A SimEvents Model for the Analysis of Scheduling and Memory Access Delays in Multicores

Caroline Brandberg, Marco Di Natale
Scuola Superiore Sant'Anna, Pisa, Italy,
caroline.brandberg@santannapisa.it, marco.dinatale@sssup.it

***Abstract*—Model-based development of CPSs is based on the capability of early verification of system properties on a model of the controls and the controlled physical system. In the development of multicore systems, the scheduling and contention delays may significantly affect the behavior of the controls, and therefore need to be suitably represented and analyzed. We present a framework for adding the model of schedulers, tasks and multicore memory accesses to Simulink models and to verify by simulation the impact of scheduling and execution times delays on the performance of the controls. In our experiments, different memory access policies, including the use of the Logical Execution Time paradigm are tried and compared. Our framework is based on the commercial SimEvents package, which allows for a much faster simulation compared to other existing approaches; it is highly modular and extensible and can be applied to existing Simulink models with limited changes.**

I. Introduction

Simulink is a graphical modeling environment implementing a Synchronous-Reactive (SR) Model of Computation (MoC) for multidomain Model-Based Design (MBD) and simulation. Continuous-time, discrete-time and discrete-events systems can be defined in the model, making Simulink suitable for the design of Cyber-Physical Systems (CPSs), where the controller (discrete-time) and the plant (continuous-time) must be modeled and simulated together.

Simulink models of controls are purely functional and allow the modeling and simulation in logical time, with the assumption that all reactions complete within the next event. However, in a real system implementation, the code implementing the functionality executes in finite time, and under the control of a scheduler, giving rise to latencies and jitter that may change the performance of the controls. In addition, in multicore CPSs, the memory hierarchy, the arbitration fabric and the memory access policies further affect time predictability and the time behavior.

A model of the task deployment, the scheduler and the resource management policies is not only useful to analyze the impact of the execution time delays on the controls performance but also to support the architecture exploration and optimization stage, when the control functionality needs to be matched to a suitable design for the platform, including the multicore hardware fabric, the operating system with its scheduling and resource management policies, with a task implementation and placement option.

978-1-5386-4155-2/18/$31.00

To represent computation, communication and resource contention delays in Simulink, a possible solution is to use custom blocks (defined by the user) that interface with a scheduling simulation engine in a co-simulation pattern. The TrueTime framework [9] integrates a real-time scheduling and a network simulator in a Simulink custom block, enabling the co-simulation of the control functions considering the software (task) and message implementation, and the scheduler and resource management policies. Presently, multi-core architectures are not supported and the memory contention delays or the memory access costs are not analyzable.

T-Res [12] is a modular framework for the co-simulation of control functionality and controlled system dynamics with real-time scheduling policies and communication mechanisms and networks. Similar to TrueTime, it is based on Simulink, and provides custom blocks for the representation of the task implementation model and the selection of the scheduler. These blocks can be applied to an existing Simulink model with limited and localized changes.

An implementation of the scheduler and resource management logic using SimEvents allows for a better simulation performance compared to the use of custom blocks, since the latter (TrueTime or TRes) use zero-crossing verification. This means that the solver needs to simulate the entire model around the point of zero crossing, and all blocks in the model will be recomputed at these steps, even if most of them may have nothing to do [17].

Compared with the model presented in [17] we provide a suitable selection of scheduling policies, we support a model of tasks and runnables (according to the AUTOSAR standard), and we allow for a more detailed representation of the memory hierarchy and possible memory arbitration mechanisms, and also of the memory access pattern by the application tasks. Several application-level mechanisms to control the access to shared memory resources have been proposed recently [5] to cope with the problem of the possible nondeterminism caused by memory arbitration (and waiting) in multicores. The Logical Execution Time (LET) paradigm [16] is being considered in the automotive domain as a possible way to restore predictability.

The model proposed in this paper allows to analyze not only the impact of scheduling delays on the performance of control applications, but also the impact of latencies in the access to memory resources and to analyze the selection of memory access policies, including LET. Further, four different

execution patterns have been implemented to study the effect of the timing of performing memory accesses.

Finally, we support an execution model that is compliant with the commercial AUTOSAR standard, including the model of runnables and tasks and the access to communication labels. To show the applicability to large models of control applications defined according to this standard, we analyzed the automotive benchmark provided by Bosch for the Waters challenge [15] that contains task, runnables and labels defined at the timing precision of a clock cycle.

The paper is organized as follows. In section II, we introduce the model assumptions and provide a short summary of the SimEvents semantics, its execution model and the interactions between the simulation engine and our framework. In section III, we describe the structure of the framework, with the representation of the scheduler and platform. In section IV we provide an example that shows a use case in which different task models and scheduling policies impact the performance of the controls, as shown by the simulation results. Finally, in section V, we provide our conclusions and a discussion of future work.

A. Related Work

Model-Based Design is widely used in several application domains. Simulink allows to create synchronous reactive models, that can be validated and verified based on a Zero Execution time (ZET) semantics, abstracting from hardware and software latencies [25]. However, execution, communication, and scheduling delays play a role in the definition of the controls performance. Several research works investigate the consequences of computation (scheduling) and communication delays on controls. An overview on the subject can be found in [3] and [10]. Also, an analysis of control activation models based on events rather than periodic triggers (and the possible improvements with respect to the CPU resource utilization) are discussed in [2] and [18].

To study the effects of the model implementation on a given software (task) architecture and execution platform, previous researchers have provided frameworks for modeling the task structure (including CPU placement), the execution under the control of a scheduler, and the communication times in Simulink models. TrueTime [9] [8] is a *freeware*[1] Simulink-based simulation tool developed at Lund University since 1999. It provides models of multi-tasking real-time kernels and networks that can be used for networked embedded systems. TrueTime provides a collection of C++ and MATLAB files to represent networks and schedulers by means of a library with two main simulink blocks: *TrueTime Kernel* and *TrueTime Network*. A task is defined within the context of a Kernel block, with its functionality and execution time defined by MATLAB scripts. The modular framework T-Res [12] provides explicit blocks for tasks, schedulers, messages and networks. T-Res can invoke functionality defined by Simulink subsystems (not only scripts) that are transformed into triggered blocks. Tasks are modeled as sequences of subsystems, executed according to their specified execution times. Inspired by TRES and TrueTime, Naderlinger [21] propose to introduce software execution times by replacing Simulink subsystems by customized xTask blocks, where timing is incorporated by means of a dedicated delay function. Other approaches have been proposed to combine Simulink with schedulability analysis tools. In [**lampke2015resource**], runtime profiles are extracted from SymTA/S and introduced into the functional Simulink model using blocks derived from triggered Simulink subystems: TriggeredSignal, TriggeredRead and TriggeredWrite. These blocks control the availability of data between communicating subsystems based on the estimated execution and communication times.

Wei Li et al. used SimEvents [17] to model the effect of execution times and scheduling in multicores. SimEvents [19] is a commercial toolbox providing a discrete-event simulation engine and component library for Simulink. SimEvents models allow a clear separation between the pure functional model and the architectural components and allow significant performance improvements with respect to the previous approaches (not quantified in [17], but found to be at least 40% in our comparative experiments with TRes on simple models).

When the integration with the model of the controls behavior is not required, a very large number of projects target the evaluation of scheduling policies and the analysis of task implementations. A necessarily incomplete list includes Yartiss [11], Storm [24], ARTISST [13], Cheddar [23], Schesim [20], and Stress [4]. Most of these projects do not allow the representation of the details related to the access to the memory resources in multicores and perform the simulation at the level of the tasks.

The Bosch automotive benchmark provided as a challenge for the WATERS workshop highlights the need to evaluate the impact of resource contention at the level of the memory hierarchy in multicores and presents a model that is characteristic of the AUTOSAR automotive standard [1], in which the behavioral elements are runnables (executable function), rather than tasks [15]. The AUTOSAR standard provides two communication (and the related memory access) mechanisms: explicit and implicit communication.

To ensure deterministic output values and timing, the concept of the Logical Execution Time (LET) was introduced with the programming language Giotto [16]. A LET program consists of multiple periodic tasks. Each task performs all its read operations at the time it is activated and performs all writes at the end of its cycle. In the case of multicore architectures, the exact placement in time of all the memory reads and writes can be leveraged to improve the predictability of the application [5].

Finally, for the modeling and simulation of the time performance of hardware architectures and features, several modeling languages and approaches exist. They are simply too many to be cited here in a comprehensive way. For architecture-level hardware modeling, MARTE by the OMG

[1]http://www3.control.lth.se/truetime/LICENSE.txt

provides an extension to the UML and SysML languages that covers all the typical software concepts (such as threads, resources, schedulers) and hardware elements down to the pin level [14]. However, there seems to be no tool at this time that can fully exploit MARTE models for the purpose of simulation of timing behavior at the level of the memory accesses. MathWorks provides its own XML-based solution for the high-level description of a target platform, with the purpose of customizing code generation. The XML is not fully documented but made available through a GUI, which allows the user to specify additional resources, such as the number of cores available but also the partitioning of tasks to cores.

There exist several other simulation possibilities for architectural exploration, and SystemC is among the most popular languages for architecture description (going practically to the level of a Register Transfer Language). SystemC provides a C++ class library for modeling hardware implementing sequential and parallel functionality, with the purpose of verification by simulation. The language allows the modeling of a virtual platform, possibly consisting of several cores, executing cooperating applications [22]. SystemC is also used in conjunction with MATLAB/Simulink in a co-simulation framework, to enable the co-simulation of the execution of complex (control) functions represented in Simulink on hardware models defined in SystemC [6], [7].

II. System model and Tools

A. Abstract model

This paper considers platforms consisting of a set of symmetric cores $C_1 ... C_n$, each provided with a local memory $M_1 ... M_n$ and an additional shared global memory M_{n+1} connected over a crossbar with FIFO arbitration. Each core is statically assigned a set of tasks scheduled according to a fixed priority policy.

A task set $\Gamma(\tau_p, \tau_s)$ is composed by periodic (τ_p) and sporadic (τ_s) tasks. A task τ_i is associated with a period T_i if periodic, whereas a sporadic task is activated with a minimum $T_{i_{\min}}$ and maximum $T_{i_{\max}}$ inter-arrival time. Each task may be assigned a deadline D_i, with $D_i \leq T_i$ for periodic tasks and $D \leq T_{i_{\min}}$ for sporadic tasks. A task can either be preemptive or cooperative, where a preemptive task can be interrupted by another task when executing and between memory accesses, while a cooperative tasks can be interrupted at runnable boundaries.

Task executes a set of functions in a sequential order $\rho = \{\rho_1, \rho_2, ... \rho_m\}$, referred to as runnables according to the AUTOSAR standard. Runnables communicate by means of *labels*: variables that can be read from and written to in an atomic manner, denoted ℓ_r and ℓ_w respectively. The processing of the data is expressed in terms of instruction, with each instruction taking a fixed number of clock cycles.

Labels are statically assigned a memory placement. The number of cycles required for a runnable to access a label depends on its memory placement and the core on which the task executing the runnable is allocated.

The access to memory by the runnables with the execution of the associated instructions can be performed in different modes. The AUTOSAR standard supports two patterns. In the *explicit communication* model, AUTOSAR runnables read and write data throughout their execution, using spin locks and resource protection mechanisms when required. The *implicit data access* ensures data consistency throughout the execution of the runnable by having all reads performed before the execution of the runnable (with the result stored in local copies of the variables) and all writes performed on local copies and actually moved to the communication variables only after the end of the runnable execution [1].

A natural extension of the previous policy consists in having all the variable reads into local copies performed at the beginning of the task for all the runnables executed by the task, and all the writes at the end of the task execution. Throughout the execution of the task, the code will only operate on local copies, referred to as the task consistency pattern throughout the paper. To ensure deterministic output values and timing, the Logical Execution Time (LET) paradigm read the input at task activation and delay the results until the deadline. The four different patterns are illustrated in Figure 1.

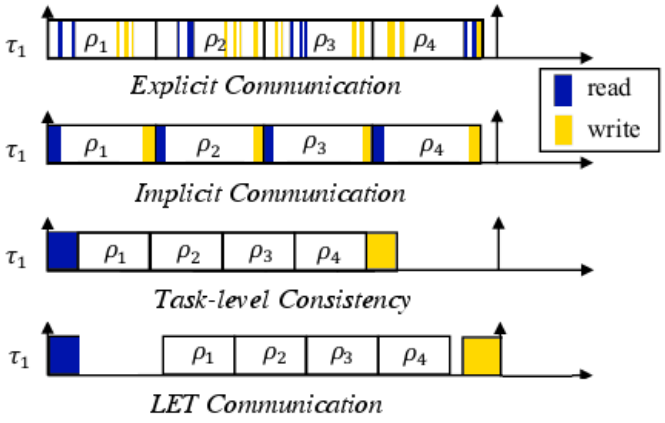

Fig. 1. The memory access patterns with the corresponding accesses in read and write mode by runnables and tasks.

B. SimEvents

SimEvents is a discrete-event modeler and simulation engine with a library of components by the MathWorks. A SimEvent model consists of a set of connected blocks exchanging *Entities*. An entity can be associated with attributes that can change during its lifetime. The lifetime of an entity starts when generated from a SimEvent source block called Entity Generator. Throughout its lifetime, the entity traverses a network of Queues and Servers.

An Entity Server will serve entities for a specified amount of service time. Several entities can be served simultaneously where the capacity is indicated in the graphical block. A server is typically preceded by an Entity Queue which stores entities as an advancement is blocked. Along the network, Entity Input- and Output Switches can be used to affect the

path of an entity. An Entity Input Switch allows entities to arrive at all input ports, whereas an Entity Output Switch will advance an entity based on a criterion to one of its output ports. Additionally, Entity Gates can be used to control the advancement of an entity.

Upon completion of its traversal, the entity ends in an Entity Terminator. Entities are not shown in the model, but the Entity generators and terminators, along with the network of connections and processing servers are represented graphically, see Figure 4.

There are two types of Entity Generators: Time-based and Event-Based. A Time-based Generator is associated with a time source defining when a new entity shall be created. The generation can be defined by a constant period or by a MATLAB script that encodes a stochastic generation. In addition, Event-Based Entity Generators create new entities upon the detection of an event.

An *event* represents a point in time associated with the detection of an incident. An event can be associated with an action resulting (among others) in the generation of one or more new events, a change in the state of an item, and/or a computation of an output. Events are typically used to mark the occurrence of the following:

1. Generation of an Entity
2. Exit of an Entity from a Generator block
3. Entry or Exit of an Entity to a Server block
4. Service completed on an Entity by a Server block
5. Entry of an Entity to a Terminator block

The actions for an event are defined by the user using Simulink functions or a restricted set of the MATLAB language supporting code generation. An action can be everything from invoking MATLAB functions and/or scripts to affect the next destination of the entity to the generation of debug information, defined in the associated SimEvent building block.

As for Entities, Events do not have a graphical representation. Nevertheless, an interaction with the events is allowed by *Observer* classes. By observing certain events, a trace of the timeline of events in the model can be created.

As in Simulink, functionalities are wrapped inside subsystems, providing a hierarchy to the model, where input- and output ports are used to allow for connection to the outside.

C. Implementing LET on a multicore platform

The main principles in realizing LET communication for a multicore platform are:

R1 For each task, all read operations must be scheduled at its activation.
R2 For each task, all write operations must be executed at the end of its period.
R3 For each pair of communicating tasks, all write operations must be completed before the read operations.

Figure 2 illustrates the realization of a schedule compliant with the previous rules and deployed in a multicore, with the help of inter-core synchronization. The implementation is based on the work described in [5] and assumes that all

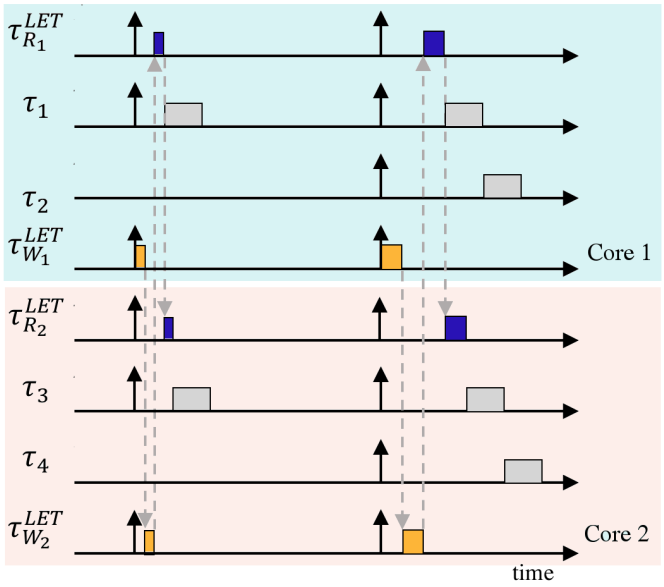

Fig. 2. Illustration of the memory access synchronization realizing the LET pattern.

global memory accesses are performed by two LET tasks for each core, τ_R^{LET} and τ_W^{LET} that are activated at the greatest common divisor of the periods of the tasks on their core, and execute with highest priority (preempting if necessary, other tasks). In this implementation, writes and not performed at the end of the period, but right at the beginning of the following cycle, with a minimum time difference. This scheduling example assumes an execution on two cores, each assigned two tasks in addition to the LET tasks performing the communication as accesses to global memory.

At initialization of the application, the writer LET tasks on each core are executed according to a round robin pattern (orange/light color in the figure). Upon completion of each writer task, a signal is sent to the following core to activate its LET writer task. Next, the reader LET tasks are executed in order, one for each core. Upon completion or the reader LET task, each core is ready to process its application tasks.

III. SimEvents Model of the Multicore Platform, Schedulers and Tasks

The automotive benchmark model of the WATERS challenge consist of 21 tasks statically assigned to four symmetric cores connected to local and global memories. These elements are presented at the top level of the Simulink model shown in Figure 3.

The leftmost area in the model represents the task set composed by sporadic and periodic tasks. The application consist of 1250 runnables and 10000 labels, defined in a table by a MATLAB script. A core is modeled as a subsystem with three inports and two outports. When a task performs a memory access, it leaves the core subsystem through its output port and enters back through the first input port upon the completion of the memory operation. The second outport is used to signal

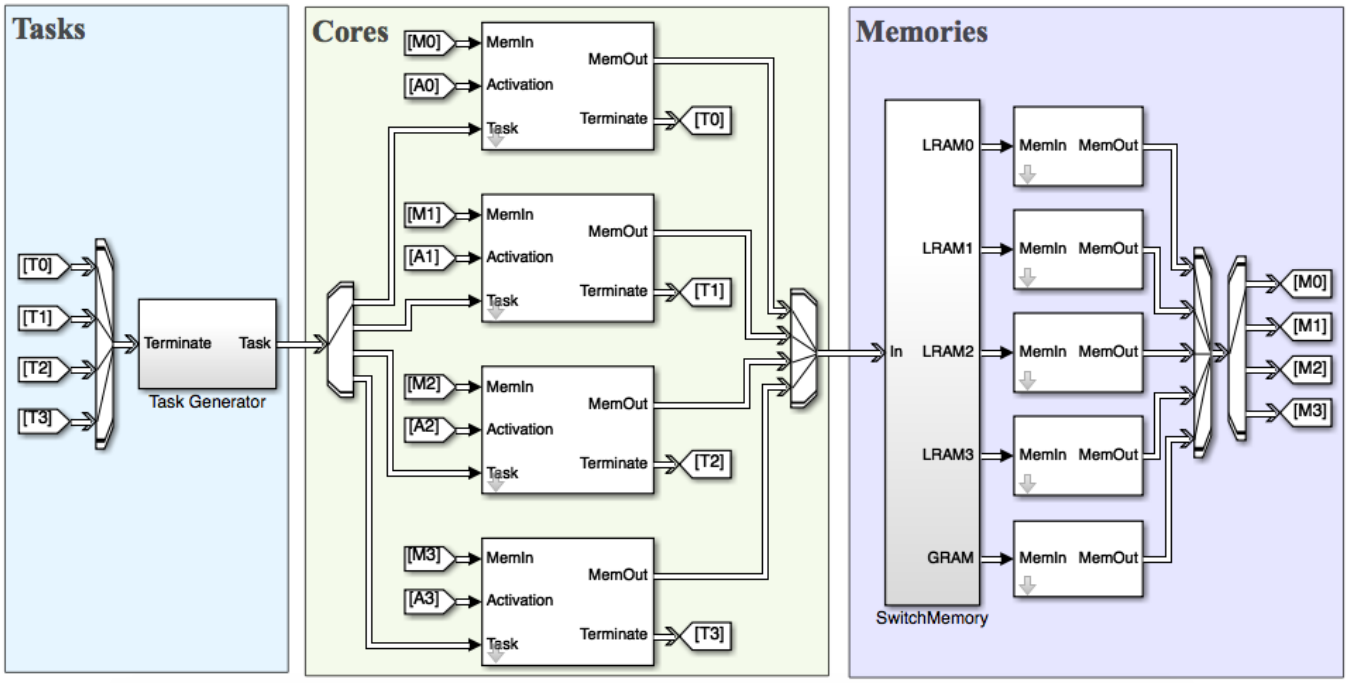

Fig. 3. Bosch automotive benchmark modeled using SimEvent. The model consist of 21 tasks, 1250 runnables and 10000 labels processed by four symmetric cores. Each core is provided with a local memory and an additional memory is shared amongst the cores.

the termination of the currently executing task; and the second inport is dedicated to inter-core synchronization. Finally, the third inport is an entry point for the tasks assigned to the core.

The five memories defined in the model are located in the Simulink model to the right of the cores. All memories can be accessed from all cores, as in the Infineon Aurix Tricore TC27x controllers for automotive systems.

a) Task: A task is modeled with a time-based SimEvent Entity. Entities representing periodic tasks are generated periodically. Entities representing sporadic tasks are generated by Matlab code that activates them with an inter-arrival time selected from a uniformly distributed random set between a lower and an upper bound. Figure 4 shows the user interface for the definition of all the customizable characteristics of a task. A task is associated with a set of runnables executed in a sequential order (defined using a table), and represented in the model by a Matlab data structure, as in

```
struct('name', 'T1', 'core', 1,
'runnables', [1], 'period', 4)
```

b) Runnable: A runnable is modeled as a data structure with an associated name, a set of labels to read from, a set of labels to write to and a number of cycles to execute, such as for example:

```
struct('name', 'R1', 'readLabels', [1],
'writeLabels', [2], 'instructions', 23,
'function', 1)
```

where the `'function', 1` section defines the mapping between the runnable (in the functional model) and the execution platform.

c) Label: A label is modeled as a data structure with an associated name and a memory placement. The structure is also specified in a table.

```
struct('name', 'L1', 'placement', 'LRAM0')
```

d) Tasks, runnables and lables relationship: The relationship between tasks, runnables and labels are defined by connections among the corresponding tables. An example of the relationship is shown in Figure 5. Task T1 executes the runnables with index 1 and 3. In the runnable table, index 1 corresponds to runnable R1 and index 3 to runnable R3. Runnable R1 reads from a label placed at index 4 and writes to a label at index 2. The third runnable, R3, reads from labels at index 5, 6 and writes to a label defined at index 3.

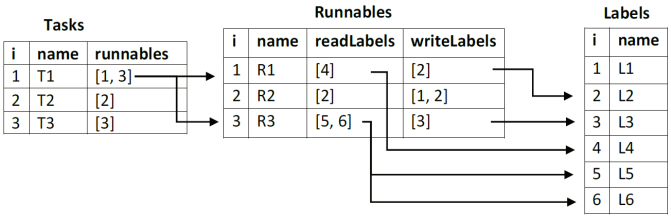

Fig. 5. Relationship between tasks, runnables and labels defined using tables in MATLAB scripts. The index in the table maps to a corresponding row for the related component.

e) Global scheduler: Tasks are assigned by a global dispatcher to the core to which they are statically assigned (as specified by their attribute *core*), and then scheduled on the core according to a local scheduling policy. Upon the

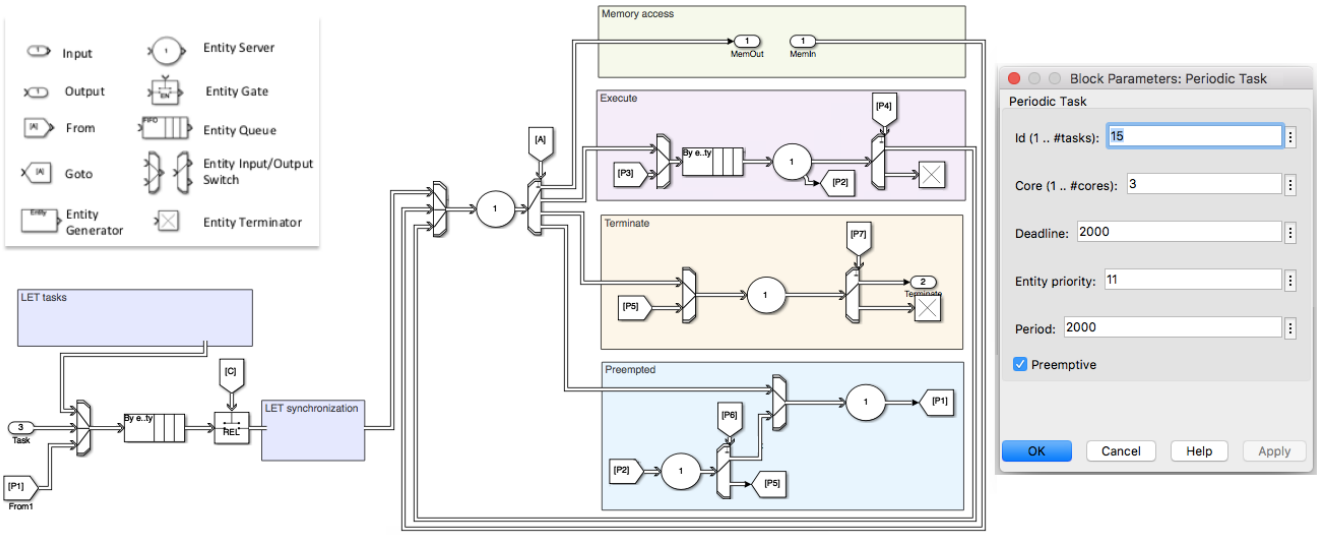

Fig. 4. Internal model of the core as relates to Task execution and a task configuration window.

activation of a task, the entity will enter a server which forwards the entity to the core the task is assigned to.

f) Core and local scheduler: Each core (the high-level model is shown in the left side of Figure 4) includes a scheduler and an execution simulator, realized by processing one task entity at the time, under the control of an Entity Gate. The gate is initially open and then advances to the following task only as a consequence of a task termination. The gate is preceded by a queue ordered according to the local scheduling policy. The scheduling policies that are supported are: fixed priority policy (priority), FIFO or LIFO where each policy can be preemptive or non-preemptive.

Upon the activation of a task with higher priority, the scheduler preempts the current task (if preemptive) while executing and between memory accesses, while cooperative tasks run until completion. A non-preemptive scheduler executes all tasks until completion without interruption.

The core is programmed to process the tasks in four different execution modes: memory access, execute, terminate and preempted. The sequence of runnables to execute, the labels that are accessed by runnables, and the current mode of the task are controlled by a MATLAB script, where four different memory access patterns have been provided: explicit, implicit, task consistency and LET. A core can be released either when a task is preempted or upon completion.

In SimEvents, entities can be assigned priorities but events can not (formally, SimEvents does not support super-dense time). Therefore, it is not possible to enforce a precedence order between a task termination event and a task preemption event and an additional check needs to be performed for an entity entering the preemption path, to ensure it has residual service time.

Debug information can be retrieved from a core in form of text and/or a graphical representation. The debug text provides a time-stamped log of the system and the graphical representation presents the load of the core as well as an illustration of when a task is running, where an example of the graphical representation can be found in Figure 6. The example consist of three preemptive tasks, τ_1, τ_2 and τ_3 with periods $T_1 = 0.004, T_1 = 0.005, T_1 = 0.006$ with the execution time of 0.002 scheduled according to a fixed priority policy supporting preemption by a single core. τ_1 is provided with the highest priority followed by τ_2 and last τ_3.

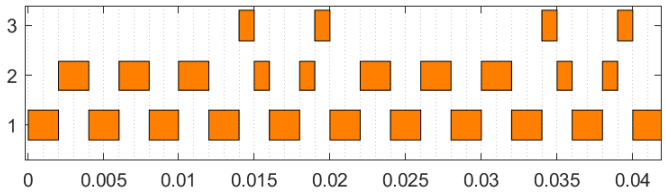

Fig. 6. Graphical debug information presenting current task acquired by the core.

The architectural model can be configured to invoke a functional Simulink model by enabling the ”Invoke functional model” option. This option requires a mapping between the architectural and functional models.

g) Logical Execution Time: The support for the LET execution policy requires additional developments, as described in section II-C. The τ_R^{LET} and τ_W^{LET} tasks are activated and executed for each core. The LET tasks performing read and write operations are provided with a higher priority than the ordinary tasks of the model. The writing task has a higher priority than the reading task and the reading tasks shall be generated at simulation start, while the writing task shall not. This will initially result in only performing read operations followed by write and read operations performed at a given rate periodically. The mode of the task shall also initially be assigned according to the operation of the task, i.e., read or

write. The SimEvents subsystem shown in Figure 7 realizes the sequential activation of the LET tasks (in a round-robin fashion) at the beginning of their cycles.

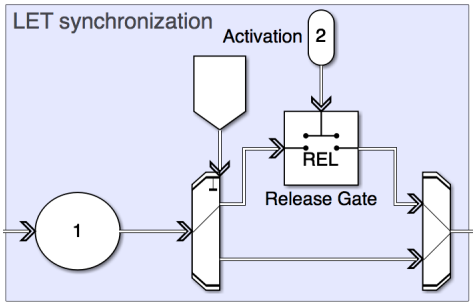

Fig. 7. Internal structure for the block of LET execution.

h) Memory: The access to the local and global memories (the memory subsystem) is modeled with an Entity Queue where entities are ordered according to a FIFO policy followed by an Entity Server. The model contains both local and global memories. The time to access a memory is constant for the global memory; one cycle for the local memory of the core on which the task is executing and 9 cycles for the local memories of the other cores.

When a task is about to perform a memory access, the entity exits from the core subsystem and enters the top level of the model, see Figure 3. The index of the memory label to access is defined by one of the tasks attributes and is used to retrieve its memory placement. The placement will then be used to forward the task entity to the memory subsystem it needs to access. Upon completion of the memory access, the task returns to its core subsystem using one of the cores input ports.

The entire model developed for this architecture and the execution of the WATERS challenge task set is available freely on the Matlab Central website [2].

IV. Experimental results

To assess the applicability of our proposed SimEvents framework, we measured the compilation time for models with increasing complexity, as defined by the number of labels accessed in memory. In this case, the quantity under analysis is the pre-compilation time that is required to build the queue of events that need to be evaluated during the simulated time. Figure 8 shows the results. For models with a level of complexity that is typical of industrial systems (around 10000 labels), the compilation times remain at levels that are manageable at design time (approx 40 minutes for the most complex setting) and grows with the complexity of the system. The increase in the compilation time is hard to quantify. In Figure 8 we show a linear, quadratic (apparently the best fit) and exponential fit. Of course the model only needs to be compiled once.

[2]https://it.mathworks.com/matlabcentral/fileexchange/66173-analysis-of-scheduling-and-memory-access-delays-in-multicores?s_tid=prof_contriblnk

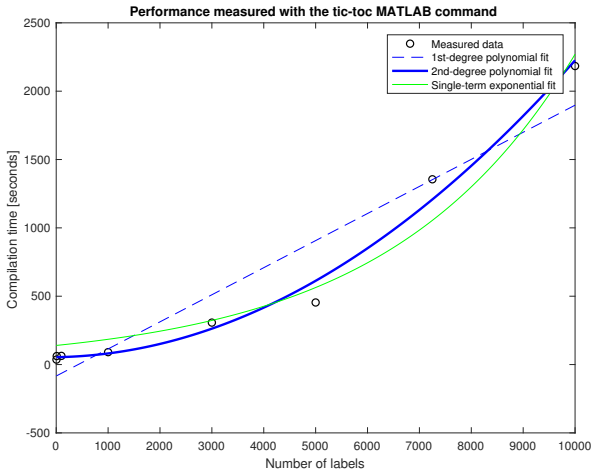

Fig. 8. Time required to compile the automotive benchmark model for models of different complexity

An event based simulator allows an improvement over competing approaches on the simulation time. Figure 9 shows the real execution time that is required to simulate a given number of clock cycles with respect to the complexity of the model. As for the previous set of experiments, the complexity of the model was configured by modifying the number of accessed labels. The simulation time depends on the number of events occurring during the simulated time. The simulation is done with the automotive benchmark model with full debug information, including a text output as well as a graphical representation of the load of the processors and an illustration of when a task is running.

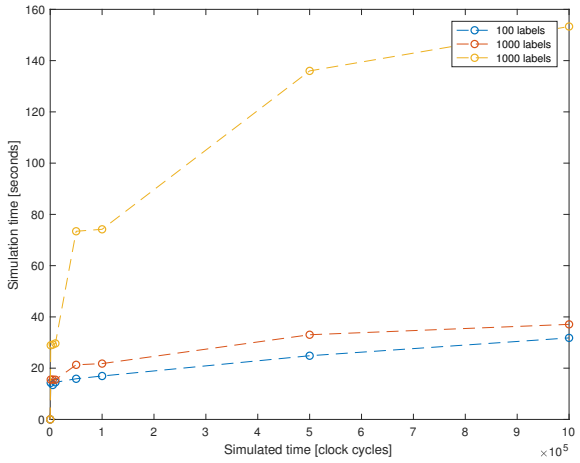

Fig. 9. Time required to simulate a given amount of clock cycles with respect to the complexity of the model

To evaluate the impact of the different memory access patterns, we compared the average and worst-case response times of the tasks in the set. Since the original task set is composed by both periodic and sporadic tasks, the application

of the LET pattern requires to define a periodic server task for each sporadic, with a period lower than the task inter-arrival time and the same worst-case execution time. The response time was measured for all the task instances in the system cycle or hyper-period (the least common multiple of the period of all the tasks).

In the first set of experiments, the average and worst-case response time of the tasks have been compared for the explicit, implicit, and Task-level consistency with local copies patterns. As shown by the Figures 10, and 11, there is a very limited difference in the worst-case response times for the given application. The implicit and task-level consistency patterns can lead to a higher probability of contention given that the accesses are concentrated in a relatively small set of time windows (smaller for task-level copies giving rise to higher response times for some tasks).

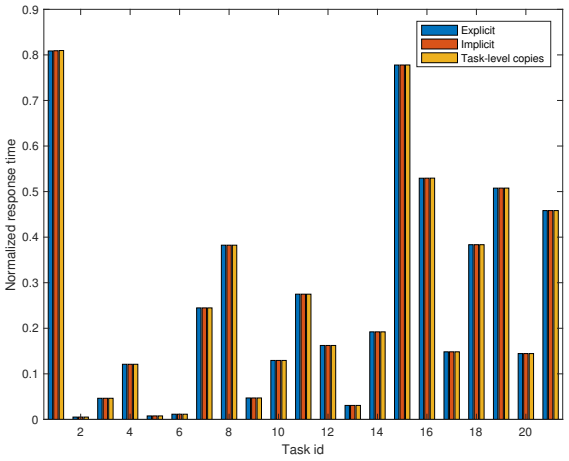

Fig. 10. Average response time for the tasks in the set when using different memory access patterns

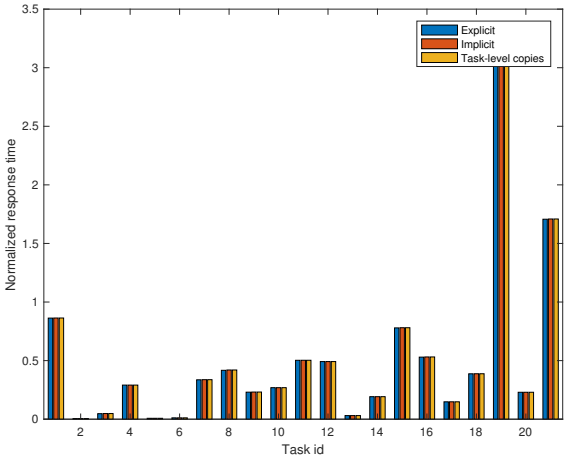

Fig. 11. Worst-case response time for the task set at initialization using different memory access patterns

When using the LET pattern, however, there are significant changes. The task set requires the addition of the LET copy tasks (for writes and reads). Further, these tasks can change the order of scheduling of the other tasks, resulting in significant difference when non-preemptive or cooperative tasks are in the set. In addition, the LET implementation proposed in [5], and implemented in this paper, requires a specific label placement, in which all the labels for inter-core communication are in global memory and label duplicates are provided in the local memory for intra-core communication.

To provide for a relatively fair comparison with other methods, we changed all tasks to be preemptive, and we compared the executions for the label placement as required by our LET implementation. Labels shared within the core are ”represented” as one additional instruction to execute, i.e., the task set never waits in the access to the memory. The two models share the same label placement (all shared labels between cores are in global memory, otherwise they are in the local memory of the core with the tasks accessing them. Table I shows the task configuration. The challenge periods and inter-arrival times are in the column denoted with T_i. The set of the LET periods with the server periods handling the sporadic tasks are labeled as T_i^*.

Core	Id	Task	T_i	T_i^*	prio
Core 0	3	ISR_10	700us	500 us	40
Core 0	8	ISR_5	900us	500 us	39
Core 0	9	ISR_6	1100us	1000us	38
Core 0	7	ISR_4	1500us	1500us	37
Core 0	11	ISR_8	1700us	1500us	36
Core 0	10	ISR_7	4900us	4500us	35
Core 0	4	ISR_11	5000us	5000 us	34
Core 0	12	ISR_9	6000us	5000us	33
Core 1	2	ISR_1	9500us	5000 us	32
Core 1	5	ISR_2	9500us	5000 us	31
Core 1	6	ISR_3	9500us	5000 us	30
Core 1	15	Task_10ms	10ms	10ms	11
Core 2	16	Task_1ms	1ms	1ms	15
Core 2	1	Angle sync	6660us	6500us	14
Core 3	19	Task_2ms	2ms	2ms	13
Core 3	21	Task_5ms	5ms	5ms	12
Core 3	18	Task_20ms	20ms	20ms	9
Core 3	20	Task_50ms	50ms	50ms	8
Core 3	14	Task_100ms	100ms	100ms	7
Core 3	17	Task_200ms	200ms	200ms	6
Core 3	13	Task_1000ms	1000ms	1000ms	5

TABLE I
TASK SET FOR THE EVALUATION OF THE LET ACCESS PATTERN

Figures 12 and 13 show the results for the average and worst-case response times when comparing the explicit memory access pattern with our LET implementation. In all the plots, the Y-axis represents the response time normalized with respect to the task period. The worst-case response times show a significant increase for the tasks with smaller period, because these tasks need to wait for all the LET tasks in the system at the beginning of the cycle. Furthermore, given that the LET tasks execute the copies of all the communication labels on behalf of all the application tasks, they execute according to a multi-frame pattern. Depending on the task

periods, there are instances of the LET tasks that will access many labels (in our case, there is one instance that access labels for all tasks except one), and other instances that access only a limited number. This is why, in the worst case, the LET implementation requires copy times in the order of 100 μs, which can be a significant additional delay for the tasks with the smallest execution times (up to 4 times for one of the tasks). In the average case the situation is better, and the overheads are typically much smaller, less than 5% for all tasks except the six with the smallest execution times.

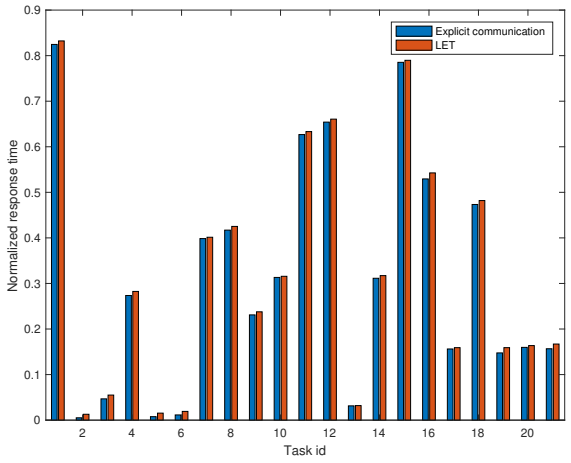

Fig. 12. Average response time for the task set using different memory access patterns

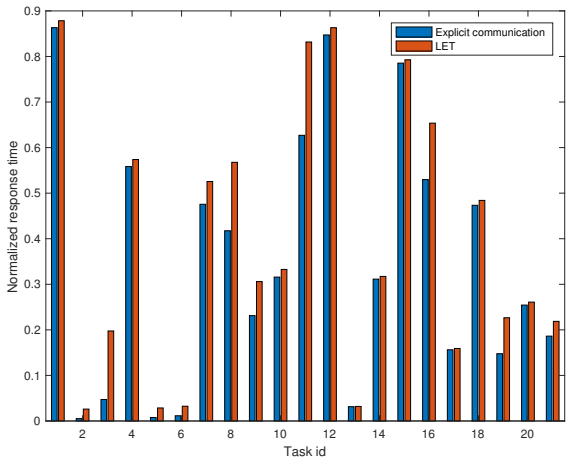

Fig. 13. Worst-case response time for the task set at initialization using different memory access patterns

As discussed in [5], the LET overhead is a price that may be worth paying in exchange for predictability. The LET tasks interference is known and deterministic, while the memory contention for the other access patterns is quite hard to upper bound; and any formula for estimating it may be quite pessimistic. LET also requires additional memory, an increase off 7.5 % compared to the AUTOSAR explicit communication was found in [5]. It is also worth to highlight that further optimizations can be done for the realization of the copies from and to global memory, including the use of DMA and a possible partitioning of the LET tasks to reduce the blocking time for the application tasks (as discussed in [5]). This analysis is left as future work. Nevertheless, the results obtained highlight the opportunity to study the complex relationship between the task set, the memory access pattern, the impact of the label placement and the platform resources.

V. Conclusions

In this paper, we present how to integrate models of scheduling and platform resources using SimEvents to analyze scheduling and memory contention delays in a Simulink model. The model consists of periodic and sporadic tasks executed on an architectural platform of symmetric cores, local- and global memories. The model is AUTOSAR compatible, where the task execute a set of runnables communicating by means of labels. We also provide the implementation for four memory access patterns: AUTOSAR implicit and explicit communication, task level data consistency, and the LET execution model.

References

[1] AUTOSAR. *Specification of RTE Software AUTOSAR CP Release 4.3.1*. 2017.

[2] Amir Aminifar et al. “Designing Bandwidth-Efficient Stabilizing Control Servers”. In: *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. IEEE. 2013, pp. 298–307.

[3] Karl Johan Åström and Bjorn Wittenmark. *Computer-controlled systems: theory and design*. Courier Dover Publications, 2011.

[4] Neil C. Audsley et al. “STRESS: A simulator for hard real-time systems”. In: *Software: Practice and Experience* 24.6 (1994), pp. 543–564.

[5] Alessandro Biondi and Marco Di Natale. “Achieving Predictable Multicore Execution of Automotive Applications Using the LET Paradigm”. In: *Proceedings of the 24th RTAS Conference, Porto*. IEEE. 2018.

[6] Jean-François Boland, Claude Thibeault, and Zeljko Zilic. “Using MATLAB and Simulink in a SystemC verification environment”. In: (2005).

[7] F Bouchhima et al. “A SystemC/Simulink co-simulation framework for continuous/discrete-events simulation”. In: *Behavioral Modeling and Simulation Workshop, Proceedings of the 2006 IEEE International*. IEEE. 2006, pp. 1–6.

[8] Anton Cervin et al. “How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime”. In: *IEEE control systems* 23.3 (2003), pp. 16–30.

[9] Anton Cervin et al. “How does control timing affect performance?” In: *IEEE control systems magazine* 23.3 (2003), pp. 16–30.

[10] Anton Cervin et al. “Optimal online sampling period assignment: theory and experiments”. In: *Control Systems Technology, IEEE Transactions on* 19.4 (2011), pp. 902–910.

[11] Younès Chandarli et al. “Yartiss: A tool to visualize, test, compare and evaluate real-time scheduling algorithms”. In: *Proceedings of the 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*. 2012, pp. 21–26.

[12] Fabio Cremona, Matteo Morelli, and Marco Di Natale. “TRES: a modular representation of schedulers, tasks, and messages to control simulations in simulink”. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM. 2015, pp. 1940–1947.

[13] David Decotigny and Isabelle Puaut. “ARTISST: An extensible and modular simulation tool for real-time systems”. In: *5th IEEE ISORC Symposium*. 2002, pp. 365–372.

[14] OMG Group et al. *Modeling and Analysis of Real-time and Embedded systems (marte)*. 2007.

[15] A Hamann et al. “FMTV 2016 verification challenge”. In: *Inf Process Lett* (2016).

[16] Christoph M Kirsch and Ana Sokolova. “The logical execution time paradigm”. In: *Advances in Real-Time Systems*. Springer, 2012, pp. 103–120.

[17] Wei Li et al. “Simulating a multicore scheduler of real-time control systems in simulink”. In: *Proceedings of the Summer Computer Simulation Conference*. Society for Computer Simulation International. 2016, p. 11.

[18] Camilo Lozoya et al. “Resource and performance trade-offs in real-time embedded control systems”. In: *Real-Time Systems* 49.3 (2013), pp. 267–307.

[19] MathWorks. *SimEvents Getting Started Guide, R2017b*. The MathWorks, Inc., 2017.

[20] Yutaka Matsubara et al. “An open-source flexible scheduling simulator for real-time applications”. In: *15th IEEE ISORC Symposium*. 2012.

[21] Andreas Naderlinger. “Simulating preemptive scheduling with timing-aware blocks in Simulink”. In: *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2017, pp. 758–763.

[22] Andrzej Pułka, Łukasz Golly, and Adam Milik. “SystemC hardware-software design and simulation platform based on AMBA bus”. In: *Mixed Design of Integrated Circuits and Systems (MIXDES), 2011 Proceedings of the 18th International Conference*. IEEE. 2011, pp. 644–649.

[23] Frank Singhoff et al. “Cheddar: a flexible real time scheduling framework”. In: *ACM SIGAda Ada Letters*. Vol. 24. 4. ACM. 2004, pp. 1–8.

[24] Richard Urunuela, A Deplanche, and Yvon Trinquet. “Storm a simulation tool for real-time multiprocessor scheduling evaluation”. In: *IEEE ETFA Conference*. 2010.

[25] Guoqiang Wang, Marco Di Natale, and Alberto Sangiovanni-Vincentelli. “Improving the size of communication buffers in synchronous models with time constraints”. In: *IEEE Transactions on Industrial Informatics*. IEEE. 2009.