Flexible Mapping of Concurrent Object-Oriented Applications to MPSoC Platforms

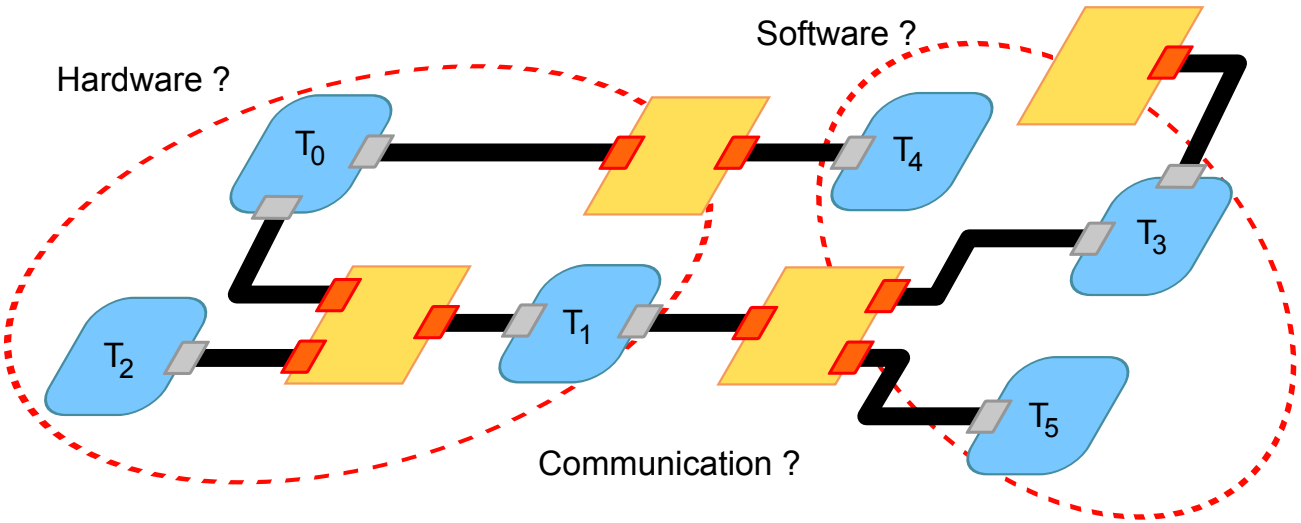
Verkehr
Transportation

Philipp A. Hartmann, Kim Grüttner,
Frank Oppenheimer, Wolfgang Nebel
OFFIS Institute for Information Technology
Oldenburg, Germany

June 28, 2010
Map2MPSoC'2011
St. Goar, Germany

1 HW/SW Communication and Partitioning

Motivation



- **Mapping** of components (Hard-/Software) is often subject to **changes**
  - but usually requires expensive design modifications
- **Communication** between Tasks is a **critical part** of concurrent embedded systems

2 Outline

3 Outline

Introduction

4 OSSS – Overview (I)

Oldenburg System Synthesis Subset

- SystemC-based Design Methodology for **Object-Oriented HW/SW Co-Design**
  - modelling library available under LGPL → `http://system-synthesis.org`
- **Layered refinement flow** towards implementation

- Application is modelled as composition of (hardware or software) **tasks** and synchronising **objects**.
- User-defined **Shared Objects** enable transaction-based modelling.
  - Communication via abstract method calls.
  - Synchronisation via guard conditions.

- Separate modelling of **application** und **architecture**
  - Initial description of the system as composition of **tasks** and communication **objects**
  - **Explicit allocation and mapping** of processing elements for scheduling and resource sharing.

5 OSSS – Overview (II)

Oldenburg System Synthesis Subset

- Abstract, host-based **simulation of embedded software** multi-tasking
  - → enables exploration of **software architecture**
  - local scheduling of multiple tasks on a (virtual) processor cores
  - based on annotated execution times
  - task priorities, deadlines, preemption, synchronisation, resource access policies
- Exploration of (hardware) platform variants, esp. for **communication refinement**.
  - Mapping of abstract communication to concrete channels
  - Buses, point-to-point, . . .
  - incl. serialisation, and HW/SW communication
- Path towards implementation
  - Cross-compilation for target against **software run-time** based on (RT-)Linux
  - **Hardware synthesis for** dedicated hardware **Shared Objects** objects and hardware tasks via synthesis tool *Fossy*.
  - Generic **driver framework** for HW/SW and core/core communication

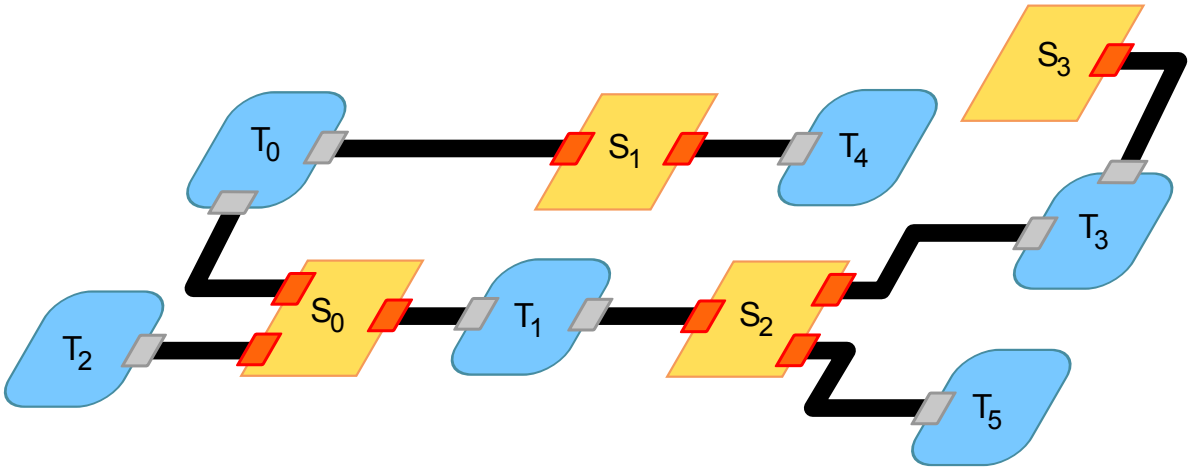
6 OSSS– Layered Design Flow

Oldenburg System Synthesis Subset

Application Layer

Task

(Shared) Object



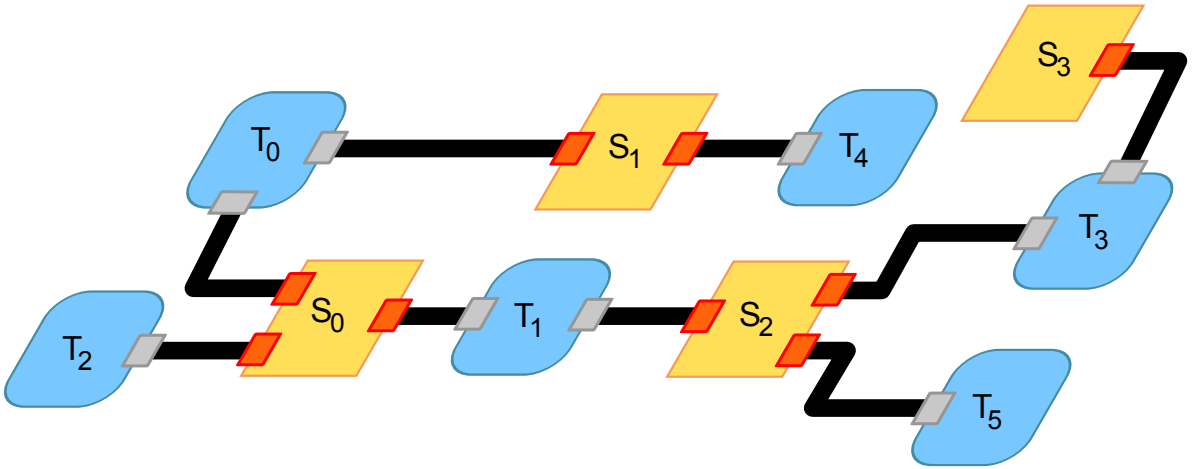
6 OSSS– Layered Design Flow

Oldenburg System Synthesis Subset

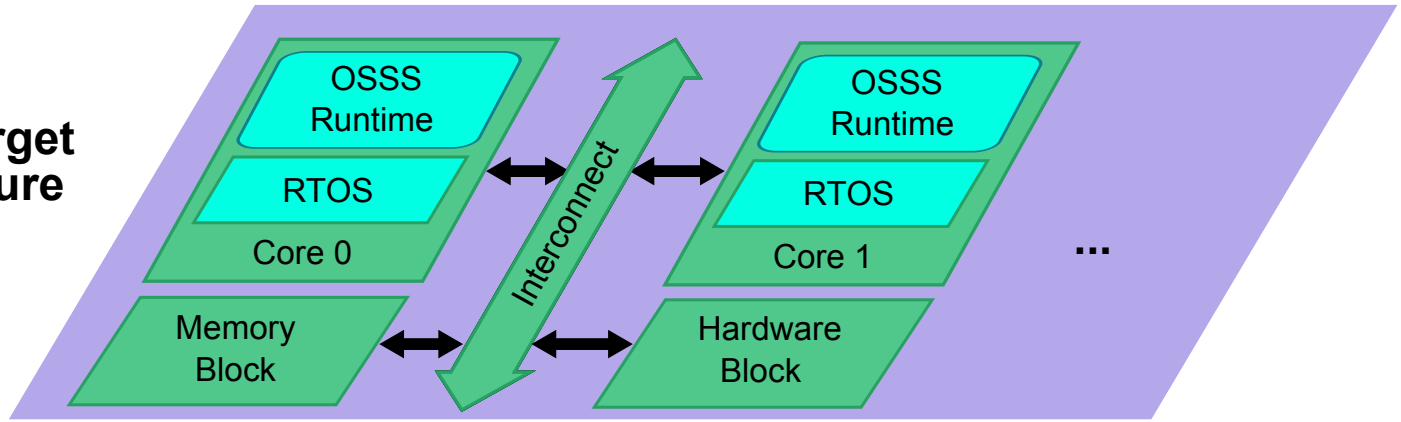
Application Layer

Task

(Shared) Object

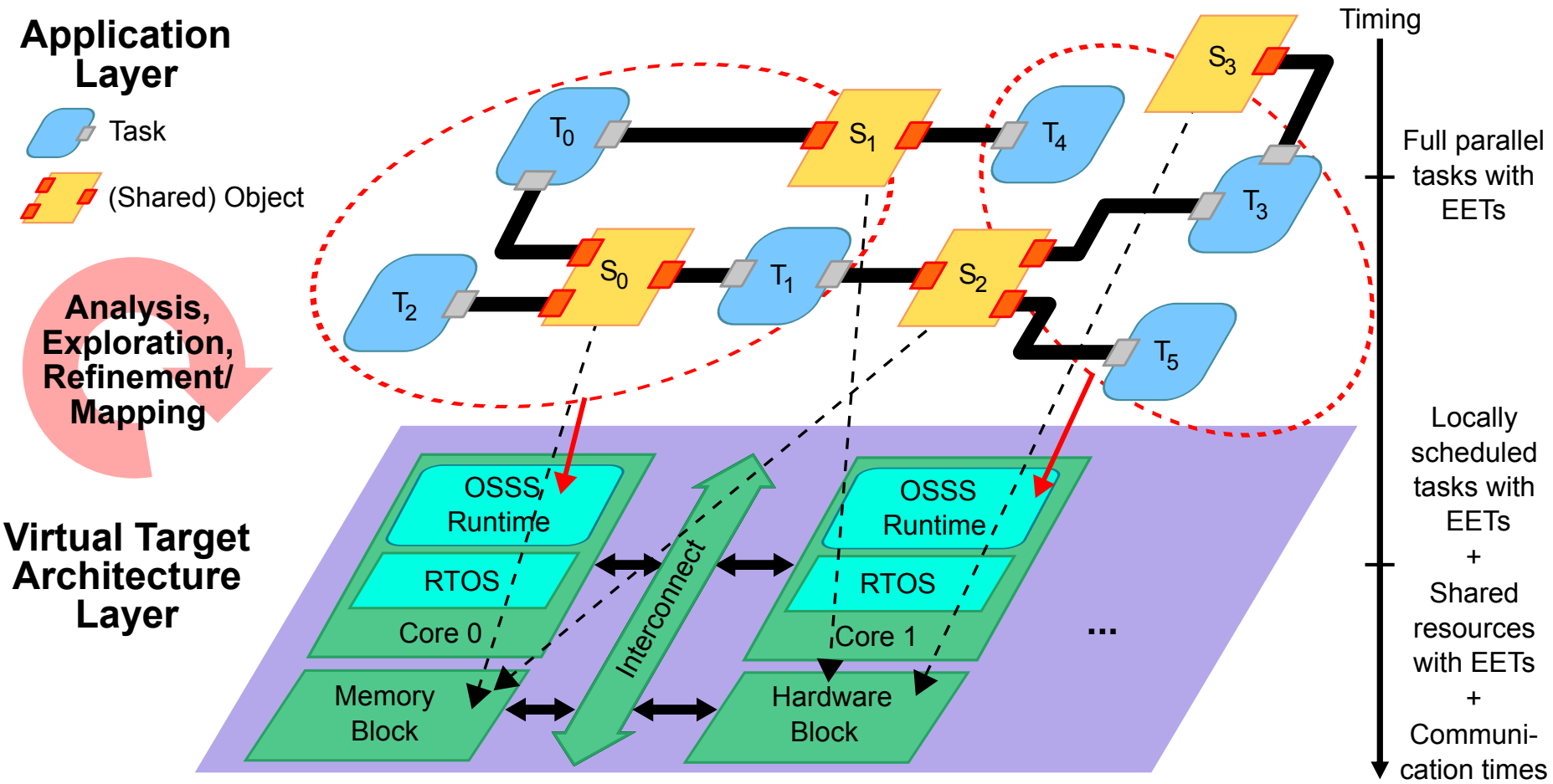


Virtual Target Architecture Layer



6 OSSS– Layered Design Flow

Oldenburg System Synthesis Subset



7 Outline

Application Layer model

8 Communication via Shared Objects

Method-based communication

Communication between Tasks is modelled in terms of so-called **Shared Objects**.

- User-defined, **method-based** interfaces (services)
- Guaranteed **mutual exclusive** access
  - concurrently accessing tasks (clients) are **synchronised transparently**
  - ensures consistency

8 Communication via Shared Objects

Method-based communication

Communication between Tasks is modelled in terms of so-called **Shared Objects**.

- User-defined, **method-based** interfaces (services)
- Guaranteed **mutual exclusive** access
  - concurrently accessing tasks (clients) are **synchronised transparently**
  - ensures consistency
- Services can have **logical pre-conditions (Guards)**
  - boolean conditions, based on **inner state** of Shared Object
  - can block a caller, until condition holds
  - a blocked service call can only be released by **another unblocked service**, changing the object's inner state

FIFO method	guard
`put(int item)`	`full`
`int get()`	`!empty`

Table: Simple guards example

8 Communication via Shared Objects

Method-based communication

Communication between Tasks is modelled in terms of so-called **Shared Objects**.

- User-defined, **method-based** interfaces (services)
- Guaranteed **mutual exclusive** access
  - concurrently accessing tasks (clients) are **synchronised transparently**
  - ensures consistency
- Services can have **logical pre-conditions (Guards)**
  - boolean conditions, based on **inner state** of Shared Object
  - can block a caller, until condition holds
  - a blocked service call can only be released by **another unblocked service**, changing the object's inner state
- Different flavours, transparently handled during simulation/implementation
  - **Local** objects, all clients under the same RTOS
  - **Software Objects**, explicitly mapped to shared memory
  - Dedicated **Hardware Objects**, e.g. as accelerators

FIFO method	guard
`put(int item)`	`full`
`int get()`	`!empty`

Table: Simple guards example

9 Timing annotations in OSSS

Application Layer model

- **E**stimated **E**xecution **T**ime (EET) annotations
- Block-wise annotation, can not be nested
- Flexible granularity
- Must not contain communication

```
while( some_condition )
  // block has to be finished within 1ms
 OSSS_RET( 1, SC_MS )
{
  OSSS_EET( 20, SC_US )
  {
    // .. some computation .
    max_i = compute_max_i();
  }
  // estimate data-dependent loop
  for( int i=0; i<max_i; ++i )
    OSSS_EET( 100, SC_US )
    {
      // .. loop body .
    }
  if( my_condition )
  {
    // comm. outside of EET blocks onl
    result = my_shared->get();
  }
}
```

9 Timing annotations in OSSS

Application Layer model

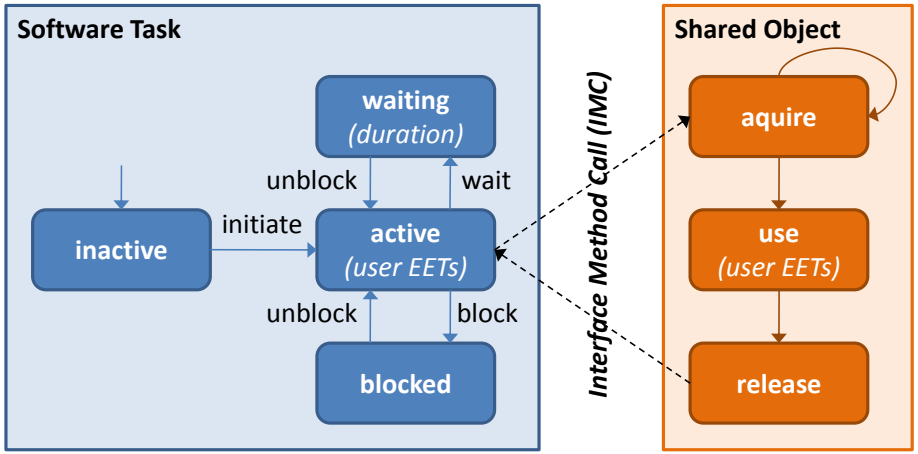
- **E**stimated **E**xecution **T**ime (EET) annotations
- Block-wise annotation, can not be nested
- Flexible granularity
- Must not contain communication

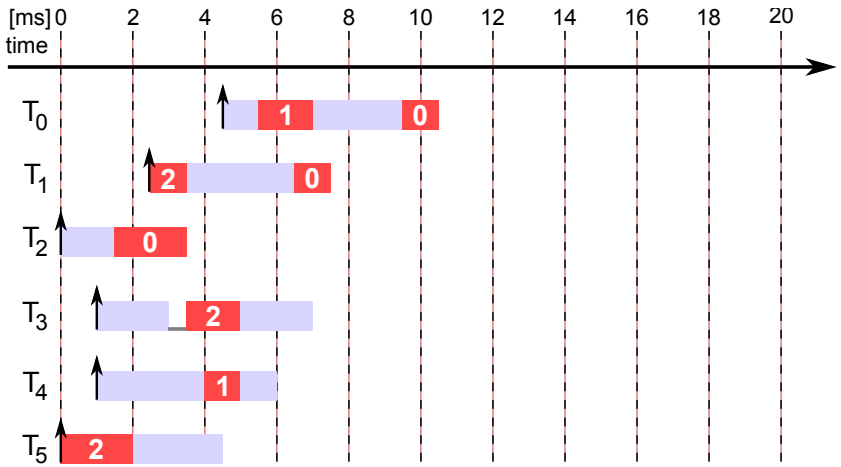
- **R**equired **E**xecution **T**ime (RET) annotations
- Check of relative deadlines during simulation
- Can be nested, and enclose communication

```
while( some_condition )
  // block has to be finished within 1ms
  OSSS_RET( 1, SC_MS )
{
  OSSS_EET( 20, SC_US )
  {
    // .. some computation .
    max_i = compute_max_i();
  }
  // estimate data-dependent loop
  for( int i=0; i<max_i; ++i )
    OSSS_EET( 100, SC_US )
    {
      // .. loop body .
    }
  if( my_condition )
  {
    // comm. outside of EET blocks onl
    result = my_shared->get();
  }
}
```

10 Application Layer simulation

Application Layer model





- **Highest simulation performance**, used for functional validation
- No task scheduling, all tasks run **fully parallel**
- Only EETs are considered (if available)
- Resource contention only due to **conflicting acceses**

11 Outline

Architecture model

12 Virtual Target Architecture model

Architecture model

- **Mapping** of **Application** Layer elements **to** concrete **Virtual Target Architecture** elements
  - processor cores
  - hardware blocks
  - memories
  - interconnects
- Abstract **communication links** mapped to explicit interconnects
  - point-to-point channels, buses,. . .
  - (Remote) Service calls handled via **Remote Method Invocation** protocol (RMI)

12 Virtual Target Architecture model

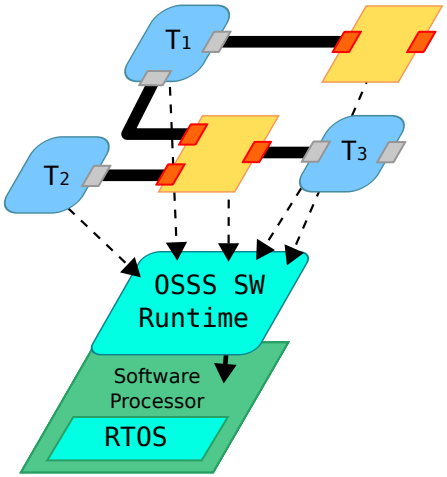
Architecture model

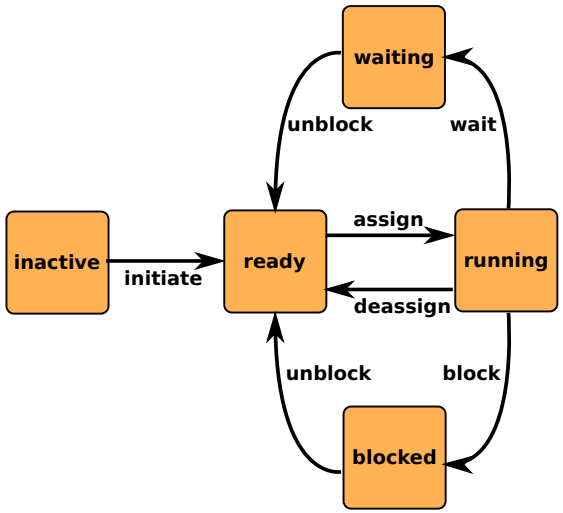
- **Mapping** of **Application** Layer elements **to** concrete **Virtual Target Architecture** elements
  - processor cores
  - hardware blocks
  - memories
  - interconnects
- Abstract **communication links** mapped to explicit interconnects
  - point-to-point channels, buses,. . .
  - (Remote) Service calls handled via **Remote Method Invocation** protocol (RMI)
- **Task mapping**
  - Hardware tasks can only be mapped exclusively (no scheduling).
  - Multiple software tasks may share one processor, handled by a local RTOS/runtime
    $\rightarrow$ **Asymmetric multi-processing**
- **Shared Object mapping**
  - Mapping to Hardware, Software (SHM), or Local objects
  - **Remote** objects require explicit mapping.
  - **Local** objects can be mapped implicitly.

13 OSSS Software Multi-Tasking

Architecture model

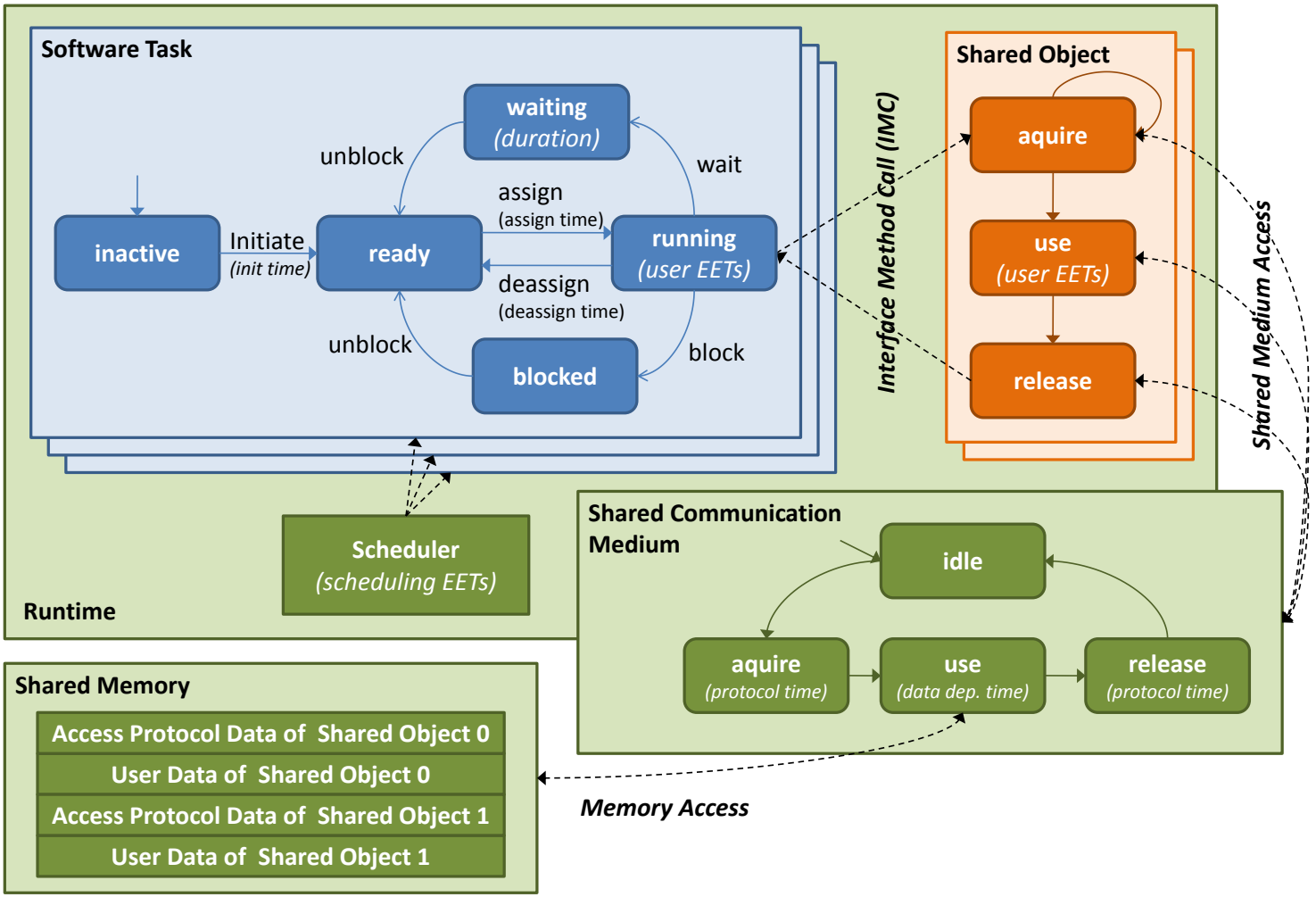
- Abstraction of underlying RTOS: **OSSS Software Runtime**
  - hide low-level, error-prone RTOS primitives from designer
- Main purpose: **scheduling** of mapped tasks
  - **Task state** management and guarantee, that only one task is running at a time
  - Periodic task activation, deadline observation
  - Time synchronisation and preemption model
  - Handling resource (Shared Object) access, signaling of **unblocked** tasks.
- Several predefined **scheduling policies** supported
  - Static priorities, preemptive and cooperative
  - Time-slice based round-robin
  - EDF, RMS
  - . . . and **user-defined schedulers** via interface class
- **Synchronisation** with other runtime instances.





14 Virtual Target Architecture Layer simulation

Architecture model

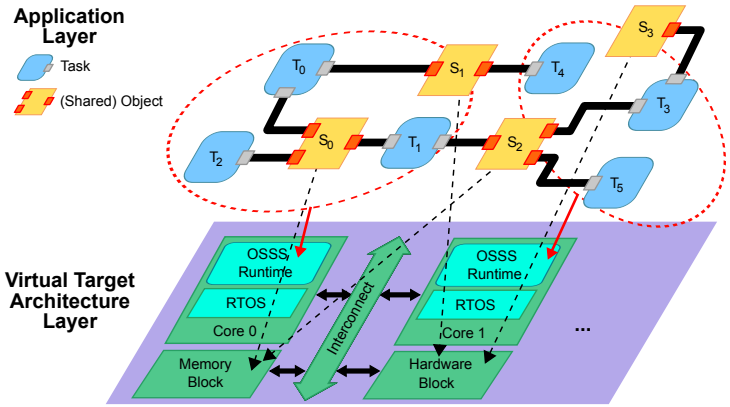


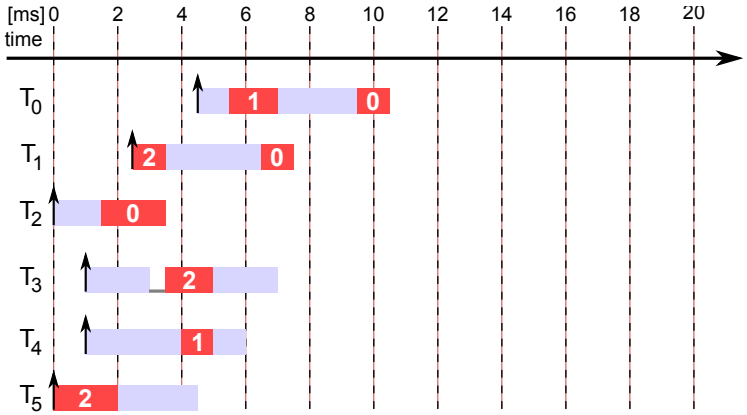
15 Outline

Simulation of platform artefacts

16 Simulation of Platform Artefacts

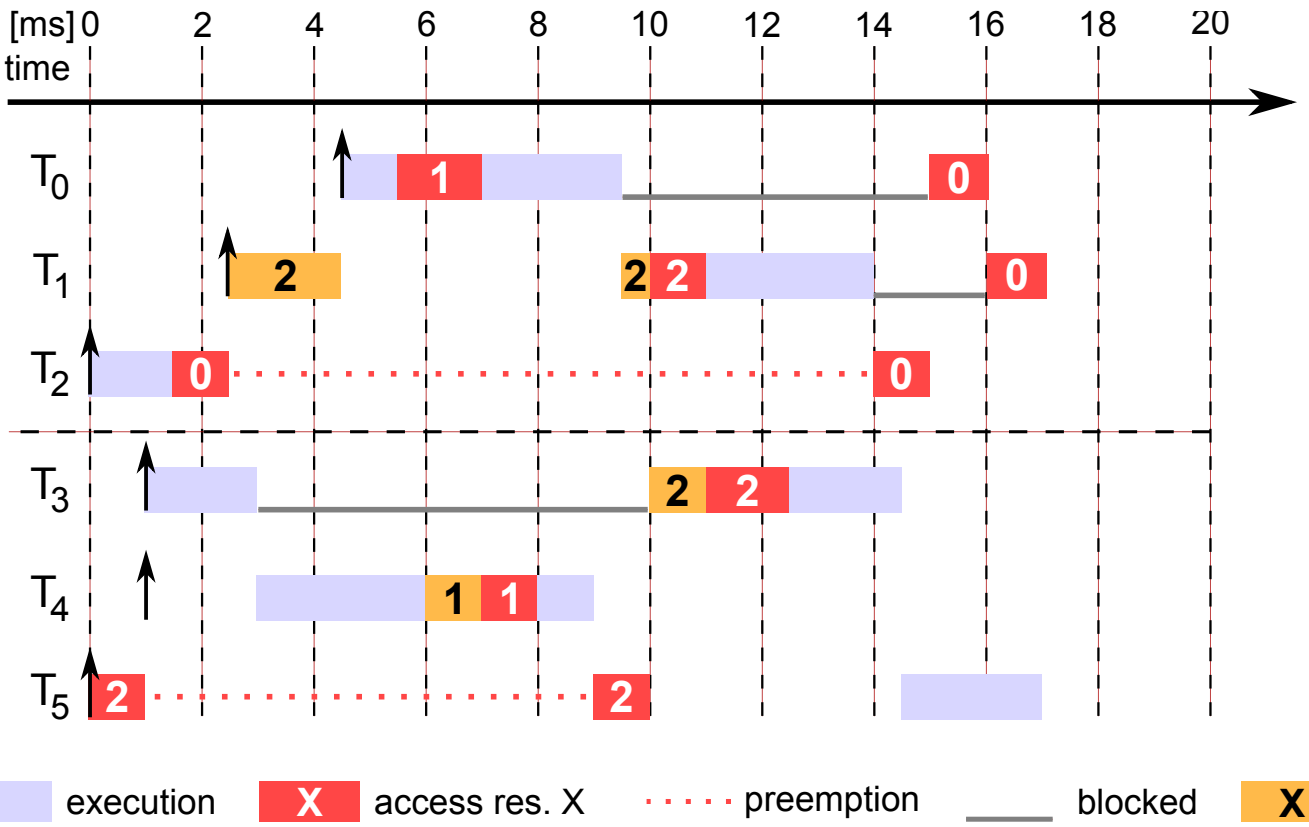
- Virtual Target Architecture Layer simulation enables **analysis of (software) architecture artefacts**
- Effects of design parameters on system behaviour
  - **application mapping**
  - (local) scheduling policies, task priorities
  - distributed **resource access** strategies
- Some examples in the following, based on artificial system
  - Static priorities: $T_0 > T_1 > \ldots > T_5$
  - Core 0 higher priority than Core 1
  - Distributed resource access (no central arbiter)
  - Blocking behaviour either by **suspend** or **busy-waiting**
  - Task **preemption during resource access**
    - **allowed** (not handled specifically)
    - allowed, but support (local) **priority inheritance**
    - explicitly **suppressed**





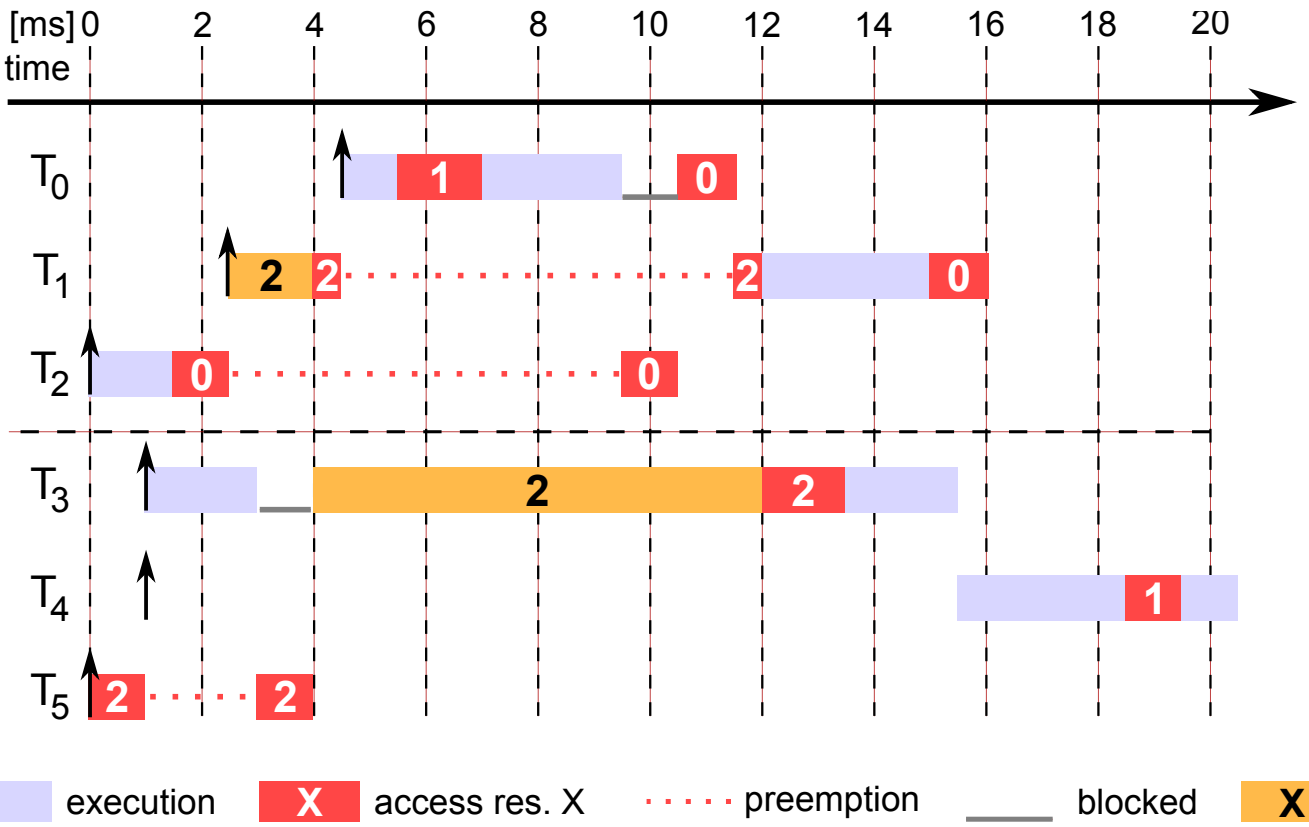
17 Priority inversion – busy waiting

Resource access example (I)



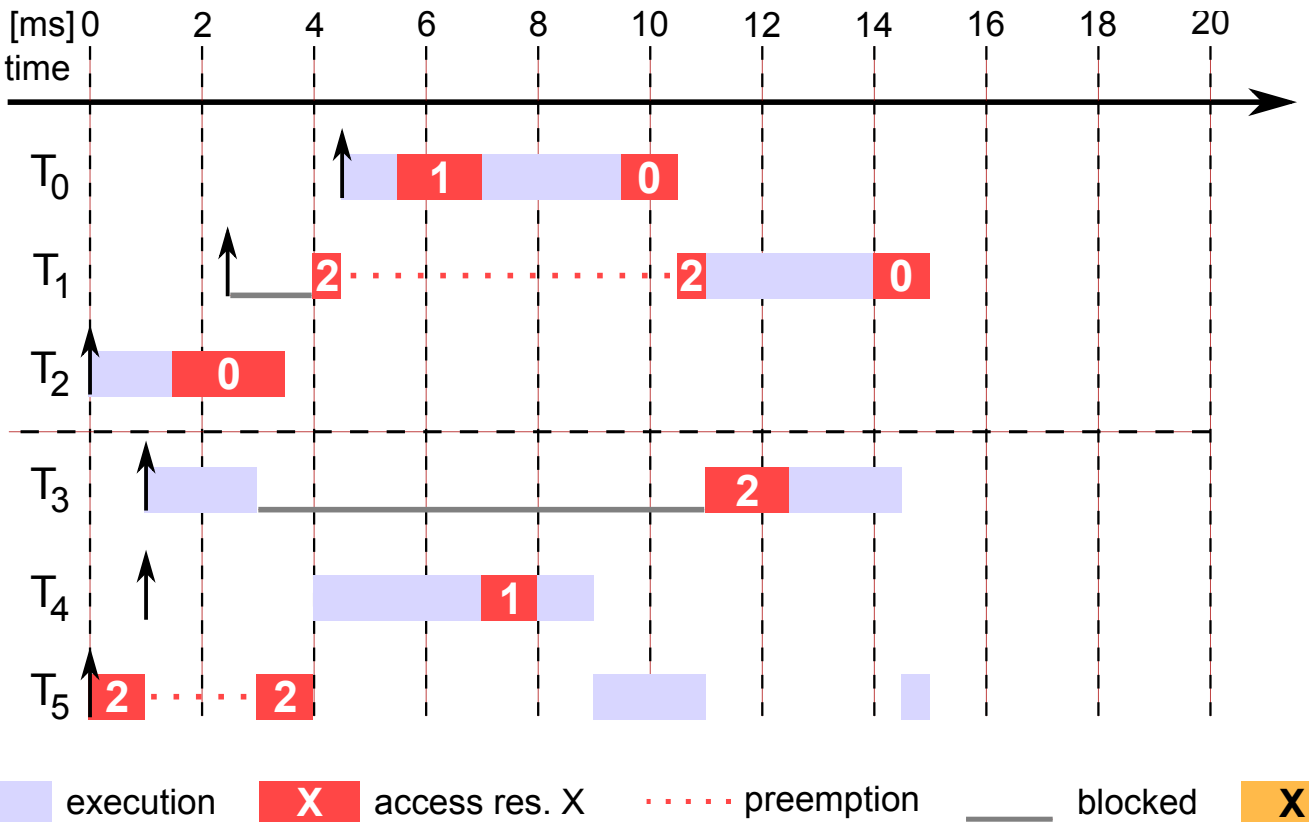
18 Priority inheritance – busy waiting

Resource access example (II)



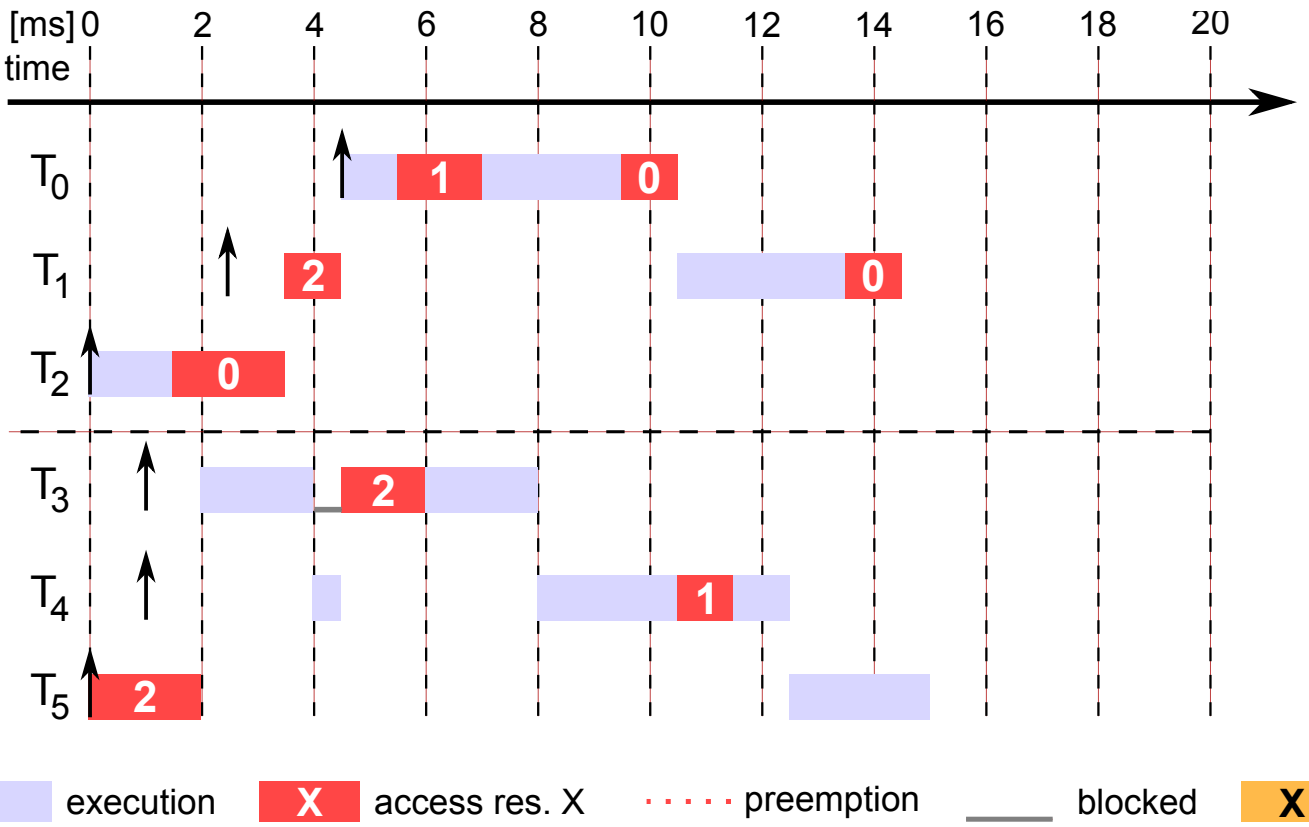
19 Priority inheritance – suspend

Resource access example (III)



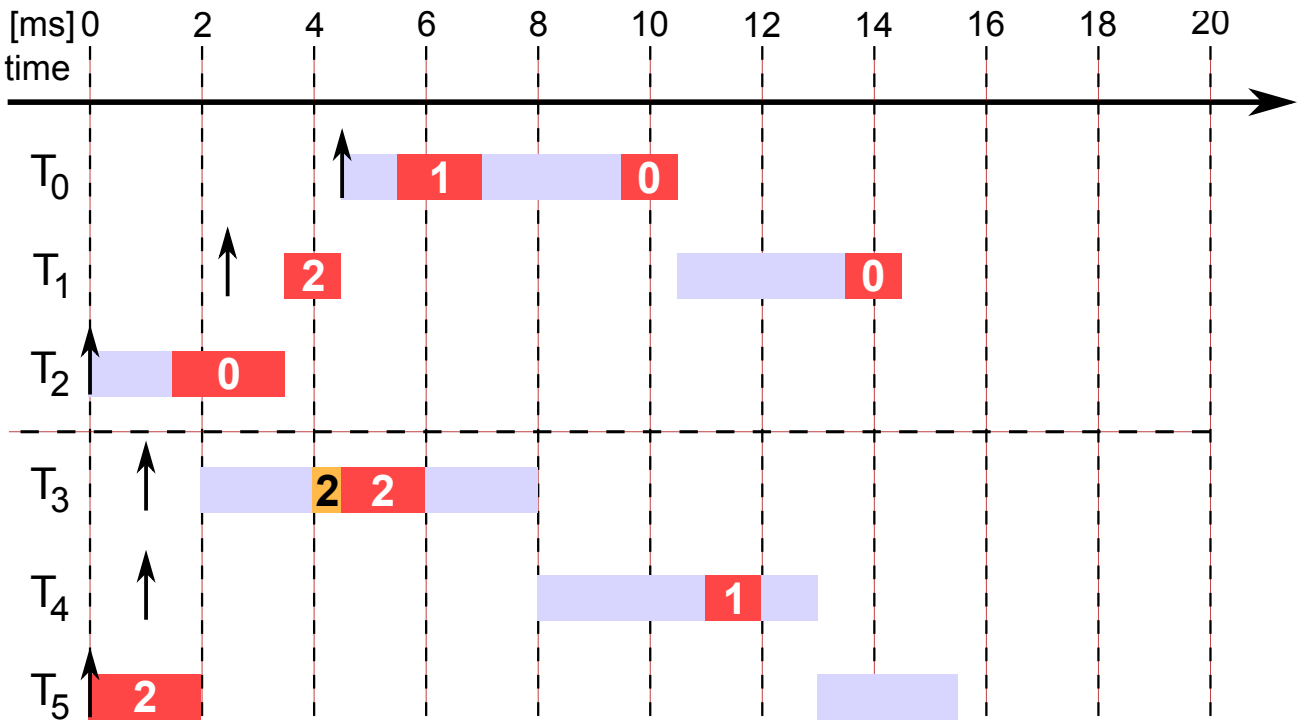
20 No preemption – suspend

Resource access example (IV)



21 No preemption – busy-waiting

Resource access example (V)



22 Outline

Conclusion

23 Summary & Outlook

Conclusion

- Extension of OSSS Design Methodology for MPSoC.
  - Provides **flexible mapping** of Tasks and Objects to explicit virtual architecture elements.
  - separate modelling of Application and Architecture avoids **costly design changes** during exploration
  - direct **path to implementation**
- Abstract software modelling
  - Based on **annotated execution times** and explicit modelling of **shared resources**.
  - Enables early analysis of **software architecture** artefacts by showing impact of
    - scheduling policies
    - priorities, periods, deadlines
    - resource access strategies

23 Summary & Outlook

Conclusion

- Extension of OSSS Design Methodology for MPSoC.
  - Provides **flexible mapping** of Tasks and Objects to explicit virtual architecture elements.
  - separate modelling of Application and Architecture avoids **costly design changes** during exploration
  - direct **path to implementation**
- Abstract software modelling
  - Based on **annotated execution times** and explicit modelling of **shared resources**.
  - Enables early analysis of **software architecture** artefacts by showing impact of
    - scheduling policies
    - priorities, periods, deadlines
    - resource access strategies
- Outlook
  - Extend software multi-tasking model towards (locally) **symmetric multi-processing** (multiple cores per RTOS instance)
  - Extend existing **mapping to formal model** based on **Extended Real-Time Task Networks**
  - Enable design decisions through **back annotation of communication delays** to simulation & analytical model.

Thanks for your attention!

Questions?