Compositional Analysis of the WATERS Industrial Challenge 2017

Kai-Björn Gemlau, Johannes Schlatow, Mischa Möstl and Rolf Ernst
Institute of Computer and Network Engineering, TU Braunschweig
{gemlau,schlatow,moestl,ernst}@ida.ing.tu-bs.de

I. Introduction

With the introduction of multicore processors for real-time embedded systems, the influences of the memory subsystem got in focus for timing analysis. Therefore, the 2017th FMTV challenge extends the previous one with additional information on the memory subsystem. We show how Compositional Performance Analysis (CPA), as introduced in [1] and [2], can be used to compute worst-case timing guarantees for a given processor system with focus on implicit communication or LET. Further we discuss the different implementation options that are not clarified by the given Amalthea model and their influences on our analysis. Particularly, we utilize the non-preemptive FIFO behavior of the memory arbitration to limit the amount of possible interference.

II. Summary of Contributions

In the pages that follow, we present our contributions to WATERS Industrial Challenge 2017. First, we propose how we implement implicit and Logical Execution Time (LET) communication with copy operations, double buffering and additional tasks (Section III). In Section IV, we will have a look at the assumptions we derive from the challenge model in order to perform a timing analysis of the memory accesses, compute the tasks' response times using pyCPA [3] and, ultimately, calculate worst-case end-to-end latencies for particular cause-effect chains. More precisely, our solution handles memory contention on the crossbar and blocking from critical sections on the Local RAM (LRAM). W.r.t. end-to-end latencies, we cover both semantics (data age and reaction time) without restrictions on the activation patterns/periods of the tasks. We provide the details of our pyCPA analysis extension in Section V and evaluate the memory overheads (in time and space) as well as the achieved latency bounds for the given cause-effect chains.

III. Implementing implicit and LET communication

With implicit communication, a task takes a local copy of all data it is working on (copy-in) from the shared regions, executes all runnables, and writes the results back (copy-out) to its shared region when it is finished (cf. Fig. 1). Fig. 2 illustrates the tasks' memory structure in the LRAM (cf. Task 3): The private region holds all the local copies of the read and write labels. During the copy-out phase, the write labels are copied from the private to the shared region.

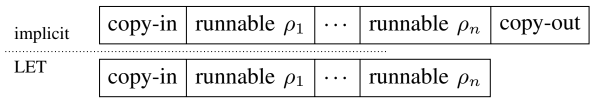


Fig. 1. Task structure for implicit and LET communication with copy-in/copy-out phases and runnable execution.

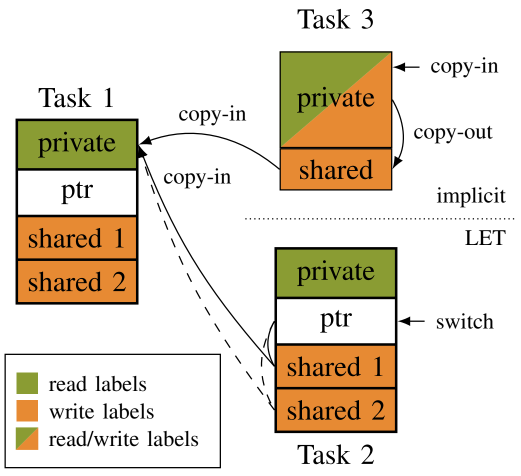


Fig. 2. Task memory structure in the implicit and LET communication scenario.

LET separates the computation and communication of a task such that the write labels are published at specific time intervals rather than at the end of a task's execution. We refer to these as LET intervals. As depicted in Fig. 1, the copy-out phase is omitted in contrast to the implicit communication. As presented in [4], LET can be efficiently implemented as a form of double buffering, which avoids time intensive copy operations. This is illustrated in Fig. 2 (cf. Task 1&2): In the copy-in phase, Task 1 copies the required read labels from global memory to its private region. In addition, the task holds two separate shared regions (double buffer). During the runnables' execution, the labels are either read from the private region or written to one of the shared regions. A pointer (ptr region) stores which of the shared regions is currently published (and therefore not written to). At the corresponding LET intervals, a separate instance will switch the ptr in an atomic operation. Note that we assume this switch never happens while the task is executing. This can be achieved by adjusting the LET intervals conservatively and furthermore be ensured by a response-time analysis that verifies that a task always finishes within its LET interval. For each task, we add a higher-priority task that executes at the LET intervals and implements the switching between the buffers. We refer to this tasks as LET tasks. Furthermore, analysis must ensure that the pointer is not switched while a copy operation from the shared region is in progress. Note that the double buffering requires an additional memory access to the ptr region for the task itself and all tasks that access the shared regions during the copy-in phases.

Both approaches induce a memory overhead in the LRAMs while the DRAM usage remains constant. For implicit communication, the LRAM needs to be partitioned such that a private region can store all labels accessed by the task. With the assumed LET implementation, two regions for the write labels and one private region for the copied labels are needed. In summary, this equals the memory overhead of implicit communication plus one data word for the ptr region. Further evaluation of memory and runtime overhead takes place in Section V.

IV. Modeling and assumptions

For our system model we make a set of assumptions which are not explicitly covered by the challenge model:

- The LRAMs are dualported, i.e. the local core and the crossbar can access the LRAM at the same time.
- Task internal communication between runnables is always done via the private memory area.
- Copy-in/copy-out phases are implemented as critical sections.
- One 32bit memory access is one arbitration in the crossbar.
- FIFO arbitration in the crossbar is fair.

The remainder of this sections moves on to describe in further detail how we model the memory subsystem and the entire multi-core processing system in order to perform a CPA and calculate end-to-end latencies.

A. Modeling of the memory subsystem

The presented processor consists of four cores, each with its own LRAM and a shared DRAM. Access to foreign LRAMs or DRAM is done through a central crossbar. Fig. 3 shows the architecture with the access latencies marked at the arrows. Notice that any memory access from the crossbar to an LRAM or DRAM takes one cycle whereas the cores experience an additional (protocol) overhead of 8 cycles for each access to the crossbar.

The FMTV challenge assumes the following memory mapping:

1) Labels that are only read (and never written by any runnable) are stored in the DRAM.
2) Labels are only written by one runnable and stored in the core-local LRAM belonging to the core to which their task is mapped.

In consequence each copy-in operation into an LRAM sees contention at the FIFO scheduler of the crossbar in addition to the access latencies between crossbar and LRAMs.

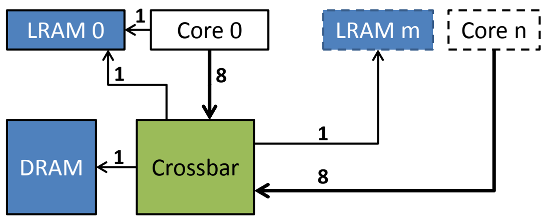


Fig. 3. System overview with access latencies.

1) Local Memory Contention: As illustrated in Fig. 1 the copy-out operation is performed at the end of a task. We assume that copy operations are implemented in a critical section, meaning no context switching can be done on the core while it performs copy operations. As tasks are scheduled in a Static-Priority Preemptive (SPP) manner, we only need to consider the largest critical section on a core as blocking in the timing analysis. Furthermore, as copy-out operations only happen in the local memory, no contention from the crossbar can be seen (due to the dualported LRAM).

2) Crossbar Contention: Due to our assumptions on the implementation, we only have to consider crossbar accesses for copy-in operations, as for both implicit and LET, the copy-out operation only takes place in the local memory. As a result, the contention on the crossbar for a task's copy-in phase only depends on the number of other cores and the length (number and size of labels) of the phase. For the arbitration of the crossbar we assume a fair First-in-First-out (FIFO) scheduler, where the worst-case scenario is that each foreign core places exactly one access in the scheduler's queue right before the access under consideration. As the crossbar has a specified memory word width of 32bits, each label that is less or equal than this, is represented by one access. Labels that are larger than 32bits are split into accesses of this granularity and each access is arbitrated separately. Note that labels that are produced by a runnable/task on the same core do not need to be copied-in via the crossbar, and do not see contention from its scheduling.

B. Timing Model

W.r.t. CPA we need to consider two types of resources: First, the SPP scheduled cores, that execute the tasks containing the runnables, and second, the crossbar. The crossbar is treated as a FIFO scheduled resource.

Fig. 4 shows the two CPA model variants, depending on whether LET or implicit communication is applied. In the implicit communication variant (the right core in Fig. 4), each task τ_i has a corresponding read tasks r_i on the crossbar resource. Here r_i models all the accesses necessary to copy-in remote labels. Note that we group all remote label accesses into one task, although we assume that each access can be interfered with from other cores. Due to our fair FIFO queuing assumption, the grouping is possible. The effect is that the worst-case response time (WCRT) of a task on the crossbar resource only depends on the number and size of accesses (and thus the transferred labels and their size).

In contrast to that, the LET case is shown on the left resource in Fig. 4. Each task τ_i comes with an independent high-priority LET task p_i on the same processing resource, as well as a read task r_i on the crossbar to copy-in remote labels. The read task takes care of copying-in labels that are produced on other cores and is handled as in the implicit communication case. Furthermore, the LET task p_i takes care of switching the pointer(s) at the LET interval boundaries. The pointer in the LRAM ensures that always the current public dataset is read by other cores.

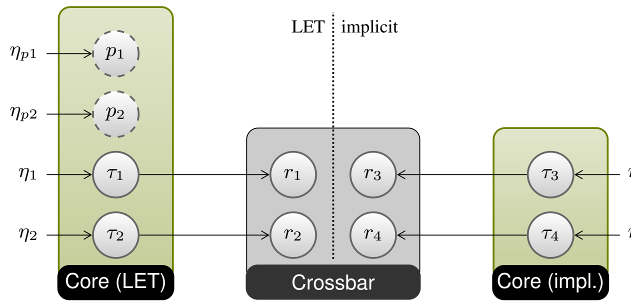


Fig. 4. Timing model variants, depending on the communication type. The left variant shows the necessary tasks for LET, whereas the right illustrates the implicit communication variant.

As explained earlier, the WCRT of a task on the memory resource only depends on the number and size of the transferred labels due to our fair FIFO queuing assumption. Consequently, we do not need the input event models of the memory tasks (and therefore no iteration in the CPA analysis between memory and processing resources). The best-case/worst-case execution times of the processing tasks are therefore calculated as a sum of the following terms:

- best-case/worst-case execution times of each runnable,
- best-case/worst-case response time (BCRT/WCRT) of the corresponding memory task,
- memory transfer time for read/write accesses to the LRAM during copy-in,
- memory transfer time for label accesses during runnable execution,
- (implicit) memory transfer time for the read/write accesses to the LRAM during copy-out.

Note that for LET communication, we also need to include the read accesses to the ptr regions in the memory tasks and during the runnables' execution.

Since we assume implicit deadlines and synchronous activation of all tasks, we can set the offsets of the LET tasks as follows: We reserve an *LET-band* Δ_{LET} at the end of each period. The length of Δ_{LET} is statically defined such that every LET task can execute once during this interval. The offset for each LET tasks is thus defined as $T_i - \Delta_{LET}$. By additionally replacing the implicit deadlines with these offsets, we ensure that an LET task can not overlap with its computation task.

C. *End-to-end latency*

One part of this challenge consists in the computation of end-to-end latencies for particular cause-effect chains. Such a cause-effect chain is a sequence of communicating runnables. For each pair of neighboring runnables (ρ_i, ρ_{i+1}) we either have intra-task (they execute in the same task) or inter-task communication. Intra-task communication does not impose any additional delay if the runnables are in correct order. Only if ρ_{i+1} executes before ρ_i in the task (backward communication), we must consider the additional delay that results from the fact that the data produced by ρ_i is processed in the next activation of the task. This can also be seen as inter-task communication between two instances of the same task. We therefore model the cause-effect chain as a sequence of tasks in which we only consider inter-task communication. Moreover, in case of LET communication, the inter-task communication only takes place between the LET tasks.

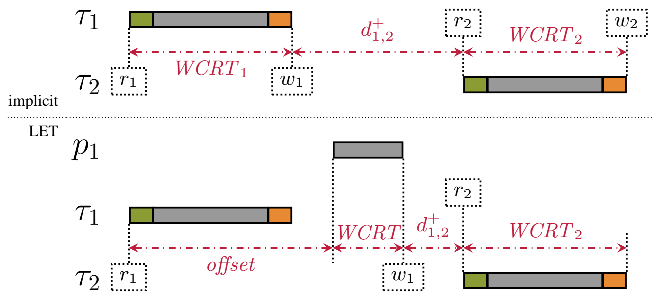


Fig. 5. Latency computation for implicit and LET communication from τ_1 to τ_2 based on WCRTs, offsets and maximum distances between write (w_i) and read (r_i) times. Note that p_1 represents the LET task for τ_1.

We base our end-to-end latency analysis on the evaluation of read and write events of all labels consumed and produced by an LET task. In particular, we formulate a cause-effect chain as a sequence of read/write events and bound the distance between each pair of neighboring events. Fig. 5 shows such an exemplary cause-effect chain (r_1, w_1, r_2, w_2) for implicit and LET communication. It also illustrates the write time for task τ_1 in the LET case, that resides at the end of p_1.

In Section V, we will present the details how the event times are computed from the best-case/worst-case response times (BCRT/WCRT) and relative activation offsets of the (LET) tasks as illustrated in Fig. 5. Furthermore, we will provide upper bounds for the maximum distance (d^+ in the Fig. 5) between write and read times for each neighboring pair of tasks, and between the read and write times for the data from each task. In particular, we formulate different bounds for the data age and reaction time semantics such that end-to-end latencies can be computed as the sum of these maximum distances in both cases.

V. Analysis and Evaluation

For the analysis of the challenge scenario, we use pyCPA [2]. Among others, it computes the worst-case/best-case response times for each task τ_i, which we denote by R_i^+/R_i^- in the remainder of this section. The calculated response times serve as basis for the end-to-end latency analysis of the given cause-effect chains. The source code which implements the challenge model and analysis is publicly available[1].

Our analysis results are grouped as follows: Section V-A provides the response-time results on task level for implicit communication and LET. Furthermore, we elaborate on the necessary additional execution time for copy operations described in Section IV-B. The additional memory overhead for the LET implementation is presented in Section V-B. Section V-C gives the detailed result for the three effect-chains that are included in the Amalthea model.

A. *Timing results*

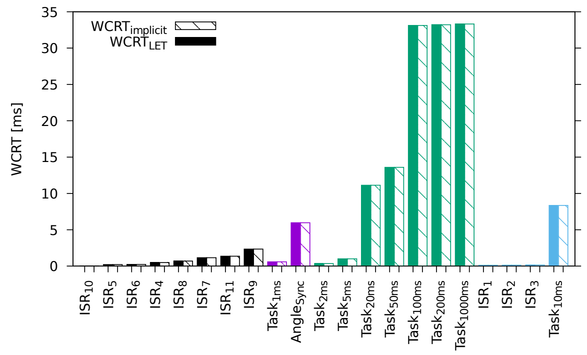
The results of the task-level timing analysis are presented in Fig. 6. Since the given Amalthea model leads to processor loads over 100%, we used a common scaling factor of 0.7 for the execution times of all runnables. This is equivalent to a 40% increase of the system frequency and results in a maximum load of 94% on Core 1.

Fig. 6a shows the calculated WCRTs for all tasks and ISRs for LET (filled boxes) and implicit communication (striped boxes). One can see that the LET overhead on the WCRT is negligible due to the low-overhead double-buffering. The WCRT is mainly influenced by the priority, e.g. a task sees interference from all higher priority ones, and the size of the largest critical section on each core. The results show that interference is mainly driven by higher priority tasks, as the critical section blocking is even observed by the highest priority task on each core.

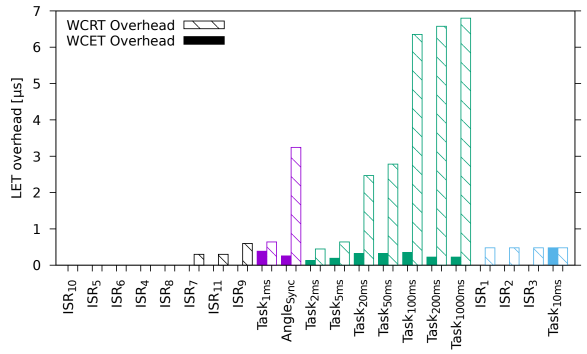
Fig. 6b provides a more detailed view of the timing overhead due to the LET implementation. The overhead is displayed for the Worst-case execution time (WCET) (filled boxes) as well as the WCRT (striped boxes). The WCET is only influenced by the modified copy-in phase, as it needs one label access for each foreign producer task. As an example, the 10ms task reads data from 15 foreign tasks and therefore has the highest WCET overhead. The analysis is slightly pessimistic, as it conservatively assumes that every read originates from a task on a foreign core. Keep in mind that the WCRT overhead as displayed in Fig. 6b is visible because the scale is more than an order of magnitude lower than in Fig. 6a. To investigate this, Fig. 6c provides additional information. On the left y-axis, the ration of each task's WCRT over its period is displayed, which gives an indication of the task's activity (filled boxes). In addition to the task activity, the right y-axis provides a comparison of the task's read-activity during the copy-in phase. Therefore, the share of copy-in time compared to the task's WCET is displayed.

Combining this information, the WCRT overhead in Fig. 6b can be evaluated. For example, the Angle-Sync task has a high WCRT overhead due to its high utilization since it has the lowest priority on Core 1 and is influenced by the additional LET Task for the 1ms task. In contrast to that, the 200ms and

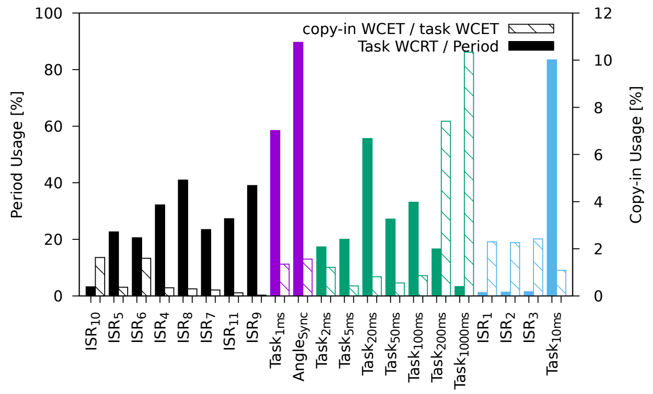
[1]https://bitbucket.org/pycpa/waters2017



(a) Worst-case response times of tasks.



(b) Overhead of the LET implementation on WCET and WCRT.



(c) Execution profiles of the tasks.

Fig. 6. Timing analysis results. The colors represent the used processor core and for each core the priority decreases from left to right.

1000ms tasks have a low utilization but read more labels thus seeing higher interference.

B. *Memory overhead*

As we described in Section III, the core-local LRAMs can be divided in a shared (write) and a private (read) region. For the implicit communication, all data from foreign memories is loaded in the private region during the copy-in phase. Table I shows the required memory sizes in 32Bit words per core and task for share (S), private data produced by foreign cores (P_{rw}) and read-only data from GRAM (P_{ro}). We assume a fixed 32Bit alignment here, e.g. an 8Bit label requires a 32Bit memory cell. One can see that the ISRs only produce data for other tasks and read static data from GRAM (not shown in Table I). Therefore, their private regions only consist of read-only data. An optimization would be to preload read-only data for Core 0 at boot-time, avoiding unnecessary copy-in phases, resulting in reduced interference on the crossbar by 25%.

TABLE I
LRAM CONSUMPTION IN 32BIT WORDS

Core 0				Core 2			
Task	S	P_{rw}	P_{ro}	Task	S	P_{rw}	P_{ro}
ISR_10	8	0	11	Task_200ms	182	196	40
ISR_11	13	0	9	Task_1000ms	291	209	131
ISR_6	10	0	11	Task_100ms	1218	974	809
ISR_7	18	0	18	Task_20ms	1017	913	964
ISR_4	7	0	28	Task_5ms	41	23	58
ISR_5	5	0	21	Task_50ms	206	222	143
ISR_8	5	0	20	Task_2ms	28	11	94
ISR_9	7	0	3				
$\sum_{implicit}$	73	121		$\sum_{implicit}$	2983	4787	
$\sum_{LET}$	155	121		$\sum_{LET}$	5973	4787	
Core 1				Core 3			
Task	S	P_{rw}	P_{ro}	Task	S	P_{rw}	P_{ro}
Angle_Sync	742	838	475	ISR_2	10	0	9
Task_1ms	76	71	146	ISR_3	12	0	13
				ISR_1	12	0	18
				Task_10ms	2075	1815	979
$\sum_{implicit}$	818	1530		$\sum_{implicit}$	2109	2834	
$\sum_{LET}$	1638	1530		$\sum_{LET}$	4222	2834	

For the LET case, sizes of the private regions remain the same and sizes for the shared regions are doubled due to the double buffering. In addition, one write pointer is required for each task. The resulting memory sizes are shown in Table I as $\sum_{implicit}$ and $\sum_{LET}$. One can see that for LET on Core 2, the resulting LRAM consumption is above 10.000 words. The challenge model specifies 256KByte of LRAM per core which is in principle sufficient to hold this as there is no information about the code size of the runnables.

In contrast to implicit communication, LET adds additional runtime overhead for the LET tasks and pointer access. We assume a fixed execution time of 100ns for each LET task, as it only updates the buffer pointer in the local LRAM. The additional runtime overhead for the normal tasks depend on their amount of data consumption by foreign tasks. For each task where data is read from, one access to the foreign LRAM is required to read the buffer pointer before the copy-in phase is started.

C. End-to-end latency

As was motivated in Section IV-C, we compute the end-to-end latency by the distances between the read (r) and write (w) events of the corresponding tasks. A cause-effect chain is given as a sequence of these events in the form $(r_0, w_0, \dots, r_i, w_i, r_{i+1}, w_{i+1}, \dots, w_n)$. Note that the indexes only resemble the relative order of tasks in the sequence. In order to generalize the latency computation, we must therefore formulate upper bounds for the distance between any r_i and w_i and between any w_i and r_{i+1}. For backward intra-task communication, we further allow cause-effect chains in the form of $(r_0, w_0, \dots, r_i, w_i, r_i, w_i, \dots, w_n)$. We therefore also require bounds between a write event w_i and read event r_i of the same task. As a result, the end-to-end latency of the chain under analysis is calculated as the sum of maximum distances between each pair of events in the given sequence. For this, we assume the read/write events to align with the activation/completion events of a task.

Before we summarize the latency results from our analysis, we formulate the upper bounds for the three different types of read/write event distances: the *read-to-write* distance $(w_i - r_i)$, the *inter-task write-to-read* distance $(r_{i+1} - w_i)$ and the *intra-task write-to-read* distance $(r_i - w_i)$.

1) Analysis of cause-effect chains: In order to bound the read-to-write distance, we must distinguish implicit and LET communication. For implicit communication, this distance is easily bounded by the worst-case response time of the task τ_i. On the other hand, for LET communication, we only need to account the logical execution time of τ_i, i.e. its period:

$$w_i - r_i \leq \begin{cases} R_i^+ & (implicit) \\ LET_i & (LET) \end{cases} \tag{1}$$

Here, we assume that offsets have been chosen carefully such that the LETs can be enforced correctly (cf. Section IV-B).

Regarding the intra-task write-read distance, we leverage the assumption of implicit deadlines (i.e. the response time of a task τ_i is never greater than its period) such that it is easily bounded as follows:

$$r_i - w_i \leq \begin{cases} \delta_i^+(2) - R_i^- & (implicit) \\ 0 & (LET) \end{cases} \tag{2}$$

where $\delta_i^+(2)$ is available from the CPA and denotes the maximum distance between two consecutive activations of τ_i, and R_i^- denotes the best-case response time of τ_i.

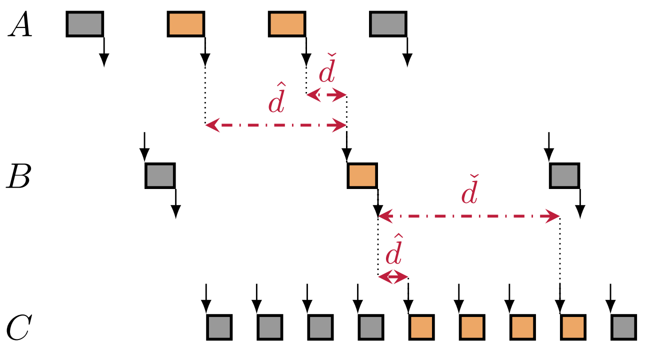


Fig. 7. Write-read distances for implicit communication. Data produced by A is undersampled by B, and subsequently oversampled by C.

With respect to the inter-task write-read distance, we need to compute the distance between the relevant jobs of different tasks in the cause-effect chain. Depending on the latency semantics (data age or reaction time), the jobs considered relevant may differ as soon as over- or under-sampling comes into play. Let us have a look at Fig. 7, which shows a chain of three implicitly communicating tasks (A, B, C) with under-sampling between task A and B and over-sampling between task B and C. Between each pair of tasks, the figure shows two semantically different distances: the forward distance $\hat{d}$ from a write event to the next read event, and the backward distance $\check{d}$ from a read event to the nearest preceding write event. Based on these distances, we can formulate the maximum inter-task write-read distance for data age and reaction time as follows:

$$r_{i+1} - w_i \leq \begin{cases} \hat{d}_{i,i+1} & (\mathit{reaction\ time}) \\ \check{d}_{i,i+1} & (\mathit{data\ age}) \end{cases} \tag{3}$$

For under-sampling (i.e. period $T_{i+1} > T_i$), $\hat{d}^+_{i,i+1}$ is bounded by the maximum distance between two activation events of τ_{i+1}, i.e. $\delta^+_{i+1}(2)$. Similarly, for over-sampling, $\check{d}^+_{i,i+1}$ is bounded by the maximum distance between two completion events of τ_i, which we denote by $\tilde{\delta}^+_i(2)$. This can be bounded by the maximum input event distance $\delta^+_i(2)$ plus the output jitter [1]. In the other cases, the data is exchanged between the neighboring jobs as shown in Fig. 7 (Eq. (4c) and (5c)).

For LET, this results in (logical) zero-time communication between τ_i and τ_{i+1} if both have have a harmonic period. Otherwise the worst case distance is given by the modulo of both periods. For implicit communication and data age semantic, we differentiate whether τ_i is an ISR or whether it is synchronously activated with τ_{i+1}. In the former case, we bound the distance by the distance between two output events of the ISR. In the latter case, τ_{i+1} can either read the data from the previous activation of τ_i or from its current activation. Again, for non-harmonic periods, an worst-case offset has to be taken into account (Eq. (7b), (7c)). Here, due to the synchronous activation of both tasks and the priority-based scheduling, we can assume that if τ_{i+1} finishes after τ_i, it can only start after τ_i finished and will therefore read the new data. For the reaction time with oversampling, a slight optimization can be done if τ_i has a shorter WCRT R^+_i than the period of of τ_{i+1} (Eq (6a)). Consequently, we calculate $\hat{d}$ and $\check{d}$ as follows:

$$\hat{d}^+_{i,i+1} = \begin{cases} \delta^+_{i+1}(2) & \text{if } T_{i+1} > T_i & (4a) \\ \Phi_{i,i+1} & \text{else } (\mathit{LET}) & (4b) \\ f^+_{i,i+1} & \text{else } (\textit{implicit}) & (4c) \end{cases}$$

$$\check{d}^+_{i,i+1} = \begin{cases} \tilde{\delta}^+_i(2) & \text{if } T_{i+1} < T_i & (5a) \\ \Phi_{i+1,i} & \text{else } (\mathit{LET}) & (5b) \\ b^+_{i,i+1} & \text{else } (\textit{implicit}) & (5c) \end{cases}$$

with $\tilde{\delta}^+_i(n) = \delta^+(n) + R^+_i - R^-_i$ and $\Phi_{k,l} = T_k \bmod T_l$.

$$f^+_{i,i+1} = \begin{cases} \delta^+_{i+1}(2) - R^-_i & \text{if } R^+_i \leq T_{i+1} & (6a) \\ \delta^+_{i+1}(2) & \text{else} & (6b) \end{cases}$$

$$b^+_{i,i+1} = \begin{cases} \tilde{\delta}^+_i(2) & \text{if } \tau_i \ \mathit{is\ ISR} & (7a) \\ \delta^+_i(2) - R^-_i + \Phi_{i+1,i} & \text{if } R^+_i \geq R^+_{i+1} & (7b) \\ \Phi_{i+1,i} & \text{else} & (7c) \end{cases}$$

2) Results: The challenge model specifies three different cause-effect chains that emphasize prominent cases: Chain 1 only consists of intra-task communication with forward and backward dependencies. Chain 2 contains inter-task communication with oversampling, i.e. decreasing periods whereas Chain 3 covers the undersampling case and starts with an ISR.

Table II summarizes our data age and reaction time results for the three cause-effect chains given in the challenge model. Note that we calculated the latencies from the activation of the first task involved in the chain to the write action of the last task in the chain. As there is no under-/oversampling involved in Chain 1, the data age and reaction time results are the same. LET provides slightly better results, as it does not suffer from execution jitter. As also expected, for Chain 2, the data age is longer than the reaction time due to the oversampling and vice versa for Chain 3 (cf. Fig. 7). More interesting is the impact of LET and implicit communication on the latencies of Chain 2 and 3. In general, LET results in higher latency bounds. One reason for this is that LET delays the write time of the last task in the chain, which, in our interpretation, determines the end of the chain. This can be observed at Chain 3, which ends with the 50 ms task and receives an additional delay of approx. the difference between the task's LET (50 ms) and its WCRT.

TABLE II
END-TO-END LATENCY RESULTS

	Data Age			*Reaction Time*		
Chain	implicit	LET	Δ	implicit	LET	Δ
1	23	20	3	23	20	3
2	188	222	−34	53	112	−59
3	14	53	−39	65	104	−39

REFERENCES

[1] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System Level Performance Analysis - the SymTA/S Approach," in *IEE Proceedings Computers and Digital Techniques*, 2005.

[2] J. Diemer, P. Axer, and R. Ernst, "Compositional Performance Analysis in Python with pyCPA," in *3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Jul. 2012.

[3] (2010-2017) pyCPA. [Online]. Available: https://bitbucket.org/pycpa/pycpa

[4] M. Beckert, M. Möstl, and R. Ernst, "Zero-Time Communication for Automotive Multi-Core Systems under SPP Scheduling," in *Proc. of Emerging Technologies and Factory Automation (ETFA)*, Berlin, Germany, Sep 2016. [Online]. Available: http://dx.doi.org/10.1109/ETFA.2016.7733563