Achieving Predictable Multicore Execution of Automotive Applications Using the LET Paradigm

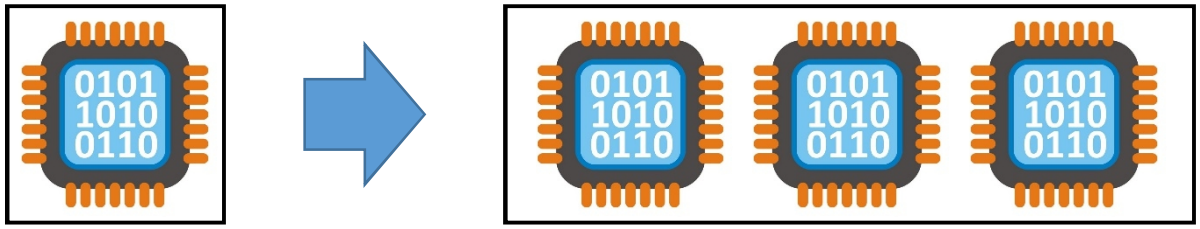
Alessandro Biondi and Marco Di Natale

Scuola Superiore Sant'Anna, Pisa, Italy

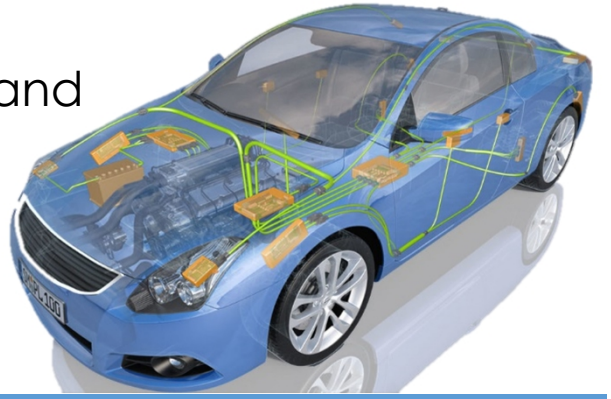
Sant'Anna
Scuola Universitaria Superiore Pisa

Introduction

- The introduction of safety-critical functions in *automotive systems*, together with the advent of multicore platforms, brings the need to **rethink** the development and execution paradigms for embedded functionality

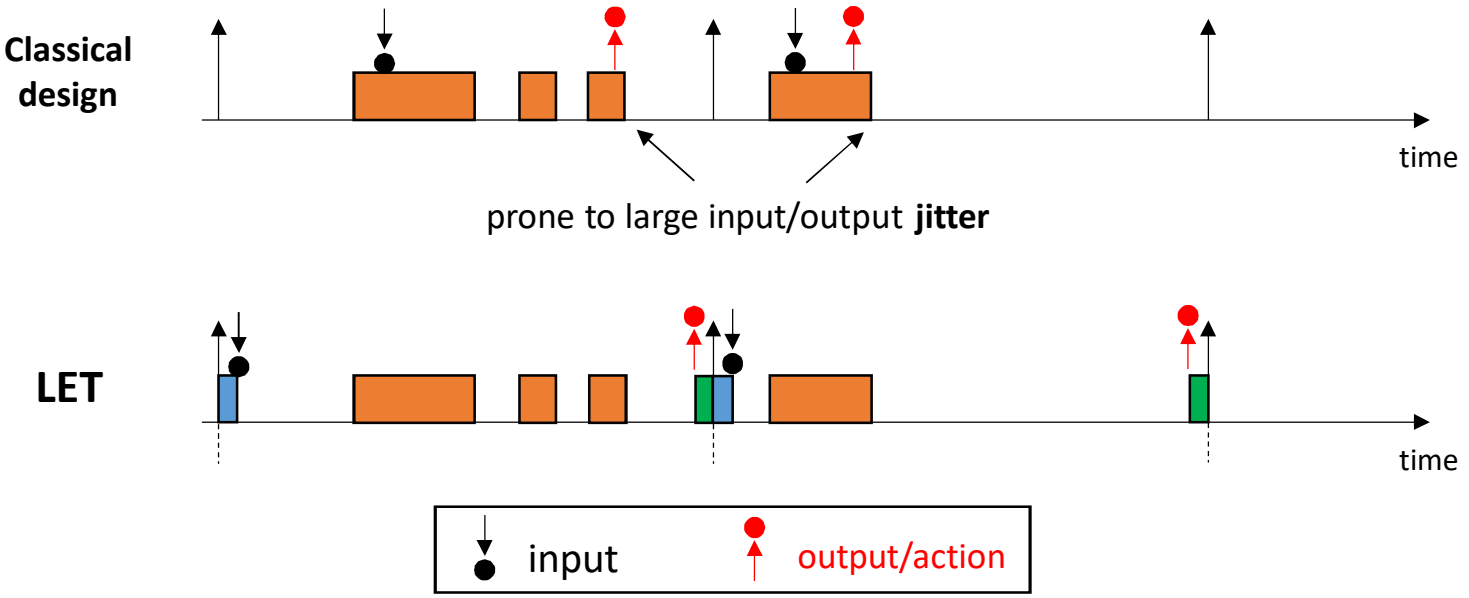


- Several issues in switching to multicores…
  - Lack of appropriate modeling for partitioning applications
  - Legacy SW with causality implicitly verified on single core
  - Need for a portable timing model
  - Achieving timing predictability is not trivial
- …plus increasingly stringent legal regulations and *certifiability* requirements



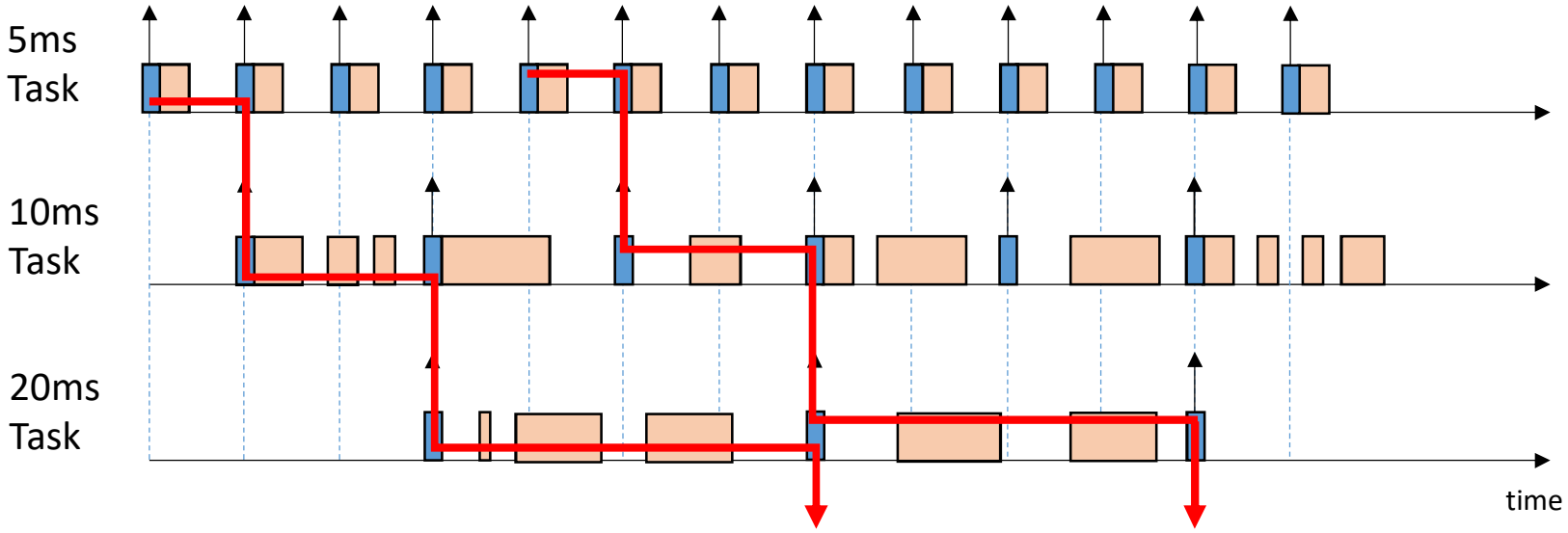
Logical Execution Time

- Logical Execution Time (**LET**) introduced as a method to eliminate output **jitter** and provide time determinism in the implementation of control algorithms [*Henzinger et al.* 2003]
- LET can be realized with *different scheduling strategies* provided that the desired semantic is respected



Logical Execution Time

- Recent renewed attention on LET by automotive industry
- Several players are adopting LET to provide deterministic end-to-end latencies of chains of **communicating tasks**
- LET seems a promising solution to also solve other issues in the design and development of real-time systems (e.g., SW portability, interface with control engineers, etc.)



Example from Dirk Ziegenbein's talk at Dagstuhl seminar on LET

This Talk

1

Scheduling strategy for realizing LET communication in **multicore** platforms to achieve execution predictability

2

Implementation on Aurix Tricore TC275 and evaluation with a pseudo-realistic case study (*WATERS Challenge 2017* by Bosch)

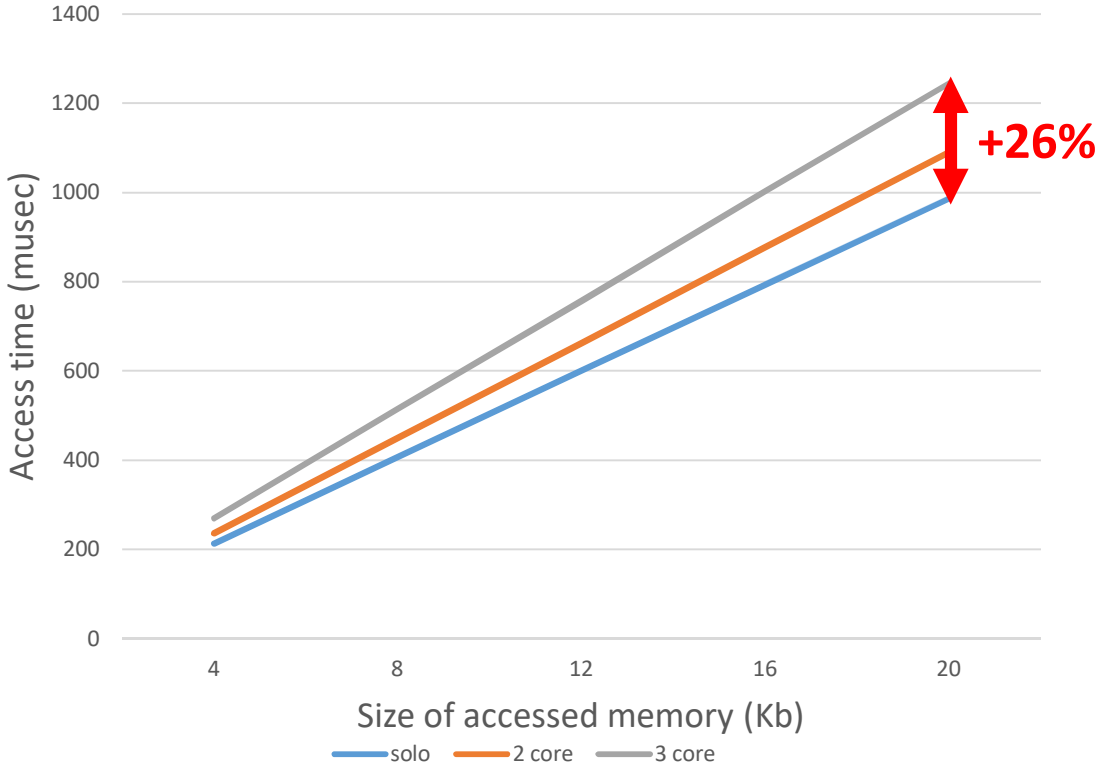
LET AS AN OPPORTUNITY TO CONTROL MEMORY CONTENTION

Memory Contention

- **Contention** in accessing *shared memories* strongly harms the predictability of software running upon multicores
- Any-time access to shared memories carries considerable **pessimism** in timing/schedulability analysis

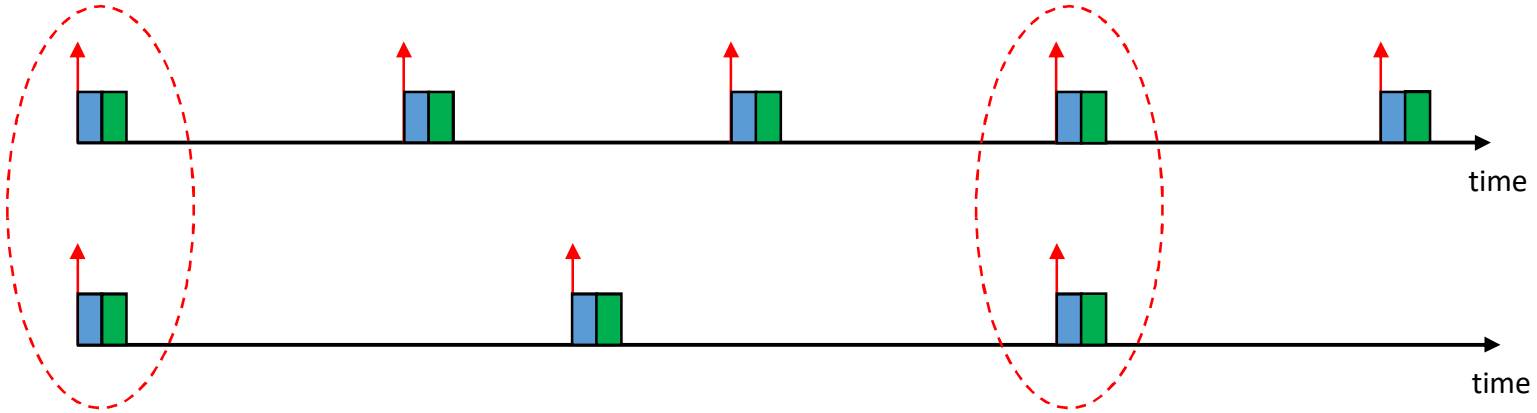
Testbed: Aurix Tricore TC27x

- Shared buffer of 20Kb in global memory
- Core 0 is under analysis and accesses a portion of the buffer
- Cores 1 and 2 are continuously writing into the buffer to generate interference



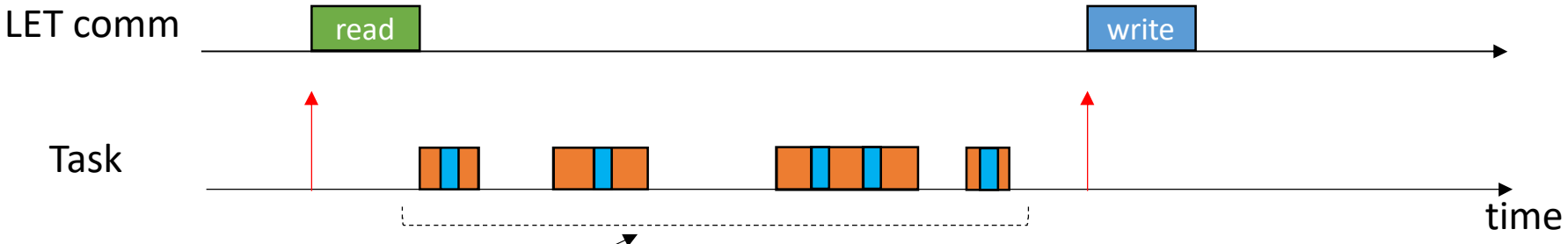
Controlling Memory Contention

- **Selling point**: scheduling LET communication at the beginning of periodic instances allows *localizing* the access to shared memories in precise time windows
- Such time windows are determined by the tasks' periods
  - they are hence *predictable* (off-line)
  - and can host explicit arbitration to avoid contention

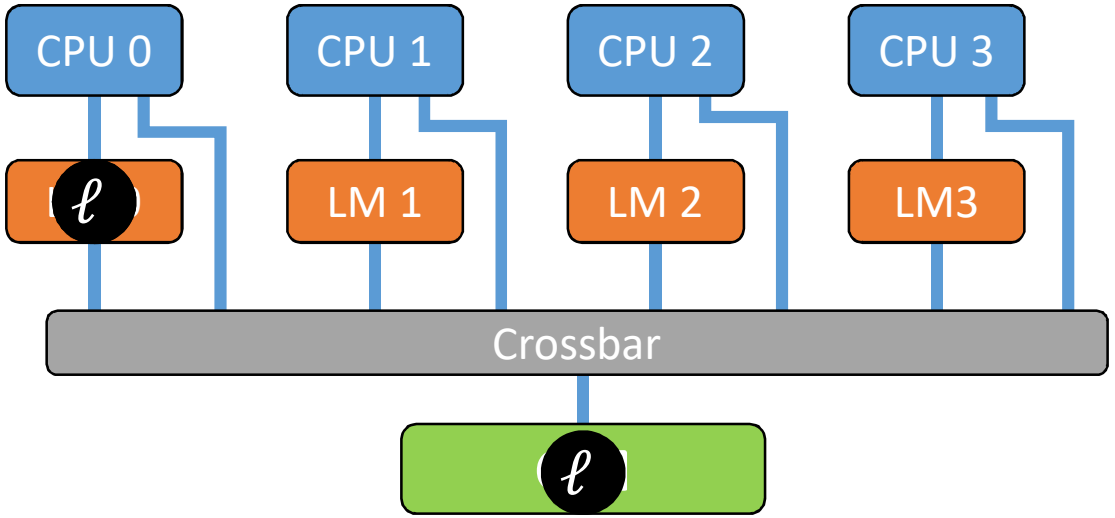


Realizing LET Communication

- Local copies of data allocated in the scratchpads
- Shared copies allocated in the global memory
- LET communication stack moves data from global to local memories and viceversa

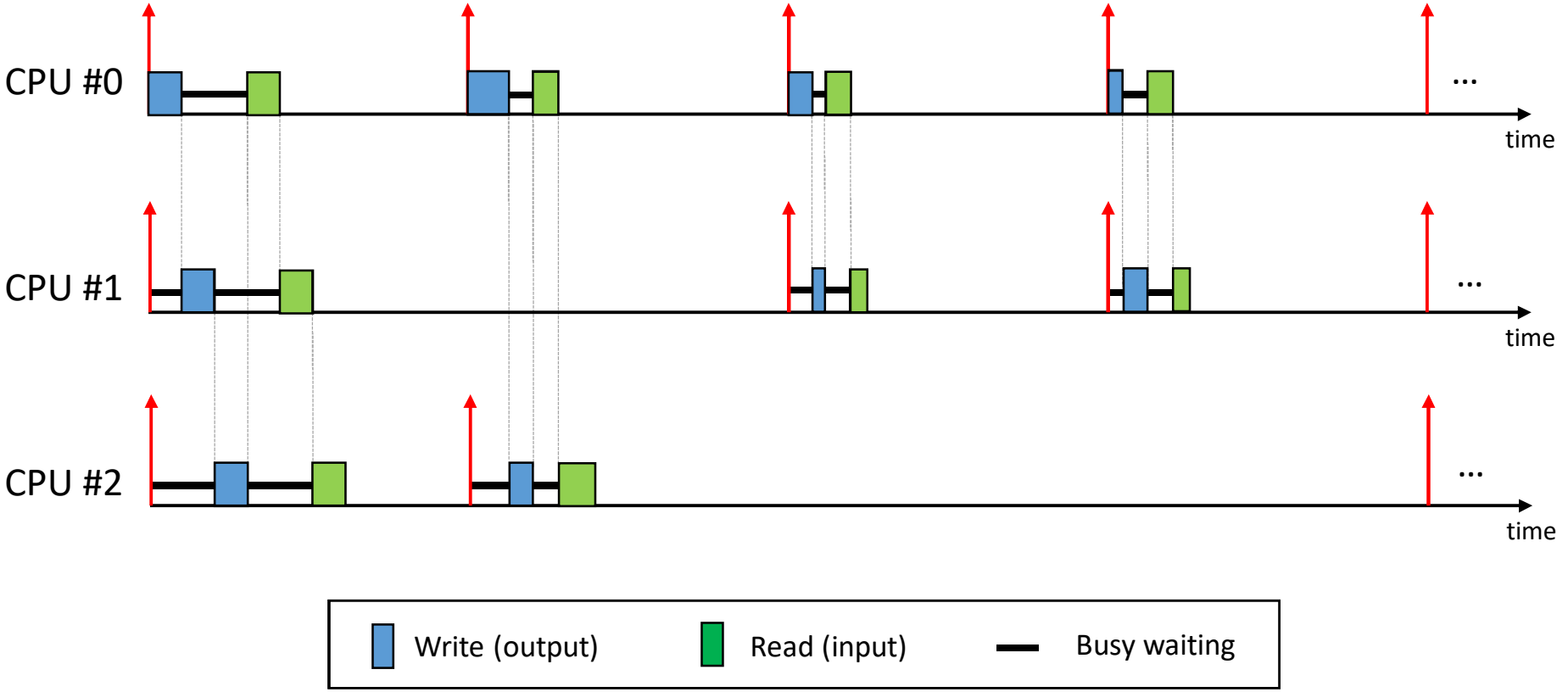


Preemptable task execution with **contention-free** accesses to local memory



LET Tasks: Synchronization

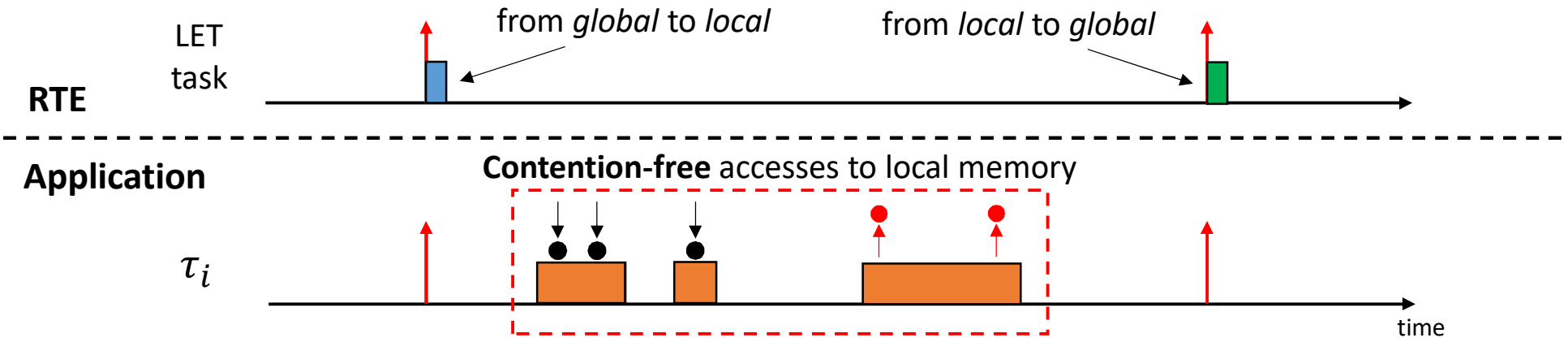
- One task running at the *highest priority* in each core to implement LET communication
- Access to shared memory is regulated by lightweight *spin-based synchronization*



LET and AUTOSAR RTE

- The tasks implementing the LET communication can be *automatically generated* as part of the AUTOSAR RTE
- Our approach is prone for being implemented in a model-based design flow

- Integration within **AUTOSAR**
  - RTE takes care of *mirroring* **local** copies according to the LET paradigm
  - RTE offers an API to access the local copies
  - Local copies are accessed with *explicit communication*



IMPLEMENTATION

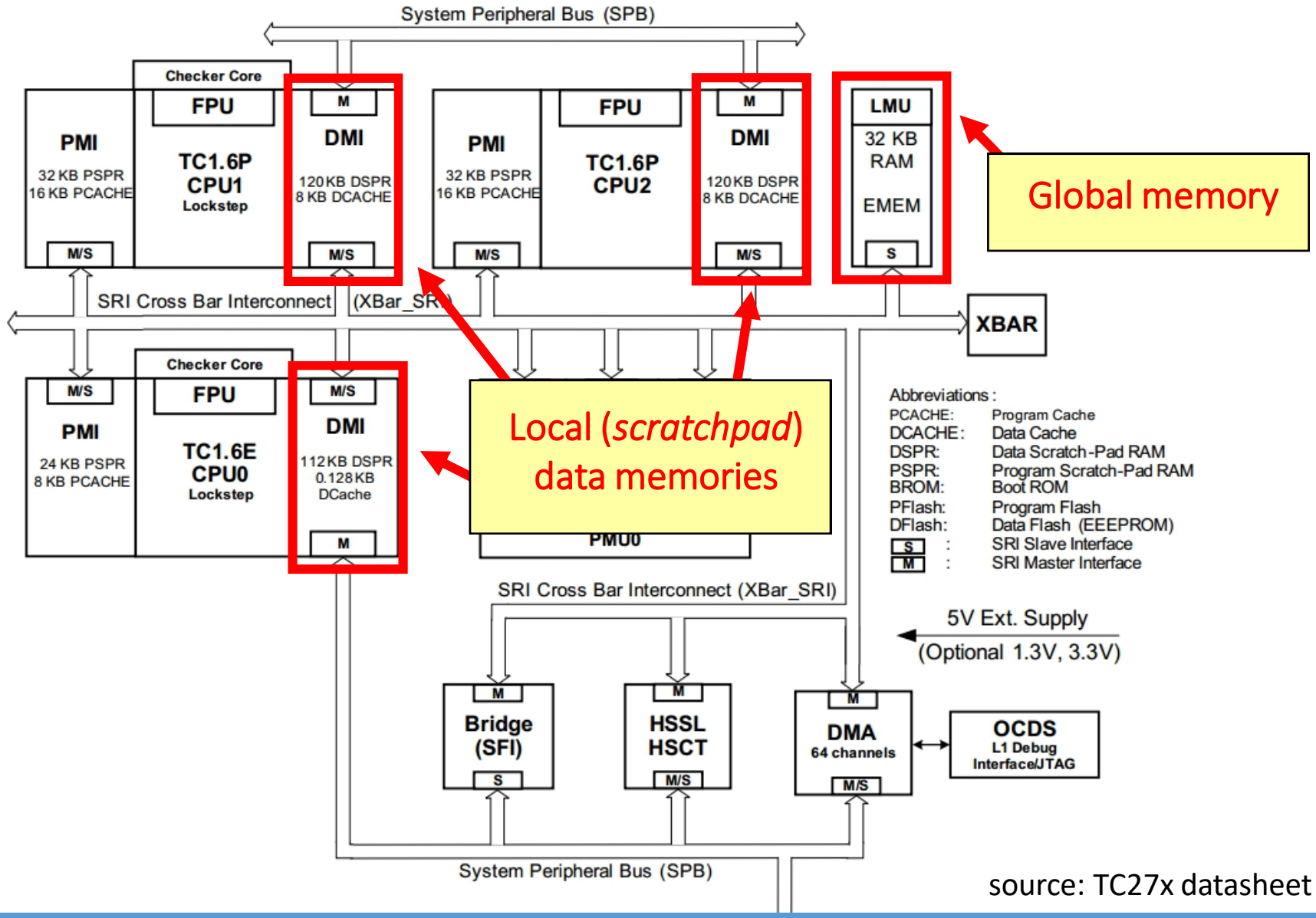
Implementation

- **Reference platform**: Infineon Aurix TC275
  - Asymmetric Tricore with scratchpads
  - Widely adopted by the automotive industry
  - Can be configured to match the abstract model introduced before

- **RTOS**: Implementation based on ERIKA Enterprise v2
  - OSEK certified
  - De-facto representative of the typical behavior of AUTOSAR OSes
  - Open-source

ERIKA ENTERPRISE

Aurix TC27x

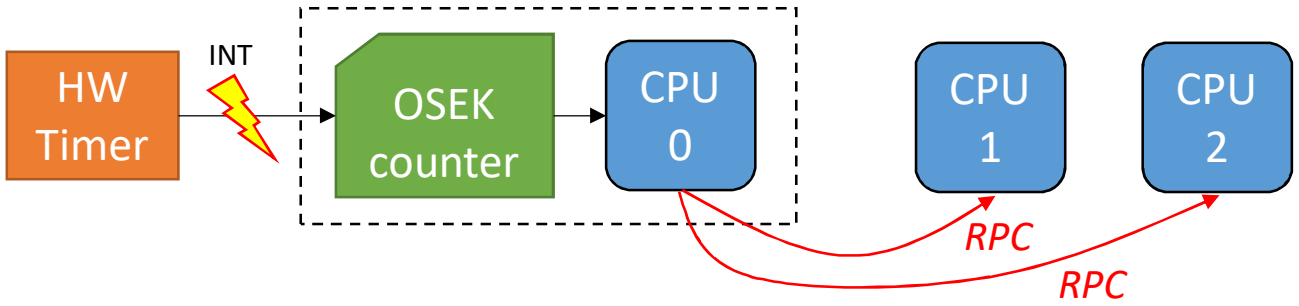


source: TC27x datasheet

Implementation (1)

The implementation required facing with three major issues:

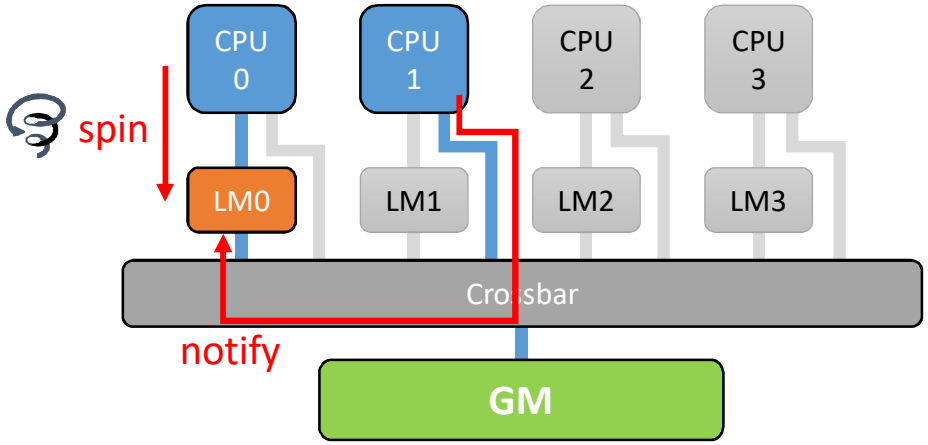
1. Synchronization of task activations across cores
   - Solved using *remote procedure call* (**RPC**) features available in ERIKA
   - Single timer connected to an *OSEK counter* handled in CPU #0
   - CPU #0 uses RPC to activate the tasks in all the cores by means of *OSEK alarms* (inter-core interrupts are leveraged)



Implementation (2)

The implementation required facing with three major issues:

2. Inter-core synchronization to access global memory
   - Similar strategy as for *Mellor-Crummey & Scott* locks
   - Spin variables allocated to local scratchpads
   - Each core can directly access all local scratchpads, hence making notification of a spinning core easy (*baton passing*)
   - Need to pay attention to achieve sequential consistency (barriers with `DSYNC`)

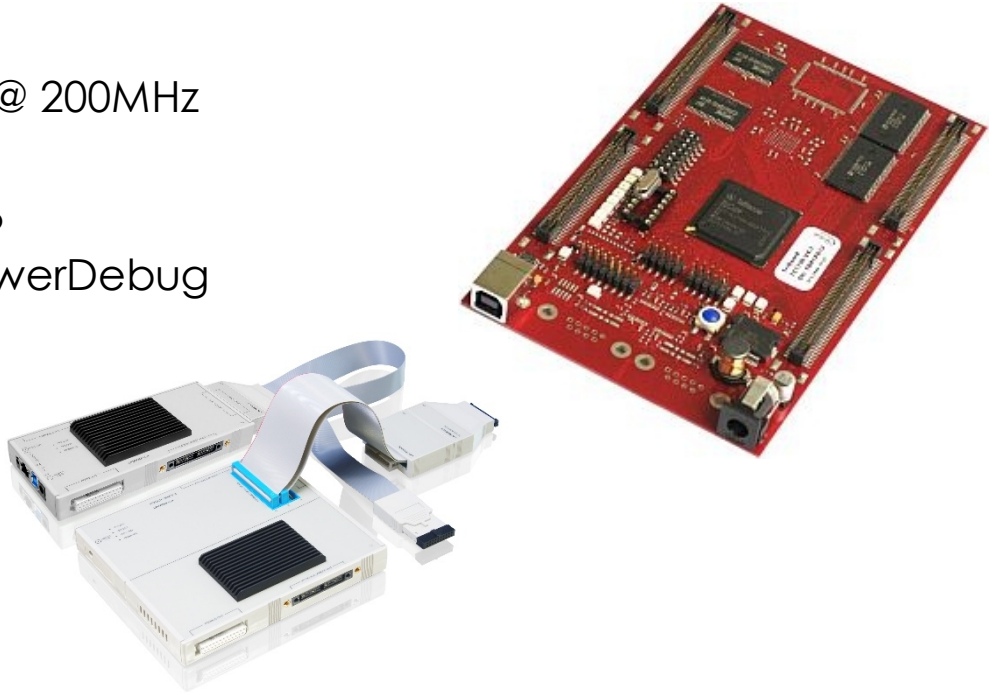



```
1:  procedure LET_TASK_P_X( )
2:      do_write_tick()
3:      busy_wait( spin_P_x_write == 0 )
4:      spin_P_x_write = 0
5:      do_write()
6:      notify_next_processor_write()
7:
8:      do_read_tick()
9:      busy_wait( spin_P_x_read == 0 )
10:     spin_P_x_read = 0
11:     do_read()
12:     notify_next_processor_read()
13: end procedure
```

Case Study

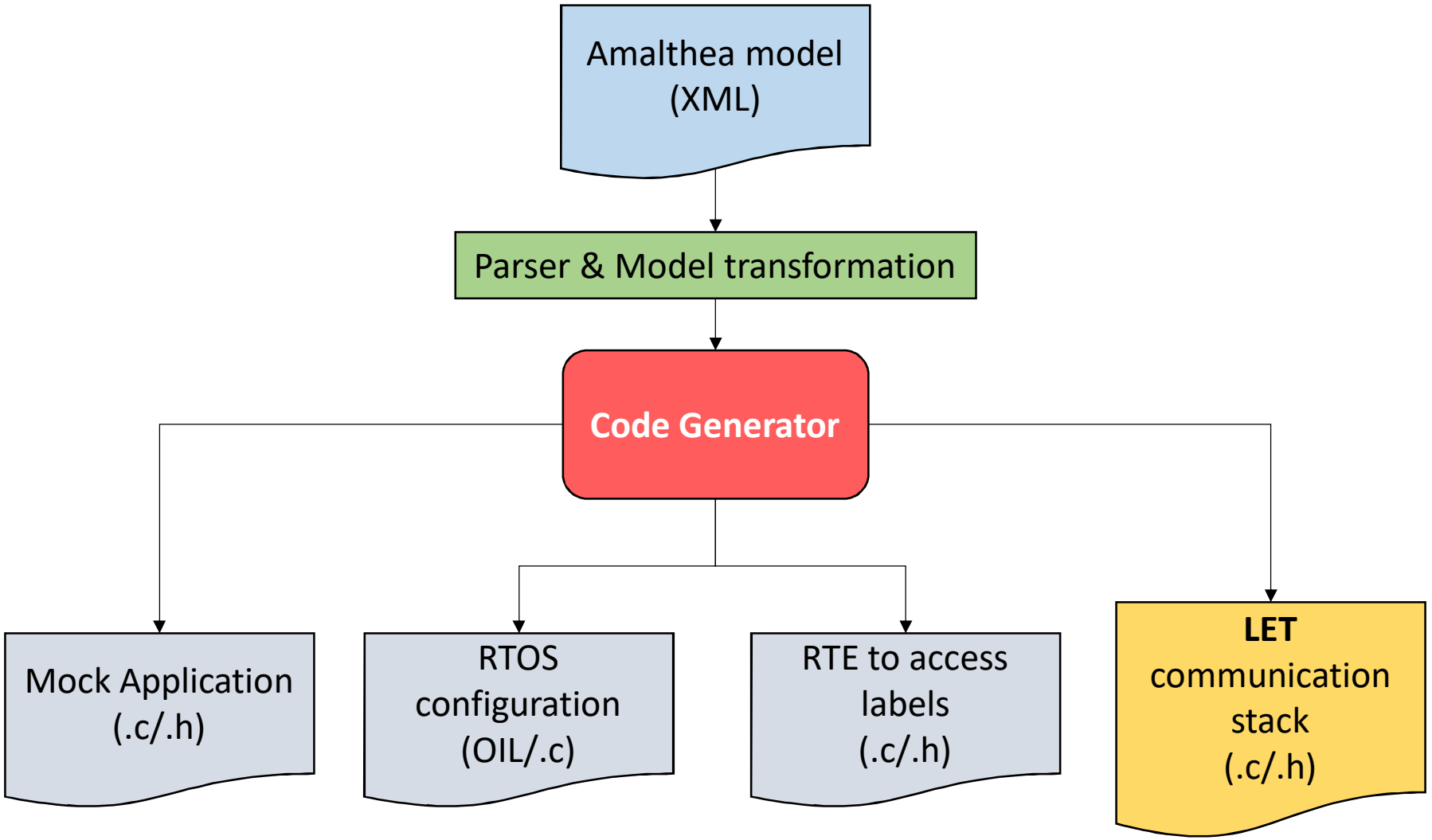
- Implementation tested with a case study
- Mock application generated from the model provided by Bosch for the *WATERS 2017 challenge* – representative of an engine control application
  - ~**20** tasks partitioned into the three cores of the TC275
  - ~**5000** labels (atomic variables) used by the tasks to communicate

- Experimental setup
  - Infineon TriBoard v2 with TC275 @ 200MHz
  - ERIKA Enterprise v2.7
  - HIGHTECH Aurix C compiler v4.6
  - Lauterbach PowerTrace-II & PowerDebug



Code Generation

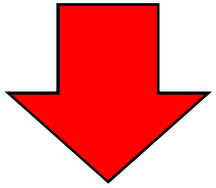
WATERS 2017 Challenge
provided by Bosch



Experimental Results

- The adoption of the LET paradigm significantly increase time determinism –- it's an additional system feature!
- From a scheduling (timing) perspective, our realization faces with two conflicting trends

Worst-case delays due to memory contention are reduced. Pessimism is removed by design and schedulability analysis is simplified.



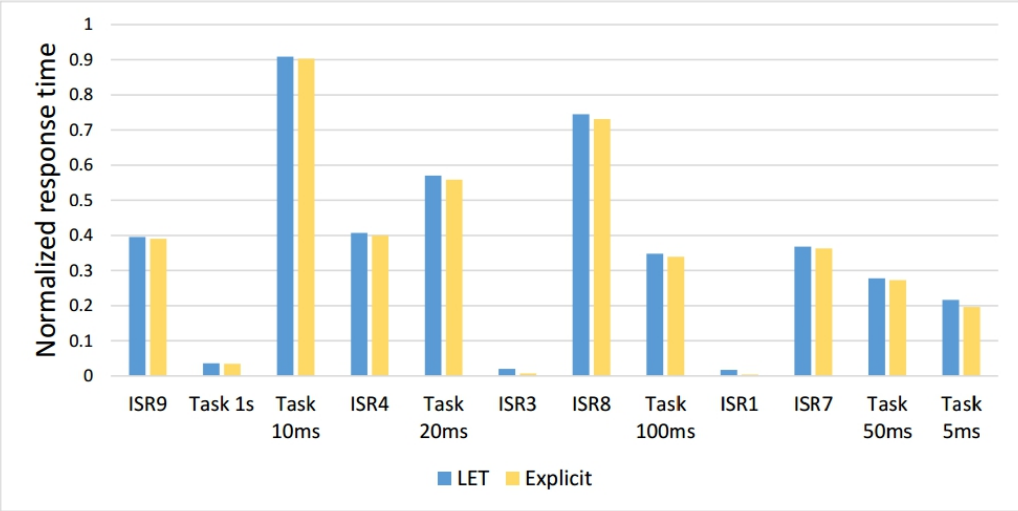
High priority workload is required to perform LET communications, which may harm latency-sensitive tasks (priority inversion).

Experimental Results

- What's the impact of the proposed approach in terms of run-time *overhead* and memory *footprint*?
- Despite the benefits in controlling *memory contention*, is the priority inversion introduced by LET communication harmful?

Exec time first frame of LET tasks (most expensive)

core	net execution time [μs]
1	3.8
2	108.76
3	148.2



Footprint (in bytes)

	text	data	bss
LET	393064	4904	88328
Explicit	359872	4784	80752

+7.5% (can be lower for a real appplication)
Mostly due local copies of labels and code of LET communication

Conclusions

- Presented a scheme for *practical* implementation and analysis of LET communication for **multicore** systems
- LET taken as an opportunity to *control memory contention*
- **Implemented** upon ERIKAv2 on Infineon Tricore TC275 and tested with a case study based on an application model by Bosch

- **Take-away messages**
  - Impact on run-time overhead has been found negligible
  - The only concern may be the increase of footprint
  - There are a lot of open problems and possible improvements

- **Future works**
  - Ad-hoc schedulability analysis under the proposed scheme
  - Holistic synthesis methodology that optimizes label placement, the generation of the LET communication stack, # of buffers, and possibly the runnable placement

Thank you!

Alessandro Biondi
alessandro.biondi@sssup.it

Do you want to know more about this work?
Check it out our RTAS2018 paper!

Achieving Predictable Multicore Execution of Automotive Applications Using the LET Paradigm

Alessandro Biondi and Marco Di Natale
Scuola Superiore Sant'Anna, Pisa, Italy
E-mail: {alessandro.biondi, marco.dinatale}@ssup.it

***Abstract*—Next generation automotive applications require support for safe, predictable, and deterministic execution. The Logical Execution Time (LET) model has been introduced to improve the predictability and correctness of time-critical applications. The advent of multicore architectures, together with the need to ensure time predictability despite the complex memory hierarchy and the hardware resources shared by the cores, is an additional motivation for the use of the LET paradigm in conjunction with a suitable scheduling and memory access model. In this paper, we show how an implementation of the LET model on actual multicore platforms for automotive systems brings the potential to improve time determinism at the price of a modicum run-time overhead. Multiple implementation options are discussed using the automotive AUTOSAR model and operating system standard, and a realistic application defined by Bosch for the 2017 WATERS challenge. Experimental data of executions on the Infineon Aurix platform show the feasibility of the proposed approach. The paper also provides a discussion on further implementation optimizations and other issues related to the general problem of memory-aware analysis of automotive applications on multicores.**

I. Introduction

The introduction of safety-critical functions in automotive systems, together with the advent of multicore platforms, brings the need to rethink the development and execution paradigms for embedded functionality. Developers need high levels of predictability, testability, and ultimately determinism in the execution of their code. The LET model was introduced as part of the GIOTTO framework [1] to eliminate output jitter and provide time determinism in the code implementation of controls. Recently, there has been a renewed interest in the LET execution paradigm by automotive electronics vendors, as witnessed by the recent WATERS challenges [2].

In essence, the LET delays the program output of a task (or any function executed inside the task) at the end of the task period, trading delay for output jitter. The LET model is also characterized by an execution model of functional units with execution order (causality) constraints. The adoption of this model brings to the foreground not only the concept of timeliness, but also of causality, which is typical of synchronous languages and their implementations.

A key observation is that the LET execution model not only avoids output jitter but has the additional benefit of scheduling precisely in time the accesses to the communications variables. This can be extremely valuable in the multicore execution of tasks communicating remotely. Several techniques have been proposed to analyze the time performance of real-time tasks on multicores in the face of the sharing of memory and other hardware resources, including interconnects, arbiters and I/O devices. Unfortunately, COTS multicore platforms are not designed with the aim of providing predictability, with the consequence that conventional analysis techniques can be at best pessimistic. The LET execution model can improve and restore predictability by controlling the time when memory resources are accessed.

For modern automotive systems, the AUTOSAR standard [3] provides a reference model for the development of applications, including a model of the functions and the tasks, a standard API for communication and execution, and a standard platform architecture. In AUTOSAR, the application consists of a set of communicating runnables grouped into tasks and statically allocated and scheduled on the system cores. The AUTOSAR model is based on the concept that the task model and the communication implementation are automatically generated by dedicated tools based on configuration information, the model of the application, and platform constraints. Such aspects are of paramount importance when designing a LET implementation for automotive applications.

This paper. In this paper, we draw analogies from all these concepts and propose an integrated approach to face the problem of implementing and scheduling task communications in multicores. We first provide a characterization of possible variants of the LET paradigm. Next, we discuss the implementation of the LET paradigm in agreement with the AUTOSAR model and API on a multicore platform that is very common in the automotive domain and representative of typical HW configurations: the Infineon Aurix microcontroller. Then, we provide an analysis of possible actual implementation options based on the ERIKA RTOS (compliant with the OSEK automotive standard and a de-facto representative of the typical behavior of AUTOSAR OS kernels). Finally, we provide our results on the evaluation of a code implementation of the application proposed by Bosch in the context of the WATERS 2017 challenge [2], executed with our LET implementation on the Aurix. Other related issues will be shortly discussed but are not the main concern of this work, including the schedulability analysis with explicit consideration of memory contention.

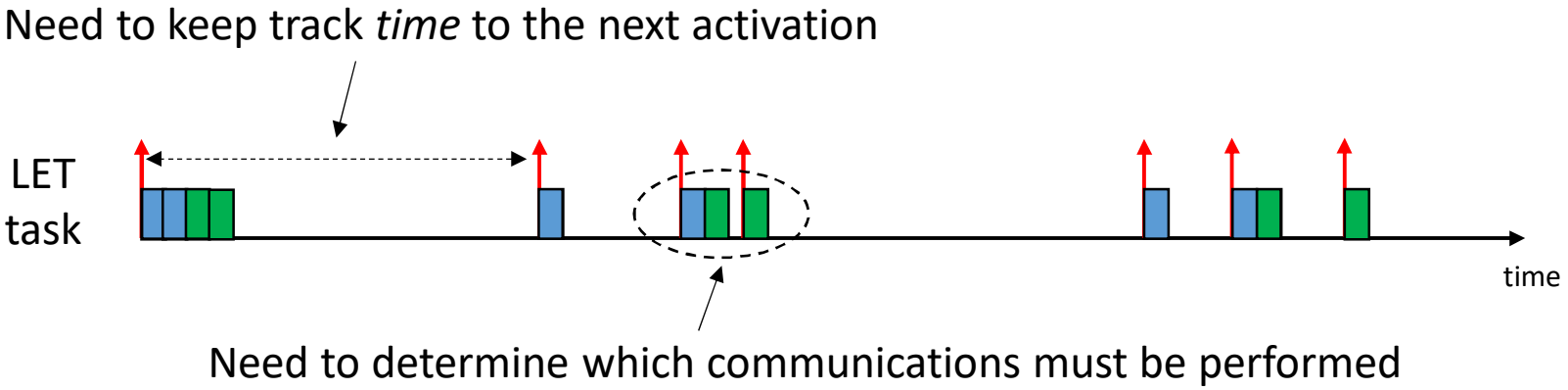
II. Modeling and Background

This paper considers applications composed of a set of n periodic tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$, each characterized by a worst-case execution time (WCET) C_i, a period T_i, and a relative deadline $D_i \leq T_i$. A bound on the response time of τ_i is denoted by R_i. The tasks execute upon a platform that comprises m processors $P_1, \ldots, P_m$, with local memories $M_1, \ldots, M_m$ (one for each core), and a global memory M_{m+1}. The platform disposes of a crossbar switch that enables point-to-point communication between each core and each memory. Concurrent accesses to memories are arbitrated with a FIFO policy. Blocking memory access is assumed, i.e., no write or read buffers. Tasks are scheduled according to *partitioned fixed-priority* scheduling, and $hp(i)$ denotes the set of tasks with higher priority than τ_i. Each task is statically allocated to a

Implementation (2)

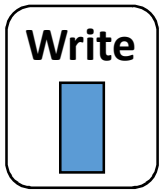
The implementation required facing with three major issues:

2. Realization of GMF tasks
   - Memory vs. time trade-off
   - Possible approach: scheduling table
     - Potentially needs to store information up to the hyper-period
     - It would introduce a lot of duplicate information
   - **Hint**: We are dealing with specific instances of GMF tasks!
     - Leveraging some analytical properties of LET timing, GMF tasks can be implemented with counters for each pair of communicating tasks

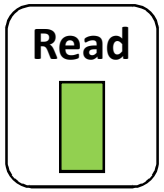


LET Tasks: Synchronization

- One GMF Task running at the *highest priority* in each core to implement LET communication
- Access to shared memory is regulated by lightweight *spin-based synchronization*



Update shared copy of data (copy *from* local memory *to* global memory)



Read shared copy of data (copy *from* global memory *to* local memory)

Avoid contention when accessing the global memory (explicit *synchronization*) **removing pessimism** in the analysis

Limited jitter

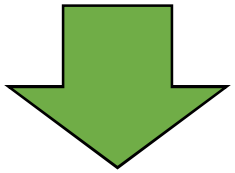
Potential priority inversion due to high-priority communication

OTHER CONTRIBUTIONS

LET semantic options & analysis

Memory-aware RTA

- **Objective**: extend the response-time analysis for partitioned fixed-priority scheduling to explicitly account for delays due to memory contention
- **Analysis design principles**:
  1. Use a simple task model (no execution traces)
     - Contention-free WCET
     - Period and deadline
     - Per-job max. number of accesses to global memory
  2. Do not inflate WCETs but rather account for contention at the stage of response-time analysis [inflation-free analysis, *Brandenburg 2013*]

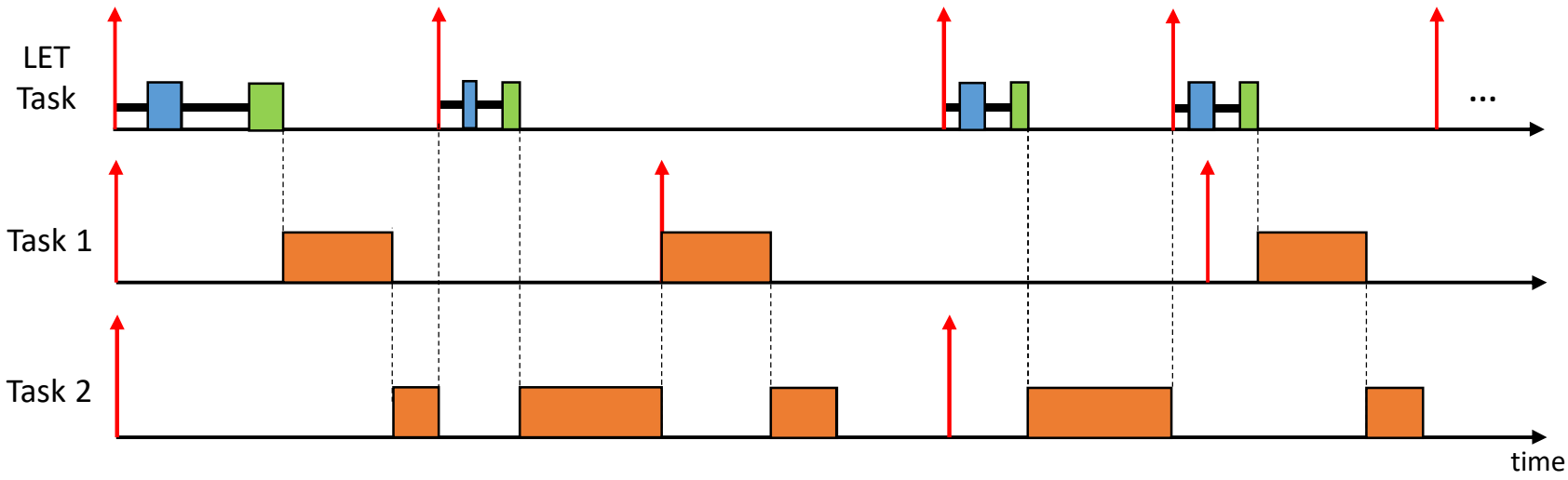


- *Provides a taste of the impact of any-time memory accesses on response times*
- *Still, it is affected by considerable intrinsic pessimism…*

Memory-aware RTA

With the proposed approach the analysis is simplified:

- Standard RTA for partitioned fixed-priority scheduling…
- …plus a high-priority GMF task
- Parameters of the GMF tasks can be derived as a function of
  - Periods of the periodic tasks
  - Labels accessed by each task
  - Configuration of the inter-core synchronization mechanism



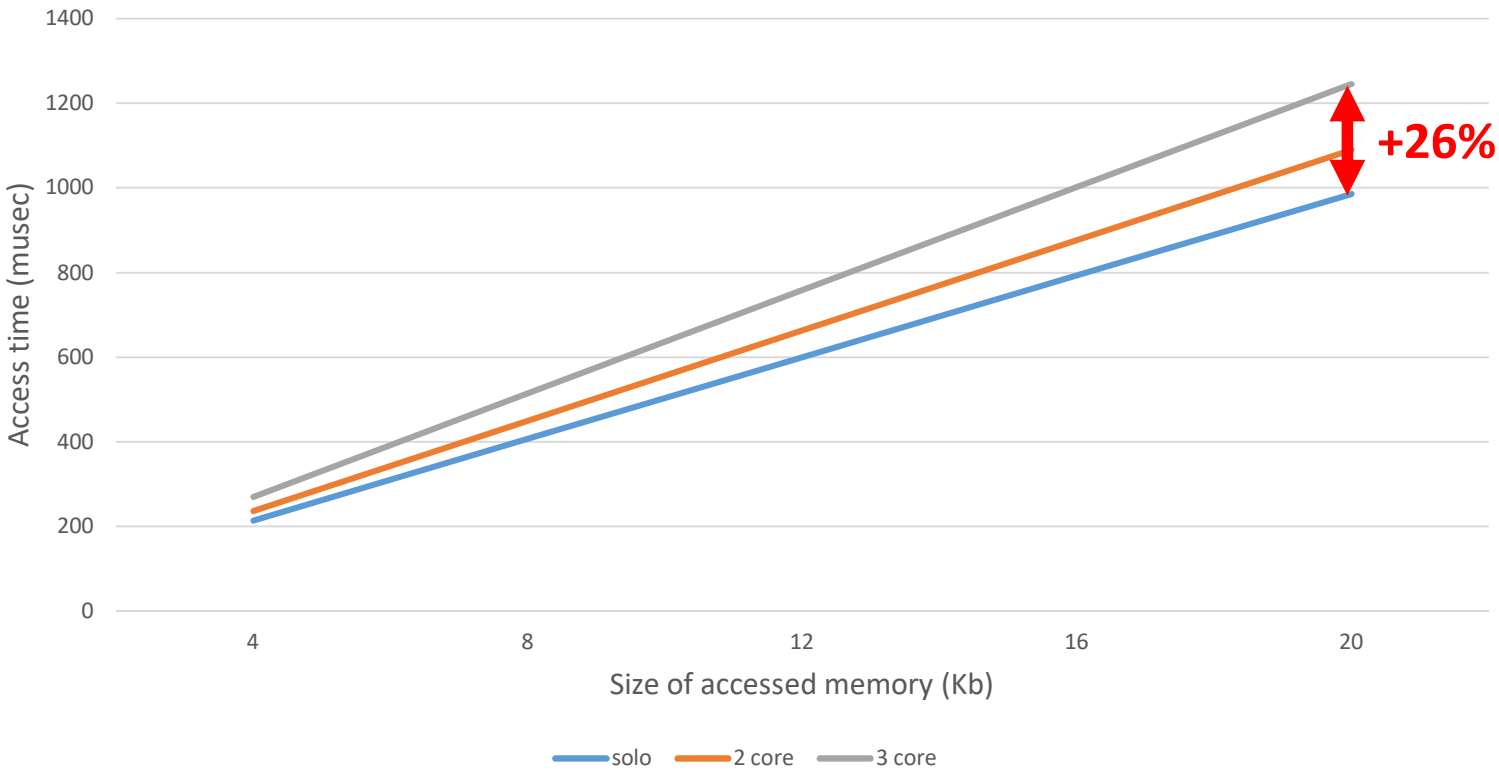
Clarifications on LET Semantics

- **Warning**: different scheduling decisions for LET communications may lead to completely different LET semantics!
- The order with which read and write operations are performed is really important
  - Different orders also lead to different worst-case *end-to-end latencies* in task chains
- A clear formalization of the adopted semantic is needed to avoid misunderstanding when talking about LET

*To shed the light on possible **pitfalls**, the paper also discusses three different LET **semantic options**, focusing on the impact of scheduling decisions on end-to-end latencies*

Memory Contention on TC27x

- Shared buffer of 20Kb allocated in LMU (global memory)
- Core 0 is under analysis and accesses a portion of the buffer
- Cores 1 and 2 are continuously writing into the buffer to generate interference
- The buffer is accessed in a sequential fashion

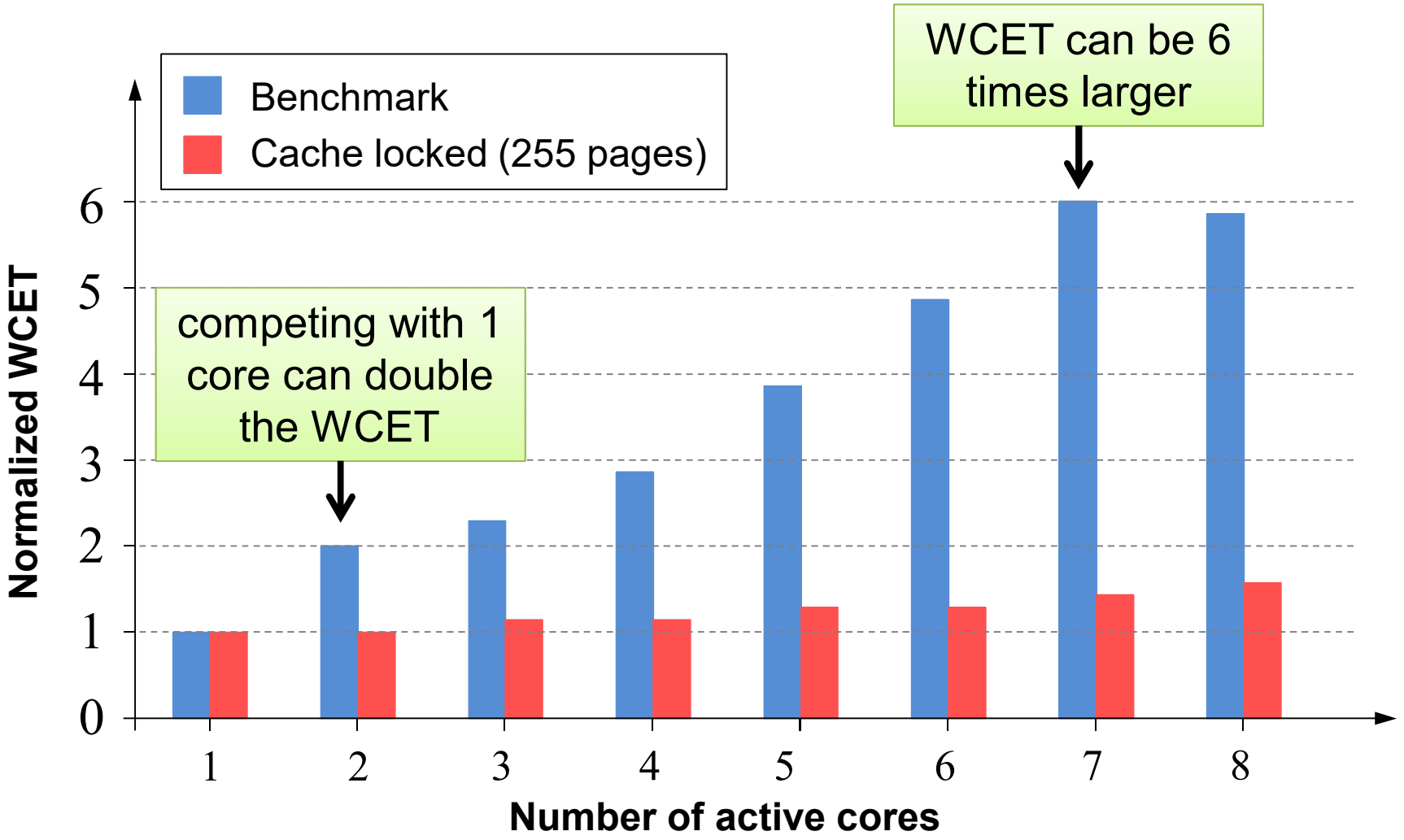


Handling counters

```
1:  procedure DO_WRITE_TICK( )
2:      ⟨...⟩
3:      cnt_write_T6_T8 = cnt_write_T6_T8 -1
4:      if (cnt_write_T6_T8 == 0) then
5:          k_{6,8} = (k_{6,8} + 1) mod k_{6,8}^{max}
6:          cnt_write_T6_T8 = jobs_T6_T8[k_{6,8}] ·T_6/T_1^{LET}
7:          write_flags_T6 | = TURN_ON_FLAG_T6_T8
8:      end if
9:      ⟨...⟩
10: end procedure
```

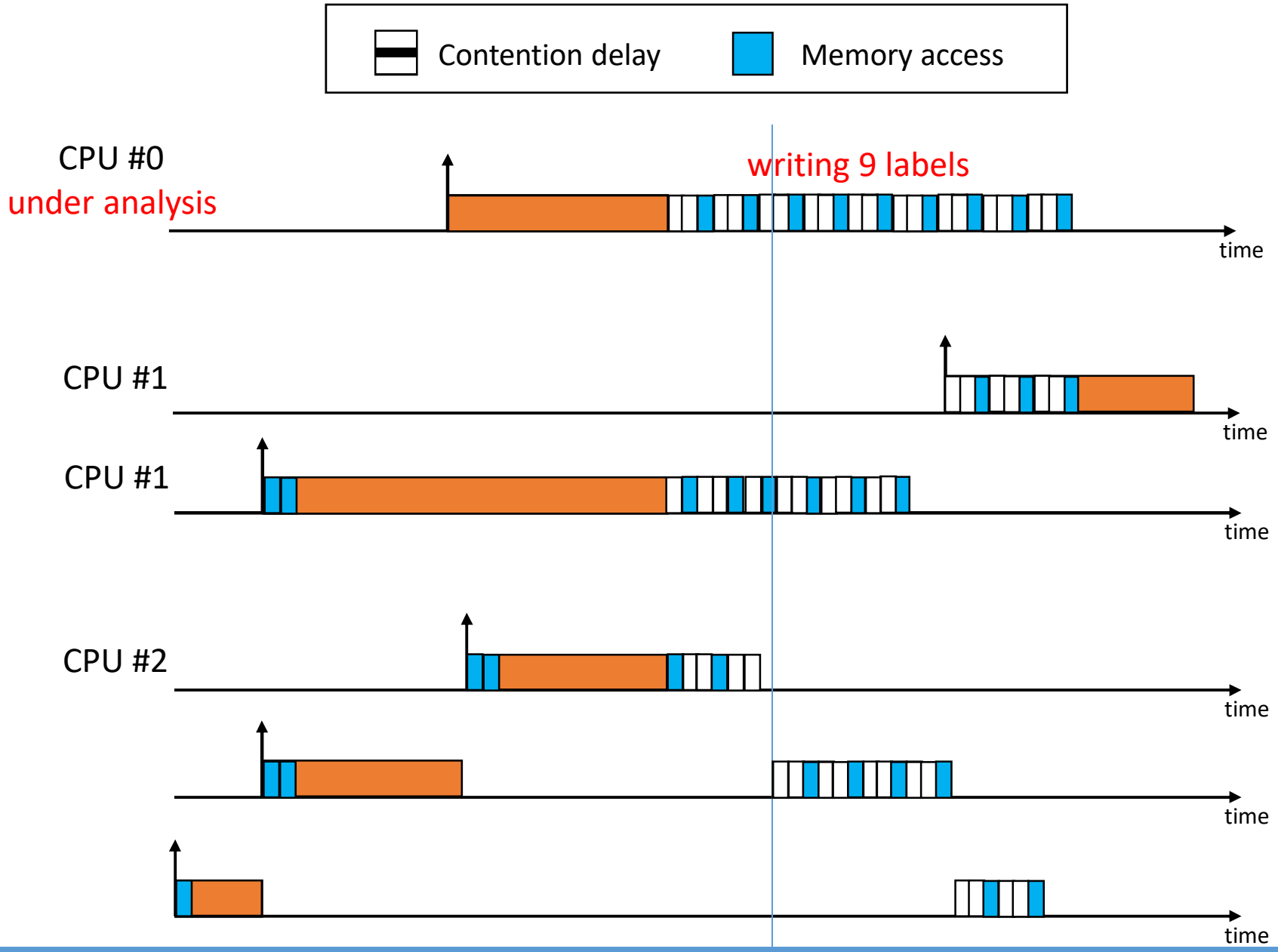
The WCET Issue

Test by Lockheed Martin Space Systems on 8-core platform

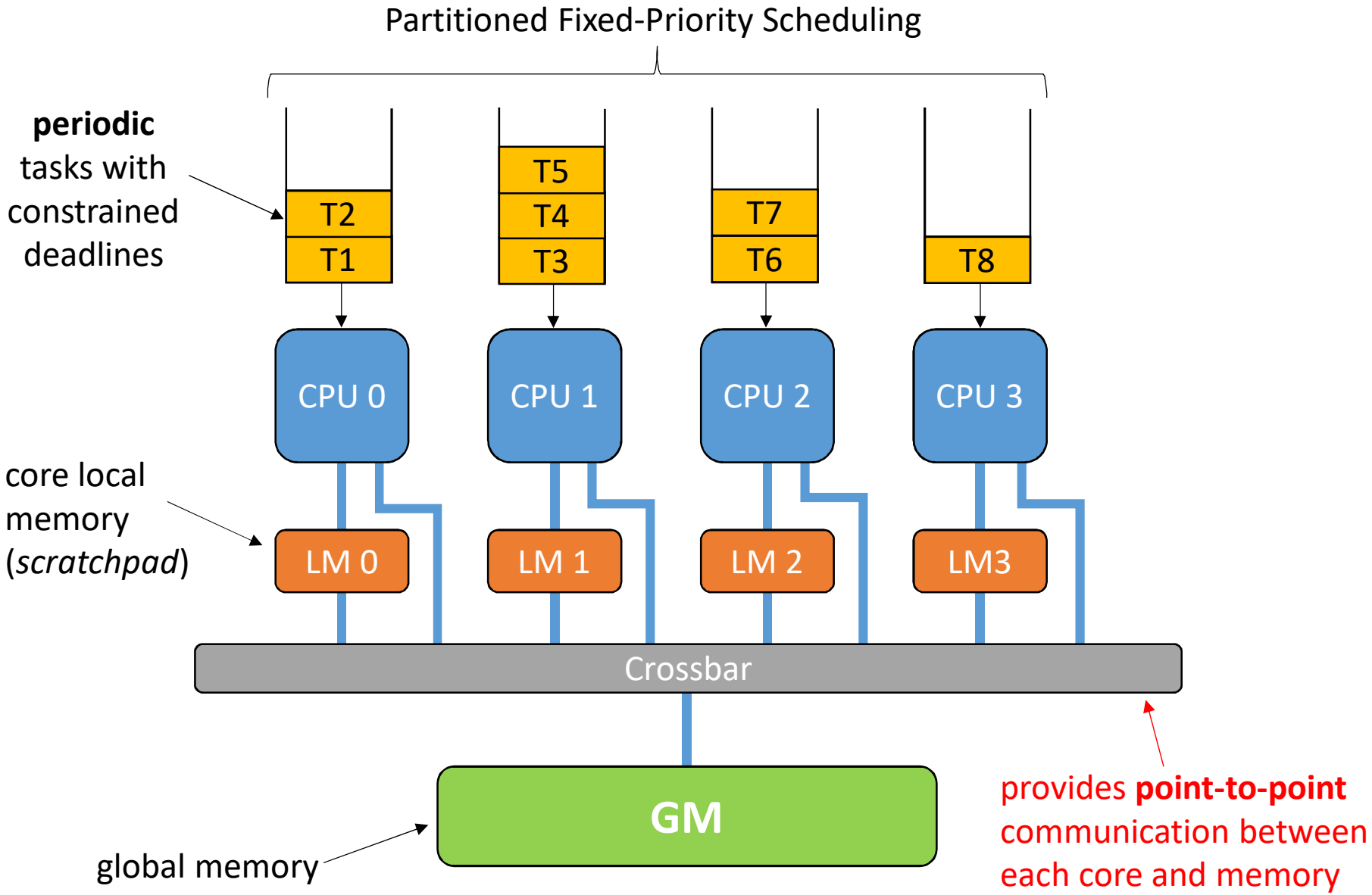


Source: http://rtsl-edge.cs.illinois.edu/SCE/

Memory contention



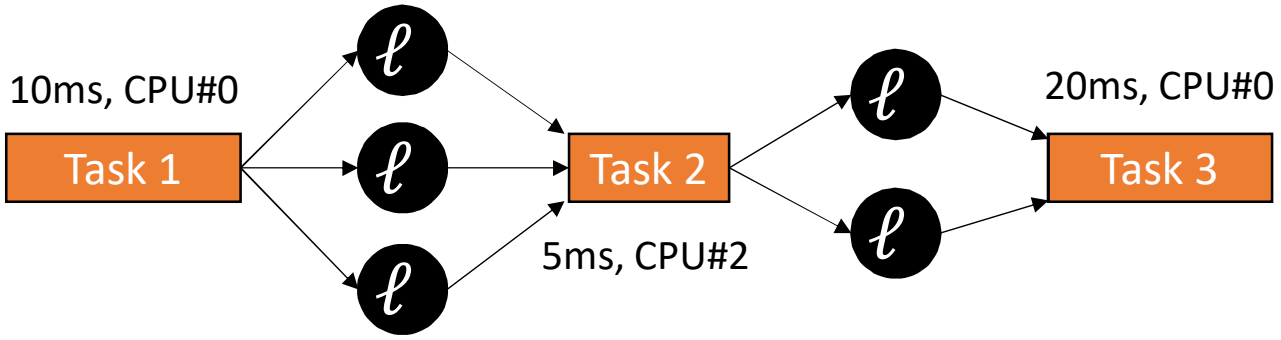
Platform & System Model



Inter-task Communication

Tasks communicate by means of labels, i.e., atomic shared variables (size ≤ processor word)

- Realistic applications include thousands of labels, as witnessed by the 2017 WATERS Challenge data provided by Bosch
- Communications through labels originate causality dependencies and task chains, typically also across *different cores*



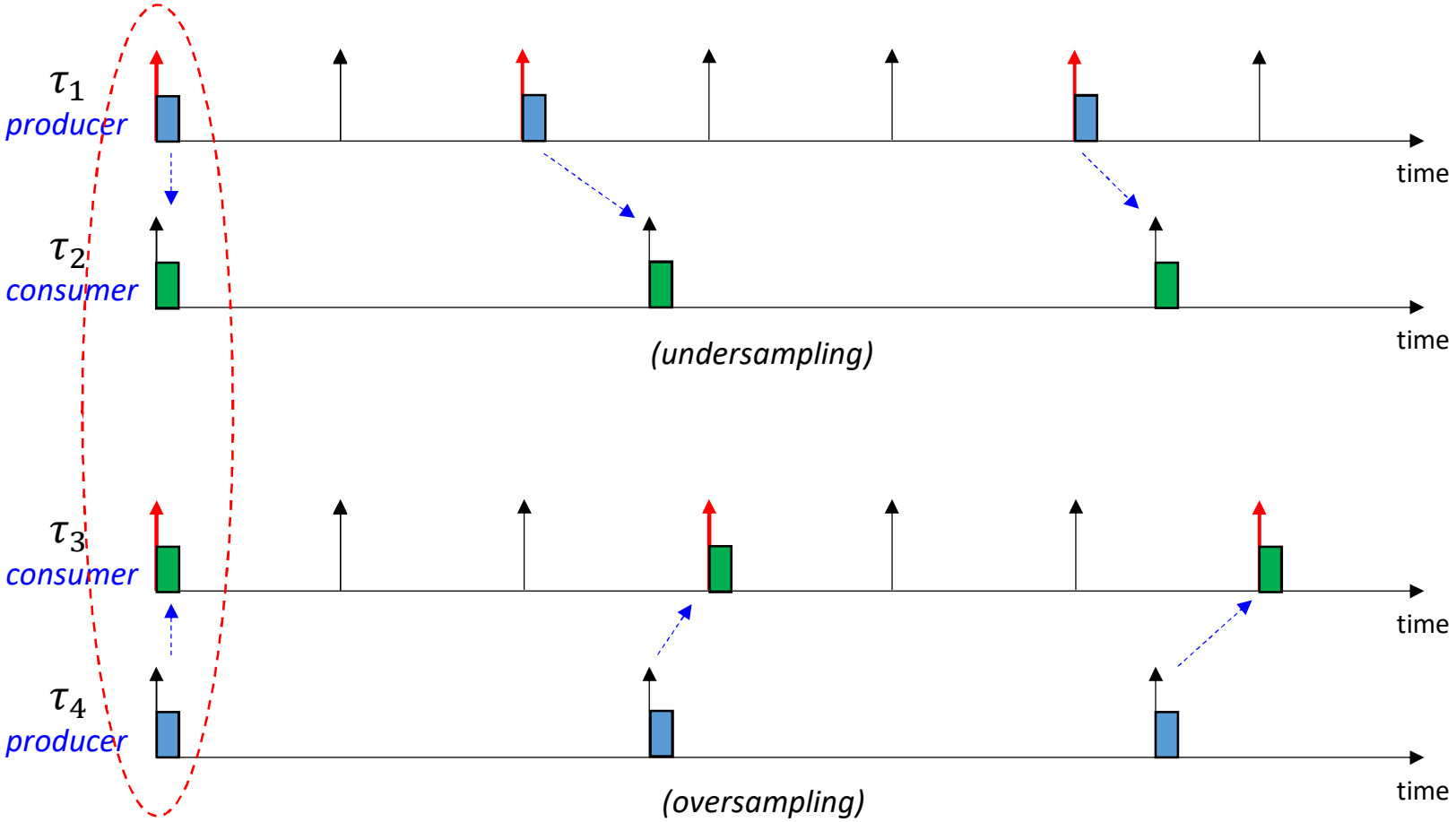
Realizing LET Communication

Understanding & Modeling the **timing** of LET communications

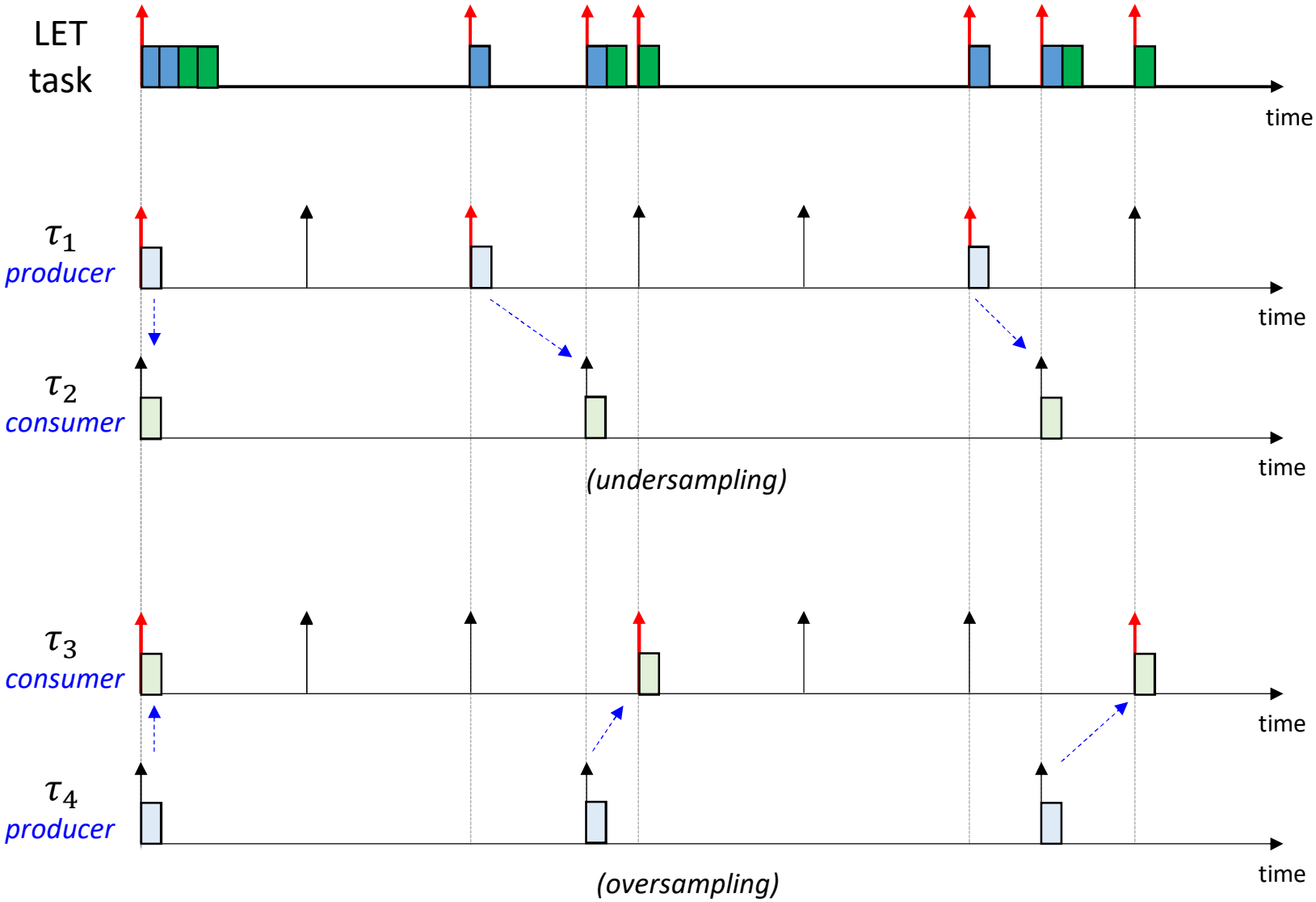
Coordination of LET communications **across cores** (controlling the access to global memory)

LET Timing: Logical View

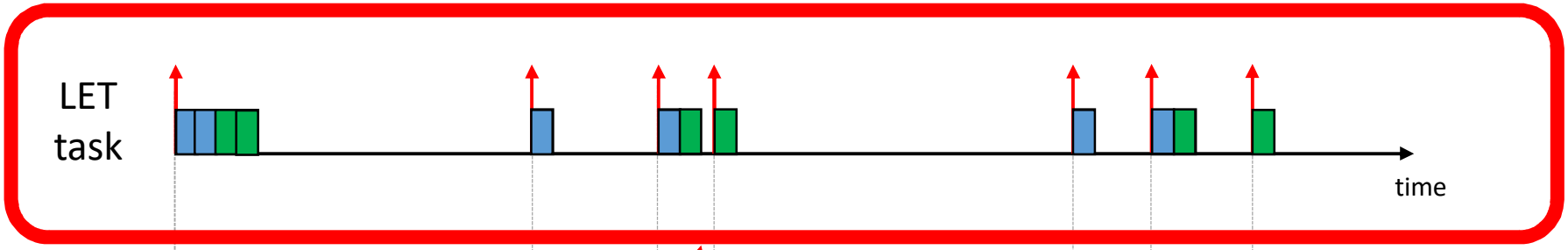
Understanding LET timing: not all reads and writes are actually necessary

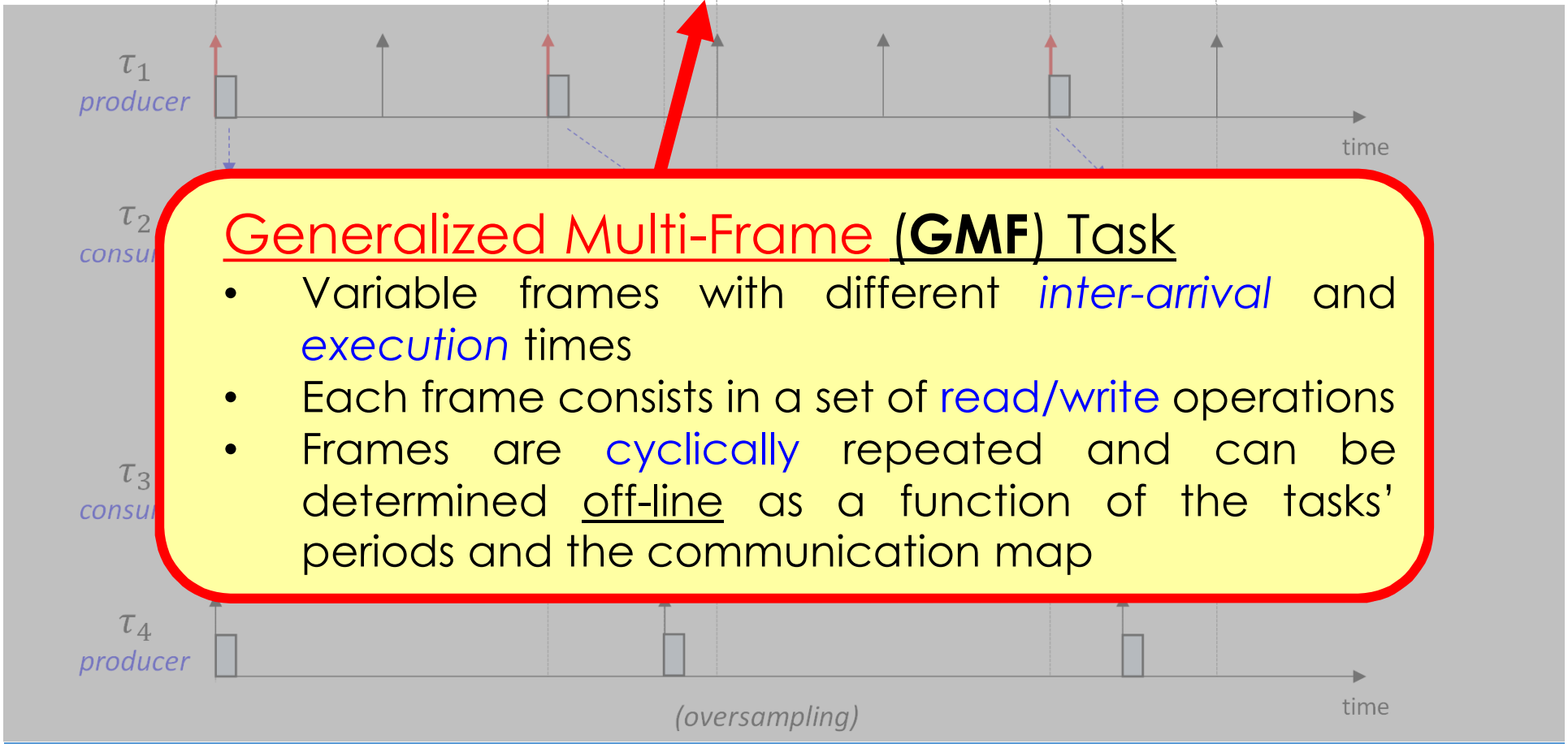


LET Timing: Scheduling



LET Tasks





Generalized Multi-Frame (**GMF**) Task

- Variable frames with different *inter-arrival* and *execution* times
- Each frame consists in a set of read/write operations
- Frames are cyclically repeated and can be determined off-line as a function of the tasks' periods and the communication map