The IDA LET Machine - An efficient and streamlined open source implementation of the Logical Execution Time Paradigm

Matthias Beckert and Rolf Ernst
Institute of Computer and Network Engineering
TU Braunschweig, Germany
{beckert | ernst }@ida.ing.tu-bs.de

***Abstract*—Multi-core CPUs are getting more and more important in the field of automotive electronic control units. In order to make use of the advantages of those multi-core platforms, new techniques for system design need to be considered. This includes scheduling and communication as well as function placement onto several cores. 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. In this interactive session we show, how those mechanisms can be implemented efficiently with a low memory and runtime overhead. We provide a generic open source implementation, which profits from special software/hardware features in the automotive domain and can sill be used on the other platforms. The approach is evaluated with an automotive use-case in an OSEK/VDX conform implementation based on ERIKA/OS.**

I. Introduction

Each car might include more than 50 electronic control units (ECUs) and several buses (or other interconnect mechanisms). In general, software on automotive ECUs is implemented in a set of periodical tasks. The scheduling is done based on fixed priorities with a rate monotic priority assigned for all tasks. Each task consists of a set of so called *runnables* which implement the applications functionality. Inside a task those runnables are executed in a static order. Due to cost and time, functionality developed for singlecore ECUs is usually reused also on multicore ECUs.

In automotive singlecore systems, the communication along several tasks and runnables is often achieved via shared memory variables following a publisher subscriber paradigm. When dependent runnables are executed on different cores, the communication between those needs to be synchronized. Otherwise data consistency cannot be guaranteed, especially if shared-memory communication is used. A possible approach to mitigate this problem is to insert fixed points in time where data is read or written. This mechanism can be covered with the paradigm of *logical execution time* (LET) [5]. The paradigm defines, that at task activation all input data is read and after the results will not be written before or after the defined LET. As LET enforces a deterministic write behavior, data consistency among several cores can be achieved if this theoretical concept can be accurately implemented. The IDA-LET-Machine (ILM)[3] is an open source LET implementation based a zero-time communication with double buffering[4].

II. Integration into the Automotive Domain

Modern automotive software is often developed and integrated according to the *AUTomotive Open System ARchitecture* (AUTOSAR)[1] set of standards. In general it consists of three parts. First, the *AUTOSAR Software* which contains the applications. Second, the *AUTOSAR Runtime Environment* (RTE) which abstracts inter- and intra-ECU communication to the application. And third the *AUTOSAR Basic Software* (BSW) which integrates driver, services and hardware abstraction layer (HAL). In the AUTOSAR context, the operating system is considered a system service. As the LET paradigm needs to be tightly coupled to operating system and underlying hardware, it therefore should be integrated as a part of the AUTOSAR BSW.

Fig. 1 shows an overview of our implementation proposal. Each LET related action is coupled to an so called *LET Event*. Those actions can be the start of an LET task, a pointer swap or also the backup of a read pointer. As an example, at t_0 in Fig. 1 the LET task λ_1 is started and the corresponding block b_1 will be executed within λ_1. In order to generate the LET events on different cores, we use an *LET IRQ* which is generated on one core by a single timer and is then redirected to other cores. In our example, the IRQs are generated on C0 and redirected to C1 and C2 if necessary. If an LET IRQ should also result in an LET event on C0, it is directly processed afterwards. Therefore, the IRQ pattern generated on C0 contains events from all other cores and is repeated periodically. The period of the IRQ pattern is defined by the hyperperiod of all LET tasks. As Fig. 1 contains tasks with an 10ms and 20ms period, the hyperperiod is equal 20ms in this example. From pattern and period we derive an *IRQ Table*. Each entry consists of an offset relative to the hyperperiod and a bitfield which marks all relevant cores for event redirection. The table is used to configure a *Hardware Port* which generates the needed IRQ pattern in a most efficient way. This can be done with a timer and/or capture compare unit. Inside the hardware dependent IRQ handler the *LET Handler* on each relevant core is activated, which usually can be done with a software IRQ. While the hardware generated IRQ is only handled on one core (e.g. C0 in our example), the LET handler can be executed on all cores in parallel with a barrier synchornization at the end of each LET handler. Each handler accesses an own LET *Event Table*, which contains again an relative offset, an action to be performed and an additional payload field. In contrast to the IRQ table, the event table might contain multiple entries with

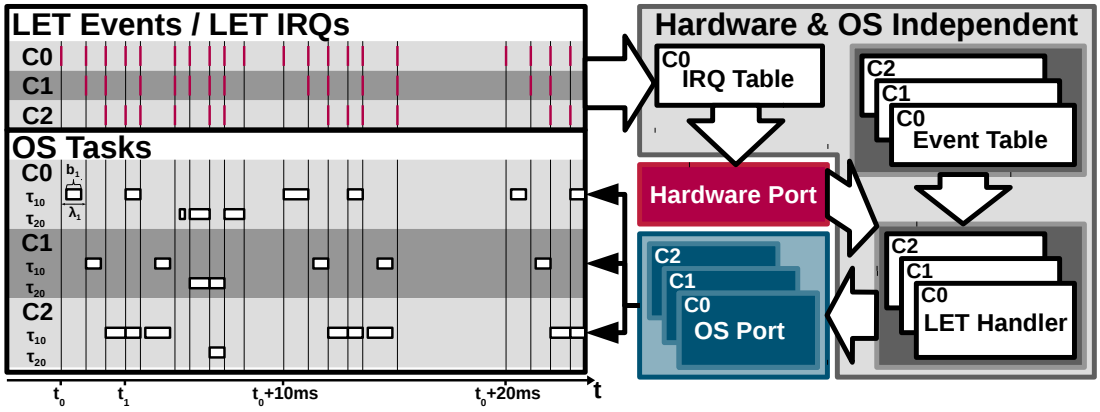

Fig. 1. Overview of the proposed LET implementation

the same relative offset. This is the case when two or more actions must be performed at the same time. An example for this behavior is a simple back-2-back execution of two LET tasks (e.g. on C2 at t_1), where first the pointer of the previous LET task is swapped and second the following LET task is activated. The activation of an LET task is performed through the *OS Port* abstraction, which maps the LET task activation to an OS task activation. At this point the LET implementation hands over the actual OS task dispatching to OS scheduler. This way the LET implementation activates the OS task but does not dictate the actual task execution or the scheduling scheme.

The generation of the needed IRQ pattern can be realized with common hardware periphery usually used for the generation of a pulse width modulation (PWM). This is also the case for the prevalent Infineon AURIX microcontroller series, which we used for our first hardware port. Most of the AURIX microcontroller do support the generation of PWM signals in two different ways. Either based on the *Capture Compare Unit* (CCU6) or on the *Generic Timer Module* (GTM). We used in our implementation the CCU6 module, as the implementation was straight forward this way. Using the GTM instead is more challenging due to its complexity.

For the software port we use ERIKA OS[2] as underlying operating system, which implements an OSEK/VDX and AUTOSAR OS compliant API. Both define so called *basic conformance classes* (BCC) and *extended conformance classes* (ECC). Tasks of both conformance classes can be started with the *ActivateTask* function. At the end of each task the OS function *TerminateTask* is called, to announce the end of execution to the OS. The mapping of LET tasks to operating system tasks can be performed direct to BCC or grouped to either BCC or ECC.

III. Evaluation

The evaluated setup consists of an AURIX TC275, ERIKA OS 2.8.0 and is included in the ILM repository on GitHub[3]. The memory overhead shown in Table I has been extracted from the optimized and stripped .elf file with the binutils tool *nm*. First, the LET implementation including the relevant event handling, OS and hardware port and second, the hook functions used for pin toggling and tracing. The LET implementation primarily consists of instructions in .text and a few uninitialized system variables in .bss, which get initialized through an *init* call during runtime. Additional on C0 also the

	.text	.bss	.data	.rodata	.global
LET implementation					
C0	638	62	0	0	8
C1	200	17	0	0	0
C2	200	17	0	0	0
Hook functions					
C0	166	0	184	192	0
C1	146	0	0	184	0
C2	158	0	0	184	0

TABLE I
Memory overhead in bytes

AURIX specific hardware init and 8bytes for bitmask/barrier in .global used for core-2-core synchronization are included. For additional tracing via a logic sniffer, the hook functions implement a pin toggling resulting in additional code overhead in .text. The overhead in .data and .rodata is based on pin mapping tables which provide an abstraction layer to the used evaluation platform.

The generated runtime overhead $U_{Hdl,x}$ on core x depends on the number of LET handler activations per hyperperiod and on the actual execution time of the LET handler $C_{Hdl,x}$.

$$U_{Hdl,x} = \frac{NoOfHandlerActivations_x}{Hyperperiod} \cdot C_{Hdl,x} \quad (1)$$

In the example from [3], C0 is used for IRQ generation and the LET based application is executed on C0/1/2. Therefore $C_{Hdl,0}$ includes CCU6 IRQ and LET handling ($6.2\mu s$), while $C_{Hdl,1}$ and $C_{Hdl,2}$ do only include LET handling ($4.5\mu s$). This lead to a runtime overhead of $\sim 0.58\%$ on C0, $\sim 0.34\%$ on C1 and $\sim 0.36\%$ on C2 for the given example.

IV. Conclusion

The results show a small memory footprint and a low impact with respect to runtime mechanisms. Overall the generated overhead is primarily based on the actual application and not by the ILM implementation itself. The ILM therefore represents an efficient and streamlined implementation of the LET paradigm, especially for prototyping or academic research.

References

[1] Automotive open system architecture. https://www.autosar.org/.
[2] ERIKA Enterprise RTOS. http://www.erika-enterprise.com/.
[3] ILM. https://github.com/matthiasb85/IDA-LET-Machine.
[4] M. Beckert, M. Möstl, and R. Ernst. Zero-time communication for automotive multi-core systems under spp scheduling. In *Proc. of ETFA*, Berlin, Germany, Sep 2016.
[5] C. M. Kirsch and A. Sokolova. The logical execution time paradigm. In *Advances in Real-Time Systems*, pages 103–120. Springer, 2012.