Optimizing the Functional Deployment on Multicore Platforms with Logical Execution Time

Paolo Pazzaglia, Alessandro Biondi and Marco Di Natale

Scuola Superiore Sant’Anna, Pisa, Italy

paolo.pazzaglia@santannapisa.it

RTSS 2019
Introduction

- Multicore platforms are replacing conventional single-core architectures in many embedded application domains, requesting the developers to find effective ways to partition functions on the cores.

- Several issues in switching to multicores...
  - Lack of appropriate modeling for partitioning applications
  - Causality properties in legacy software are only implicitly verified on single core
  - Achieving timing predictability on multicores is not trivial

- ...plus increasingly stringent legal regulations and certifiability requirements
This work

Problem:
Finding an optimal mapping of real-time applications to a multi-core platform, satisfying the following requirements:

- All timing requirements are guaranteed (all deadlines are satisfied)
- Maintaining a set of functional dependencies
- Providing a suitable response-time analysis
- Flexible and applicable to complex realistic system

Our formulation:
- We leverage the Logical Execution Time paradigm, with multiple synchronization points to communicate data between the cores, enforcing causality and determinism
- The problem is coded as an optimization problem with linear constraints (MILP). Timing and functional requirements are coded as constraints for the placement of functions and related variables
The real-time application to be mapped has both periodic and sporadic tasks $\Gamma_i$, scheduled by fixed-priority with implicit hard deadlines.

Each task $\Gamma_i$ consists in a sequence of **runnables** (functions) that read and write data in **labels** (variables), compliant to the **AUTOSAR** standard.

Each runnable $r_i$ is associated with a WCET $e_i$.
• A **message** represents a communication between a producer runnable and a consumer runnable, via a **shared** label.

• **Intra-task messages** (i.e., when the runnables involved in the message belong to the same task) have specific timing properties:

  - Immediate message
  - Delayed message
  - Loop message

• These timing properties are mapped to **precedence constraints** between runnables that must be enforced in the final mapping.
Our target multicore platform has $N_P$ identical cores, each with a local memory, plus a (possibly larger) global memory.

A crossbar enables point-to-point communication. Accessing memory $M_j$ from core $P_i$ has a (timing) cost $\lambda_{i,j}$.

If multiple processors access the same memory, contention may happen! This must be properly managed...
Logical Execution Time paradigm

• We adopt the **Logical Execution Time** (LET) paradigm: all communications are performed at specific points in time

With LET we trade a variable output jitter for a **fixed (longer) end-to-end latency**. This helps achieve timing determinism.

• LET also provides deterministic end-to-end latencies of chains.
Realizing LET Communication

- The local copies of data are allocated in the local memories (LM), while the shared copies are allocated in the global memory (GM).
- The LET communication stack moves data from global to local memories and vice versa.

Preemptable task execution with only contention-free accesses to local memory.
Multicore LET design

Proposed implementation:

- Sporadic tasks are mapped “as a whole” in a dedicated subset of cores
- For the other cores, a container task $\Gamma^p_i$ is defined for each periodic task $\Gamma_i$ of the starting application. Each container task will “contain” (some of) the runnables of the corresponding initial task $\Gamma_i$
- The activation of all periodic tasks is synchronized between cores
- One LET communication task $\Gamma^p_L$ is defined in each core with highest priority and with the only purpose of copying variables involved in LET communications
Multicore LET design

- We use a generalization of LET paradigm where communication can be performed at multiple synchronization points, common to all cores.

- One instance of the LET communication task $\Gamma^p_L$ is activated at each synchronization point, copying variables involved in LET communication.
Multicore LET design

- We use a generalization of LET paradigm where communication can be performed at multiple synchronization points, common to all cores.

![Diagram showing Core #1 and Core #2 with synchronization points and time spans labeled as $T_1$, $T_2$, and $T_3$.]

- The time span between a pair of consecutive synchronization points is called LET interval.
- Each runnable must then be mapped in a specific LET interval.
Multicore LET design

- LET tasks of different cores may incur in contention when simultaneously accessing the global memory. To address this issue, we adopt a *baton-passing protocol with busy waiting*.

- A priority is assigned to each core. At each synchronization point, LET reads start only when all cores finished their LET writes.

- Enforcing the write-read order guarantees causality following the original LET semantic.

1 A. Biondi and M. D. Natale, “Achieving predictable multicore execution of automotive applications using the LET paradigm,” *RTAS 2018*
Optimal code partitioning under LET

- The allocation of runnables to the LET intervals must satisfy the **precedence constraints** extracted from all messages.
- We define **mapping rules for runnables** to guarantee these dependencies.

**Example:** direct message

Associated precedence rule

\[ r1 \prec r2 \]

When mapping in different cores, communication is realized with LET.

**Selection of possible mappings**

- (a) \( \Gamma_i^1 \):
  - \( r2 \)

- (b) \( \Gamma_i^1 \):
  - \( r1 \) \quad \( r2 \)

- (c) \( \Gamma_i^1 \) and \( \Gamma_i^2 \):
  - \( r1 \) \quad \( r2 \)
Optimal code partitioning under LET

- Similarly, we also define rules for mapping labels in memories.
- Read-only, write-only, and loop labels are mapped in the local memory of the core in which the corresponding runnable is allocated to.
- Shared labels are mapped in global memory if involved in LET communication, and require additional local copies.

The mapping rules guarantee exclusive and contention-free access to LMs:
- Runnables mapped in $P_i$ access only labels in $M_i$.
- The LET communication task $\Gamma^i_L$ accesses only $M_i$ and $GM$.

- Additionally, some LET communications involving different tasks may be skipped in case of over- or under-sampling.
Response-time analysis

- At each synchronization point, an instance of the LET communication task may be invoked. In order to achieve data consistency, the execution of all runnables mapped to the preceding LET interval must complete before the synchronization point.

Each LET interval of a container task $\Gamma^p_i$ is called child task $\tau^p_{i,k}$, and can be treated as a periodic task with period $T_i$, WCET $C_{i,k}$, offset $\phi_{i,k}$, and deadline $D_{i,k}$ that corresponds to the synchronization point.

- The body of each child task $\tau^p_{i,k}$ is composed of the subset of runnables mapped in the corresponding $k$-th LET interval of $\Gamma^p_i$.
- The actual WCET of $\tau^p_{i,k}$ is the sum of the WCETs of the runnables, plus the cost of all the accesses to the local memory for the corresponding labels.
Response-time analysis

• Each container task is modeled as a special case of transactional task\(^2\), i.e., children tasks can be considered tasks grouped in a periodic transaction

• The interference on a child task is composed of:
  – The execution of higher-priority container tasks
  – The communication and busy-waiting of the LET task at the synch points

\(^2\) J. C. Palencia and M. G. Harbour, “Schedulability analysis for tasks with static and dynamic offsets” RTSS 1998
Response-time analysis

• We identify the **worst-case interference** generated by an arbitrary container task on a child task, considering both execution and LET communication of other container tasks
  
  – *Please see the paper for more details about the response time analysis*

• The multicore deployment is **feasible** if $R_{i,k}^p \leq D_{i,k}$ for each child task $\tau_{i,k}^p$
MILP formulation

- The design problem leverages a *mixed-integer linear programming* (MILP) formulation. The response-time analysis and the mapping rules are converted into a set of linear constraints, with a linear objective function.
- **Objective function**: minimizing the maximum ratio $R_{i,k}^p / D_{i,k}$ of any child task $\tau_{i,k}^p$.
- We leverage approximations to linearize the response time analysis:
  - Considering a fixed number of equally-spaced synchronization points
  - **Approximate schedulability test** (limited number of checkpoints) $^3$

We check schedulability conditions only at **last activation instants** of interfering tasks. We obtain a **sufficient test** but with **high accuracy** (precision loss of $\sim 1\%$). This approach can be generalized to other task models (see ref: $^3$)

- Mapping rules and response-time analysis use ad-hoc Boolean variables
  - *Please see the paper for a more detailed description*

---

$^3$ Paolo Pazzaglia, Alessandro Biondi, and Marco Di Natale ”Simple and General Methods for Fixed-Priority Schedulability in Optimization Problems”, DATE 2019
Case study: WATERS challenge 2017

- The **FMTV Challenge 2017** provides a realistic engine control architecture
  - 4 CPUs with local memories + GM
  - 21 tasks (10 periodic and 11 ISRs)
  - 1250 runnables
  - 10000 labels
  - Thousands of functional dependencies!

- **Goal**: optimize runnables and labels allocation of periodic tasks
- All ISR tasks are statically mapped in core #1, while the other cores are reserved for periodic tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>$T_i$ ($\mu$s)</th>
<th># Runnab.</th>
<th># Accessed $\ell$</th>
<th>WCET</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma_1$</td>
<td>1000</td>
<td>42</td>
<td>293</td>
<td>764 $\mu$s</td>
<td># 2</td>
</tr>
<tr>
<td>$\Gamma_2$</td>
<td>6660</td>
<td>147</td>
<td>2055</td>
<td>3805 $\mu$s</td>
<td># 2</td>
</tr>
<tr>
<td>$\Gamma_3$</td>
<td>2000</td>
<td>28</td>
<td>133</td>
<td>404 $\mu$s</td>
<td># 3</td>
</tr>
<tr>
<td>$\Gamma_4$</td>
<td>5000</td>
<td>23</td>
<td>122</td>
<td>931 $\mu$s</td>
<td># 3</td>
</tr>
<tr>
<td>$\Gamma_5$</td>
<td>10000</td>
<td>304</td>
<td>4869</td>
<td>11712 $\mu$s</td>
<td># 4</td>
</tr>
<tr>
<td>$\Gamma_6$</td>
<td>20000</td>
<td>307</td>
<td>2894</td>
<td>10468 $\mu$s</td>
<td># 3</td>
</tr>
<tr>
<td>$\Gamma_7$</td>
<td>50000</td>
<td>46</td>
<td>571</td>
<td>3084 $\mu$s</td>
<td># 3</td>
</tr>
<tr>
<td>$\Gamma_8$</td>
<td>$10^5$</td>
<td>247</td>
<td>3001</td>
<td>9418 $\mu$s</td>
<td># 3</td>
</tr>
<tr>
<td>$\Gamma_9$</td>
<td>$2 \cdot 10^5$</td>
<td>15</td>
<td>418</td>
<td>138 $\mu$s</td>
<td># 3</td>
</tr>
<tr>
<td>$\Gamma_{10}$</td>
<td>$10^6$</td>
<td>44</td>
<td>631</td>
<td>137 $\mu$s</td>
<td># 3</td>
</tr>
</tbody>
</table>

The initial mapping provided by the challenge has whole tasks mapped in cores and the scheduling is unfeasible with the given WCETs.
Case study: results

- Our MILP optimization is coded in C++ using CPLEX libraries
- We compared the resulting mapping with the one obtained via a genetic algorithm and with the mapping proposed in the FMTV 2017 challenge
- Different tests scaling the WCETs with a scaling factor [0.65, 0.7, 0.75, 0.8]
- Mapping runnables and labels with our approach leveraging LET and multiple synchronization points produces a gain of ~40% w.r.t. original mapping
- Limited pessimism due to the approximated schedulability test is of ~1%

![Comparison of max(R/D)](image)
Thank you!

Contacts:

Paolo Pazzaglia: paolo.pazzaglia@santannapisa.it
Alessandro Biondi: alessandro.biondi@santannapisa.it
Marco Di Natale: marco@santannapisa.it

Check our RTSS 2019 paper for more details!

Optimizing the Functional Deployment on Multicore Platforms with Logical Execution Time

Paolo Pazzaglia, Alessandro Biondi, and Marco Di Natale
Scuola Superiore Sant’Anna
Pisa, Italy
Email: {paolo.pazzaglia, alessandro.biondi, marco}@sssup.it

Abstract—The move to multicore systems requires methods and tools to support the designer in the partitioning of functions among the available cores and the definition of the task model. In this paper we present the formulation of a functional partitioning for real-time systems and we provide an optimization method for an efficient implementation of the Logical Execution Time (LET) paradigm, to enforce causality and determinism in the development of time- and safety-critical applications. A novel schedulability analysis for partitioned tasks executing according to the LET paradigm is also provided. Our methods are applied to the industry-size model of the WATERS challenge and compute solutions that easily outperform the initial solution provided

(deadlines), as well as preserve the correct behavior (causality) of the functional model of the runs. This means that every functional dependency of the original system must be guaranteed in every possible execution of the multicore system.

The proposed solution leverages the LET paradigm and the definition of (multiple) synchronization points to achieve both determinism and flexibility in parallelizing the application. An accurate response-time analysis of the resulting architecture is developed and presented. We provide an optimization method based on a formulation of the problem as a mixed-integer