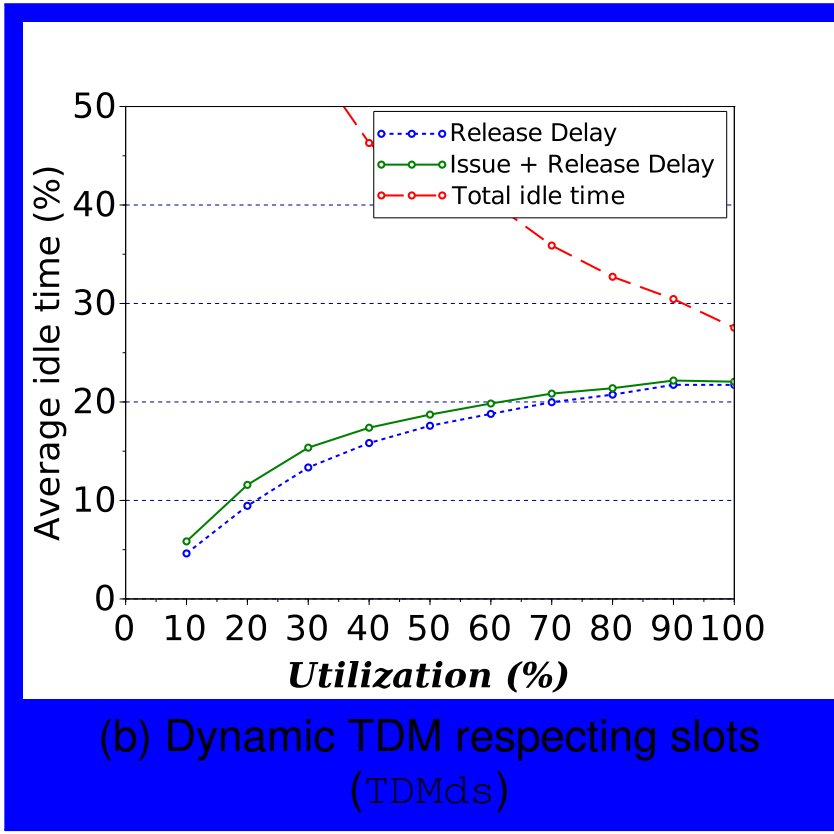
- Objective:
  - Average-case impact on memory utilization

    (issue & release delays, memory idling)
  - How *work-conserving* can we get?

Results: Baseline



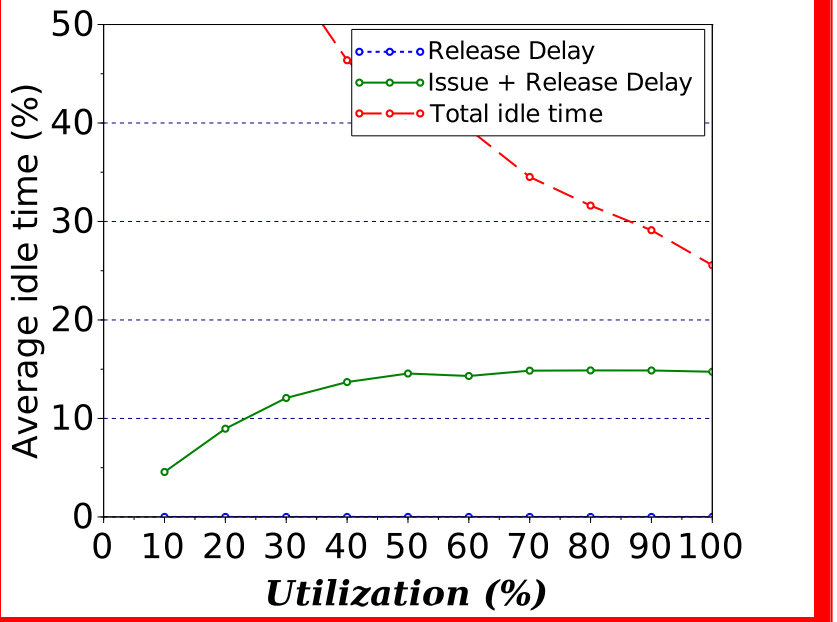
(a) Strict TDM (`TDMfs`)



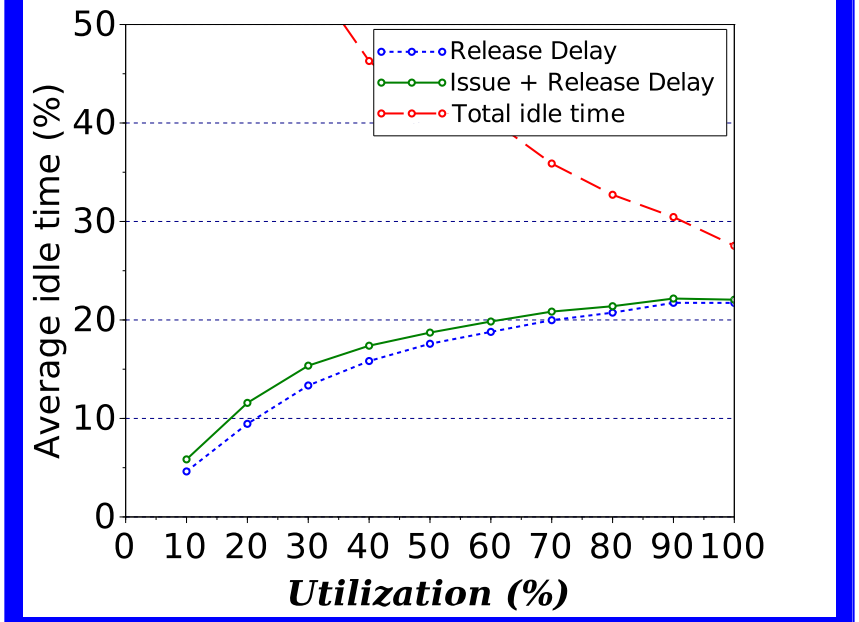
(b) Dynamic TDM respecting slots (`TDMds`)

Dynamic TDM arbitration largely eliminates issue delays.

Results: Dynamic TDM



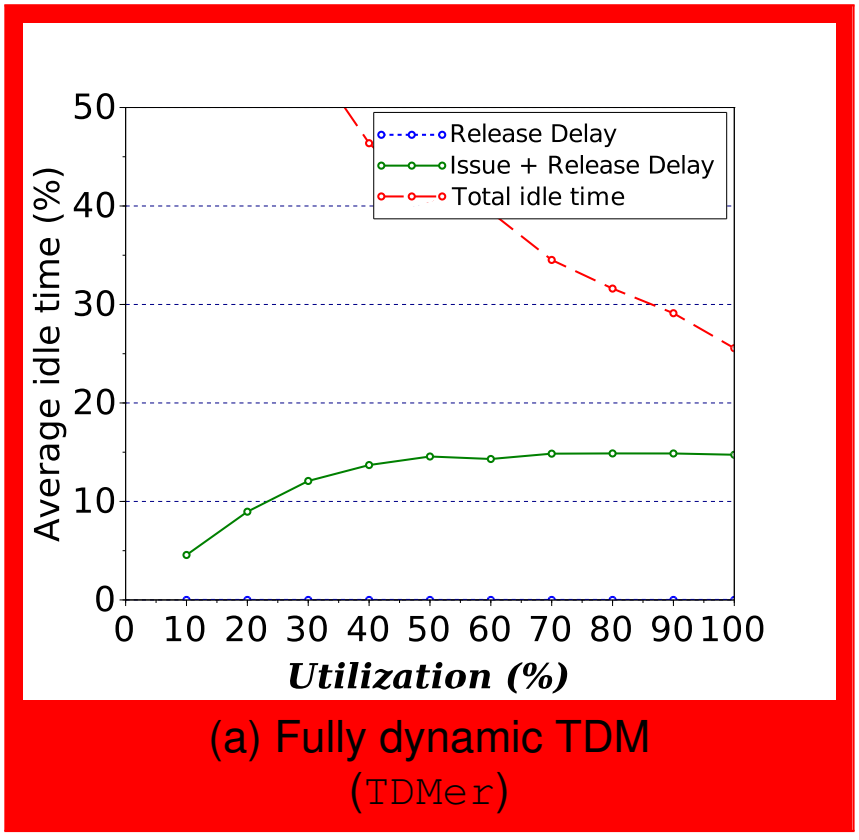
(a) Fully dynamic TDM (`TDMer`)



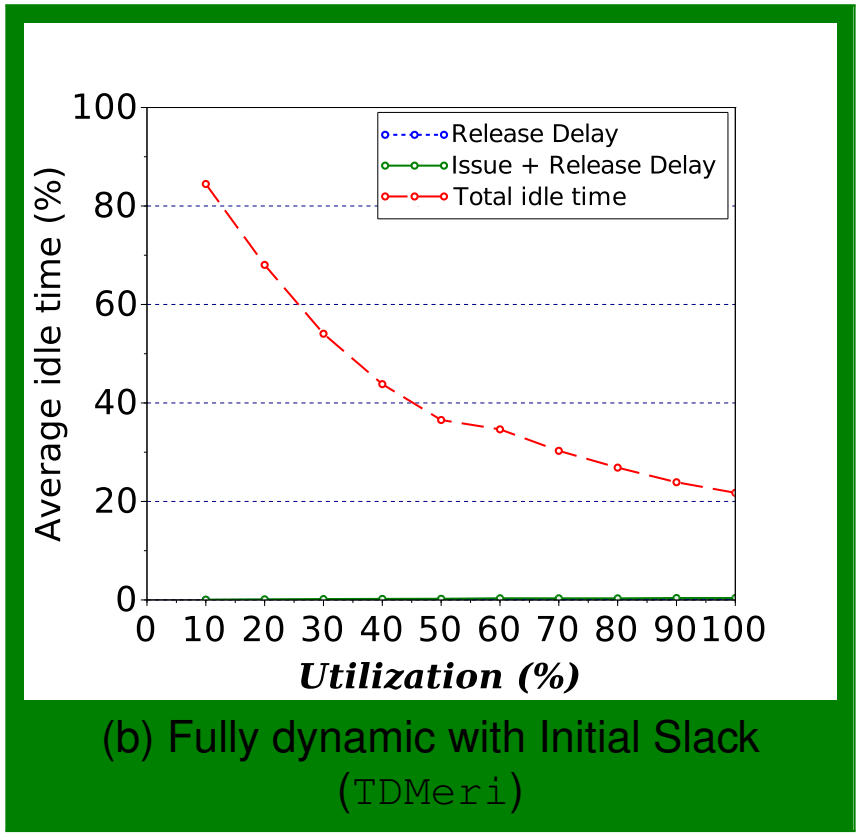
(b) Dynamic TDM respecting slots (`TDMds`)

Fully dynamic TDM arbitration noticeably better under high load
– but little change in total memory idling . . .

Results: Initial Slack



(a) Fully dynamic TDM (`TDMer`)



(b) Fully dynamic with Initial Slack (`TDMeri`)

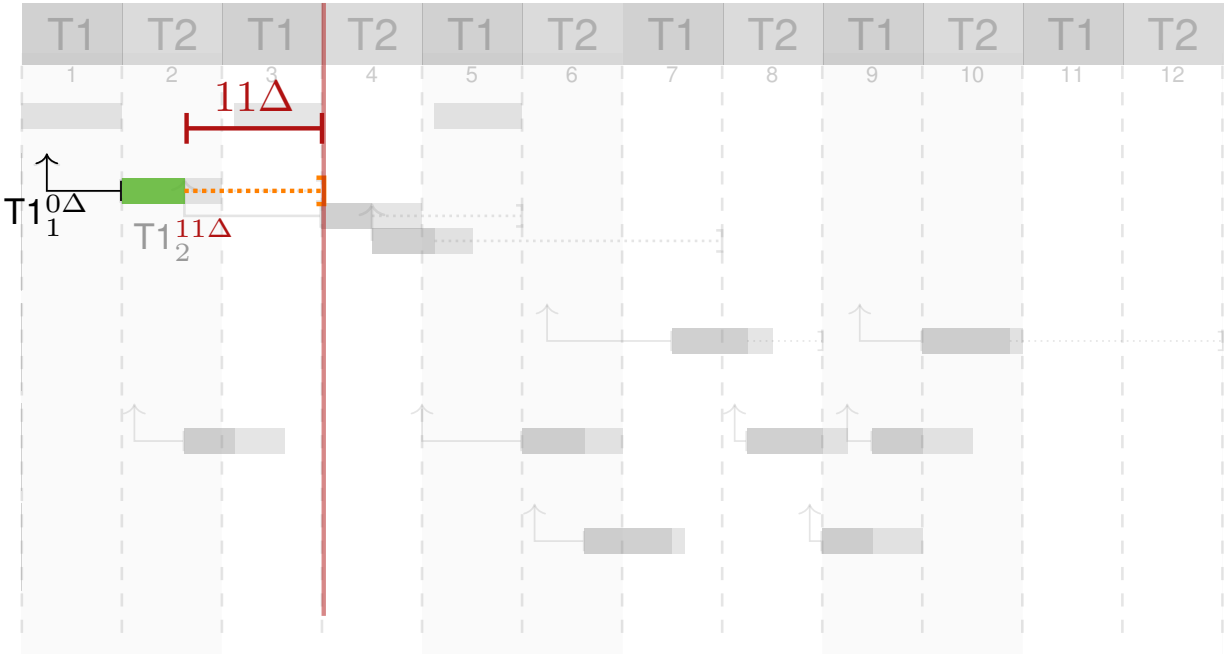
Initial slack (40 cycles) results in work-conserving arbitration — total memory idling improves considerably under high load

Conclusion

- Dynamic TDM-based arbitration
  - Preserves TDM's guarantees
  - Simple analysis of critical tasks
  - Essentially work-conserving – Yay!
- What else?
  - Multiple tasks on a single core $\Longrightarrow$ preemption (WiP)
    - Large slack counter values may delay task preemption
    - How can we control this delay?
  - Hardware implementation
    - Efficiently checking for the smallest deadline?
    - How to scale to large multi-/many-cores?

Accumulating Slack:

Subtract actual completion time from deadline.



Slot Independence:

Check slack (deadline) of owner of upcoming TDM slot.