View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Communication Centric Design in Complex Automotive Embedded Systems

Arne Hamann[1], Dakshina Dasari[2], Simon Kramer[3], Michael Pressler[4], and Falk Wurst[5]

1 Corporate Research, Robert Bosch GmbH, Germany
arne.hamann@de.bosch.com
2 Corporate Research, Robert Bosch GmbH, Germany
dakshina.dasari@de.bosch.com
3 Corporate Research, Robert Bosch GmbH, Germany
simon.kramer2@de.bosch.com
4 Corporate Research, Robert Bosch GmbH, Germany
michael.pressler@de.bosch.com
5 Corporate Research, Robert Bosch GmbH, Germany
falk.wurst@de.bosch.com

Abstract

Automotive embedded applications like the engine management system are composed of multiple functional components that are tightly coupled via numerous communication dependencies and intensive data sharing, while also having real-time requirements. In order to cope with complexity, especially in multi-core settings, various communication mechanisms are used to ensure data consistency and temporal determinism along functional cause-effect chains. However, existing timing analysis methods generally only support very basic communication models that need to be extended to handle the analysis of industry grade problems which involve more complex communication semantics. In this work, we give an overview of communication semantics used in the automotive industry and the different constraints to be considered in the design process. We also propose a method for model transformation to increase the expressiveness of current timing analysis methods enabling them to work with more complex communication semantics. We demonstrate this transformation approach for concrete implementations of two communication semantics, namely, implicit and LET communication. We discuss the impact on end-to-end latencies and communication overheads based on a full blown engine management system.

1998 ACM Subject Classification C.3 Real-Time and Embedded Systems, D.4.4 Communications Management

Keywords and phrases communication semantics, logical execution time, implicit communication, automotive, embedded systems, scheduling simulation, Amalthea

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.10

1 Introduction

Most of the innovation in the automotive area is happening on the electronics front and as a result, the number of Electronic Control Units (ECUs) in the cars has increased, with a high end car now typically having 80 to 100 ECUs. With increased functionality arises the responsibility to address the issue of handling the increased complexity in these systems. On the application front, embedded automotive applications, like the engine management system (EMS), have various application requirements (timing, data consistency, performance). Additionally, parallelizing such applications at task level to leverage the

© Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst;
licensed under Creative Commons License CC-BY
29th Euromicro Conference on Real-Time Systems (ECRTS 2017).
Editor: Marko Bertogna; Article No. 10; pp. 10:1–10:20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

capabilities of newer multi-core platforms is non-trivial, since all functional and non-functional properties of the software must be preserved after the transition. An EMS is inherently complex due to the existence of multiple tightly coupled functions executed by multiple tasks with various types of activations (periodic, sporadic, angle synchronous) exchanging data with different communication semantics. These communication semantics set rules on how and when data is communicated across functions to ensure data consistency and temporal determinism. While "implicit communication" proposed by AUTOSAR targets data consistency, Logical Execution Time (LET) has been proposed to solve the problem of temporal non-determinism by decoupling computation and communication, especially so when the software is deployed across multiple processors. To our knowledge, it is very uncommon to have uncontrolled/unregulated or direct communication between concurrent tasks in industry grade deployments and more complex communication semantics are employed.

Thus there is a need to factor-in the effects of these semantics in each phase of the system design (including task creation, task distribution, data and code mapping) for effectively using the platform and meeting all application objectives. In this work, we address the often overlooked problem in system design: a communication semantic aware timing analysis.

The problem is imminent since commercial tools that are currently adopted by the industry for timing analysis, like the Timing Architects Simulator [21] or SymTA/S from SymtaVision [9], currently *do not consider the effect of different communication semantics* and assume the most basic communication protocols (direct communication). Therefore the provided analysis may be unreliable and non-applicable in an industry setting where data consistency and temporal determinism must be ensured. Hence there is a need for *Model Transformation* that can convert complex communication semantics into equivalent direct communication mechanisms facilitating the use of these tools more meaningfully.

Contributions: In this paper, we highlight the importance of communication semantics when computing end-to-end latencies of effect chains in an engine management system. We provide a description of different execution models and communication semantics and their role in the entire design process. Next, we propose model transformation methods to transform LET and implicit communication into equivalent direct communication semantics. We also propose a concrete implementation of such a model transformation taking into account platform specific overheads. This transformed model can then be fed into existing scheduling analysis engines without worrying about the underlying communication semantics. Based on experiments conducted with SymTA/S, we then demonstrate the model transformation on a full blown engine management system, showing the impact on end- to-end latencies and communication overheads.

The rest of the document is organized as follows. We first describe the execution model in Section 2 followed by the hardware platform description in Section 3. In Section 4 we describe the communication semantics used in this paper. This is followed by Section 5, where we describe the model transformations along with discussions about the incurred communication overhead, backed up with experiments presented in Section 6.

2 Execution Model

In an earlier paper [14], an engine management system was characterized. Here we add more details, exploring the design space further. In general, automotive applications are organized as software components according to the AUTOSAR specification. These components consist of schedulable entities called runnables, which are grouped by their activation into tasks. We now describe the different elements in detail.

2.1 Runnables and Tasks

Runnables are atomic functions that constitute the smallest executable unit in a software component. They interact with each other over shared variables called labels. Each runnable is characterized by its code footprint, execution time (best case, average case, worst case), activation (as described later) and the set of labels it accesses. Typically runnables with the same activation (period) are grouped into tasks, with each runnable occupying a particular position in a task. Tasks are the units of scheduling by the operating system. Runnables with the same activation scheme can also be grouped into multiple tasks, e.g. for separation or distribution purposes.

2.2 Task Model: Activation Semantics

Different kinds of tasks co-exist in an EMS and they primarily differ in their activation triggers.

2.2.1 Periodic, Sporadic and Single Activation tasks

Periodic tasks are time triggered exactly every P time units and typically an EMS has different tasks with periods in $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ ms. Sporadic tasks are typically defined by a minimum interarrival time $minT-$ that is two activations are separated by at least $minT$ time units. Single activation tasks, as the name suggests, occur only once in the system. Typically system setup and initialization tasks fall into this category.

2.2.2 Angle Synchronous tasks

These tasks are asynchronously activated at specific angles of the crankshaft and their periodicity is determined by the speed of the crankshaft, generally represented in rotations per minute, rpm, and the number of cylinders, $nCyl$ in the engine. The period P is then given by $P = 120/(rpm * nCyl)$. As a result, these tasks are also called adaptive rate tasks since their rate changes with the speed of rotation. As seen above, these tasks are activated more frequently with increase in rotation speed.

2.2.3 Chained Tasks

Unlike other tasks that are triggered independently, these tasks are typically triggered by a predecessor task. Here the predecessor task is terminated and the thread of control is handed over to the newly activated task. Tasks can also be chained across cores.

2.2.4 Interrupts Service Routines (ISR)

Interrupt Service Routines are functions that are directly triggered by a hardware event. They usually execute their functionality directly without any task switching overhead, and thus respond faster to events.

2.3 Scheduling Model

Tasks in the EMS are scheduled using fixed priority preemptive scheduling like rate monotonic scheduling (RMS) [15]. With RMS, the priority of the tasks depends on the period (rate) – shorter the period, higher is the priority. In general, only higher priority tasks may preempt lower priority tasks. Furthermore, tasks are scheduled either in a fully preemptive

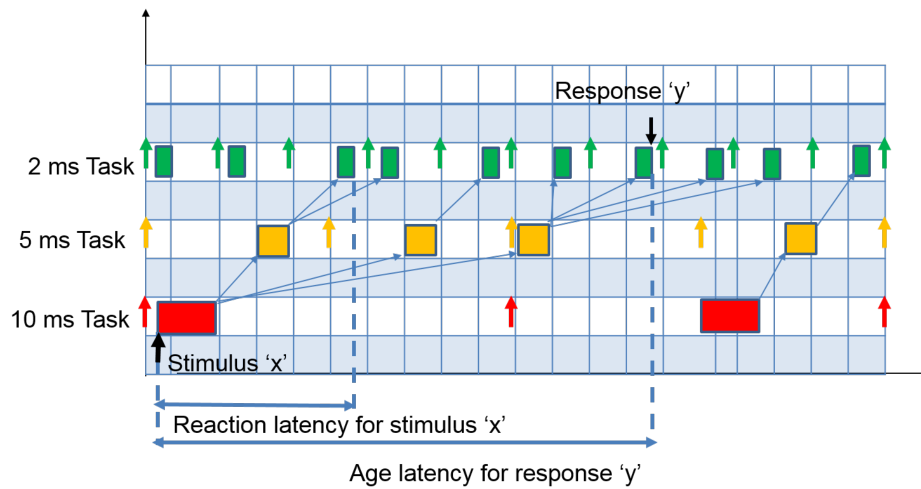


Figure 1 Age and Reaction Latency.

or cooperative manner. Tasks that participate in preemptive scheduling can preempt every other task at any time. However, tasks that are scheduled cooperatively may be preempted by higher priority cooperative tasks only at specified schedule points which correspond to runnable boundaries within a task. This ensures data consistency at the granularity of the runnables. Since preemption can occur at runnable boundaries only, it may lead to cases in which higher priority tasks are blocked by lower priority tasks still executing the current runnable. Note that preemptive tasks may preempt cooperative tasks at any point.

2.4 Event Chains

An event chain, also called effect chain or signal flow, is an abstraction of a data flow through a system. Typically an EMS has multiple event chains wherein data is sensed by the sensor nodes, passed on to control functions which act on this data and finally the output is used to configure the actuation functions to perform the desired action. Thus these event chains are a sequence of successive stimulus-response segments, where the response of one segment is the stimulus of the next segment. Each of these event chains is associated with an end-to-end latency requirement which is specified via one of the two delay semantics: an *Age* or *Reaction* latency constraint (see Figure 1). Event chains may be based on arbitrary events that occur in a system. In this paper we focus on event chains that are based on runnables as well as on start and termination events. With this, runnables could be part of different tasks and hence a stimulus and response segment will be realized by runnables that may belong to two different tasks.

2.4.1 Reaction Latency Constraint

Reaction latency denotes the time between the occurrence of the response to a specific stimulus. In other words, it denotes the time lapsed between a specific (sensor) input value or signal to a corresponding (actuator) output value, specifying how long a value or signal needs from one end to the other. A reaction latency constraint of k time units to a particular stimulus implies that the response should occur no later than k time units after that stimulus. An example from the chassis domain is the time from the brake pedal is pressed until the

brakes are activated [6]. In this work, we will deal with reaction latency constraints.

2.4.2 Age Latency Constraint

For the age latency, the freshness of the data producing the response is important and hence the focus is from the response perspective rather than from the stimulus perspective. In other words, this is the maximum time a specific output (actuator) value is available from a corresponding (sensor) input value or signal. This also equals the validity of a specific value or signal before a new value arrives. A (max) age constraint of “k” time units for a cause-effect chain mandates that for an occurrence of a response event, the corresponding input data is not older than “k” time units [6]. When for example an actuator value is periodically updated, it is of importance that the corresponding input values are not too old.

2.4.3 Multi-Rate Event Chains

The complexity of computing end to end delays (age or reaction) in event chains arises due to the fact that an event chain may consist of tasks executing at different rates or periods. Such multi-rate event chains thus often suffer from effects of i) undersampling in which the producer produces data at a higher rate than the consumer task can process it or ii) oversampling in which the consumer is activated at a higher rate than the producer. With these effects, there is a problem of either produced results being lost as they are not consumed as frequently, or duplicate inputs being acted upon by different consumer task activations. Besides, the effects of jitter during the sensing, scheduling, control and actuation, together with the possible presence of multiple clocks in the system, add to uncertainty in the delay calculations.

3 Platform Model

A cost model of the hardware platform is necessary as it is required to compute the overheads of data accesses while employing different communication semantics. In this work, we consider the AURIX [4] platform as seen in Figure 2, which is widely used for deploying automotive embedded real-time systems. The 32-bit platform consists of three cores, each equipped with a local data and code scratchpad memory. Additionally the platform provides a persistent global flash memory which is used to store code and persistent data, and a global dynamic RAM for storing non-persistent data. The memory is distributed and each core can access all scratchpads and the global memory via a crossbar interconnection.

The AURIX platform has a write buffer to decouple memory write operations from the CPU instruction execution. When the buffer is full, the priority of the buffer is raised, as a consequence of which, the buffer is flushed. Due to this mechanism, additional write access latencies are negligible and of a non-blocking nature for automotive control software categorized in [14]. Hence, we ignore additional write access latencies in the overhead calculation in this work.

Contention arises when multiple cores try to access cross-core/flash data or when high priority DMA blocks the memory. A precise analysis must ideally calculate the exact timing of concurrent accesses but dynamic hardware effects leading to execution jitters make a realistic formal calculation of these overheads nontrivial, and these aspects have been addressed to some extent in [13, 16]. Therefore, contention modelled by α is an additional overhead in the calculations which is highly dependent on the actual software and deployment configuration.

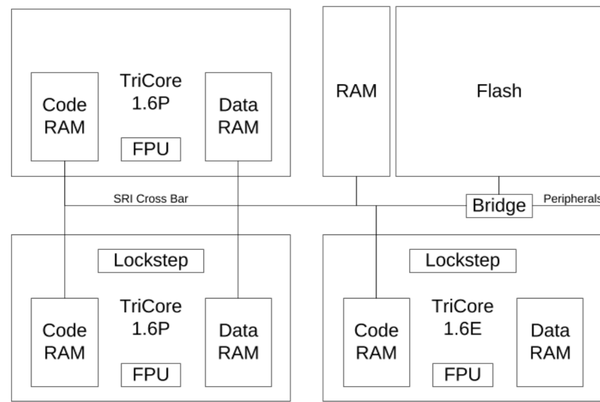


Figure 2 Simplified AURIX Architecture [20].

In this paper, to keep the focus on the semantics, we do not consider the overheads in the experimental section.

To compute the remaining memory access latency, we distinguish between read and write accesses. Data read from local scratchpads is directly available and therefore load instructions need one execution cycle[1]. In the following, we calculate the read access latency A_r, by $A_r(x) = CC_x + 1 + \alpha$, where CC_x represents the CPU access latency to memory x, and the additional cycle accounts for the execution of the load instruction. As stated above the contention α is ignored in our experiments. The CPU access latency CC_x to a local scratchpad costs zero cycles and eight cycles for a remote RAM access. The write access A_w always needs one cycle. We only consider the store instruction execution time and ignore the interconnect latencies due to the write buffer mechanism. Additionally, we assume that all labels in the system under analysis fit into the word size of the AURIX architecture (32 bits), and thus can always be fetched with a single access. This is the case for the EMS considered in this paper.

4 Communication Semantic Centric Design

As mentioned earlier, the importance of data consistency is integral to the engine management system due to the high coupling of combustion control functions [17]. Multiple functions are involved in computing the dynamic process starting from air intake to the end of the exhaust system. Highly dynamic physical values (like engine speed, manifold pressure, air temperature) are constantly exchanged at high rates between different functions. Since many sensed values are required by multiple functions, race conditions are quite probable. This problem is aggravated by the fact that the involved functions are executed by tasks allocated to different cores communicating with each other using shared data that need to be protected by mechanisms that guarantee mutually exclusive access with bounded worst-case blocking time. To cope with this problem, platform mechanisms enforcing data consistency and temporal determinism are employed in industrial systems to constructively ensure functional correctness. The definition of the term data consistency includes two different dimensions. The first dimension is the consistency in value, meaning that the value of a variable/message is not affected by action outside the current execution context. The other dimension is

[1] We ignore micro-architecture effects, e.g. result latencies, here, which could lead to additional delays.

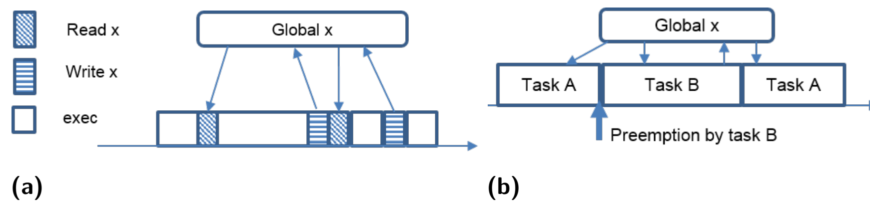


Figure 3 (a) Direct access: task performs read and writes on a global variable during its execution. (b) Example showing how task A uses 2 different values at different points in execution.

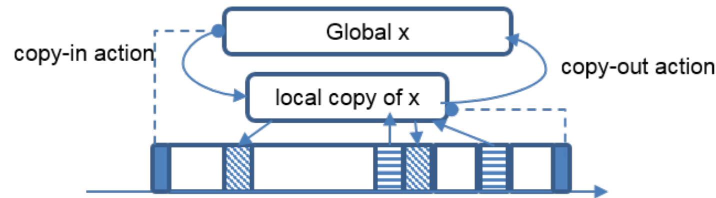


Figure 4 Implicit communication: CopyIn and CopyOut runnables read and write copies at beginning and end of the task.

consistency of a variable with other variables. This means that multiple variables are only valid if all (within the consistency scope) are stemming from the correct (same) point in time. In this work we deal with the first dimension of the data consistency problem by employing inter-task and intra-task communication semantics as described below.

4.1 Inter-Task Communication Semantics

4.1.1 Explicit or Direct Communication

This semantic, often also called direct access, allows unrestricted access to shared variables (labels) across tasks. As seen in Figure 3a, the task works on the global variable of the label. This may work for labels which are strictly read-only, but with labels which may be overwritten, data inconsistency may result. Interleaving of task activations will result in different values of the data. In a single core setting, a simple scenario is one in which a preempting task changes a shared value and so the preempted tasks works on two different values at two different points in time, leading to inconsistencies as seen in Figure 3b.

4.1.2 Implicit Communication

This semantic, proposed by AUTOSAR, is primarily focused towards maintaining data consistency to avoid the pitfalls of explicit communication. It essentially follows a read-execute-write paradigm – Implicit communication mandates that the task always makes local copies of the shared data it needs at the beginning of its *execution*, works on the local copies and writes the data back at the end of its execution (see Figure 4). This ensures that the task works on the same copy throughout its execution, and also preserves consistent state of the data.

From the access latency perspective, all the variables that are read during task execution will have to be pre-fetched into local memory from its source and then the task may execute by referencing readily available local copies (see Figure 5), hence not incurring the cost of remote accesses multiple times.

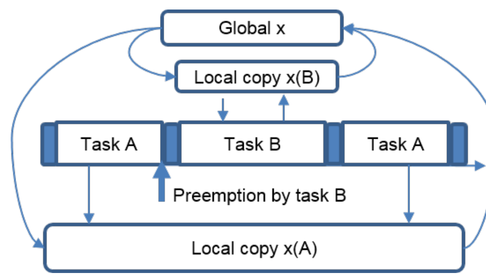


Figure 5 Implicit communication: each task works on its local copies.

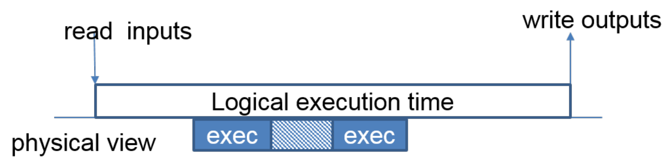


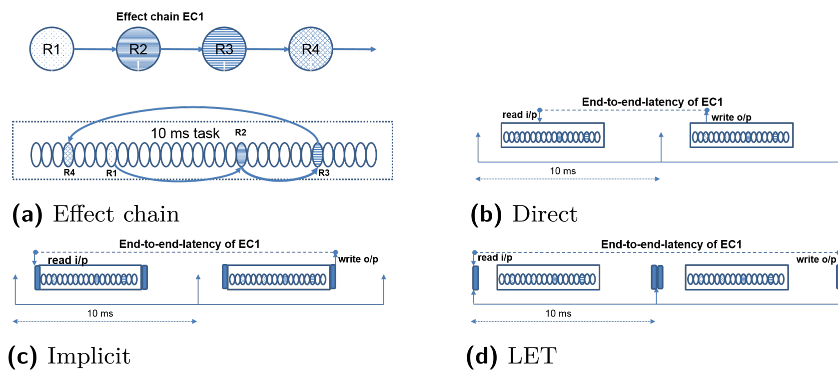
Figure 6 LET: The observed output is independent of the time a task executes in its LET interval.

The access latency in case of implicit communication is therefore dependent on multiple factors including, the cost of access to remote and local memory, number of accesses to the label during one execution to the local memory, and the period/activation rate of the task. However on the memory storage front, more local storage is required, since for every task which accesses the label, an extra local copy is required.

4.1.3 Logical Execution Time Model

Logical Execution Time (LET) was introduced with the time-triggered programming language Giotto [11]. It is a real-time programming concept which ensures temporal determinism by decoupling computation and communication. The problem with an unconstrained communication method, i.e, allowing tasks to read and write arbitrarily is non-determinism due to "execution jitter". The result is highly dependent on possible interferences of other tasks executing within a tasks activation interval (say from its release to the end of its period). The effects of this jitter becomes more prominent in event chains, leading to large variations in end-to-end delays. The LET model is robust against these jitters by enforcing strict communication rules. With the LET model, tasks always read data at the beginning of the activation interval and write data at the end of the activation interval (see Figure 6). As with implicit communication, LET requires that a local copy is available for each variable accessed by a task. Using LET, the observable temporal behavior of a task is independent from its physical execution. That is irrespective of the exact time a task executes within its execution interval, the result will be always available only at the end of its activation interval. LET also ensures portability, i.e, the same behavior of the tasks when migrated to another hardware (core), integrability on addition of newer software and interoperability, which is verified by deterministic communication.

With LET, the end-to-end latencies in case of synchronous stimuli is always equal the sum of the periods of the tasks involved in the chain. However, with asynchronous stimuli it may happen that each task in the effect chain executes as early as possible in its activation window but the data arrives just after it begins execution (meaning it is operating on an older value of the data). Thus the newer data is consumed only one time period later. The



(a) Effect chain **(b)** Direct

(c) Implicit **(d)** LET

Figure 7 The effect chain spans over two task activations due to backward communication.

same scenario could occur with every pair of tasks in the chain. Eventually, the worst case latency in the case of such asynchronous arrivals is *twice the sum of the periods of all the tasks in the chain.*

LET thus leads to longer latencies in event chains. But on the other hand, with LET, there is no need for complex synchronization mechanisms to handle race conditions or priority inversions, given its well-defined semantics.

4.2 Intra-task communication

Runnables within a task may communicate with each other in two different ways. With forward communication, the producer runnable completes before the consumer runnable and hence there is no delay in getting the latest data by the consumer and communication is therefore fast.

With backward communication however, the consumer runnable executes before the producer, and thus there is a delay of one period in this case to receive the latest data.

4.3 Event chains and communication semantics

Figure 7 summarizes how the different communication mechanisms affect the resulting latency of a simple event chain in which all runnables are mapped to one task, but there is a backward communication, since the consumer is positioned before the producer in the task, as shown in Figure 7a, and has a different order in the event chain.

5 Model transformation

Currently existing timing analysis tools like SymTA/S [9] typically do not consider the implementation overheads of the underlying communication semantics or are only equipped to handle direct communication. As a result, these tools cannot be reliably used in actual industry-grade evaluations where typically complex semantics like *implicit* or *LET* communication are deployed. Additionally, since each design phase like task creation, task distribution, data/code mapping and computation of end-to-end latencies of event chains are heavily influenced by the underlying semantics, the implementation-specific details cannot be ignored. We transform the models as shown below – these models can then be analyzed using state-of-the-art timing analysis tools to effectively compute the end-to-end latencies along event chains by considering the implementation specific overheads for copying data.

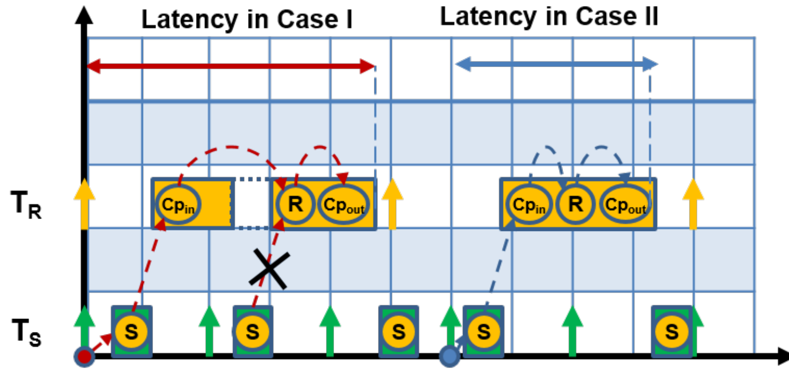


Figure 8 Example event chain exhibiting non deterministic latencies with *implicit communication semantics*. Case I: shows that newer data from the sending runnable might be discarded due to the copy-in mechanism. Case II: shows shorter latency in a different execution scenario.

5.1 Transformation of event chains to model latency effects

In this section we describe how event chains consisting of runnables communicating according to *implicit* and *LET communication* semantics can be transformed to equivalent event chains in terms of end-to-end latencies assuming direct communication between all runnables. The transformations are explained on *event chain segment* basis[2], meaning that they can be mixed along event chains to address systems with heterogeneous communication semantics, which is usually the case in real systems.

5.1.1 End-to-end latency with implicit communication semantic

As mentioned earlier, *implicit communication* mandates that the task makes local copies of the shared data it needs at the beginning of its execution, works on these local copies and writes the data back at the end of its execution. In order to realize this aspect of *implicit communication*, auxiliary Copy-in Cp_{in} and Copy-out Cp_{out} runnables are added at the beginning and the end of each task, respectively. The Cp_{in} runnable prefetches all the labels that are needed by the task during execution, into local memory. Then, during task execution, only these local copies are accessed. Similarly, all the labels that are written by a task are written into local memory and then, after task execution are written back to the original labels by Cp_{out}. The labels that a task needs during execution are extracted by program analysis, which is possible in such embedded programs, where pointers and other complex programming artifacts are not encouraged.

Figure 8 visualizes how event chains need to be transformed to take *implicit communication* semantics into account.

Each event chain segment $\{S \rightarrow R\}$ where the stimulus runnable S and response runnable R, *belong to different tasks*, needs to be replaced by three new consecutive event chain segments with stimulus and responses pairs and the modified event segments are: $\{\{S \rightarrow Cp_{out}(S)\}\{Cp_{out}(S) \rightarrow Cp_{in}(R)\}\{Cp_{in}(R) \rightarrow R\}\}$, where the notation $Cp_{out}(S)$ represents the copy-out runnable of task containing S. For completeness, we similarly model the initial copy-in to the first runnable in the event chain and the copy-out of the last runnable of the event chain.

[2] We assume that event chains are defined on runnable level.

As can be observed in Figure 8, Case I, the copy-in operations potentially increase the end-to-end latency along event chains. More precisely, this is the case when the sending runnable S updates the label after the copy-in procedure of the task containing the receiving runnable R has taken place and before R is executed. In the given example, this leads to the situation where fresher data of T_S is discarded, which would not be the case with direct communication. Obviously, such situations leading to increased latency due to *implicit communication* semantics do not occur systematically as is visualized in Case II. However, similar data race effects occur due to the copy-out mechanisms.

Please note that it is assumed that there is *exactly* one task responsible for writing each label. In case multiple tasks with different priority levels perform write accesses to the same label, there can be data races when all of them try to perform a copy-out to the master copy of the same variable. Lock mechanisms for exclusive update rights must then be provided for maintaining consistency. Also task priorities that prevent data race conditions can be exploited to optimize the number of created label copies.

5.1.2 End-to-end latency with LET communication semantic

Modeling LET communication is similar to modeling *implicit communication*, in that copy in and copy-out tasks must be augmented to the model. However, positioning these auxiliary tasks correctly is important to achieve the expected behaviour. The activation rates of the pair of communicating tasks (relatively harmonic or non-harmonic) and their priorities will have an influence on the resulting end-to-end latencies. Note that with LET, tasks can communicate data *only* at the beginning and the end of their activation interval. Since we consider a system that employs fixed priority preemptive processing, we prioritize the copy operations by elevating their priorities to the highest in the system. Hence the copy-out task is given the highest priority while the next highest priority is assigned to copy-in tasks.

In this paper, *LET communication* semantic is applied at task level, i.e. deterministic communication is ensured at task activation boundaries. For this reason, event chain segments with the stimulus runnable S and the response runnable R need to be transformed if, either, they belong to different tasks, or if they belong to the same task and exchange data with backward communication (see Section 4.2).

Figure 9 visualizes the transformation, where T_S and T_R denote the tasks containing the runnables S and R, respectively. We transform the segment $\{T_S \rightarrow T_R\}$ to $\{T_S \rightarrow Cp_{out}(T_S)\}, \{Cp_{out}(T_S) \rightarrow Cp_{in}(T_R)\}, \{Cp_{in}(T_R) \rightarrow T_R\}\}$. For completeness, we similarly model event chain segments modelling the initial copy-in to the first runnable and the final copy-out from the last runnable. Please note that these tasks are only needed logically to mimic *LET communication* semantics for analysis engines that are based on direct communication. Therefore, the execution time attributed to those copy tasks is equal to zero. How the overhead induced by these copy operations can be taken into account is discussed later.

5.2 Calculating the communication overhead

In this section we explain how to calculate the communication overhead of the system with respect to the different data synchronization mechanisms. We use the cost model of the widely used AURIX [4] platform explained in Section 3. Obviously, there are different implementation alternatives on software level for the mentioned communication mechanisms. Depending on the chosen alternative, runnables for modeling the overhead need to be added

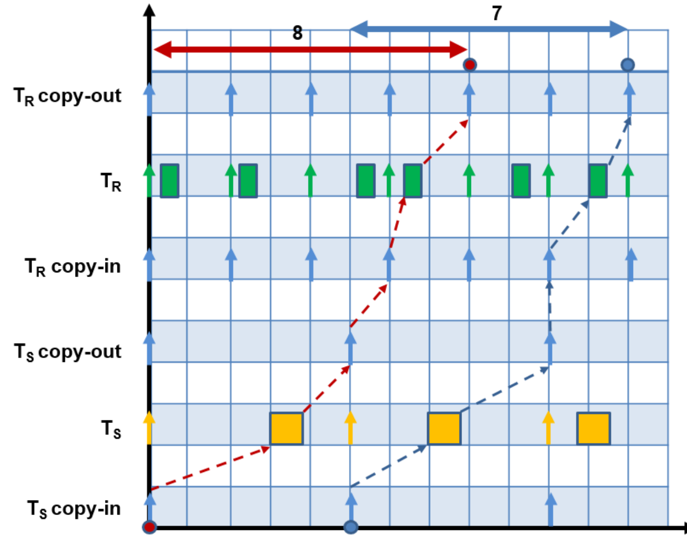


Figure 9 *LET communication* semantic is achieved by 1) adding highest priority copy-out and second highest priority copy-in tasks with 0 execution time for each task, and 2) placing these in event chains between communicating tasks.

to different tasks. We assume that the execution time for each runnable contains code-fetching overhead, but excludes any load/store execution times.

We assume that constants are mapped to the global RAM and that accesses are never cached. Variables have exactly one writer but can have multiple readers and are mapped to the local scratchpad of the core hosting the writer (task or runnable). We also assume that labels are not persisted in registers when multiple accesses to labels in local scratchpads are made, and therefore a load instruction is necessary.

The communication cost per task is calculated as described in Section 3 for each read label access and considers one cycle for each write accesses as described earlier. Let $\eta_i(l)$ be the number of accesses of the label l from task τ_i during one execution. Let π_l denote the memory where label l is mapped. Then,

$$\begin{aligned} C_{com} &= \sum_{l \in R_i} \eta_i(l) * A_r(\pi_l) + \sum_{k \in W_i} \eta_i(k) * A_w(\pi_k) \\ &= \sum_{l \in R_i} \eta_i(l) * A_r(\pi_l) + \sum_{k \in W_i} \eta_i(k) \end{aligned} \tag{1}$$

where R_i and W_i denotes the set of labels read and written by task τ_i and $A_r(\pi_l)$ describes the time for a read access from memory π_l and correspondingly $A_w(\pi_l) = 1$ for any π_l. Eventually C_{com} is added to the effective execution time of the task. Note that for the experiments, we similarly apply the above formula at runnable level to compute their gross execution time. For direct communication, the overall communication cost is given by Equation 1 and so $C_{dc} = C_{com}$. However, additional costs are incurred for the implicit case as seen below.

5.2.1 Communication cost for implicit communication

As explained in Section 5.1.1, two runnables are added to each task for realizing *implicit communication*: Cp_{in} for copying all labels read inside the task into local copies, and Cp_{out} for writing back all local copies of labels written inside the task.

The cost for Cp_{out} depends on the number of written labels, since the access latency overhead is minimal due to the write buffers of the considered AURIX platform (refer Section 3).

To create copies of labels, a read operation followed by a write operation is executed to load a label into the local scratchpad. The read operation loads the label into the CPU register and the write operation stores it into the local scratchpad. The runtime C_{in} for the Cp_{in} runnables is calculated by

$$C_{in} = \sum_{l \in R_i} (A_r(\pi_l) + A_w(x)) = \sum_{l \in R_i} A_r(\pi_l) + |R_i| \quad (2)$$

where again π_l describes the memory location of the label l, which could be mapped locally or remotely. $A_w(x)$ denotes the cost of writing a label to the local scratchpad memory x, which like any other write access always costs 1 cycle. Note that we do not include the frequency of accesses here since a copy is done once for each task that accesses the label.

The cost C_{out} for the write-back of the label copies in Cp_{out} is calculated by the below equation where y denotes the target memory location.

$$C_{out} = \sum_{l \in W_i} (A_r(\pi_l) + A_w(y)) = |W_i| * 2\,. \quad (3)$$

Due to the fact, that the copies are located in the local memory and so $A_r(\pi_l) = 1$, the overhead for the copy is always two cycles per written label.

In addition to the copy runnables, we need to consider the access latencies within the task to compute the overall communication cost $C_{implicit}$ for implicit communication. The formula to calculate the communication cost is given by Equation 1, but considering that the labels to be accessed are now *located in the local scratchpad incurring a single cycle access latency*. Only the constants remain in the global RAM. In the following equation, C_{com} includes the accesses from the runnables of the task whereas C_{in} and C_{out} contain the costs of the auxiliary copy runnables, leading to:

$$C_{implicit} = C_{com} + C_{in} + C_{out}\,. \quad (4)$$

5.2.2 Communication cost for LET communication

In contrast to implicit communication, with LET, communication happens at period boundaries. Thus we add additional runnables for copying the required data for each task into the corresponding local scratchpad memories as with implicit communication, but we trigger these copy operations at different time points. In our proposed implementation, this copying is facilitated via highest priority interrupts.

The first step of the model transformation is to add highest priority interrupts for each periodic task in the system. These interrupts are activated with the same period as the tasks they correspond to.

In the second step of the model transformation, runnables performing the necessary copy operations to realize *LET communication* have to be added to the system *for each pair of communicating tasks*. In the following, the two communicating tasks are denoted by T_A and T_B, whereas their periods are denoted by P_A and P_B.

In order to describe the timing of the copy operations, the notions of *prescale* and *offset* that can be associated to runnables are required. The *prescale* describes the activation frequency of the runnable in relation to the activation frequency of the task it is mapped to. For instance, a *prescale* of 2 denotes that the runnable is executed only every second

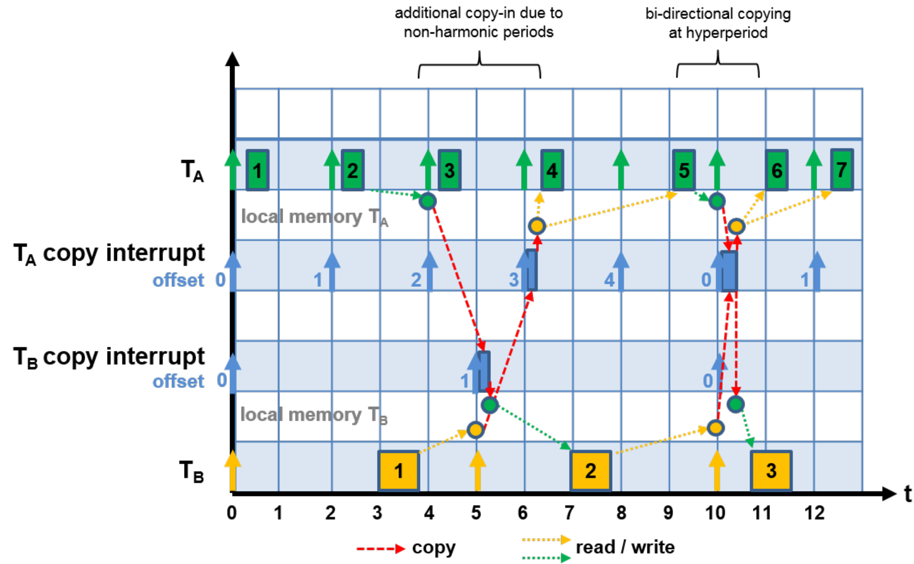


Figure 10 Copy operations (realized by high-priority interrupts) that need to be performed between two tasks with non-harmonic periods in order to realize *LET communication*.

time the parent task is activated. In relation to that, the *offset* describes the initial shift of the first execution starting to count from 0. For instance, a *prescale* of 3 and an *offset* of 1 denote that the runnable is executed at the activations $2, 5, 8, etc.$ of the parent task.

Figure 10 shows the necessary copy operation for the more general case where T_A and T_B have non-harmonic periods. In this case $P_A = 2$ and $P_B = 5$.

5.2.2.1 Costs for label accesses

Firstly, as in implicit communication, the execution times of both T_A and T_B are increased by C_{com}, as in Equation 1, to factor-in the label access costs. Note that for the calculation of C_{com}, it has to be taken into account *that all tasks operate on local copies*.

5.2.2.2 Bi-directional copy at hyperperiods

Then, a bi-directional update of the local label copies needs to be performed at the end of the hyperperiod of the two communicating tasks. This is necessary since according to the *LET* semantic, new written data becomes visible for all reading tasks at that point. These bi-directional copy operations are performed by a single runnable that is mapped to the copy interrupt corresponding to the faster task. Given that $P_A < P_B$, the following copy operations need to be performed:

- The local copies of labels written by T_A and read by T_B need to be updated in the local scratchpad of the core that T_B is mapped to. The cost for these operations is denoted by C_{AB}.
- The local copies of labels written by T_B and read by T_A need to be updated in the local scratchpad of the core that T_A is mapped to. The cost for these operations is denoted by C_{BA}.

An important point in the following cost computations is that labels are mapped to the local memory of the core hosting the task that writes the label. Let S_{AB} be the set of labels

written by task T_A and read by T_B. Likewise S_{BA} denotes the set of labels written by task T_B and read by task T_A. Then given that $A_w(x) = 1$ for any destination memory x, and π_l is the local memory of the core where T_A is mapped to, we have:

$$C_{AB} = \sum_{l \in S_{AB}} (A_r(\pi_l) + A_w(x)) = 2 * |S_{AB}| . \tag{5}$$

Similarly given π_k is the local memory of the core where T_B is mapped to and y is the local memory of the core where T_A is mapped to, we have:

$$C_{BA} = \sum_{k \in S_{BA}} (A_r(\pi_k) + A_w(y)) = \sum_{k \in S_{BA}} A_r(\pi_k) + |S_{BA}| . \tag{6}$$

5.2.2.3 Handling non-harmonic tasks

In the case of harmonic task periods, the copy operations at the hyperperiod are sufficient to realize *LET communication* semantics. However, in case of non-harmonic task periods additional copy operations need to be performed. Figure 10 visualizes the extra operations necessary for *LET communication* between the tasks T_A and T_B with periods $P_A = 2$ and $P_B = 5$, respectively.

As observed, the results of T_A's second execution become visible at time instant 4, and, thus, before the second activation of T_B at time instant 5. For this reason, the local copies of the labels written by T_A that are read by T_B must be updated in the local memory of the core T_B is mapped to *before* its second activation is executed. In the implementation discussed in this paper, the necessary copy operations are performed by a runnable that is mapped to the copy interrupt corresponding to T_B with $prescale = 2$ and $offset = 1$. In the reverse communication direction, the results of T_B's first execution that become visible at time instant 5 need to be made available for T_A's fourth activation at time instant 6. This is achieved by adding a runnable to the copy interrupt corresponding to T_A with $prescale = 5$ and $offset = 3$.

Algorithm 1 Calculate additional copy points for tasks with non-harmonic periods

Input: Periods P_A and P_B of two tasks T_A and T_B involved in *LET communication*
Output: List of copy points (prescale p and offset o) at which T_A needs to copy-in new available data from T_B

1: $CopyPoints \leftarrow \{\}$
2: $bCur \leftarrow P_B$
3: **while** $bCur < lcm(P_A, P_B)$ **do**
4: $\quad aCur \leftarrow P_A \times \left\lceil \frac{bCur}{P_A} \right\rceil$
5: $\quad bCur \leftarrow P_B \times \left\lceil \frac{aCur}{P_B} \right\rceil$
6: $\quad$ **if** $aCur \neq lcm(P_A, P_B)$ **then**
7: $\quad\quad CopyPoints \leftarrow \left(\mathrm{p} = \frac{lcm(P_A,P_B)}{P_A}, \mathrm{o} = \frac{aCur}{P_A}\right)$
8: $\quad$ **end if**
9: **end while**
10: **return** $CopyPoints$

Algorithm 1 describes how these uni-directional additional copy points can be calculated in the general case. Given the periods of two communicating tasks, the algorithm returns a list of copy points (tuples of prescale and offset) at which the first specified task needs to fetch the results of the second task in the above explained manner.

The communication costs for each of these copy points is the calculated by Eq. 5 and Eq. 6. The calculations must be done for every pair of periodic tasks that exchange data. The communication costs for one execution of the task differ on whether no updates are needed, a bi-directional hyperperiod data exchange is performed, or non-harmonic task pairs perform intermediate copy updates.

6 Experimental Results

6.1 Experimental Setup

We base our experiments on the model of an engine management system provided in the context of the industrial challenge of the WATERS 2017 workshop [2], an extension of that provided in [14, 1]. The earlier model is augmented to specify the frequency of label accesses from each runnable. The platform consists of 4 cores, running at 200 MHz and executing a set of periodic tasks, an angle synchronous task as well as interrupt service routines (ISR). Please note that the model transformations proposed in this paper are only applied to the periodic tasks. The application consists of 1250 runnables grouped into 21 tasks/ISRs which communicate via 10000 labels. Since the focus is on modeling the communication semantics, we assume that the mapping of labels and tasks is already provided. Constant calibration data, i.e. labels that are only read but never written, is mapped to the global RAM. Variables, i.e. labels that are written by a single task and potentially read by multiple tasks, are mapped to the local memory of the core hosting the writer task. We further assume that the underlying platform does not support data caching for the data mapped into the global RAM. We assume synchronous releases of all periodic tasks for these experiments, whereas the angle synchronous task and all ISRs are asynchronously released. The calculated end-to-end latency does not contain extra delays for sampling effects with external stimuli. In case of implicit communication, the beginning of the event chain is the point in time when the copy-in runnable of the task containing the first runnable of the effect chain is executed.

We use the AMALTHEA [3] meta-model for describing the engine management system. AMALTHEA provides model elements to express event chains, different tasks models, different constraints and the hardware platform. We implemented the above proposed transformations to realize the direct, implicit, and LET communication semantics. This transformed AMALTHEA model is then fed into the SymTA/S tool, wherein the execution behavior is simulated over multiple runs to generate the access latency distributions we present.

6.2 Comparison of the end-to-end latencies

In this experiment, we compare the end-to-end latencies of effect chains across the three communication semantics. For the experimental results we highlight the applicability on two important types of chains for evaluation.

6.2.1 Single-rate effect chain with backward communication

We analyze effect chain $EC1$, composed of four runnables, all positioned in a single task with activation of 10ms similar to the example in Figure 7a. There is backward communication in this scenario, since the relative order of the producer and consumer in the effect chain is different from the positions in the task.

The different latencies observed for around 3500 runs are shown in Figure 11. As seen, the latency for direct communication is the least, since the available input is read immediately

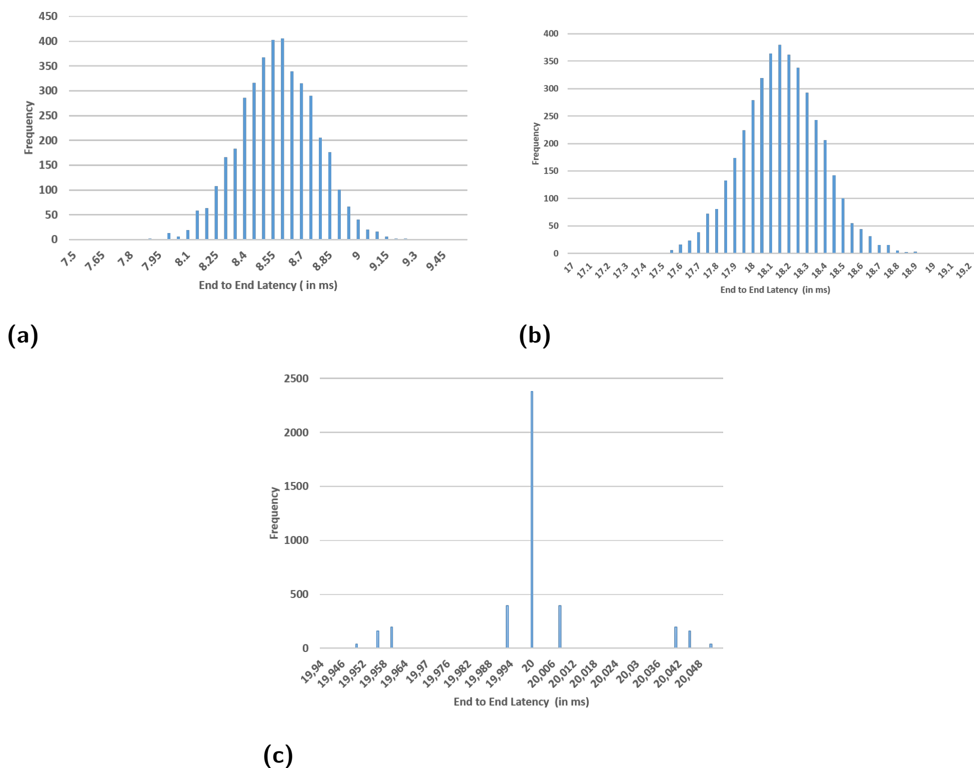


Figure 11 Comparison of end-to-end latency distribution between direct, implicit and LET communication.

by the first runnable of the effect chain and the produced output is immediately reflected across the event chain segments, therefore reducing the effective end-to-end latency. However with implicit and LET semantics, as seen in Fig. 7c and Fig 7d, there is, as expected, an increase in the end-to-end latencies in the chain. As described earlier the output using implicit communication is globally available only at the end of the task execution, while with LET at the end of the activation interval. Note that with LET irrespective of where it executes, the end-to-end latency is always centered around 20ms with a negligible execution jitter in the order of microseconds. This jitter is due to the execution times of the copy-in and copy-out runnables which are executed with highest priorities at the points in time the LET communication takes place. Obviously, this leads to deviations compared to the idealized LET semantics.

6.2.2 Multi-rate chains

We next analyze effect chain $EC2$ composed of 3 runnables, with task periods 100, 10 and 2 ms. The resulting end-to-end latency distributions are presented in Figure 12. As expected, direct communication results in the least latency (say x), while implicit communication and LET have higher (almost 4x and 14x) latencies. This increased latency for implicit communication is attributed to the fact that results are not directly available after runnable completion, but rather after task completion. Obviously, this leads to situations where several instances of the receiving task are "missed" before the receiving runnable can read the data. The negative effect on the end-to-end latency is aggravated with increasing response time of

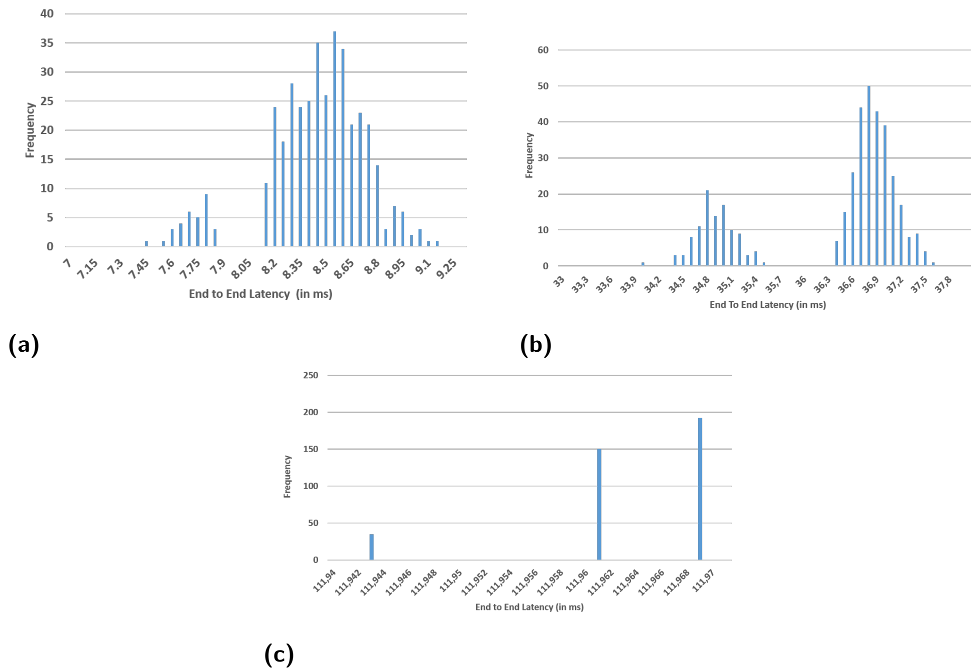


Figure 12 Comparison of latency overheads and variations across direct, implicit and LET communication.

the sending task. The increased latency for LET is purely attributed to the involved task periods since results are available only at the end of the task interval. As seen, the average latency is almost equal to the sum of the periods of the involved tasks. The reason for the jitter is the same as for event chain $EC1$.

6.3 Comparison of the communication overhead

In this experiment, we compute the overhead incurred by applying different mechanisms (see Figure 13). In order to compute it, we consider the cost of accesses of all labels in the application as described earlier, without factoring-in contention. The higher communication overhead for direct access is because remotely stored labels are fetched each time they are accessed. In contrast, references to local copies in implicit and LET communication result in slightly lower overhead. However, each time a new task accesses a label, new copies are to be made – and so, while direct communication is influenced by the frequency of label accesses, implicit and LET communication are influenced by the number of tasks accessing the label, since copies need to be maintained for every task.

6.4 Summary of experiments

We observe that data consistency via implicit communication or temporal determinism via LET comes at a cost of higher end-to-end event chain latencies and reduced communication overheads – but this trade-off is reasonable considering the intangible gains towards functional correctness of the systems and development, validation and deployment efforts.

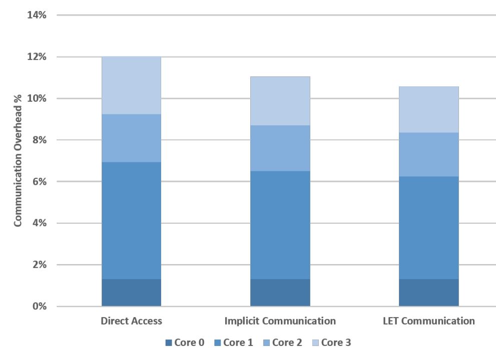


Figure 13 Comparison of communication overheads.

7 Related Work

For automotive embedded systems, in addition to meeting deadlines of individual tasks, meeting end to end latency requirements of event chains is crucial. These end to end timing requirements are described in standards such as AUTOSAR [5] and EAST-ADL [7]. In this regard, [18] compute end-to-end latencies considering the AUTOSAR implicit communication model and describe age and reaction constraints while introducing the first-to-first, first-to-last, last-to-first, last-to-last semantics. Kluge et al. [12] show how an LET based approach can facilitate compositionality by extending the MOSSCA multicore OS for LET. Henzinger et al. [10] proposed a methodology that supports distributed realtime code generation for distributed real-time systems considering LET. Farcas et al. [8] further demonstrate how a transparent task distribution is facilitated via LET and hence irrespective of where a task is mapped on a distributed system, the logical timing behavior is undisturbed. Pellizoni et al. [19] also proposed and analyze tasks under the "Predictable Execution Model" which have semantics similar to implicit communication. Our work is different in that we emphasise on the end-to-end latency implications of event chains in a real-world system. We also focus on how an existing tool can be extended to express these semantics.

8 Conclusion

In this paper we presented communication semantics used in the industry and their role in ensuring data consistency and temporal determinism in real-time multi-core embedded systems, and in particular an engine management system. We proposed model transformations to increase the expressiveness of existing tools and demonstrated the resulting impact on the system behavior. The consideration of communication semantics in widespread modeling approaches allowed us to evaluate their significant impact using a well-known timing analysis tool. These results can guide system designers to evaluate different optimization goals like minimizing communication overheads or reducing event-chain latencies, as well as ensuring deterministic system behavior.

References

1 S. Kramer A. Hamann, D. Ziegenbein and M. Lukasiewycz. Demonstration of the FMTV 2016 timing verification challenge. *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 00:1, 2016.

2 A. Hamann, S. Kramer, M. Pressler, D. Dasari, F. Wurst, and D. Ziegenbein. The industrial challenge of WATERS 2017 provided by Robert Bosch GmbH. URL: `http://waters2017.inria.fr/challenge/`.

3 AMALTHEA. An open platform project for embedded multicore systems. URL: http://www.amalthea-project.org.

4 AURIX. Aurix – safety joins performance. URL: http://www.infineon.com.

5 AUTOSAR – Spec. of Timing Extensions, 2014.

6 M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte. Synthesizing job-level dependencies for automotive multi-rate effect chains. In *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 159–169, Aug 2016. doi:10.1109/RTCSA.2016.41.

7 EAST-ADL – Domain Model Specification, 2014.

8 E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. *SIGPLAN Not.*, 40(7):31–39, June 2005. doi:10.1145/1070891.1065915.

9 R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *Computers and Digital Techniques, IEEE Proceedings -*, 152(2):148–166, March 2005. doi:10.1049/ip-cdt:20045088.

10 T. A. Henzinger, C. M. Kirsch, and S. Matic. Composable code generation for distributed giotto. *SIGPLAN Not.*, 40(7):21–30, June 2005. doi:10.1145/1070891.1065914.

11 C. M. Kirsch and A. Sokolova. *The Logical Execution Time Paradigm*, pages 103–120. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

12 F. Kluge, M. Schoeberl, and T. Ungerer. Support for the logical execution time model on a time-predictable multicore processor. *SIGBED Rev.*, 13(4):61–66, November 2016.

13 L. Kosmidis, D. Compagnin, D. Morales, E. Mezzetti, E. Quinones, J. Abella, T. Vardanega, and F. J. Cazorla. Measurement-Based Timing Analysis of the AURIX Caches. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–11, 2016. doi:10.4230/OASIcs.WCET.2016.9.

14 S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2015.

15 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973. doi:10.1145/321738.321743.

16 E. Mezzetti M. Ziccardi, A. Cornaglia and T. Vardanega. Software-enforced Interconnect Arbitration for COTS Multicores. In *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, volume 47 of *OpenAccess Series in Informatics (OASIcs)*, pages 11–20, 2015. doi:10.4230/OASIcs.WCET.2015.11.

17 L. Michel, T. Flaemig, D. Claraz, and R. Mader. Shared SW development in multi-core automotive context. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, January 2016. URL: https://hal.archives-ouvertes.fr/hal-01284591.

18 F. Nico, R. Kai, N. Johan, and J. Jan. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *Int. Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2008.

19 R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, April 2011. doi:10.1109/RTAS.2011.33.

20 D. Reinhardt and G. Morgan. An embedded hypervisor for safety-relevant automotive e/e-systems. In *SIES*, 2014.

21 Timing Architect. Model-based development tools for embedded multi-core systems. URL: https://www.timing-architects.com.