Report from Dagstuhl Seminar 18092

The Logical Execution Time Paradigm: New Perspectives for Multicore Systems

Edited by

Rolf Ernst[1], Stefan Kuntz[2], Sophie Quinton[3], and Martin Simons[4]

1 **TU Braunschweig, DE,** ernst@ida.ing.tu-bs.de
2 **Continental Automotive GmbH - Regensburg, DE,** stefan.kuntz@continental-corporation.com
3 **INRIA - Grenoble, FR,** sophie.quinton@inria.fr
4 **Daimler Research - Stuttgart, DE,** martin.simons@daimler.com

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 18092 “The Logical Execution Time Paradigm: New Perspectives for Multicore Systems”. The seminar brought together academic and industrial researchers working on challenges related to the Logical Execution Time Paradigm (LET). The main purpose was to promote a closer interaction between the subcommunities involved in the application of LET to multicore systems, with a particular emphasis on the automotive domain.

Seminar February 25–28, 2018 – https://www.dagstuhl.de/18092

2012 ACM Subject Classification Computer systems organization → Embedded systems, Computer systems organization → Real-time systems, Software and its engineering → Scheduling

Keywords and phrases Automotive domain, logical execution time, multicore architectures, real-time systems

Digital Object Identifier 10.4230/DagRep.8.2.122

Edited in cooperation with Borislav Nikolić

1 Executive Summary

Rolf Ernst (TU Braunschweig, DE)
Stefan Kuntz (Continental Automotive GmbH - Regensburg, DE)
Martin Simons (Daimler Research - Stuttgart, DE)
Borislav Nikolić (TU Braunschweig, DE)
Sophie Quinton (INRIA - Grenoble, FR)
Hermann von Hasseln (Daimler Research - Stuttgart, DE)

The Logical Execution Time (LET) abstraction, which was originally introduced as a real-time programming paradigm, has gained traction recently in the automotive industry with the shift to multicore architectures. The objective of this Dagstuhl Seminar was to investigate new opportunities and challenges raised by the use of LET as a basis for implementing parallel execution of control software.

LET abstracts from the actual timing behavior of real-time tasks on the physical platform: Independent of when a task executes, the time interval between its reading input and writing output is fixed by the LET. This introduces a separation between functionality on the one

The Logical Execution Time Paradigm: New Perspectives for Multicore Systems, *Dagstuhl Reports*, Vol. 8, Issue 02, pp. 122–149
Editors: Rolf Ernst, Stefan Kuntz, Sophie Quinton and Martin Simons
DAGSTUHL REPORTS Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

hand, and mapping and scheduling on the other hand. It also provides a clean interface between the timing model used by the control engineer and that of the software engineer.

The LET paradigm was considered until recently by the automotive industry as not efficient enough in terms of buffer space and timing performance. The shift to embedded multicore processors has represented a game changer: The design and verification of multicore systems is a challenging area of research that is still very much in progress. Predictability clearly is a crucial issue which cannot be tackled without changes in the design process. Several OEMs and suppliers have come to the conclusion that LET might be a key enabler and a standardization effort is already under way in the automotive community to integrate LET into AUTOSAR.

The seminar brought together researchers and practitioners from different backgrounds to discuss and sketch solutions to the problems raised by the use of LET in multicore systems, with a focus on the automotive domain. The program was structured around the following topics: (i) Implementations of LET; (ii) LET and related paradigms; (iii) LET and control; (iv) Future directions of LET. The fruitful discussions covered the following issues:

- LET was designed as a programming paradigm but is now being used as a mechanism for predictable communication. How can the principles of LET be adapted accordingly? How should LET values be chosen?
- LETs act as deadlines for tasks, which means that they must be dimensioned for the worst-case response time of tasks. This may be too inefficient in practice. Alternatives exist where a bounded number of deadline misses may be tolerated. How should LET exceptions (violations of the specified LET) be handled then? How can deadline miss patterns which still guarantee functional correctness (e.g., system stability) be established?
- How should the LET constructs be integrated into AUTOSAR? More generally, how should the design and verification process in the automotive industry be modified to integrate the LET paradigm?
- How does the use of the LET paradigm for multicore systems fit into the more general context of achieving predictability of multicore systems?

This seminar provided a unique opportunity for participants from the automotive industry to get feedback from academia on their effort to adopt the LET paradigm. At the same time, it allowed other participants to confront their own models and/or solutions with industrial reality and identify new research challenges. This seminar furthermore brought together research communities which do not so often interact with each other, e.g. the synchronous, control and real-time communities.

Organization of the seminar

The seminar took place from 25th to 28th February 2018. The first day started with an introduction by the organizers, followed by a talk from one of the co-founders of the LET paradigm – Christoph Kirsch. The following two sessions included talks providing an industrial view on the challenges of implementing LET in the multi-core automotive setting. The first day continued with a session comprised of talks presenting the academic view on LET-related challenges, and concluded with breakout sessions (detailed below). The second day of the seminar started with two sessions in which LET was compared to related paradigms, such as the synchronous model. The afternoon talks focused on the connection between LET and control as well as on a possible application of the LET approach to the domain of graphical processing units. The second day concluded with another set of breakout sessions. The third day included talks exploring future directions of LET, and a final set of breakout sessions.

Breakout sessions led to very interesting and fruitful discussions, and covered, among others, the following aspects:

- **Dimensioning of LET intervals**: The main focus was on how to efficiently dimension LET intervals to fit specific applications, which is currently a very pragmatic and experience based activity. Moreover, the two uses of LET in the automotive setting were identified: (i) Functional LET and (ii) Implementation LET.
- **Buffer optimization within LET**: The main focus was on the management of buffers in a LET-based implementation. The following topics were identified as relevant and thus discussed: minimizing the number of used buffers, strategies to handle memory contentions when accessing buffers, location of buffers in the memory hierarchy of hardware platforms and locality affinities between buffers, impact of spatial partitioning or periodicity of LET frames (harmonic or not) the buffers.
- **The synchronous approach vs LET**: The focus was on the comparison between the synchronous and LET models, with a discussion of their advantages and limitations, and their positioning in the context of the needs of the automotive industry, with a special emphasis on a transition from a singlecore to a multicore setting.
- **Control and LET**: The main focus was on the use of the LET paradigm to implement controllers. The following topics were identified as relevant and thus discussed: Is LET the correct paradigm for controller implementation? What is a viable period choice? How are potential deadline misses handled? Can a proper fault model be conveniently incorporated into the LET methodology? Can LET lead to new contributions in the control research domain?

More details on breakout sessions are available in a dedicated section of this document, after the overview of the talks given during the seminar.

Outcome of the seminar

The seminar has already enabled several collaborations: (i) a white paper on the topic is under preparation; (ii) a special session at EMSOFT'18 will be proposed. In addition, since participants expressed very positive opinions about the seminar and were in favor of reproducing the experience, a follow-up seminar will be considered.

Finally, as organizers, we would like to thank all of the participants for their strong interaction, interesting talks, fruitful group discussions, and work on open problems.

2 Table of Contents

Executive Summary

Overview of Talks

Working groups

3 Overview of Talks

3.1 Integration of the Logical-Execution-Time Paradigm in the Automotive E/E Architecture

Matthias Beckert (TU Braunschweig, DE), Leonie Ahrendts (TU Braunschweig, DE), Rolf Ernst (TU Braunschweig, DE), and BorislavNikolić (TU Braunschweig, DE)

More often the logical execution time (LET) paradigm is considered to ensure synchronization among multiple cores. In theory LET introduces an zero-time communication model, which can be used to provide a consistent core-to-core communication at fixed points in time. This contribution to the Dagstuhl Seminar 18092 provides a possible implementation of the LTE paradigm on a multicore ECU, as a test framework for future research. As first topics the handling of LET misses as well as the integration of LET into the in-vehicle network are discussed.

3.2 Achieving Predictable Multicore Execution of Automotive Applications Using the LET Paradigm

Alessandro Biondi (Sant'Anna School of Advanced Studies - Pisa, IT) and Marco Di Natale (Sant'Anna School of Advanced Studies - Pisa, IT)

3.2.1 Introduction

This document is an extended abstract in support of a talk proposal for the Dagstuhl Seminar on the Logical Execution Time (LET) paradigm (February 25-28 2018). The abstract is a short summary of a work that has recently been submitted by the same authors to the 24th IEEE RTAS conference.

3.2.2 Realizing LET with GIOTTO semantic on multicores

The GIOTTO LET semantic

At a high level, the LET paradigm assumes that the input and output operations of a periodic task τ happen in zero time at the beginning and the end of each periodic instance of τ, respectively. In practice, the actual input/output operations must be scheduled for execution, and can also take place at various time instants (i.e., not necessarily in a strictly periodic fashion) provided that the order of their execution preserves the desired logical semantic.

Note that the order with which they are executed influences the timing properties of the system, especially when flow preservation along communication chains is required. To ensure time determinism, the GIOTTO programming paradigm [1] specifies an order of execution for the communication operations, with a particular focus on those that should happen at the same logical time (see GIOTTO micro steps in [1]).

LET as an opportunity to control memory contention

Due to the contention of architectural shared resources in the memory hierarchy (such as levels of caches and shared memories), real-time applications that are executed upon multicore platforms may experience several delays that are difficult to predict, hence making the timing analysis of the system arduous. That is, without a proper synchronization mechanism, in the worst-case the memory accesses issued by one core can interfere with the other, and viceversa, leaving room for pathological scenarios that inevitably affect the tasks' response times.

Several works in the literature (e.g., see [2], [3]) addressed this problem by proposing clever solutions to improve the predictability of memory accesses. Nevertheless, leveraging the periodic access to the communication variables, the adoption of the LET paradigm brings the potential to improve time determinism by design.

In fact, although the LET paradigm can be realized by scheduling data write and read operations at various time instants, scheduling the communication phases at the beginning of the periodic instances of tasks carries considerable benefits in controlling the memory traffic. Specifically, this approach allows localizing the memory accesses within precise time windows that are determined by the task periods, which are hence known off-line. This rationale enables the possibility of realizing an explicit arbitration of the accesses to shared memories that become under the control of the system designer.

LET tasks with inter-core synchronization

As a reference abstract platform, the presented work assumed that the tasks execute upon $m > 1$ processors each disposing of a local scratchpad memory. The platform also includes a global memory and a crossbar switch that enables point-to-point communication between each core and each memory. All the memories are accessible from all the cores. For instance, this model matches the popular AURIX Tricore platform by Infineon, which is widely adopted in the automotive domain.

Tasks work on local copies of the communication variables that are managed under the LET paradigm, which are stored in local memories. Global (i.e., shared) copies of the communication variables are allocated to the global memory.

The proposed approach to realize LET communication is based on the following design principles:

- Synchronous activation of all the tasks in the system (i.e., all the tasks on all the cores are synchronously released at the system startup).
- Definition of a LET task for each processor that moves data from the local copies to the global copies (write operations), and viceversa (read operations). Such tasks run at the highest priority.
- Adoption of an inter-core synchronization protocol to arbitrate the accesses to the global memory performed by the LET tasks.

Note that since tasks work only on local copies, their execution is not affected by memory contention. Conversely, the accesses to the global copies performed by the LET tasks are subject to contention.

Thanks to a timing analysis of LET communications, it has been identified that, as a function of the task periods, a producer does not need to always update the shared copies of the accessed variables at every periodic instance. A dual conclusion has been reached for consumer tasks. Overall, given a task set, the subset of memory accesses that are required to safely realize the LET paradigm can be analytically characterized. This fact has been leveraged to realize the LET tasks, which resulted to require a variable behavior job by job, but with a cyclically repeating order of the job behaviors. For this reason, LET tasks can be modeled and analyzed as generalized multi-frame (GMF) tasks [4].

To match principle (iii), a simple synchronization protocol has been implemented. The protocol is based on baton passing and enforces an order with which the processors access the global memory. The order can change frame by frame. Spinbased busy-waiting has been adopted.

3.2.3 Implementation and Evaluation

The proposed approach has been implemented on the popular Aurix Tricore platform produced by Infineon and by building upon the ERIKA open-source real-time operating system [5], which is certified OSEK/VDX and implements most of the AUTOSAR OS requirements.

The synchronization of the tasks' periods among the cores has been realized by exploiting the remote procedure call (RPC) features that are available in ERIKA. The resulting design consists in the first core that is in charge of activating all the tasks in the system as a function of a common time reference (a hardware timer).

The realization of the LET tasks required facing a memory vs. time trade-off. A literal implementation of the approach may require the definition of a table that stores the frames of the LET tasks up to the hyperperiod of all the tasks in the system. While this choice would have a limited impact in terms of runtime overhead, it is memory eager for realistic applications. To contain the memory footprint, the solution adopted in our implementation is based on providing counters for each pair of communicating tasks. Such counters can be used to identify the time instants in which the LET communications for a pair of tasks must be performed.

Finally, thanks to the characteristics of the Aurix Tricore, it was possible to devise a lightweight implementation of the inter-core synchronization mechanism. For each processor, two spin variables allocated to the corresponding local memory are provided: one to wait for write operations, and another to wait for read operations. Notification of a LET task that is spinning is then performed by simply updating one of its spin variables from a remote core, i.e., passing the baton.

A case study

An experimental evaluation has been conducted to assess the feasibility of the proposed approach and its impact in terms of timing performance. The proposed LET implementation has been adopted for a synthetic application that has been automatically generated from a model provided by Bosch for the WATERS 2017 challenge [6], which is claimed as representative for a realistic engine control application. The tests have been performed on an Infineon TriBoard v2 equipped with an Aurix TC275 microcontroller running at 200MHz and connected to a Lauterbach PowerTrace to perform debugging and tracing.

The WATERS 2017 challenge came with a model of an engine control application consisting of 1250 runnables grouped into 21 tasks/ISRs that access 10000 labels. The model specifies the labels accessed by each runnable, the type of access (read or write), and the number of accesses. Execution times are also provided together with the tasks' periods.

Based on set of assumptions, a code generator has been developed. The generator inputs the XML file that encodes the system model and generates C code for each runnable where execution segments are realized with for loops including a nop operation in the body. The generator is also in charge of producing (i) the definition of all the communication variables accessed by the tasks (both the local and the global copies), (ii) the corresponding accesses within the runnable code, (iii) the tasks' code (to call a sequence of runnables), (iv) the configuration for the operating system, and (v) the code to setup the OSEK alarms to periodically activate the tasks.

Furthermore, the generator is in charge of generating the code of the LET tasks starting from the information available in the challenge model (i.e., communication relationships between tasks and task periods).

The collected results demonstrated that LET communication – with all the benefits that it brings in terms of predictability of the timing of control outputs and end-to-end latencies – can be realized without significantly harming the timing of the application with respect to the case of direct accesses to the global memory, which by definition lacks of the benefit provided by LET.

The major impact of the realization of LET has been found in terms of memory footprint, which increased by the 7.5% (about 40KB) with respect to the case of explicit communication (i.e., without LET).

By looking at the collected measurements, evident benefits in terms of reduced memory contention have not been observed. Although this is mainly attributed to the fact that the tested application was not sufficiently memory-intensive, note that, in general, the usage of LET does not bring average-case improvements, but rather it allows avoiding worst-case pitfalls and simplifying (by removing pessimism) worst-case analysis. More investigation is required to compare the worst-case performance of LET with the case of explicit communication : a detailed theoretical analysis is in the research agenda of the authors.

References

1 T. A. Henzinger, B. Horowitz, and C. M. Kirsch. *Giotto: a time-triggered language for embedded programming.*. Proceedings of the IEEE, vol. 91, no. 1, pp. 84–99, Jan 2003.

2 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. *Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms.*. in 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013, pp. 55–64.

3 H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. *PALLOC: DRAM bankaware memory allocator for performance isolation on multicore platforms.*. in 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), April 2014.

4 S. Baruah, D. Chen, S. Gorinsky, and A. Mok. *Generalized multiframe tasks.*. Real-Time Systems, vol. 17, no. 1, pp. 5–22, Jul 1999.

5 ERIKA Enterprise. *ERIKA Enterprise: Open-source RTOS OSEK/VDX kernel.*. Available: http://erika.tuxfamily.org

6 A. Hamann, D. Dasari, S. Kramer, M. Pressler, F. Wurst, and D. Ziegenbein. *WATERS Industrial Challenge 2017.*. Available: https://waters2017.inria.fr/challenge/#Challenge17

3.3 Beyond the LET and back to Synchronous Models

Marco Di Natale (Sant'Anna School of Advanced Studies - Pisa, IT)

Considering the objectives for which LET is planned for use in the automotive industry (determinism and enforcing causality), there is a clearly strong relationship between the

LET model and the general class of synchronous (reactive, or SR) models of computation [1, 2, 3, 4, 5]. The LET model has attracted significant attention for its capability of providing causality and time determinism. However, it can be considered a restriction of the SR class of systems (in which a constant delay is applied at the end of each computation step), which provide the same properties (flow preservation and determnism), but with a much greater choice of the possible delays to be applied at each input/output stage. This connection can be leveraged by reusing several results from the research on SR systems that define how to provide efficient or even optimal implementations of tasks and communication primitives [6, 7, 8, 9, 10, 11, 12].

References

1 D. Potop-Butucaru, R. De Simone, J. P. Talpin, "The Synchronous Hypothesis and Synchronous Languages," in R. Zurawski, ed., The Embedded Systems Handbook, CRC Press, 2005.

2 A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Guernic, and R. de Simone. "The synchronous languages 12 years later," in Proceedings of the IEEE, 91, January 2003.

3 G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," in Sci. Comput. Program, vol. 19, pp. 87–152, Nov. 1992.

4 F. Boussinot and R. de Simone, "The Esterel language," in Proceedings of the IEEE, vol. 79, pp. 1293–1304, Sept. 1991.

5 P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: A declarative language for programming synchronous systems," in ACM Symp. Principles Program. Lang. (POPL), Munich, Germany, 1987, pp. 178–188.

6 P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis. "Semantics-preserving multitask implementation of synchronous programs," in ACM Trans. Embed. Comput. Syst., 7(2):1–40, January 2008.

7 J. Forget, F. Boniol, D. Lesens, and C. Pagetti. "A multiperiodic synchronous data-flow language". In 11th IEEE High Assurance Systems Engineering Symposium (HASE'08), Nanjing, China, Dec. 2008.

8 J. Forget, "A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints", Ph.D. Thesis, University of Toulouse, 2009.

9 C. Sofronis, S. Tripakis, and P. Caspi. "A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling." In Proc. 6th ACM International Conference on Embedded Software, 2006.

10 Guoqiang Wang, Marco Di Natale, and Alberto L. Sangiovanni-Vincentelli. "Optimal synthesis of communication procedures in real-time synchronous reactive models." in IEEE Trans. Industrial Informatics, 6(4): 729–743,2010.

11 Guoqiang Wang, Marco Di Natale, and Alberto L. Sangiovanni-Vincentelli. Improving the size of communication buffers in synchronous models with time constraints, in IEEE Trans. Industrial Informatics, , Volume 5, Number 3, August 2009.

12 Haibo Zeng and Marco Di Natale. "Mechanisms for Guaranteeing Data Consistency and Time Determinism in AUTOSAR Software on Multi-core Platforms." In Proceedings of the 6th IEEE Symposium on Industrial Embedded Systems (SIES), June 2011.

3.4 A Time-Triggered execution model for the automotive field and some perspectives

Mathieu Jan (CEA LIST - Gif-sur-Yvette, FR)

Main reference Damien Chabrol, Didier Roux, Vincent David, Mathieu Jan, Moha Ait Hmid, Patrice Oudin, Gilles Zeppa: "Time- and angle-triggered real-time kernel", in Proc. of the Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013, pp. 1060–1062, EDA Consortium San Jose, CA, USA / ACM DL, 2013.
URL http://dx.doi.org/10.7873/DATE.2013.223

CEA LIST ended in 2013 the technology transfer to a spin-off company of a Real-Time Operating System (RTOS), successively called OASIS and then PharOS. This was the end of 18 years of work in this area, the initial idea behind this RTOS being stated in 1995. In the PharOS RTOS, CEA LIST introduced in 2012 a new paradigm in order to address the shortcomings of this TT approach when applied two case studies. Theses two case studies were an automotive powertrain system that mixes angular and physical time scales and an protection relay, which uses an electrical sample time scale.

In this talk, we will briefly describe the steps of these 18 years of work done by CEA LIST on the subject of TT. Besides, we will shortly describe the custom flavor of the TT paradigm that was initially developed at CEA and compare it against the LET paradigm. The main focus of the talk is then to present why our xT extension was needed to fit automotive requirements and show its design within the two aforementioned case studies

3.5 On Event- and Time-triggered Communication in Networked Control Systems

Karl Henrik Johansson (KTH Royal Institute of Technology - Stockholm, SE)

Main reference Bart Besselink, Valerio Turri, Sebastian H. van de Hoef, Kuo-Yun Liang, Assad Al Alam, Jonas Mårtensson, Karl Henrik Johansson: "Cyber-Physical Control of Road Freight Transport", Proceedings of the IEEE, Vol. 104(5), pp. 1128–1141, 2016.
URL http://dx.doi.org/10.1109/JPROC.2015.2511446

In an event-triggered control loop the sensing, communication, computation, or actuation takes place only when needed. This paradigm has been developed to reduce the need for feedback while guaranteeing performance. In a large networked control system, control loops are often either event- or time-triggered. In this lecture, we discuss a few such control loops in emerging cooperative road freight transport systems based on heavy-duty vehicle platooning. It is shown how safety-critical loops, such as regulating the distance between vehicles, are triggered periodically, while other controls, like fuel-optimising the platoon velocity, are triggered whenever needed. Some open problems on how implementation can be formalised using logical execution time and other paradigms are briefly discussed.

3.6 Parallelization of Automotive Control Software

Sebastian Kehr (Denso Automotive - Eching, DE)

Joint work of Bert Böddeker, Eduardo Quiñones, Günter Schäfer, Dominik Langen, Miloš Panic, Jorge Alberto Becerril Sandoval, Jaume Abella, Francisco J. Cazorla
Main reference Sebastian Kehr: "Parallelization of automotive control software", PhD thesis, Technische Universität Ilmenau, Germany, 2016.
URL https://dblp.org/rec/bib/phd/dnb/Kehr16

The purpose of this talk is to present methods for the parallelization of automotive control software that use logical execution times (LETs).

Automotive control software is developed according to the AUTomotive Open System ARchitecture (AUTOSAR) standard. High development costs require the re-use of existing software when the hardware platform changes from a single-core to a multicore electronic control unit (ECU).

This talk focuses on the migration of AUTOSAR legacy software to a multicore ECU. Different parallelization methods are proposed and evaluated; RunPar and Supertasks on runnable-level, timed implicit communication on task-level, and the parallel schedule quality metric for quantification of combinations. The methods respect data dependencies and still enable parallel execution, they exploit the energy-saving potential of the processor, they guarantee latency constraints, and they reproduce the reference data-flow.

3.7 From Logical Execution Time to Principled Systems Engineering

Christoph M. Kirsch (Universität Salzburg, AT)

Joint work of Thomas A. Henzinger, Benjamin Horowitz, Christoph M. Kirsch
Main reference Thomas A. Henzinger, Benjamin Horowitz, Christoph M. Kirsch: "Giotto: A Time-Triggered Language for Embedded Programming", in Proc. of the IEEE, 91(1):84–99, 2003.
URL http://www.cs.uni-salzburg.at/ ck/content/publications/journals/ProcIEEE03-Giotto.pdf

The idea of Logical Execution Time was developed at UC Berkeley starting in the year 2000. In the beginning we pretty much knew what we wanted but we did not expect how controversial the idea would be seen by the scientific community around real-time and embedded systems. Just stating how long something takes to execute seemed to be inconceivable. In this talk, I am going to share the story of how the idea evolved over a number of years at Berkeley and elsewhere and how it finally found its place among the other real-time programming models. The LET story is an excellent example of how combining ideas from fields as diverse as programming languages, real-time and embedded systems, and formal verification can help to solve hard engineering problems. At the end of the talk, I am going to mention the selfie project as another example that we are currently working on. Selfie combines a self-compiling compiler of a tiny subset of C, a self-executing RISC-V emulator targeted by the compiler, and a self-hosting hypervisor that virtualizes the emulator. Selfie can compile, execute, and virtualize itself any number of times. The selfie project would not exist without the LET experience and, to our own surprise, has already received quite a bit of attention and caused some controversy. Will it be the topic of a Dagstuhl seminar in fifteen years? This is probably too much to ask for. Thanks a lot to the organizers of this seminar for inviting me!

3.8 From Physical Timing Requirements to Certifiable Real-Time Systems: How to Capture Requirements and Generate Correct Real-Time Programs?

Florence Maraninchi (VERIMAG - Grenoble, FR)

Working on various industrial case-studies in the past decade, we realized that, given the very quick evolution of hardware platforms, and the growing complexity of embedded software, it is, more than ever, necessary to start with a very general point of view. On one hand we need to understand the physical timing constraints and tolerances as determined by control engineers for a given application (sampling frequencies and jitters, end-to-end latencies, etc.), without mentioning software entities like tasks or scheduling. On the other hand we need to characterize precisely the computation and communication performances of a given execution platform, and assess their predictability.

Designing the implementation means finding space and time allocations for the application functional parts, in such a way that the physical constraints are met by construction, and in a provable way for certification authorities. The LET paradigm, or the much older "Bulk synchronous parallel" model, are elements in this broad picture, but not the only ones.

We will use several examples (with the Kalray MPPA, or a simple Arduino platform), to illustrate the nature and specification of the timing constraints, the implementation schemes, and how the constraints are met.

3.9 End-To-End Latency with Logical Execution Time

Jorge Luis Martinez Garcia (Robert Bosch GmbH - Stuttgart, DE)

Joint work of Ignacio Sañudo, Marko Bertogna, Jorge Luis Martinez Garcia

Modern automotive embedded systems are composed of multiple real-time tasks communicating by means of shared variables. The effect of an initial event is typically propagated to an actuation signal through sequences of tasks writing/reading shared variables, creating an effect chain. The responsiveness, performance and stability of the control algorithms of an automotive application typically depend on the propagation delays of selected effect chains. Indeed, task jitter can have a negative impact on the system potentially leading to instability. The Logical Execution Time (LET) model has been recently adopted by the automotive industry as a way of reducing jitter and improving the determinism of the system.

In this talk, a formal analysis of the LET model for real-time systems composed of periodic tasks with harmonic and non-harmonic periods is provided, analytically characterizing the control performance of LET effect chains. It is also shown that by introducing tasks offsets, the real-time performance of non-harmonic tasks may improve, getting closer to the constant end-to-end latency experienced in the harmonic case. The introduction of offsets not only may reduce response times and end-to-end latencies, but it also allows decreasing the jitter of important control parameters.

3.10 LET for Legacy and Model-based Applications

Andreas Naderlinger (Universität Salzburg, AT) and Stefan Resmerita (Universität Salzburg, AT)

Joint work of Andreas Naderlinger, Wolfgang Pree, Stefan Resmerita
Main reference Stefan Resmerita, Andreas Naderlinger, Stefan Lukesch: "Efficient realization of logical execution times in legacy embedded software", in Proc. of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017, pp. 36–45, ACM, 2017.
URL http://dx.doi.org/10.1145/3127041.3127054

Abstract

The Logical Execution Time (LET) paradigm has recently been recognized as a promising candidate to facilitate the migration to multi-core architectures in automotive real-time software systems. We outline several findings regarding the application of the LET paradigm that corroborate this perception. Our work in this respect deals with LET for legacy systems and LET in the context of model-based development (e.g., in MATLAB/Simulink). Furthermore, we present open issues and highlight implications on the development process when using LET as a synchronization mechanism.

3.10.1 Introduction

Since its initial introduction in the Giotto project [1] almost two decades ago, several research groups have been working on the Logical Execution Time (LET) paradigm which has by now been the foundation for several programming languages and run-time systems [2]. While the promised advantages, such as time- and value-determinism, do sound desirable for safety-critical real-time systems, the approach has long been met with skepticism. Also, with a few exceptions (e.g., [3], [4]), industry has been reluctant in the trial, let alone the adoption of LET. With the emergence of multi-core architectures in automotive system this seems to change as LET could play a key role for obtaining predictable behavior when parallelizing control software. As a consequence, it recently experienced an increase in attention from both academia and industry (e.g., [5]). Amongst other benefits, LET shall provide deterministic inter-task communication across multiple cores on automotive multi-core architectures. Central questions that need to be dealt with involve, for example, how to reconcile performance-dominated requirements of control systems with the additional memory and computational costs that come with LET, how to apply this primarily top-down and correct-by-construction approach to legacy systems that may not satisfy all the initial assumptions, and also how and where to best introduce the LET concept in a development process that is no longer centered around code but on models specified, e.g., in MATLAB/Simulink. This abstract presents two active lines of work in our group dealing with these questions: (1) LET applied to legacy automotive systems including multi-core architectures, and (2) LET in the context of a model-based development with simulation in MATLAB/Simulink.

3.10.2 LET for Legacy Systems

A substantial amount of legacy code is used in many embedded system domains, in particular in the automotive industry. When carried over to a new hardware platform, data consistency issues arise and provisions must be made to maintain proper behavior along cause-effect chains. Our first work on applying LET to an industrial engine controller reaches back to 2010 [3], where the imposed restriction of limiting code changes to top-level functions, lead to a considerable increase in memory requirements (both RAM and ROM). Abandoning this restriction leads to a drastic reduction in run-time and memory overhead [6]. Both dimensions of overhead are largely dependent on the particular application and are depending also on the degree of freedom for choosing the exact LET [7], especially for multi-core targets. There is an enormous potential for optimizations when migrating to multi-cores using LET. Naturally, different optimizations are difficult to harmonize. For example, a strategy that reduces buffers and leads to less total run-time overhead could still lead to bulky and unacceptable copy-operations at a particular LET boundary. Also, the question is how far optimality of a certain setting (in whichever respect) impacts extensibility or changeability of the software and the potential validation effort that goes along with it. In [8], we propose a transformation process from single-core legacy software to LET-based versions that can be safely run on a multi-core. It is a process that can be applied incrementally and that is centered around a static buffer requirement analysis, which can be applied at different levels of abstraction. The most abstract level determines a minimal set of buffers for a given LET specification that is independent of the underlying platform configuration (including task priorities, scheduling, and function-to-core mapping). Being a minimal upper bound, this set can be further reduced in more refined abstraction layers where restrictions and details are incorporated into the analysis. For example, we describe an optimal buffering strategy w.r.t. the number of required buffers for a known multi-core platform configuration under fixed-priority preemptive scheduling.

At run-time, automotive applications change functionality to adapt computational demands according to the crank angle in order to avoid system overload, for example, at high engine speeds. This variation in physical execution times must be reflected also in the logical timing domain, e.g. using a multimodal specification (as already supported by Giotto). So far, to the best of our knowledge, support for multiple modes in the context of LET and multi-core has not been addressed. It is unclear how this will increase the complexity of the analysis and the run-time system that ensures the LET semantics, and it is also unclear what the exact semantics should even be in the case of a mode switch and how this goes together with AUTOSAR modes.

3.10.3 LET in Model-based Development

Model-based design has become an established development approach in the field of embedded real-time systems. Clearly, LET should be an established fixture already in the modeling and simulation phase of the development. The predominant environment for modeling and simulating automotive control systems is MATLAB/Simulink, which is based on the synchronous block diagram (SBD) formalism. Being built on the synchronous reactive programming paradigm, SBD is also suited to realize LET behavior. However as we outlined in [9], in the presence of cyclic data-dependencies as found between AUTOSAR runnables, for example, care must be taken to comply with limitations implied by the simulation engine such that a valid execution order of the blocks can be found. In [10], we present a Simulink implementation of a run-time system (E-machine) for a multi-mode multi-rate LET specification involving potentially cyclic data dependencies. The simulated control algorithms may be implemented as Simulink/Stateflow blocks or in the programming language C.

Originating from a purely control-engineering oriented view, since at least the introduction of AUTOSAR support, Simulink models realign to more and more software-centric perspectives. It is not clear how a clean transition from platform-independent to platform-dependent models that support push-button code generation can be achieved. In any case, for obtaining highly optimized code with a minimal number of additional LET buffer variables, for example, the code generation for a particular runnable must not be considered in isolation. Timing and data-flow dependencies of the whole application must be taken into account.

The need for considering data-flow dependencies is not only an issue of optimization. In a classic LET-based specification, the LET interval of an individual task (or function) was mainly driven by physical requirements (expressed in the period) and inevitably by properties of the hardware/system (implied by worst-case execution/reaction times). Since in the multi-core setting LET is used as a synchronization mechanism, LET intervals must be harmonized across multiple cores and thus cannot be decided individually on task/function-level. This has implications on the whole development process (and also on the mode-switch issue discussed in the previous section). Despite this, the development process might benefit from using LET as a design contract between control and embedded software engineers as outlined in [11].

In the standard LET model, where a task's LET equals the period, end-to-end latencies are a major concern. However, when the LET is contained in the period, this issue is considerably relaxed [12].

An open issue, for example, is robustness w.r.t. the impact of a model change (e.g., adding a new data-dependency) on the generated code and how the attempt to minimize code changes relates to the resulting run-time efficiency.

3.10.4 Conclusion

This abstract touched on aspects of LET related to its application to automotive software systems, especially for a single- to multi-core migration. We hereby covered legacy and model-based applications and gave examples of open issues in this respect.

References

1 T. A. Henzinger, B. Horowitz, and C. Kirsch. *Giotto: A time-triggered language for embedded programming.*. Proceedings of the IEEE, vol. 91, pp. 84–99, January 2003.

2 C. M. Kirsch and A. Sokolova. *The logical execution time paradigm.* in Advances in Real-Time Systems, S. Chakraborty and J. Eberspacher, Eds. Springer, 2012, pp. 103–120.

3 S. Resmerita, K. Butts, P. Derler, A. Naderlinger, and W. Pree. *Migration of legacy software towards correct-by-construction timing behavior.*. in Monterey Workshop, 2010, pp. 55–76.

4 V. Belau, H. von Hasseln, and M. Simons. *Coordinating AUTOSAR runnable entities using giotto - first concepts.*. 2012, poster presented at DEPCP 2012, Dresden, Germany.

5 A. Hamann, D. Dasari, S. Kramer, M. Pressler, F. Wurst, and D. Ziegenbein. *Waters industrial challenge 2017.*. 8th International Workshop on Analysis Tools and Methodologies for Embedded Realtime Systems, WATERS 2017.

6 S. Resmerita, A. Naderlinger, M. Huber, K. Butts, and W. Pree. *Applying real-time programming to legacy embedded control software.*. in 2015 IEEE 18th International Symposium on Real-Time Distributed Computing, April 2015, pp. 1–8.

7 J. Hennig, H. von Hasseln, H. Mohammad, S. Resmerita, S. Lukesch, and A. Naderlinger. *Towards parallelizing legacy embedded control software using the LET programming paradigm.*. in Proc. of WiP Papers of the 22nd IEEE Real-Time and Embedded Technology and Apllications Symposium, ser. RTAS'16, 2016.

8 S. Resmerita, A. Naderlinger, and S. Lukesch. *Efficient realization of logical execution times in legacy embedded software.*. in Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017, 2017, pp. 36–45.

9 A. Naderlinger, J. Templ, and W. Pree. *Simulating real-time software components based on logical execution time.*. in Proceedings of the 2009 Summer Computer Simulation Conference, ser. SCSC'09. Vista, CA: Society for Modeling & Simulation International, 2009, pp. 148–155.

10 J. Templ, A. Naderlinger, P. Derler, P. Hintenaus, W. Pree, and S. Resmerita. *Real-Time Simulation Technologies: Principles, Methodologies, and Applications (Computational Analysis, Synthesis, and Design of Dynamic Systems).*. CRC Press, April 2016, ch. Modeling and Simulation of Timing Behavior with the Timing Definition Language, pp. 159–178.

11 P. Derler, E. A. Lee, M. Torngren, and S. Tripakis. *Cyber-physical system design contracts.*. in 2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS), April 2013, pp. 109–118.

12 W. Pree, J. Templ, P. Hintenaus, A. Naderlinger, and J. Pletzer. *TDL - steps beyond giotto: A case for automated software construction.*. Int. J. Software and Informatics, vol. 5, no. 1-2, pp. 335–354, 2011.

3.11 Embedding multi/many-core COTS in the avionics domain

Claire Pagetti (ONERA - Toulouse, FR)

Main reference Wolfgang Puffitsch, Eric Noulard, Claire Pagetti: "Off-line mapping of multi-rate dependent task sets to many-core platforms", Real-Time Systems, Vol. 51(5), pp. 526–565, 2015.
URL http://dx.doi.org/10.1007/s11241-015-9232-1

The last decade has seen the emergence of multi-core processors and many-core architectures. Although these architectures may allow a huge gain in terms of performance, they also face important challenges to their integration in safety critical environments, in particular aeronautics. As an example, due to the intensive resource sharing and lack of documentation, it is very difficult to ensure time predictability, one of the key elements of certification expectation.

A solution to tackle this last issue for COTS architectures is to rely on software-enforced predictability solutions. In this talk, we will review several execution models that have been developed in collaboration with aeronautical industrial partners. An execution model is a set of rules to be followed by the designer to remove or at least reduce drastically the temporal interferences. Most of them rely on the spatial and temporal static allocation of data, applications and communication. Thanks to these kinds of solutions, there is a way to masterize complex COTS architectures and prepare their embedding in the next generation of aircrafts. We then briefly present a recent project named PHYLOG. This project aims at offering a model-based and software-aided certification framework for aeronautics systems based on multi/many-core architectures in order to reduce as much as possible the amount of textual documentation and replace it with model(s) and promote automatic analysis and replace part of the testing with formal methods, as accepted by the DO333.

3.12 Reconciling the Original LET Paradigm with its Current Use in the Automotive Industry

Sophie Quinton (INRIA - Grenoble, FR)

In this brief talk, I discuss the differences between the original LET paradigm and its current implementation in the automotive industry. In particular, the original LET concept assumes that LETs are specified by the application designer. In contrast, in recent work about applying the LET concept to legacy code in automotive, LETs are based on tasks' WCETs. Additionally, the original paradigm has LET tasks as basic software blocks. In the automotive context, runnables are the smallest unit of software. One therefore has to decide how to map runnables to LET tasks, which was not considered before. Finally, not all tasks are implemented following the LET paradigm.

In the presentation, I try to reconcile the original LET paradigm with the current use of LET in the automotive industry. I also discuss a possible definition of what a correct implementation of the LET paradigm should be.

3.13 Logical Execution Time - An Industrial Perspective

Hermann von Hasseln (Daimler Research - Stuttgart, DE) and Martin Simons

Joint work of Stefan Kuntz, Hermann von Hassel, Martin Simons

In this talk we highlight the Industrial Automotive System Development with focus on control systems like engine control, battery control, or brake control. These systems are essentially characterized by mainly cyclic real time computations under tight memory and run-time constraints. It is shown how this design process suffers from severe complexity through a highly distributed development process, which include different stakeholders on different levels of abstractions, going down from system level to control unit levels. It is argued that therefore only simple design patterns are useful.

The need for multi-core processors with its multiple challenges has only intensified this complexity. The need for new methods, new tools and new development processes, together with the need of migration of big packages of legacy code to multi-core architectures posed real trouble. It became clear that predictability and timing have to become crucial elements in the whole software design process. And it turned out that most implementation patterns used by providers seem to fit the LET paradigm. Indeed, LET was on the radar in the Automotive Industry a long time as a recurring pattern, not only because LET is attractive because it supports determinism, but also because it defines a clear technical organizing principle for the coordination of software components that are developed independently by different stakeholders. Beside, the re-factoring of big legacy code packages and the re-definition of LET intervals offered a striking opportunity for the migration to multi-core architectures.

3.14 Migration of Legacy Embedded Control Software from Singlecore to Multicore Controllers

Hermann von Hasseln (Daimler Research - Stuttgart, DE)

There are continuing efforts to migrate legacy software of automotive applications, which has been developed for singlecore controllers to multicore platforms. The steadily growing demand for computing power can only be satisfied by embedded multicore controllers.

The process of migration has to be geared with existing highly agile and highly distributed processes of software development for the automotive applications. These demands put heavy restrictions on the use of existing parallelization schemes.

While the OEM is faced with the issues of parallelizing the software and specifying the requirements to the ECU supplier, the latter has to deal with implementing the required parallelization within the integrated system.

The Logical Execution Time (LET) paradigm addresses these concerns in a clear conceptual framework. We present here ongoing efforts for applying the LET paradigm in this respect: (1) Parallelization of legacy embedded control software, by exploiting existing inherent parallelism. The application software remains unchanged, as adaptations are only made to the middleware. (2) Using the LET programming model to ensure that the parallelized software has a correct functional and temporal behavior.

In this talk we want to report on these efforts, and show how the application of the LET paradigm helps to achieve to goal of parallelization by separation of concerns, helps on the path of re-designing legacy software more suitable for multicore platforms, and helps ECU-suppliers to semalessly integrate OEM-software into their frameworks.

References

1 Hennig, J., von Hasseln, H., Mohammad, H., Resemerita, S., Lukesch, S., Naderlinger, A.: Towards Parallelizing Legacy Embedded Control Software Using the LET Programming Paradigm.
2 Lowonski, M., Ziegenbein, D., Glesner, S.: Splitting Tasks for Migrating Real-Time Automotive Applications to Multi-Core E.
3 Resmerita, S., Naderlinger, A., Lukesch, S.: Efficient Realization of Logical Execution Times in Legacy Embedded Software.
4 Hu, T. C.: Parallel Sequencing and Assembly Line Problems.
5 Keller, J., Gerhards, R.: PEELSCHED: a Simple and Parallel Scheduling Algorithm for Static Taskgraphs.

3.15 LET as an interface between control engineers and SW integrators?

Dirk Ziegenbein (Robert Bosch GmbH - Stuttgart, DE)

Joint work of Arne Hamann, Eckart Mayer-John, Dirk Ziegenbein

The Logical Execution Time (LET) paradigm gained traction in the automotive industry and also at Bosch initially as a mean to master the transition to multi-core microprocessors.

The principal properties of the LET paradigm that are of interest in this use case are deployment-independent behavior (time and value determinism of cause effect chains) and resource-efficient implementations.

This talk proposes Logical Execution Time (LET) as the basis of a portable and composable specification of control functions. In this sense, LET serves as an interface and supports a separation of concerns between the primary tasks of control engineers and software integrators. The talk outlines the basic requirements on such an interface, discusses trade-offs and open challenges for research.

4 Working groups

4.1 Buffer optimization within LET

Mathieu Jan (CEA LIST - Gif-sur-Yvette, FR), Alessandro Biondi (Sant'Anna School of Advanced Studies - Pisa, IT), and Sylvain Cotard (Krono Safe - Orsay, FR)

The discussion within this breakout session was mainly centered around defining a list of issues associated to the management of buffers when implementing the Logical Execution Timing (LET) model. Buffers are indeed used to perform data exchange between LET frames. We ended the session with the following list of issues: minimizing the number of buffers used, implement strategies to handle memory contention when accessing to these buffers, location of buffers in the memory hierarchy of hardware platforms and locality affinities between buffers, management of these buffers and impact of spatial partitioning or periodicity of LET frames (harmonic or not) on this management. In the remainder of this summary, we briefly describe some of these issues. We first describe general impact/requirements on buffers due to the use of the LET model.

4.1.1 LET impact/requirement on buffers: overview

The LET model enforces isolation and synchronization between readers and writers. At the buffer level, the consequences are:

- producers and consumers never access to the same version of data at any point of time in single or even in multi-core;
- no locks (e.g. HW semaphores, spinlocks) are used enabling wait-free / lock-free implementation for managing these buffers;
- data become visible atomically to consumers at a given point of time by a single pointer switch done by a single instruction;

Buffers implies the use of multiple copies of same data. The basic implementation requires the use of at least two versions. When additional constraints have to be taken into account, more copies may be needed, for instance when spatial protection is a requirement. In this case, intermediate copies have to be done at user and/or at kernel level.

4.1.2 Number of buffers and implementation options

Automotive OEM and Tiers have to implement robust, efficient, but also low-cost systems. Industrial applications suffer from a lack of everything: a lack of computing power - we have to compute more and more and average CPU load become huge, and a lack of memory - hundreds of software components lead to thousands of data that have to be manipulated. If all software components were allocated to dedicated LET frames, the amount of memory needs to implement buffered communication would be a blocking point. That is why, the design of an application with LET has to be used carefully and the minimization of buffers is a critical point. The solution is twofold:

- From software point of view, the developers and researchers have to propose implementation that minimize the number of buffers;
- From engineering process point of view, even if LET is an efficient solution to ease implementation, a fine grain analysis of the application still has to be done in order to logically group together part of software that can share the same rights and does not need to be isolated. That means a LET-based implementation is an output of higher level tools and methods used all along the development process.

The implementation of the LET paradigm requires decoupling the data accessed within the execution of application tasks from the shared data that are published to the other tasks. Possible approaches to realize this decoupling are (i) the use of multi-buffering techniques or (ii) the introduction of local copies.

The former is based on a swapping mechanism: similarly to the realization of some wait-free algorithms, the tasks write and read from buffers that will be released for being accessed by other tasks at specific points in time (e.g., the end of the task execution). The actual number of buffers that is required depend on the parameters (e.g., the periods) of the communicating tasks. Multi-buffering is generally lightweight to implement as it does not require data copies. However, it may necessitate of specific arrangements of data in memory (e.g., wrapped into data structures) and, if no proper strategies are adopted, it may require multiple de-referentiation of memory pointers.

Conversely, when adopting local copies, tasks always write on private variables that are copied from and to shared copies by a communication stack. This approach tends to increase the system overhead due to the data copies, but it is typically simpler to implement and it allows tasks to directly access variables. Furthermore, it may increase the application footprint with respect to the case of multi-buffering.

4.1.3 Memory contentions when accessing buffers

A major issue in executing real-time applications upon multicore platforms is the contention of architectural shared resources in the memory hierarchy (e.g., levels of caches and global memories). In the worst-case, the memory accesses issued by one core can interfere with the other, and viceversa, leaving room for pathological scenarios that inevitably affect the tasks' response times. When looking at memory contention, the adoption of local copies to realize LET communication can provide considerable benefits that increase the software predictability.

Indeed, note that the multi-buffering approach implies that two communicating tasks will access the same memory areas, thus not providing any control on the way tasks access memories. Conversely, the adoption of local copies allows controlling the memory contention in scratchpad-based multicore platforms, where local copies are allocated to private scratchpads and shared copies to the global memory. By scheduling the LET communications at the beginning of the periodic instances of tasks, the accesses to the global memory can be localized within precise time windows that are determined by the task periods, which can host an explicit arbitration protocol that restores predictability of the memory traffic. Memory accesses simply comprise copy-in and copy-out phases from and to the global memory.

4.1.4 Location of buffers in a memory hierarchy

Under stringent memory space constraints, or to improve the tasks' worst-case response-time, the placement of variables into memory should be considered as part of the design space. In fact, it is common that multiprocessor platforms have different access times for different memories (e.g.., global vs. local memories) and also the contention delays are strictly dependent on the frequency with which tasks access such memories. An optimization of the data placement is therefore a desiderata in the design flow of real-time applications.

In the presence of LET communication, this optimization phase should explicitly consider the additional buffers (or local copies) mentioned in previous sections. In the case local copies are used to implement LET, note that to enable a contention-free execution of tasks the local copies must be mapped to private memories (e.g., local scratchpads).

4.1.5 Spatial partitioning of buffers

When spatial partitioning is a requirement, hardware memory protection units must be used to provide clear and explicit segmentation of binaries. Access rights associated to sections of a binary are linked the execution modes supported by core and are (for some of them) dynamically changed upon task switch to allow appropriate buffer access for the enabled task. Sections of a binary can also be aggregated to reduce the run-time overhead of dynamically updating access rights. How this aggregation is performed depends on the memory protection abilities provided by the hardware unit (number of descriptors available, granularity of protection, etc.). Access performance of the memory support, location of memories being used to store buffers between cores when multi-core architecture is targeted and similarities in the required memory access rights are also considered. In order to implement a strict write only within the current execution context or read from the other execution contexts (to perform data copy) policy, intermediate buffers are then needed.

4.2 Control and LET

Martina Maggio (Lund University, SE) and Rolf Ernst (TU Braunschweig, DE)

Joint work of The entire working group on Control and LET (The summary has been only checked by Rolf Ernst and Martina Maggio)

The discussion has been centered around the use of the LET paradigm to implement controllers. Generally speaking, in this discussion, a controller is a piece of code that should run periodically on a platform. Periodicity and predictability are both crucial for the correct system behavior. A few key issues have been discussed.

- Is LET the correct paradigm for controller implementation? We have been discussing two different LET realizations. In the first one, the sensor data acquisition is followed by the LET frame, and then actuation happens at the end of the period. In the second one, the LET frame is short and the actuation occurs befor the end of the period, but the controller task is triggered again at the end of the period. The first one is closer to the implementation of delayed actuation controllers, in which the controller is *designed* knowing that the actuation will happen only at the end of the period and this concern is taken into account in the control synthesis. The second alternative is closer to the most common implementation of control strategies, in which the code to compute a control signal is as fast as possible and additional computation time is needed after the actuation for the controller state update, estimation, and housekeeping operations.

- What is a viable period choice? In both the realizations mentioned above, the LET frame period is an important parameter for control performance (usually measured as a function of the system error, the difference between the desired system state and the current one). In controller implementations, the period choice is often dictated by physical considerations on the plant to be controlled. Can these physical considerations be extended to handle hardware and software constraints that derive from the concurrent execution of many different tasks? Is this helped by LET?
- Assuming the LET paradigm is used for the implementation, how can we handle potential deadline misses? From the control perspective, how can this be: (i) analysed, (ii) predicted, (iii) factored in the control design. Some research work uses the weakly hard real-time systems model to encode deadline misses and design stabilizing controllers for a given plant, with the deadline misses constraints in mind [1]. The question of whether the weakly hard model is the correct model to use for control design is a very open research question. One of the points that was raised is that missing a deadline when the system is around its equilibrium (close to the desired behavior) is very different with respect to missing a deadline when the system is in its transient phase and should still reach a fixed point.
- Can we incorporate a proper fault model in the LET design and methodology? To match safety requirements, hardware and software faults must be included. Hardware faults are generally represented by probabilistic fault models and are traditionally addressed by hardware redundancy. In many applications, transient faults can be tolerated if they have no permanent impact on the state of a system. In control, such effects can often be treated like deadline misses. Permanent hardware faults and degradation can sometimes be treated by redundancy in time, e.g. by load redistribution. Should the LET model already include such errors (very conservative) or should this involve a mode change with a new LET model? What is the effect on control design? Software and conceptual errors are much more difficult because of their various potential effects. Can control be tolerant to software errors? How do we prove that and can LET help here? Diversification is the typical approach to safeguard against software errors. Can LET help structuring the effect of diversification on timing and function?
- What can LET do for control research? More in particular, can new type of controllers be syntehesized *because of* the subsequent implementation with the LET paradigm? For example, in Model Predictive Control there are optimizations [2, 3] to shorten the controller code computation, that degrade the quality of the control signal but ensure faster termination of the code execution.

References

1 S. Linsenmayer and F. Allgower. Stabilization of networked control systems with weakly hard real-time dropout description. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 4765–4770, Dec 2017.

2 S. Richter, C. N. Jones, and M. Morari. Computational complexity certification for real-time mpc with input constraints based on the fast gradient method. *IEEE Transactions on Automatic Control*, 57(6):1391–1403, June 2012.

3 J. L. Jerez, E. C. Kerrigan, and G. A. Constantinides. A condensed and sparse qp formulation for predictive control. In *2011 50th IEEE Conference on Decision and Control and European Control Conference*, pages 5217–5222, Dec 2011.

4.3 The synchronous approach vs LET

Florence Maraninchi (VERIMAG - Grenoble, FR) and Alain Girault (INRIA - Grenoble, FR)

Abstract

This document summarizes the discussions that started with the breakout session entitled "synchronous vs LET", and continued until the end of the workshop. We first recall the main principles of both approaches, and then list the points that were discussed, and the points on which the participants reached an agreement. We focused on the aspects of one or the other that can help the transition between single-core and multi-core implementations of hard real-time systems as presented by the automotive industry.

4.3.1 Common Motivations and Overview of the Approaches

Both approaches are dedicated to the implementation of hard real-time systems, on top of various execution platforms, i.e., hardware with or without an operating system. Both provide general principles that help guarantee that the implementation indeed meets the hard real-time deadlines required by the application. Both accept to trade performance for predictability and safety. They impose strong constraints on the structure of the implementations, so that guaranteeing deadlines is feasible. In particular, they insist on input/output determinism: the same sequence of input samples at the same times always produce the same sequence of output samples at the same time.

Those hard real-time applications come from control engineering problems. They impose physical timing constraints like: sampling frequency for inputs from sensors, refreshment frequency for outputs to actuators, end-to-end maximum latency between inputs and the outputs they influence along a cause-and-effect chain.

The Synchronous Approach, Principles, Languages and Tools

The synchronous approach, first introduced in the 80's with languages like Esterel [2] or Lustre [1] and Signal[5], can be summarized as follows:

- It is designed to help reason on systems by decoupling logical and physical times as long as possible in the design flow; reasoning in logical time means defining variables as mathematical series indexed by N, to represent discrete time.
- Several programming languages have been proposed to write such mathematical series in a structured way. Esterel proposes an imperative style, while Lustre or Signal adopt a functional and declarative dataflow style.
- All the synchronous languages initially come with compilers into sequential code (the design concurrency being statically scheduled); a lot of approaches have been proposed to produce dynamically-scheduled code, or parallel code on multi- and many-core architectures, or even distributed systems. This is where logical time meets physical time. For instance, when a program is compiled into a single-loop sequential code (of the form: init memory; while (1) read inputs; compute outputs and update memory; write outputs;), it comes with a proof obligation: the actual WCET of the loop body on the chosen hardware should be sufficiently small w.r.t. the physical timing constraints on the inputs and outputs.

A lot of existing industrial tools belong to the synchronous family, the main example being Simulink, and the tool SCADE based on Lustre. Some of the synchronous languages and tools can be used as system-level languages, or Architecture-Description-Languages (ADLs), thanks to their dataflow style. This is for example the case of Prelude.

The LET (Logical Execution Time) Approach

The LET approach has been proposed in [4]. It is based on a simple solution to the problem of matching initial physical timing constraints of the application, and actual physical execution times of the implementation.

If $y = f(x)$, f will correspond to a piece of code, and the time it takes to execute this code will vary (because it depends on inputs, or because of other sources of variations caused by the execution platform). When a real-time system is implemented as a set of independent tasks running on top of a real-time operating system, each task i being one such function f_i, these variations mean that the order in which the tasks execute may vary, hence modifying the data exchanged between them and making the overall execution time more difficult to deduce from local execution times.

The essential idea of the LET principle is to reason as if the code of each function f_i always took exactly its WCET; and then to guarantee that the implementation is consistent with this view. Thanks to this design choice, the functional and temporal semantics are deterministic.

4.3.2 Discussion

We focused on the aspects of the two approaches that can help the transition between single-core and multi-core implementations of hard real-time systems as presented by the automotive industry, including the existence of legacy C code. The theoretical question of whether the LET principle is a subset of the synchronous approach quickly appeared as irrelevant for this matter (at least as a concept in programming language research, see for instance [3]).

In the sequel we will use SYNC when referring to the synchronous approach.

Needs of the Automotive Industry

The initial functional intention of a control engineer is to design a number of functions that define outputs in terms of inputs, with associated timing constraints (see the description of functions and tasks in Section 1). The legacy code is made of Simulink diagrams (for some applications), and a large number of lines of C code, called runnables, that can be assembled to get the implementation of one functionality. Each runnable should be executed at a given rate (i.e., they have a period expressed in terms of the physical time); runnables may exchange values by reading, or writing to, shared names (shared variables). The data dependency graph between runnables, as expressed by the read and write accesses to these names is known (or can be computed), in principle.

Building an application amounts to defining the periodic and sporadic tasks scheduled by a real-time operating system as sequences of runnables (i.e., designing their functionality f_i and their period or minimal inter-arrival time), in such a way that the runnables are run at the appropriate rate, and the data dependencies are respected. Once this design in completed, it is crucial to ensure the I/O timing determinism.

Current Practice, Pros and Cons of SYNC and LET

The main question remains at the end of the discussion: when, in the design flow from the control problem to the actual implementation on a particular execution platform, can/should we freeze the decision on the splitting of the application into components (tasks), and the periods associated with them?

With SYNC, especially with dataflow formalisms, the whole application is a dataflow diagram. It can be reorganized freely (without modifying its I/O mathematical semantics) in order to split or group components, before freezing the structure that will serve as a guide for the implementation. The implementation principles are independent of this view.

Once the execution platform, and the implementation principles are defined (for instance, we can decide to map one runnable onto one core, or one real-time task...), we start to get some information on the actual communication and execution times of the components that are indeed feasible. We can then re-import these information in the dataflow diagram (possibly modifying the semantics, e.g., by introducing delays). This approach can be used as a way to keep the model and the code "in sync". It also allows to test/debug/verify the impact of the constraints imposed by the execution platform, on the high-level model.

With LET, the situation is a bit more constrained. It is an implementation scheme, rather than a programming language construct. The structuring of the application into "components" is considered to be given, and will not be put in question again. For each of these components, a "LET frame" has to be chosen. The size of a LET frame is, by default, set to the period of the component that is assigned to it. Thanks to this choice and to the LET semantics, communications from one component to another one are deterministic.

4.3.3 Conclusions

The problem faced by the automotive industry is the transition to multi-core architectures, in the presence of a large amount of legacy sequential code that was designed for single-core platforms. The current practice relies on a very strict definition of the allowed single-core implementations, which guarantees determinism and correctness. The ideal objective would be to reuse the legacy code in new strict implementation schemes to be defined for multi-core platforms. Problems arise because real-time operating systems on single-core platforms are very particular; they have strong synchronization properties, which will become invalid with multi-core implementations.

Some degree of re-engineering seems unavoidable, in order to make the intrinsic timing and dependencies constraints explicit. This means two things: (1) deciding which of the synchronization/order phenomena observed with the implementation (e.g., a READ of a shared variable that always comes after a WRITE) were indeed required by the application constraints, and which are just artefacts (the former have to hold also on multi-core platforms, while the latter can safely be forgotten); (2) conversely, understanding how each application constraint is guaranteed by the implementation; in case it happens to be true by chance, thanks to the particular behavior of the single-core platform, new explicit mechanisms will have to be defined for those constraints to hold on multi-core platforms. This is especially crucial for causality constraints implied by the cause-and-effect chains of the application.

A generalized version of the LET principle might well be the appropriate choice for the definition of new mechanisms and strict implementation principles on multi-core platforms; it will result in sub-optimal performances, but this is not necessarily a problem if it brings determinism and clarity. However this choice alone will not help in revealing the intrinsic timing and dependency constraints of the application. Here the general ideas and tools of the synchronous approach can help re-engineer the legacy models and the legacy sequential code, in order to answer questions (1) and (2) above.

References

1 J-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. *Outline of a real time data-flow language.*. In Real Time Systems Symposium, San Diego, September 1985.

2 G. Berry and G. Gonthier.. *The Esterel synchronous programming language: Design, semantics, implementation.* Science Of Computer Programming, 19(2):87–152, 1992.

3 Matthias Felleisen. *On the expressive power of programming languages.*. Science of Computer Programming, 17(1-3):35–75, December 1991.

4 Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. *Giotto: A timetriggered language for embedded programming.*. In Embedded Software, pages 166–184, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

5 P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. *Programming real time applications with signal.*. Proceedings of the IEEE, 79(9):1321–1336, September 1991.

4.4 Dimensioning of LET Intervals

Dirk Ziegenbein (Robert Bosch GmbH - Stuttgart, DE) and Hermann von Hasseln (Daimler Research - Stuttgart, DE)

Joint work of the breakout session on dimensioning LET intervals

In a breakout session, seminar participants discussed approaches how the LET intervals can be defined for particular applications. The discussion showed that the current practice is currently very pragmatic and experience-based with little procedural guidance. Furthermore, the different use-cases of LET in the automotive industry are imposing different defining parameters for the dimensioning of LET intervals.

A general guideline is to dimension the LET intervals as large as possible in order to put the least constraints on platform integration and portability. The larger the ratio of physical execution time and logical execution time is, the smaller the degrees of freedom for the software integrator get, e.g. for itegrating several LET workloads on the same HW platform. Of course, the size of the LET intervals is bounded by the application requirements such as cause and effect chain latencies.

In the following, the two major use cases for LET in the automotive industry and their respective guidelines for dimensioning the LET intervals are described

- "Functional LET" - LET is used as an abstraction level and interface between function developers (e.g. control engineers) and software integrators. In this use case, LET interval shall only be functionally motivated, e.g. due to a certain latency requirement of a control algorithm. The reason behind this guideline is to establish a deterministic implementation-independent behavior and leave the maximum remaining freedom for the implementation.
- "Implementation LET" - LET is used as a mean to efficiently implement a deterministic behavior in a dedicated HW/SW platform, e.g. the parallelization of a software application on multi-core processors. Here, the dimensioning of LET intervals depends on the inherent pralellism of the software application as well as on the targeted load distribution between cores and the physical execution times of SW entities.

In summary, there are some guidelines or best practices to dimension LET intervals but in general the task is not well-formalized and has been seen as a field for future research.

Participants

- Leonie Ahrendts
TU Braunschweig, DE
- James H. Anderson
University of North Carolina at Chapel Hill, US
- Matthias Beckert
TU Braunschweig, DE
- Alessandro Biondi
Sant'Anna School of Advanced Studies – Pisa, IT
- Bert Boeddeker
Denso Automotive – Eching, DE
- Björn B. Brandenburg
MPI-SWS – Kaiserslautern, DE
- Sylvain Cotard
Krono Safe – Orsay, FR
- Marco Di Natale
Sant'Anna School of Advanced Studies – Pisa, IT
- Benoit Dupont de Dinechin
Kalray – Orsay, FR
- Rolf Ernst
TU Braunschweig, DE
- Glenn Farrall
Infineon – Bristol, GB
- Gerhard Fohler
TU Kaiserslautern, DE
- Alain Girault
INRIA – Grenoble, FR
- Mathieu Jan
CEA LIST – Gif-sur-Yvette, FR
- Karl Henrik Johansson
KTH Royal Institute of Technology – Stockholm, SE
- Sebastian Kehr
Denso Automotive – Eching, DE
- Christoph M. Kirsch
Universität Salzburg, AT
- Stefan Kuntz
Continental Automotive GmbH – Regensburg, DE
- Ralph Mader
Continental Automotive GmbH – Regensburg, DE
- Martina Maggio
Lund University, SE
- Florence Maraninchi
VERIMAG – Grenoble, FR
- Jorge Luis Martinez Garcia
Robert Bosch GmbH – Stuttgart, DE
- Andreas Naderlinger
Universität Salzburg, AT
- Moritz Neukirchner
Elektrobit Automotive – Erlangen, DE
- Borislav Nikolic
TU Braunschweig, DE
- Nathan Otterness
University of North Carolina at Chapel Hill, US
- Claire Pagetti
ONERA – Toulouse, FR
- Paolo Pazzaglia
Sant'Anna School of Advanced Studies – Pisa, IT
- Christophe Prévot
INRIA – Grenoble, FR
- Sophie Quinton
INRIA – Grenoble, FR
- Stefan Resmerita
Universität Salzburg, AT
- Hermann von Hasseln
Daimler Research – Stuttgart, DE
- Eugene Yip
Universität Bamberg, DE
- Dirk Ziegenbein
Robert Bosch GmbH – Stuttgart, DE