# Combining Time-Triggered Plans with Priority-Scheduled Task Sets

Jorge Real, Sergio Sáez, Alfons Crespo

Universitat Politècnica de València, Spain

21st International Conference on Reliable Software Technologies - Ada-Europe 2016 - Pisa, Italy, June 13-17, 2016

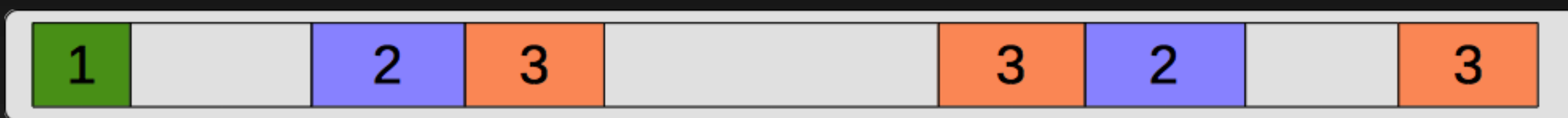
# Outline

- Introduction
- System Model
- API for Time-Triggered Plans
- API Extensions to Support Additional Task Patterns
- Comments About the Implementation
- Experimental Results
- Conclusions

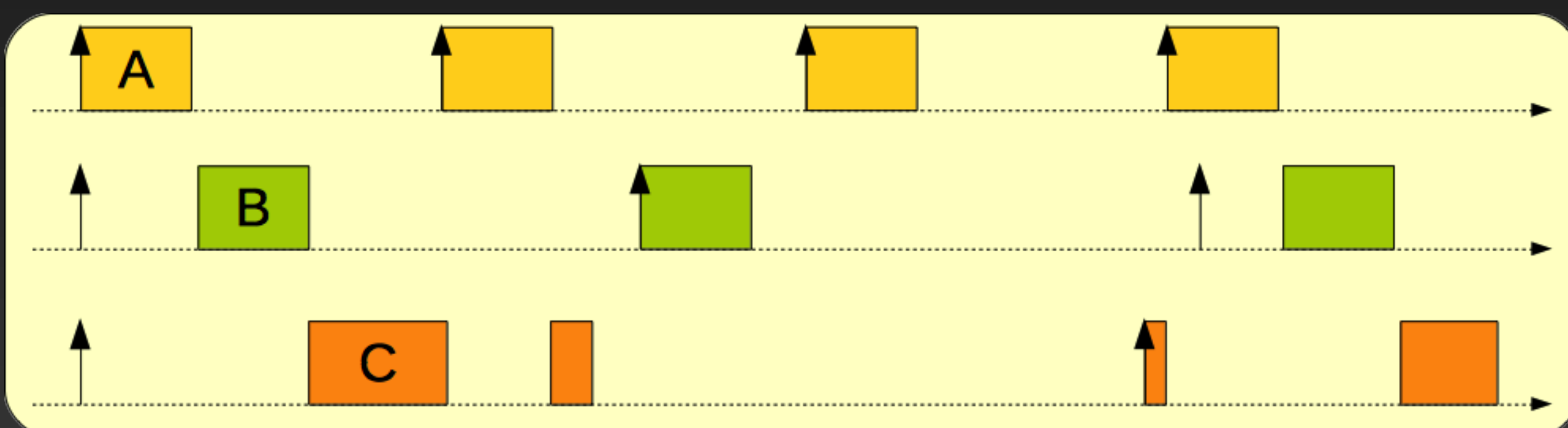
# Introduction

- Two major approaches to real-time scheduling:
  - Time-triggered (TT) – TT plans, cyclic executives
  - Priority-based (PB) – Fixed or dynamic priorities

A TT plan with three tasks




Three priority-scheduled tasks



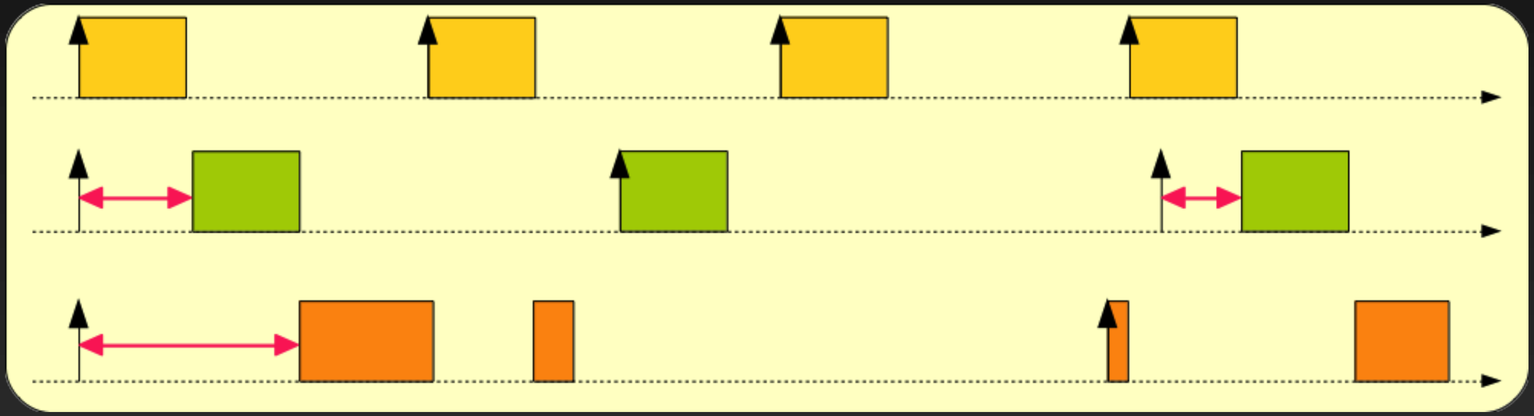
# Introduction

| Time-Triggered scheduling | Priority-Based scheduling |
|---|---|
| ✦ All events need be explicitly taken care of in advance – Building a schedule is complex | ✓ Functional and timing aspects are decoupled in the system design |
| ✓ Deterministic behaviour – Tasks start execution at predetermined points in time | ✦ Tasks have well defined release periods – but their start can be delayed due to interference |

Major issue in control systems

# Introduction

- Release Jitter
  - Actual release time - Theoretical start time
  - Degrades performance of digital controllers
  - Hinders precise distributed synchronisation



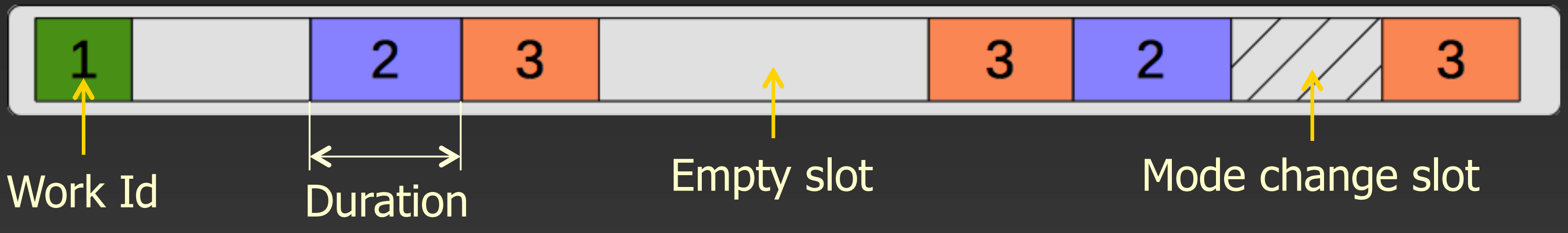
- Our goal
  - Grant short release delay for jitter-sensitive tasks

# Introduction

- The jitter issue has been tackled from different perspectives
  - Control Engineering:
    - Consider the effects of jitter in control equations
  - Priority-based scheduling:
    - Reduce deadlines – may work for a limited number of tasks
    - Control/scheduling co-design – use feedback from execution-time measurements to modify periods
    - Sub-task decomposition, giving higher priority to the most jitter-sensitive parts (initial and final)
- Our approach
  - Combine a TT plan (including jitter-sensitive tasks) with a PB schedule (jitter-tolerant tasks)

# System model

- Combined execution of:
  - A TT plan at the highest priority
  - A PB task set uses the rest of priority levels (RM, EDF, priority-specific dispatching…)
- TT Plan: ordered sequence of slots, each having:
  - A slot duration
  - An indication for what to do in each slot
    - Regular slots: a Work_Id referring to application code
    - Special slots: an indication for the TT scheduler



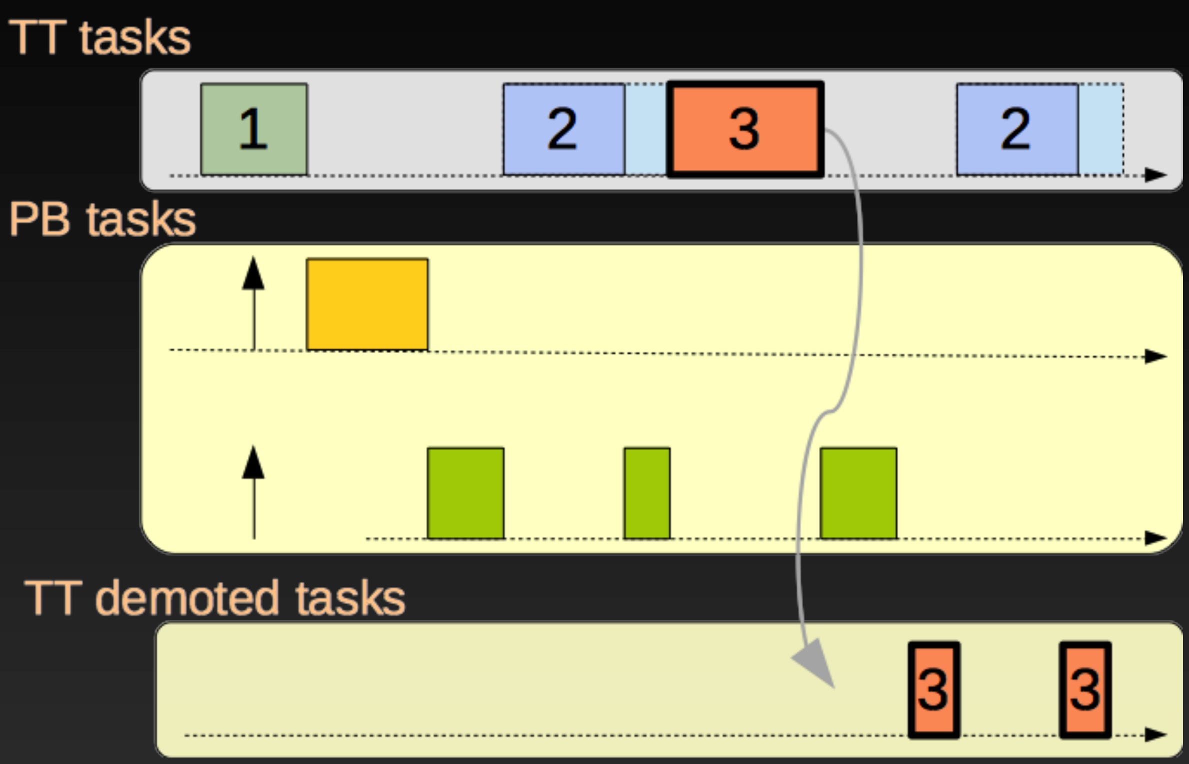
# System model

- TT scheduler
  - Triggered by plan events → slot start
- Based on the type of slot
  - Regular slot: Release the execution of the Work_Id for that slot (application code)
  - Empty slot: Time is available for PB tasks
  - Mode change slot: Empty slot + ability to change mode (plan)
    - The new plan starts at the end of the mode change slot

# System model: Overrun control

- TT tasks not allowed to execute beyond their slot duration
  - At least, not at TT priority level
- Possible corrective actions
  - Abort the offending task – perhaps too drastic
  - Mode change to degraded mode – *á la* mixed criticality
  - Continue execution at demoted priority

# System model: Overrun control




- Observation: under this policy, data shared between works must be protected
  - The plan design should take blocking times into account

# API for TT Plans

```
-- Context clauses omitted
package Time_Triggered_Scheduling is

   type Any_Work_Id is new Integer;
   subtype Special_Work_Id is Any_Work_Id range Any_Work_Id'First .. 0;
   subtype Regular_Work_Id is Any_Work_Id range 1 .. Any_Work_Id'Last;
   Empty_Slot          : constant Special_Work_Id;
   Mode_Change_Slot : constant Special_Work_Id;

   type Time_Slot is record
      Slot_Duration       : Time_Span;
      Work_Id                : Any_Work_Id;
   end record;

   type Time_Triggered_Plan is array (Natural range <>) of Time_Slot;
   type Time_Triggered_Plan_Access is access all Time_Triggered_Plan;
   ...
```

# API for TT Plans

```
...
protected type Time_Triggered_Scheduler (Nr_Of_Work_Ids: Regular_Work_Id)
      with Priority => System.Interrupt_Priority'Last is

       -- Setting a new time-triggered plan
       procedure Set_Plan (TTP : in Time_Triggered_Plan_Access; At_Time : in Time);
       procedure Set_Plan (TTP : in Time_Triggered_Plan_Access; In_Time : in Time_Span);

       -- Time-triggered tasks wait here for their next release
       entry Wait_For_Activation (Work_Id : Regular_Work_Id);
       ...

   private
       Empty_Slot : constant Special_Work_Id := 0;
       Mode_Change_Slot : constant Special_Work_Id := −1;
       ...
    end Time_Triggered_Scheduler;
end Time_Triggered_Scheduling;
```

# API for TT Plans

- A simple TT task pattern

```
TTS: Time_Triggered_Scheduler (3);     -- A scheduler for 3 different TT tasks

task type Simple_Worker (Work_Id: Regular_Work_Id; Prio: System.Priority)
  with Priority => Prio; -- Demoted priority in case of overrun

task body Simple_Worker is
begin
  loop
    TTS.Wait_For_Activation (Work_Id); -- Block here until my slot arrives
    Do_My_Work (...);                  -- Specific work actions
  end loop;
end Simple_Worker;
```

With relatively simple API extensions, the TT scheduler can support more complex task patterns

# API Extensions and Task Patterns

- API extensions (functions)
  - Get_Last_Release (Work_Id)
    - Time of last release of Work_Id
  - Get_Last_Slot_Duration (Work_Id)
    - Duration of last slot of Work_Id
  - Get_Next_Slot_Separation (Work_Id)
    - Time between start of last and next slot of Work_Id
      - This info added to Slot_Type at plan's design time
- API extensions (procedure)
  - Leave_TT_Level (Work_Id, Demoted_Priority)
    - Continue execution of Work_Id at Demoted_Priority level

# API Extensions and Task Patterns

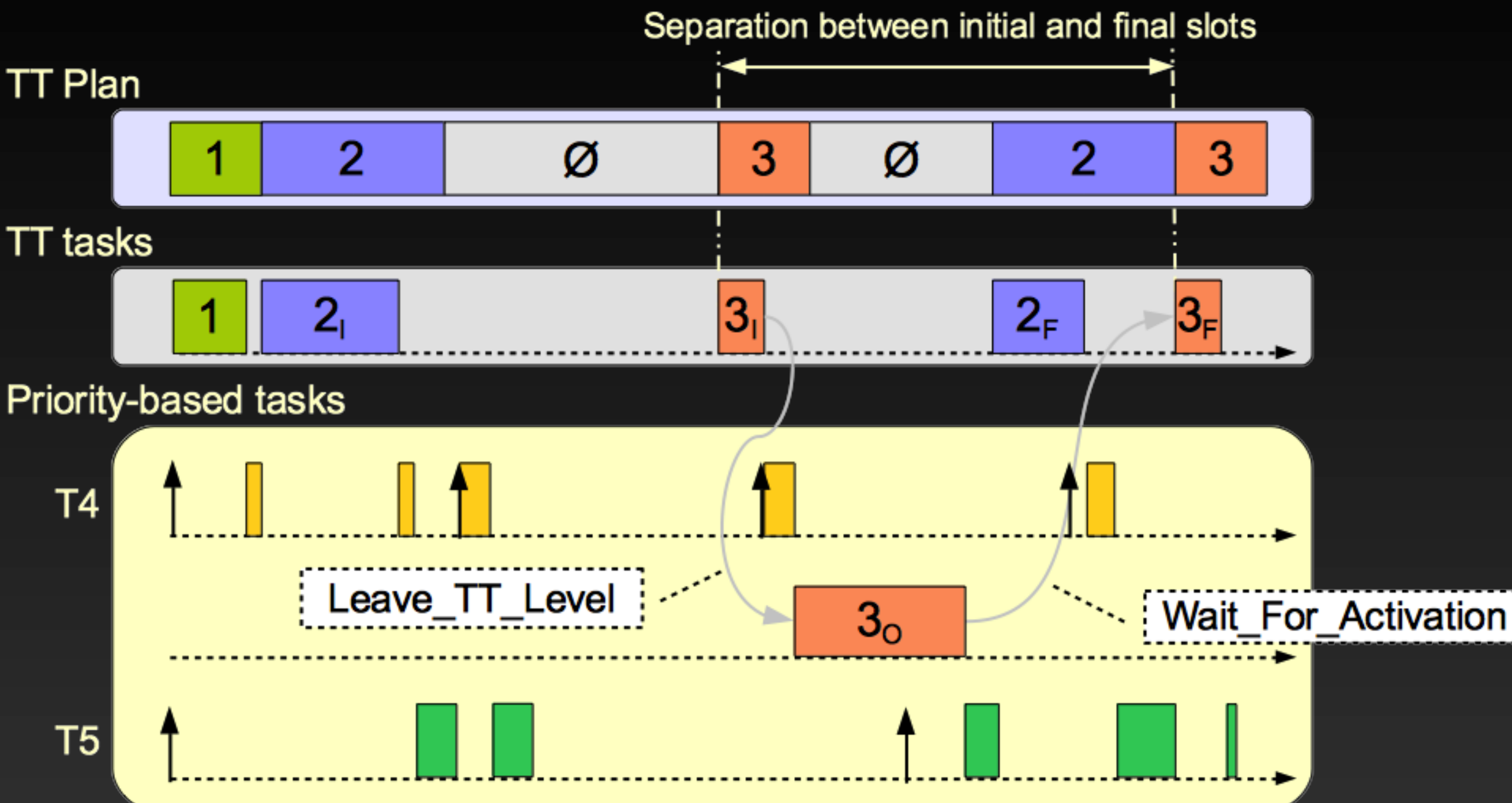
- Worker_With_Cancellation
  - The TT task cancels itself before causing an overrun
    - Tasks that cannot contribute any value after end of slot
  - Implementation
    - Task actions enclosed in ATC triggered by a delay until Last_Release + Last_Slot_Duration of Work_Id
- Worker_With_Initial_Final
  - Tasks have two clearly separated parts executed in two consecutive slots of the Work_Id
    - For TT tasks imposing deterministic I/O delays
  - Implementation
    - Concatenation of two simple workers

# API Extensions and Task Patterns

- Worker_With_Initial_Optional_Final
  - Executes an optional part between I & F
    - Optional part refines result of initial up to the point when result must be output in the final part

```
task body Worker_With_Initial_Optional_Final is
    -- Common data to all parts goes here
 begin
    loop
       TTS.Wait_For_Activation(Work_Id);
       Initial_Work;        -- Do initial part
       TTS.Leave_TT_Level (Work_Id, Optional_Part_Prio); -- Prepare to start optional part
       select
          delay until TTS.Get_Last_Release (Work_Id) + TTS.Get_Next_Slot_Separation (Work_Id);
       then abort
          Optional_Work; -- Do optional part
       end select;
       TTS.Wait_For_Activation(Work_Id);
       Final_Work;          -- Do final part
    end loop;
 end Worker_With_Initial_Optional_Final;
```

# API Extensions and Task Patterns

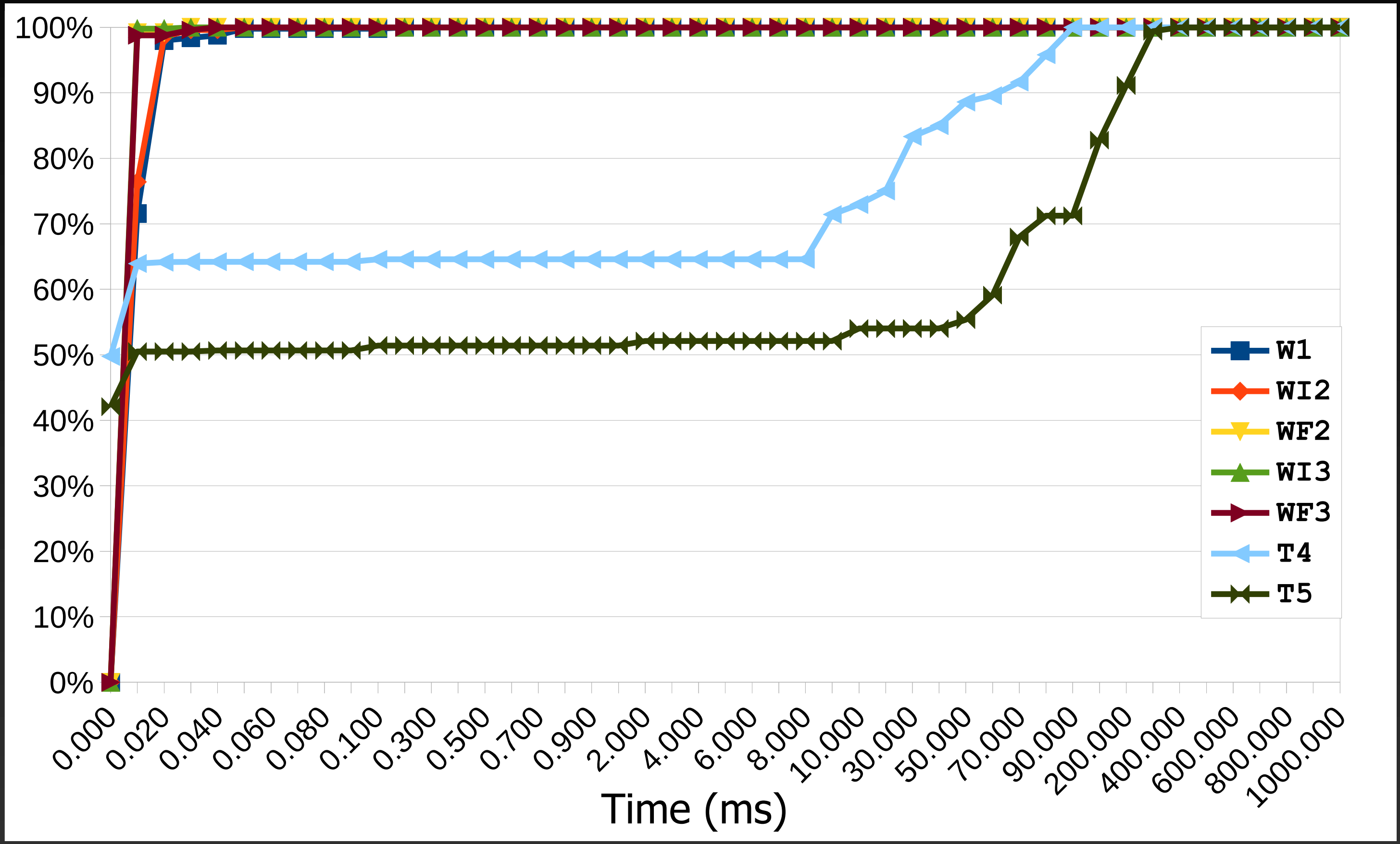


# Comments About the Implementation

- All TT scheduler's operations are O(1)
- All TT actions triggered by a timing event
- TT scheduler runs at Interrupt_Priority'Last
- All TT tasks run at Interrupt_Priority'Last - 1
- PB tasks execute below that priority
- TT tasks are implemented by actual Ada tasks
  - Cannot be simple procedures
  - A priority scheduler is already needed anyway

# Experimental Results

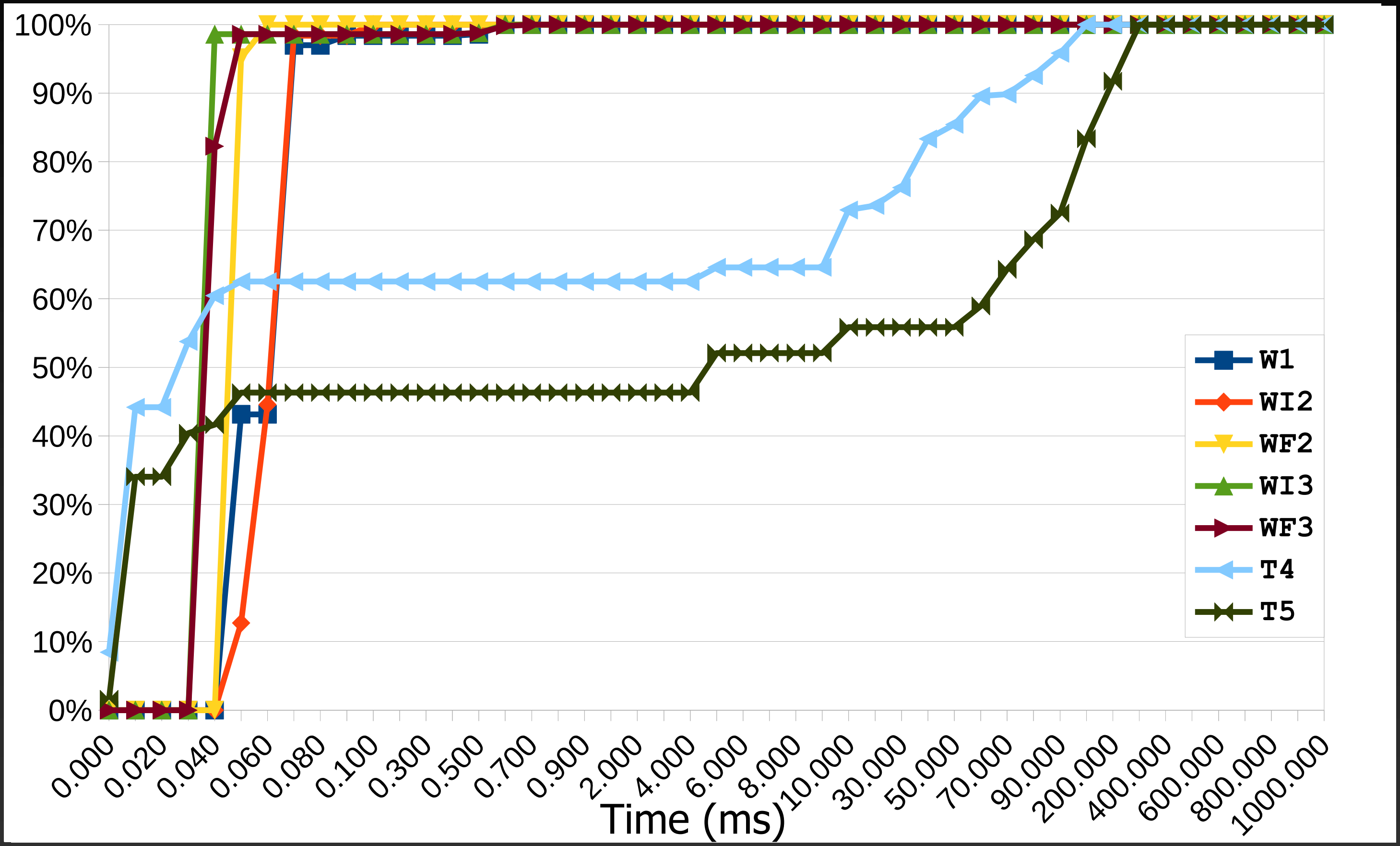
## Release jitter cummulative frequency histogram




MaRTE OS / Bare board – 6-year old Celeron @ 1.8 MHz

# Experimental Results

## Release jitter cummulative frequency histogram




MaRTE OS / Bare board – 12-year old Pentium III @ 800 MHz

# Conclusions & Further Work

- Conclusions
  - Encouraging results
  - Reuse of legacy TT plans
  - Simpler TT plan design (has only jitter-sensitive tasks)
  - Open to new TT task patterns (IMF, IMOF,…)
- Further Work
  - Schedulability analysis (End-To-End-Flow)
  - Use in Multiprocessors
    - One plan per processor; limited & controlled migration
  - Adaptation to Ravenscar
    - We're using entry families, requeue, dynamic priorities, local TE's
    - Would need to re-engineer and restrict supported patterns