Achieving Predictable Multicore Execution of Automotive Applications Using the LET Paradigm

Alessandro Biondi and Marco Di Natale

Scuola Superiore Sant’Anna, Pisa, Italy
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 (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.
• 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.)
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)
Platform & System Model

Partitioned Fixed-Priority Scheduling

periodic tasks with constrained deadlines

core local memory (scratchpad)

global memory

provides point-to-point communication between each core and memory

A. Biondi – RTAS 2018
Tasks communicate by means of labels, i.e., **atomic shared variables** (size $\leq$ 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**
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.

We like simple models → a *sustainable analysis* have to consider that memory accesses are ordered such that contention is maximized.

Bounds on worst-case response times tend to be large.
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
Realizing LET Communication

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

Coordination of LET communications **across cores** (controlling the access to global memory)
Understanding LET timing: not all reads and writes are actually necessary.

Let's consider the timing of operations in a system where we have producers and consumers. The figure illustrates the timing of operations over time. We see the following:

- **\( \tau_1 \)**: Producer operation.
- **\( \tau_2 \)**: Consumer operation. (Undersampling)
- **\( \tau_3 \)**: Consumer operation. (Oversampling)
- **\( \tau_4 \)**: Producer operation.

The figure shows that not all reads and writes are actually necessary, illustrating the concept of undersampling and oversampling in LET timing.
LET Timing: Scheduling

- **LET task**
- **$\tau_1$** producer
- **$\tau_2$** consumer
- **$\tau_3$** consumer
- **$\tau_4$** producer

*undersampling*

*oversampling*
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
Realizing LET Communication

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

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

![Diagram](image)

- **CPU #0**
- **CPU #1**
- **CPU #2**

- **Write (output)**
- **Read (input)**
- **Busy waiting**
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
LET and AUTOSAR RTE

- The GMF 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*

---

A. Biondi – RTAS 2018
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
Aurix TC27x

Local (scratchpad) data memories

Global memory

source: TC27x datasheet
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)
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

Need to keep track *time* to the next activation

Need to determine which communications must be performed
The implementation required facing with three major issues:

3. **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:   do_read_tick()
8:   busy_wait( spin_P_x_read == 0 )
9:   spin_P_x_read = 0
10:  do_read()
11:  notify_next_processor_read()
12: 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

Amalthea model
(XML)

Parser & Model transformation

Code Generator

Mock Application
(.c/.h)

RTOS configuration
(OIL/.c)

RTE to access labels
(.c/.h)

LET communication stack
(.c/.h)
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?

<table>
<thead>
<tr>
<th>core</th>
<th>net execution time [μs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.8</td>
</tr>
<tr>
<td>2</td>
<td>108.76</td>
</tr>
<tr>
<td>3</td>
<td>148.2</td>
</tr>
</tbody>
</table>

**Footprint (in bytes)**

<table>
<thead>
<tr>
<th></th>
<th>text</th>
<th>data</th>
<th>bss</th>
</tr>
</thead>
<tbody>
<tr>
<td>LET</td>
<td>393064</td>
<td>4904</td>
<td>88328</td>
</tr>
<tr>
<td>Explicit</td>
<td>359872</td>
<td>4784</td>
<td>80752</td>
</tr>
</tbody>
</table>

+7.5% (can be lower for a real application)

Mostly due local copies of labels and code of LET communication
OTHER CONTRIBUTIONS

LET semantic options & analysis
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…
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
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 WATERS 2017 Challenge model

• 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