Efficient Run-Time Environments for System-Level LET Programming

Kai-Björn Gemlau, Leonie Köhler, Rolf Ernst
Institute of Computer and Network Engineering, TU Braunschweig
Braunschweig, Germany
{gemlau, koehler, ernst}@ida.ing.tu-bs.de

***Abstract*—Growing requirements of large industrial and automotive software systems have initiated an ongoing move from monolithic and tightly integrated run-time environments (RTE) to virtual platforms implemented on fewer domain computers with heterogeneous physical architectures. This trend has given rise to new programming paradigms to enable specification, implementation and supervision of software systems that are predictable and robust under interference and change. One of those paradigms, the Logical Execution Time (LET), is now part of the automotive software standard, AUTOSAR. While originally applied to single shared-memory multicore processors, System-level LET (SL LET) extends this approach to virtual and distributed platforms providing a powerful paradigm for CPS in future industrial systems. This contribution explains and demonstrates the resulting challenges to the RTE and the opportunities to improve its efficiency, in particular the communication stack.**

***Index Terms*—Run-time environments, System-Level Logical Execution Time (SL LET), network stack**

I. Introduction

Upcoming Cyber-Physical System (CPS) architectures for industrial and automotive applications are heterogeneous, combining high performance multicore architectures with accelerators such as for graphics or machine learning, and larger memory systems for data-centric applications. High-resolution sensors including cameras and LiDARs demand high data rates, applying pressure on real-time communication. CPS programming is getting closer to high-performance computer programming, differing mainly in their large interdependent application function networks, real-time and safety-requirements. The resulting system complexity is a major challenge to runtime environments (RTEs), software architecture and network stacks.

The automotive industry has recently introduced the Logical Execution Time (LET) paradigm to master synchronization in an increasingly complex timing behavior of parallel architectures [1], [2]. It has helped to efficiently programming multicore architectures with lock-free communication. There are further advantages, such as reduced jitter in cause-effect chains. Applying LET to larger systems, however, is constrained by its requirements to tight synchronization and implicit narrow deadlines. An extension of LET to the system level overcoming these limitations has been proposed [3], [4]. A core concept of this extension, System-Level Logical Execution Time (SL LET), is the interconnect task that bridges the local LET domains. As part of this concept, the interconnect tasks monitor data life times and support data synchronization.

The research was partly funded by the German DFG under ER 168/30-2 and by the German BMBF under 16EM00285.

While providing a concept for scalability, it is not obvious that SL LET can efficiently be applied to high-performance architectures with high data rates. For that purpose, it must efficiently support the transport of large data objects with bounded latency. That is a consequence of LET, which requires setting communication timing under worst-case conditions. Traditional UDP and TCP network stacks, as needed in upcoming architectures, expose complex activation structures with wide worst case-bounds. The large data objects to be communicated add to the challenge, stressing the buffer mechanisms with hundreds of frames, each. It is not possible to just simplify the stack, because practical CPS systems include a substantial amount of non-LET traffic leading to a mixed-criticality setting that needs a complete IP/TCP/UDP stack.

With this consideration, we can formulate the following research challenge:

1) SL LET needs a network stack with a worst-case interconnect task latency at or below the average latency of a traditional stack, such that the potential overhead is mitigated. The traditional reference stack must not lose data for a valid comparison. The target are large data objects.
2) The communication timing of all non-LET traffic should not be seriously affected.
3) An increased overall load is permitted. The stack function available to SL LET communication can be limited to predefined links providing only the UDP service needed for SL LET interconnect tasks.

In earlier work, we have proposed to approach the challenge with a filter stack that works on preselecting the SL LET traffic [5]. This paper describes a proof-of-concept implementation and provides experimental data on the improvement compared to a popular light-weight IP stack [6]. As an important run-time environment for data synchronization, we use the concepts of Data Distribution Service (DDS) that is employed in Robot Operating System 2 (ROS 2) and in AUTOSAR. It is a data-centric publisher-subscriber (DCPS) model [7].

We use the predictability and simplicity of SL LET interconnect tasks to define a light-weight filter stack that speeds up SL LET communication and controls buffer size, yet can be combined with a standard IP stack. It is small enough to be implemented on a light-weight companion core that is found

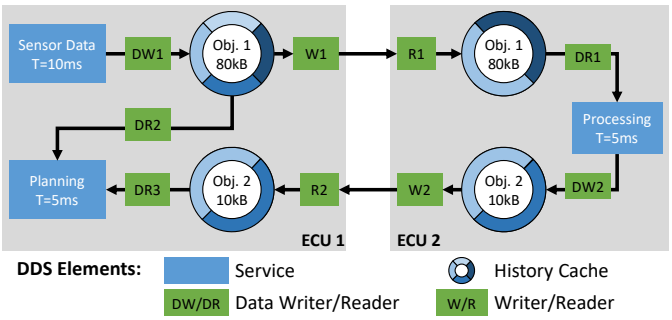

Fig. 1. Case study with processing service on a remote ECU

on many high-performance architectures for CPS, such as an ARM Cortex-R/M found on, e.g. the Renesas R-Car[1] or NXP S32[2] platform.

The paper is organized as follows: We clarify the problem statement with measurements based on a use-case in Section II. Section III gives a brief review on related work in the context of SL LET, middleware design and traffic filtering. We highlight the key concepts of our improved software architecture in Section IV and evaluate the proof-of-concept implementation in Section V. Section VI concludes the paper.

II. Problem Statement

In the following we provide a use-case related to the *Autoware.Auto* [8] benchmark and show how this can be implemented with a DDS-like middleware. We take measurements to demonstrate the negative effects of interfering background traffic with an unmodified lightweight IP (lwIP) network stack.

A. Case Study

The case study is related to the processing of LiDAR frames in the Autoware.Auto benchmark [8] and comprises three services, communicating through a DDS middleware. For trajectory planning, a classifier service identifies each element in the LiDAR point cloud as either ground- or non-ground point. Non-ground points may represent obstacles like cars, trees or buildings and are therefore sent to an object detection and tracking service. This is a processing intensive task and a candidate for outsourcing on a dedicated electronic control unit (ECU). The results are passed back to ECU 1, where the planning service combines ground-points and detected objects. At this point, the two cause-effect chains join and the quality of the planning depends on the deviation of the data ages of both, the local and the remote input.

Figure 1 shows the three services as well as the key elements of the DDS middleware. The objects are stored in *history caches* of configurable size, allowing to access different *samples* of the same object. The middleware itself is logically split in two layers, namely the DCPS communication and the underlying DDS Interoperability Wire Protocol (DDSI), as shown in Figure 2. Readers and writers are part of DDSI and utilize UDP sockets provided by the underlying network stack to synchronize samples between distributed history caches. The services use data readers and data writers to access the samples. To compensate the latency of the remote cause-effect chain, the size of the history cache of *object 1* on ECU 1 has to be chosen large enough such that the planning service is able to combine processed data (*object 2*) with raw data (*object 1*) of the same data age.

Apart from the data transport, the middleware can handle the service abstraction to ensure that only services with matching interfaces can be connected (data type representation), manage automatic service discovery to reduce the configuration effort and monitor if the services and the network fulfill their specified behavior.

However, there are also important use-cases for TCP like over-the-air software updates, which might be handled by a dedicated background task. We implemented such a background task on each ECU, communicating with a Linux host to demonstrate the effects of possible interference.

B. Interference Measurements in the Classic Stack

In the following we illustrate the possible interference that non-critical background traffic (TCP) may impose on the latency critical DDS traffic (UDP) that is required to transfer samples between history caches. We have implemented the case study from Figure 1 on two Xilinx Zynq Ultrascale+[3] boards (each equipped with a XCZU9EG multiprocessor system-on-chip (MPSoC)), which are connected to an Ethernet network. On each board, a setup as described in Figure 2 runs on an ARM Cortex-A53 performance core. As the basic software, we use the µC/OS-II [4] real-time operating system (RTOS) as well as the open-source lwIP TCP/IP stack [6], both already highly configurable to meet resource constraints in various embedded platforms. On top of that, we implemented a DDS-like demo middleware that features data-readers and writers (the interface between services and the history caches) as well as readers and writers, which use UDP sockets to synchronize data-sets between the two boards. The three services are activated periodically, synchronized in a hyper period. A writer is triggered as soon as the service has produced new data.

Depending on the priority assignment and scheduler, the network stack may suffer from interference by other applications running on higher priorities, e.g. when scheduled with the widely used rate-monotonic scheduling (RMS) (e.g. AUTOSAR) or with earliest deadline first (EDF) scheduling. On the other hand, it may itself affect applications running on lower priorities. To take this into account, we added a high priority task with a period of 1ms, causing around 35% load and stressing the level-1 caches. At the level of a background priority, we implement a task that communicates via TCP to simulate a functionality like a software update which is not time critical. It passes a 1MB data block in a logical ring between both Zynq boards and a connected Linux computer.

[1]https://www.renesas.com/us/en/solutions/automotive/soc/r-car-h3.html
[2]https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/s32-automotive-platform:S32

[3]https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html
[4]https://www.micrium.com/rtos/kernels/

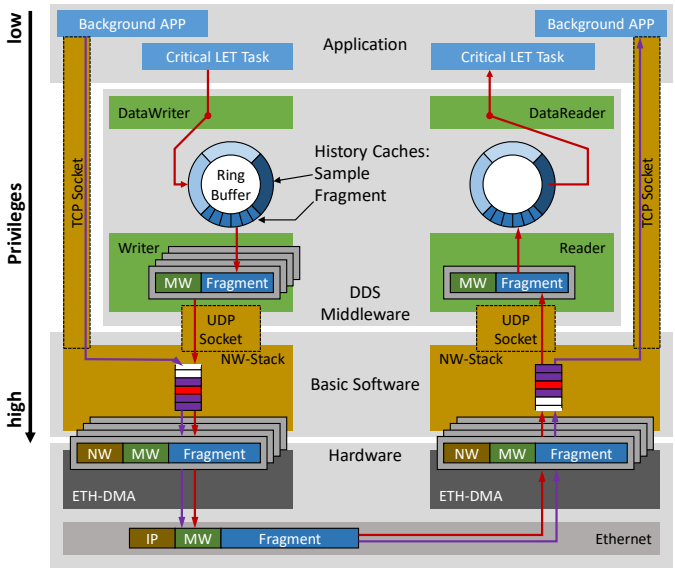

Fig. 2. Software stack for a CPS including DDS middleware

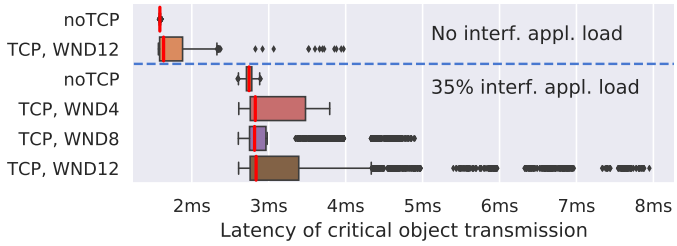

Fig. 3. Transmission latency of a critical 80kB object from ECU 1 to ECU 2 (UDP) with interference due to non-critical TCP traffic on the classic stack

Unfortunately, monolithic TCP/IP stack designs such as the lwIP are not able to bound interference between different protocols. First, the lwIP core thread processes all requests in single FIFO queue regardless of priorities and secondly, the stateful behavior of protocols such as TCP leads to a strong, unpredictable variation of the stack execution time.

One example is the TCP socket application programming interface (API) that allows to send or receive an arbitrary amount of data. It enables the background task using TCP to issue a send or receive request for a large data block, effectively handing over control to the high priority TCP/IP stack. The stack starts processing the TCP protocol until the data block is transferred, leading to a priority inversion, as the critical service as well as the middleware is starved by the running TCP/IP stack. Figure 3 shows the transmission latency for 5000 samples of *object 1* from ECU 1 to ECU 2 with and without TCP and application interference. The used TCP window is annotated as "WND" and represents the upper bound of unacknowledged TCP segments at the sender side, affecting the impact of TCP packet bursts on the receiver's input buffer. Figure 3 shows the effect of different window sizes on the transmission latency of critical samples of *object 1*. In all cases, the average latency is close to the best case and only affected by the application load. Hence, architecture and stack implementation fit well for general-purpose computing. Worst-case timing gives a completely different picture, and even in the unrealistic case of a completely unaffected network stack, the critical UDP response times grow by a factor of 4. With 35% interfering application load, the unpredictability of the TCP processing in the classic stack considerably affects the critical traffic and makes a dependable worst-case assumption impossible. This is not acceptable for safety-critical CPS, where dependable timing behavior is essential such as automated driving or industrial automation [8] [3].

Moreover, the TCP traffic stresses the buffering in the lwIP stack, which may result in lost fragments due to buffer overflows. Each sample is split in *fragments* resp. UDP packets and a missing fragment would violate the data integrity of the entire sample. In general, a low packet error rate might be handled by a retransmission protocol, ensuring the integrity of the entire sample, but any kind of retransmission protocol would also be affected by the interference.

In summary, the problem is that for the safety-critical communication in the case study, this interference has to be taken into account. Even without packet loss, any correct middleware implementation must consider the transmission latency as well as its jitter, to ensure that the planning service combining *object 1* and *object 2* is still provided with two samples of the same data age. Up to a certain point, this might be accomplished by increasing the size of the history caches and identifying corresponding samples with their timestamps or sequence numbers [3]. However, this approach is limited by (1) constraints on memory consumption due to limited platform resources and (2) latency constraints which must be satisfied to ensure a timely reaction to moving objects.

III. Related Work

Variations in computation and communication latency are a crucial aspect in safety critical CPS, as they lead to non-determinism in the data flow. Especially at the point where inputs from different cause-effect chains are read, the deviation of their data ages adds to the system complexity.

The Logical Execution Time (LET) addresses this issue for local ECUs, as it abstracts from the physical execution time by specifying explicit time instances when data is read and written [9]. Communication is assumed to be instantaneous, which can be realized e.g. by shared memory [1]. LET allows to implement lock-free multicore communication, one reason why it has successfully become a part of the AUTOSAR timing extensions [10].

SL LET extends the LET approach to distributed systems by abstracting inter-ECU communication with the SL LET interconnect task [11]. When combining input values from different cause-effect chains, SL LET provides a deterministic relation of their data ages. Moreover SL LET can be seamlessly combined with DCPS middlewares such as DDS, where the history cache already provides a form of data versioning [3]. With SL LET, each sample in a history cache is tagged with an explicit data age, allowing to specify the required size of the history caches and the access by the data-readers during

design time. Nevertheless, the complex structure of existing UDP/TCP/IP stacks, as well as the possible timing variations of high performance processing hardware, extend the required timing specification of the SL LET interconnect tasks due to their significant worst-case latency (cp. Figure 3).

Reducing the complexity of the underlying software architecture is a consequent step. MicroROS, a lightweight version of ROS 2, uses DDS for eXtremely Resource Constrained Environments (DDS-XRCE) to minimize the middleware complexity [12], targeting low power applications such as wireless sensor networks. The limited feature-set, like no multicast communication, also minimizes the requirements to the network stack. This is supported by embedded network stacks like the lwIP stack, which are highly configurable, e.g. allowing to exclude functionality already during design time. It is only applicable due to the low requirements made by the corresponding control applications while time predictable microcontrollers allow timing determinism of the overall system.

On the other hand, upcoming CPS applications demand high performance hardware and software architectures for applications such as image processing and machine learning. This introduces a large set of possible interference such as shared caches, virtual memory management and 3rd party libraries, effectively limiting time determinism. The network stacks deployed e.g. in Linux or AUTOSAR provide a large feature set, far more complex than the investigated lwIP stack.

Pre-filtering traffic in the network stack is a concept available for different use-cases. Linux provides the eXpress Data Path (XDP) as the lowest layer in the network stack, typically used for firewall applications (e.g. packet inspections or denial of service detection) [13]. In earlier work, we have proposed a filter stack that aims to combine the predictability and low latency with the option to process non-critical traffic in a full-featured background stack [5].

IV. Software Architecture

As discussed, three major aspects affect the timing behavior of a state-of-the-art RTE: (1) The interference due to resource sharing between services, middleware and basic software. (2) The limited timing determinism due to feature complexity in the RTE. (3) The trade-off between hardware performance and predictability.

In this section we present a novel RTE communication architecture, addressing those issues by strictly separating critical from non-critical features as well as exploiting the properties of SL LET to achieve deterministic and low latencies for critical middleware traffic.

SL LET enables a deterministic data-version management, since each sample in a history cache has a pre-defined data-age. The SL LET paradigm can be implemented by using any local LET scheduler on each ECU, e.g. the table based scheduler in [14], with some global synchronization with bounded error (e.g. Precision Time Protocol (PTP)). Access to the history caches is performed by utilizing sequence numbers, mapping each access to the correct history cache, which removes the need for comparing time-stamps [3].

Figure 4 shows the architecture of the communication RTE, combining a filter stack with a DDS like middleware where communication is specified with SL LET. In the following we outline the relevant parts as denoted in the figure.

A. Horizontal Separation

Apart from the vertical layers, which are known from any software stack, we apply an additional horizontal separation. Instead of reducing the functionality, a filter stack identifies each packet as either time critical or non-critical. This allows to split feature-rich elements like the network stack or middleware in time critical and non-critical parts, a kind of horizontal separation. As a direct consequence of SL LET, all features required for the implementation of the SL LET interconnect task can be identified as time critical. This includes the UDP/IP part of the network stack as well as serialization, data transport and monitoring in the middleware. Non-critical parts are potentially complex and impose requirements on correctness, but without stringent timing requirements. That way they can be run on background priorities. Examples are stateful protocols in the network stack like TCP or DHCP, as well as middleware features like service representation and discovery.

As a second step, this separation also promotes the use of a companion core in a modern MPSoC architecture such as the Xilinx Ultrascale+, Renesas R-Car H3 or NXP S32. While the high performance ARM Cortex-A cores and graphics accelerators are well suited for service implementation (e.g. image processing), those heterogeneous architectures provide lightweight but predictable Cortex-R or Cortex-M cores that are adequate for processing the critical traffic with minimal latency. The specification with SL LET further enables a lock-free implementation of the DDS history caches in a shared memory between both cores.

B. Efficient Handling of Priorities

Due to the substantial amount of non-LET traffic in a CPS, a filter stack as proposed in [5] has to provide a form of arbitration between critical and non-critical traffic. In contrast to the unmodified lwIP stack which only processes one FIFO queue, the filter stack provides priority arbitration at all layers. It supports up to 8 priority levels as known, e.g., for instance from IEEE 802.1Q [15]. Moreover, the interrupt load on a real-time core can be reduced by implementing the interface to the classic stack via polling, periodically fetching data from the classic stack. Besides providing determinism of the transmission time it is also fully configurable and can be reconfigured during runtime e.g. for network management.

C. Zero-Copy Operations with Scatter-Gather DMA

A central aspect for the efficient transport of large data objects is the memory organization. A fragment of a data sample has to be supplemented with a middleware header as well as a network header to form a network packet. Due to the lock-free behavior of SL LET, a sample in the history cache is guaranteed to be invariable during transmission. Therefore it is possible to make use of the scatter-gather direct memory access (DMA) in the network driver hardware. This does not conflict

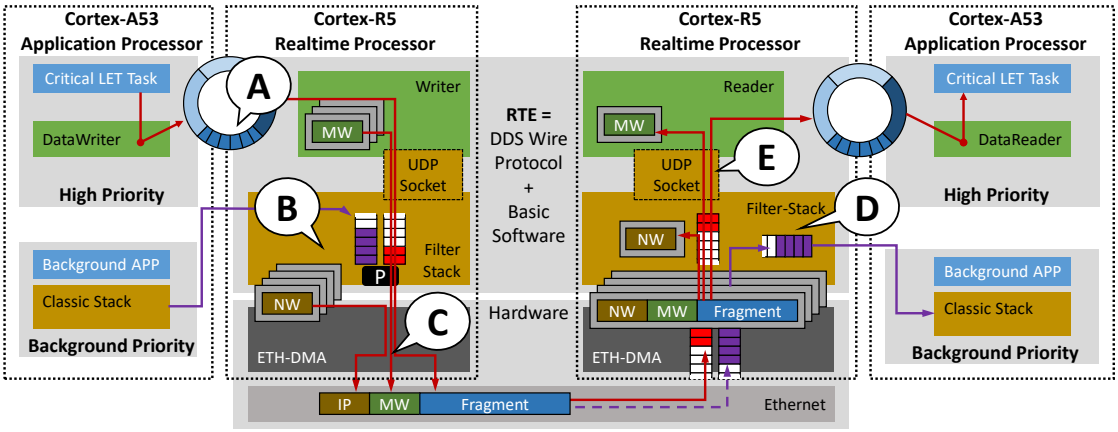

Fig. 4. Architecture of an optimized RTE with vertical and horizontal separation

with security considerations, since the memory for the history buffer is owned by the producing service and each lower layer already has higher privileges. As a result, copy operations in the middleware and network stack can be avoided, massively reducing overhead for timing as well as buffer memory.

D. Input Filtering

At the receiver side, the SL LET traffic has to be separated from non-critical traffic. The filter-stack as proposed in [5] is able to accomplish this by focusing on UDP packets for the ports used by the middleware. Due to the reduced complexity in the filter stack it is possible to make this distinction with minimal latency. The classic stack is in charge of providing buffers where non-critical packets can be stored in. Since the memory used for the DMA contains critical as well as non-critical packets, it is owned by the filter stack, and non-critical data has to be copied to the buffers owned by the classic stack. This can be implemented as a background task in the filter stack. Depending on hardware capabilities, the distinction can already be done by the network hardware, using different DMA queues for each traffic type. This would further improve the performance of the filter stack.

E. Improved Socket-API

The reception path can not directly make use of a zero-copy approach, since the middleware-header must be processed to identify the target memory for the fragment. Normal UDP sockets are assumed to be datagram oriented, discarding the remaining datagram if it has not been read at once. This induces a significant copy operation as the middleware has to read the entire packet in an internal buffer, process the middleware header and then copy the fragment from the buffer to the corresponding history cache. The situation gets even worse as the fragment in the network packet likely does not have the same memory alignment as the target memory in the history cache, prohibiting efficient copy operations.

Since the middleware is in fact a stream-oriented application (it is aware of what is expected to be received), we propose a simple modification to the socket API here, allowing to partially read from UDP sockets. That enables the middleware to receive a network packet in two steps. The first API call only receives the small middleware header to identify the target memory. The second call can directly target the memory in the history cache, avoiding an additional copy operation.

V. Evaluation

We have implemented the case study from Section II-A with the improved RTE architecture presented in Section IV. In the following, we provide measurements targeting our research challenge stated in Section I. All measurements are done on the same hardware setup as described in Section II-B and based on off-chip traces generated with Lauterbach PowerTrace-II[5] hardware. The proof-of-concept implementation and the results are available at [16].

Figure 5 is an equivalent to Figure 3 and shows the transmission latency of 80kB sized critical samples from ECU 1 to ECU 2 with SL LET and our optimized software stack (about 880k critical and non-critical Ethernet packets). The maximum worst-case latency of the filtered critical traffic is about 25% smaller than the average latency of the classic stack thus meeting research challenge 1) thereby exploiting the assumptions in 3). Moreover, the TCP traffic interference is small causing a timing overhead of at most about 6%. This interference can be attributed to the parsing of additional Ethernet traffic in the Cortex-R5 processor. The maximum difference between average and worst case timing is a few percent, as well, which can be attributed to the simpler filter stack and execution model of the Cortex-R5 (in-order pipeline, no cache related interference) with no outliers in the measurements. This way, there is high confidence in the observed worst case, different from the classical stack on the Cortex-A53 with its many outliers. With the simple Cortex-R5 processor structure and the bounded interference, even formal WCET analysis seems feasible, meaning that the critical traffic timing could likely be proven if required.

Next is research challenge 2). We measured the aggregated input and output throughput of one board by extracting both data streams with a Datacom network tap[6] and gathering

[5]https://www.lauterbach.com
[6]https://www.datacomsystems.com/products/network-taps.html

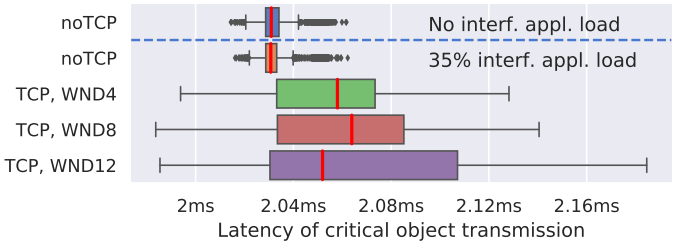

Fig. 5. Transmission latency of a critical 80kB object from ECU 1 to ECU 2 (UDP) with interference due to non-critical TCP traffic on the filter stack, showing improvement in absolute latencies and reduction in variance of latencies.

		Non-Critical Throughput	Critical Latency	Compared to WND4 Throughput	Compared to WND4 Latency
Classic	WND4	83,0 MBit/s	3,80 ms	—	—
	WND8	103,9 MBit/s	4,90 ms	25,1 %	28,9 %
	WND12	113,1 MBit/s	8,00 ms	36,2 %	110,5 %
Filter	WND4	24,1 MBit/s	2,13 ms	—	—
	WND8	37,7 MBit/s	2,14 ms	56,4 %	0,5 %
	WND12	46,3 MBit/s	2,18 ms	92,1 %	2,3 %

TABLE I
TCP (NON-CRITICAL) AVERAGE THROUGHPUT AND OBSERVED WORST-CASE UDP (CRITICAL) LATENCY DURING TEST RUN WITH DIFFERENT TCP WINDOWS

data with Wireshark[7]. Table I compares the non-critical TCP throughput as well as the observed worst-case latency of the critical traffic for the classic and the improved stack with different window sizes. The maximum sustained TCP throughput drops to less than 30%. This might be acceptable in mixed criticality applications, but is a substantial price for critical traffic predictability. The obvious reason is the comparably low average performance of the Cortex-R5 (ca. 20% of the Cortex-A53) that limits throughput. Hardware accelerated packet parsing could help but is beyond this work.

The experiments show that a deterministic transport of objects with low latency is possible due to the information gained by the SL LET specification. It enables a more efficient software architecture with a horizontal separation between critical and non-critical parts, effectively reducing the complexity on the critical path. Due to the lock-free communication, it is possible to optimize the cooperation between filter stack and DDS middleware. This allows to reduce the transmission latency of a 10kB sample by 75%, by avoiding unnecessary copy operations and ensuring memory alignment whenever data needs to be copied. The resulting software stack can also make efficient use of a real-time companion core, although the Cortex-R5 core in our experiments has only about 20% of the single-thread performance compared to the Cortex-A53 performance core. In addition, the reduced complexity also means a reduced code base in the critical path (about 60% less code in the network stack) as well a no need for a complex state machine. Consequently, this facilitates formal verification and certification in safety critical CPS as well as enables the implementation of the filter functionality in hardware.

[7]https://www.wireshark.org/

VI. Conclusion

In this paper we have shown how SL LET can be used to improve the RTE communication architecture with a filter stack and a DCPS middleware. The results of our proof-of-concept implementation show that the predictability of SL LET induced communication permits significantly faster communication than the standard stack supporting the hypothesis that SL LET programming rather reduces than extends latencies.

Despite these performance gains, the filter has a minor impact on other IP traffic in the overall stack and does not impose a limitation on non-critical functionality.

References

[1] J. Hennig, H. von Hasseln, H. Mohammad, S. Resmerita, S. Lukesch, and A. Naderlinger, "Towards parallelizing legacy embedded control software using the let programming paradigm," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Soc., 2016, p. 51.

[2] M. Beckert, "Scheduling mechanisms for efficient and safe automotive systems integration," Ph.D. dissertation, TU Braunschweig, Oct. 2020.

[3] K. B. Gemlau, L. Köhler, and R. Ernst, "A platform programming paradigm for heterogeneous systems integration," *Proceedings of the IEEE*, pp. 1–22, 2020.

[4] R. Ernst, L. Ahrendts, and K.-B. Gemlau, "System Level LET: Mastering cause-effect chains in distributed systems," in *IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2018, pp. 4084–4089.

[5] K.-B. Gemlau, J. Peeck, N. Sperling, P. Hertha, and R. Ernst, "A new design for data-centric ethernet communication with tight synchronization requirements for automated vehicles," in *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society*, vol. 1. IEEE, 2019, pp. 4489–4494.

[6] A. Dunkels and L. Woestenberg, "lwIP - A Lightweight TCP/IP stack ," https://savannah.nongnu.org/projects/lwip/, accessed: 2020-10-27.

[7] G. Pardo-Castellote, "OMG data-distribution service: Architectural overview," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* IEEE, 2003, pp. 200–206.

[8] "Autoware.Auto benchmark," https://www.autoware.auto/, accessed: 2020-10-19.

[9] C. M. Kirsch and A. Sokolova, "The logical execution time paradigm," in *Advances in Real-Time Systems*. Springer, 2012, pp. 103–120.

[10] "Specification of timing extensions," AUTOSAR Consortium, AUTOSAR Standard R19-11 (CP), Document ID 411, Nov. 2019.

[11] K.-B. Gemlau, L. Köhler, R. Ernst, and S. Quinton, "System-level logical execution time: Augmenting the logical execution time paradigm for distributed real-time automotive software," *ACM Transactions on Cyber-Physical Systems*, 2020, doi:10.1145/3381847.

[12] "micro-ROS - ROS 2 for microcontrollers," https://micro-ros.github.io/, accessed: 2020-10-27.

[13] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[14] M. Beckert and R. Ernst, "The ida let machine - an efficient and streamlined open source implementation of the logical execution time paradigm," in *International Workshop on New Platforms for Future Cars (NPCar at DATE 2018)*, March 2018. [Online]. Available: https://www.date-conference.com/date18/conference/workshop-w03

[15] "IEEE 802.1Q-2018 - IEEE Standard for Local and Metropolitan Area Networks - Bridges and Bridged Networks," IEEE, Standard, Jul. 2018.

[16] "Proof-Of-Concept Implementation Download," https://www.ida.ing.tu-bs.de/en/artifacts.