Low-Latency Real-Time Applications on Heterogeneous MPSoCs

Nicolas Coppik
ABB AG Corporate Research Center, Mannheim, Germany

Pascal Becker
ABB AG Corporate Research Center, Mannheim, Germany

Marcus Ritter
ABB AG Corporate Research Center, Mannheim, Germany

Abstract

Heterogeneous Multi-Processor Systems-on-Chip (MPSoCs) that combine multiple, heterogeneous processing units are becoming increasingly popular for a wide range of applications, including industrial applications, where complex real-time applications can benefit from the performance and flexibility they offer.

However, deploying real-time applications with low latency requirements across multiple processing units on such MPSoCs remains a challenging problem, particularly when communication between processors is required on a time-critical path. Existing solutions generally rely on the presence of at least one full-featured, general-purpose operating system on the device, and do not cater to the requirements of distributed, low-latency real-time applications.

In this paper, we investigate the performance, with a focus on latency, of different options for communication between CPUs, including inter-processor interrupts and shared memory communication via different memories, on the popular Xilinx Zynq UltraScale+ platform and propose a novel solution for communication between heterogeneous processing units that relies only on the availability of shared memory. Our solution is capable of achieving sub-microsecond latencies for signaling and the transfer of small amounts of data between processing units, making it suitable for deploying distributed, low-latency real-time applications.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded software; Computer systems organization → Multicore architectures

Keywords and phrases real-time systems, heterogeneous systems, latency, inter-core communication

Digital Object Identifier 10.4230/OASIcs.NG-RES.2025.2

1 Introduction

Embedded systems used in industrial applications are becoming both more performant and increasingly complex. As part of this trend, heterogeneous Multi-Processor Systems-on-Chip (MPSoCs) are gaining popularity. These devices typically combine multiple, heterogeneous CPUs with other computational resources, such as GPUs or FPGAs. This makes them well-suited to a wide range of applications, and popular in domains like industrial or automotive applications. A common example is the Xilinx Zynq UltraScale+ family of MPSoCs, which include both a multi-core ARM Cortex-A53 CPU as well as a dual-core ARM Cortex-R5 CPU, the latter of which is suitable for running real-time applications which require deterministic performance, while the former can be used for more computationally intensive applications.

However, in a parallel trend, real-time applications, for instance in industrial applications, have also grown more complex and computationally expensive, to the point where the individual CPUs on an MPSoC may not be sufficient for meeting their performance requirements, and distributing them across the different, heterogeneous CPUs becomes necessary.

Sixth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2025).
Editors: Patrick Meumeu Yomsi and Stefan Wildermann; Article No. 2; pp. 2:1–2:14
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

While frameworks and libraries like OpenAMP[1] to facilitate the development applications for heterogeneous MPSoCs exist, they are most commonly used in configurations that include a full-featured, general-purpose OS on one of the CPUs and a real-time operating system on another (for example, running Linux and FreeRTOS on the Cortex-A53 and Cortex-R5 cores of a Xilinx Zynq UltraScale+ device). They are not suitable for low-latency real-time applications.

In this paper, we investigate the performance of different hardware primitives for communicating between different processing units on the Xilinx Zynq UltraScale+ MPSoC, identify an approach that relies exclusively on shared memory as the most suitable, and develop a solution for inter-processor communication in low-latency real-time applications distributed across heterogeneous processing units on top of it.

This paper is structured as follows: We first provide an overview of the system we are working with and the requirements our solution is intended to meet in Section 2. Benchmarking results are presented in Section 3. We describe our solution in Section 4, summarize related work in Section 5. and provide concluding remarks in Section 6.

2 System Overview

We consider a heterogeneous MPSoC with at least two distinct processing units, which may use different instruction sets. The MPSoC additionally provides shared resources and communication between the processing units, including the possibility of issuing inter-processor interrupts from either processor and the availability of one or more memories that can be shared between the processors. We do not assume that any of the processors runs a full-fledged general purpose OS, such as Linux. Instead, all applications run either on an RTOS or bare metal. RTOS instances on each processing units may run in either an SMP configuration or with different instances on each core. This, in combination with the latency requirements described in the following, rules out the use of existing solutions such as OpenAMP. We do not consider any additional computational resources the MPSoC may provide, such as GPUs or FPGAs.

As mentioned in Section 1, a widely used hardware platform that matches these criteria is the Xilinx Zynq UltraScale+ MPSoC. Figure 1 shows the system setup described above, as mapped to this platform. To enable the different cores to communicate, we investigate both inter-processor interrupts and shared memory. For the latter, our target platform offers a number of different options, as well as a large configuration space, including various QoS mechanisms [35]. While we investigate the suitability of all memories or memory access configurations shown in Figure 2, we do not explore additional configuration options, apart from the use of the cache-coherent interconnect. We therefore end up with a total of four different memory configurations as options to build our solution on:

1. On-Chip Memory (OCM), with a total capacity of 256KiB, which is treated as non-cacheable memory by both processors.
2. DDR, using the default access paths for both processors, with a designated region of memory that is configured to be non-cacheable for both processors.
3. DDR, using the cache-coherent interconnect (CCI), with the real-time processor configured to use the coherent access path, and snooping of application processor caches from the CCI enabled (marked DDR/CCI in the following).

---

[1] `https://www.openampproject.org/`

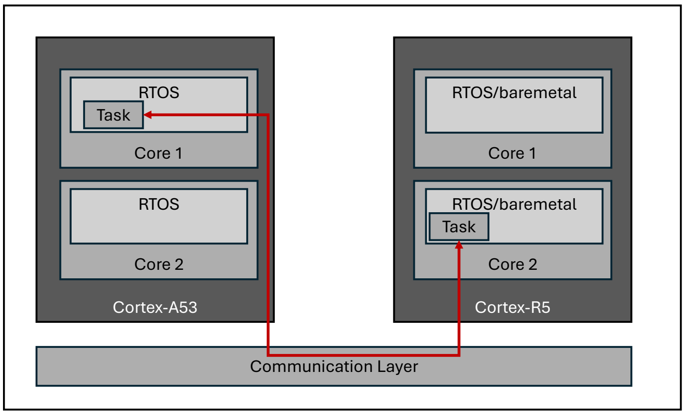


Figure 1 Our system setup as mapped to the processors of the Xilinx Zynq UltraScale+ platform, including our inter-processor communication layer for communicating between tasks on different processors.

4. Tightly Coupled Memory (TCM), with a total capacity of 128KiB for each real-time core, which is treated as non-cacheable memory by the application processor (and which is on the same level of the memory system as the caches for the real-time cores).

While only these shared memories are options for data transfer between processors, signaling can take place either over shared memory or using inter-processor interrupts ((5) in Figure 2).

The resulting solution is intended to be able to run low-latency real time applications, with response times below 30 microseconds, distributed across multiple processing units, including communication between processors in time-critical paths. Our primary aim is therefore to minimize latency, as we are only able to tolerate signaling and data transfer latencies between the processors of at most a few microseconds. Other considerations, such as reducing overall CPU utilization across the different processing units, are secondary.

We also aim to support two different kinds of communication between the processors in a distributed real-time task: Sporadic and periodic communication. The latter describes instances of communication that take place during every execution of a cyclical task - for instance, one processor sending an input for a computation to another and subsequently receiving its result. In this scenario, there is a corresponding *read* for each *write*, as in synchronous message passing, allowing for synchronization between tasks as well as data exchange. In sporadic communication, a task executing on one processor may occasionally update information that is used by a task running on the other processor, but no synchronization between the two occurs. It must be ensured that data read by the latter processor always remains internally consistent, and that its execution is not delayed by updates to the shared data (ruling out, for instance, any form of locking).

In the following, we describe our benchmarking and subsequent selection of hardware primitives to build our solution on in Section 3, and describe the solution and corresponding API in Section 4.

3 Benchmarking Results

In the following section, we will examine various metrics used to evaluate the proposed approach. These benchmarks were also instrumental in selecting the optimal memory type for fast and resilient control data transfer. All benchmarks were conducted on a Xilinx Zynq

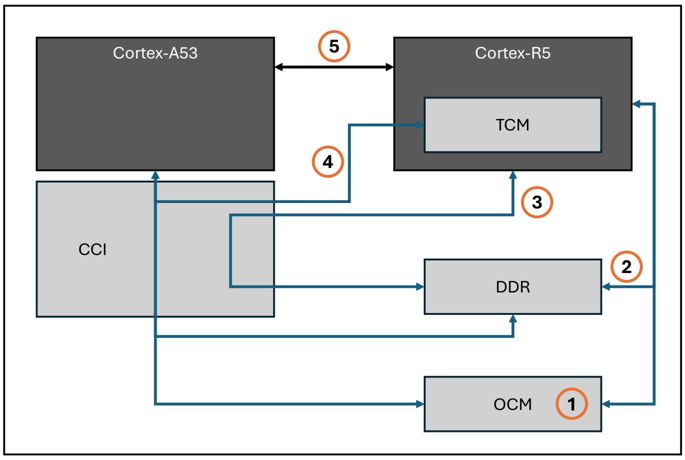


Figure 2 A simplified view of the different memory configurations and paths we consider for inter-processor communication.

UltraScale+ MPSoC, programmed using the AMD Vitis™ Unified Software Platform. The Ultra96 evaluation board used for testing is equipped with a Zynq UltraScale+ Ultra96 MPSoC, featuring a quad-core ARM Cortex-A53 CPU clocked at 1.2 GHz (the APU), a dual-core ARM Cortex-R5F CPU clocked at 500 MHz (the RPU), and 2 GiB of LPDDR4 memory.

The benchmarking scenario was structured as follows: Each test included 1,000 iterations, with 10 separate runs per test. To evaluate the impact of tasks running on another CPU core on inter-core communication performance, benchmarks were also conducted while a parallel FreeRTOS instance executed a memory-intensive task concurrently on another core of the APU. This memory-intensive task is based on a lightly modified version of the STREAM memory bandwidth benchmark[40], with changes to remove logging output and repeat benchmark runs indefinitely.

Time measurements for the benchmarks are taken on the target device itself. For measurements involving both processing units, this requires using one of the triple-timer counters (TTCs) available on the device. It is important to note that interacting with the TTCs itself impacts latency, which affects the measurements taken this way.

3.1 Shared Memory Latency Benchmarks

This section describes the benchmarks for shared memory latency in the configurations described in Section 2. We test unidirectional shared memory latencies between the processing units by starting a timer, using one of the TTCs, on one of the processors, and subsequently changing a single byte in the memory used in the given test scenario. The other processor continuously reads that byte and, as soon as the change becomes visible to it, stops the timer. These tests yield the results shown in Figure 3.

For the different memories and configurations (DDR, OCM, DDR with cache coherency, and TCM), the median latencies without additional load on the system are 1010 ns, 670 ns, 880ns, and 480 ns, respectively, going from the APU to the RPU. In the opposite direction, likewise without additional load, the median latencies for each memory type are 1000 ns, 490 ns, 840 ns, and 600 ns, respectively.

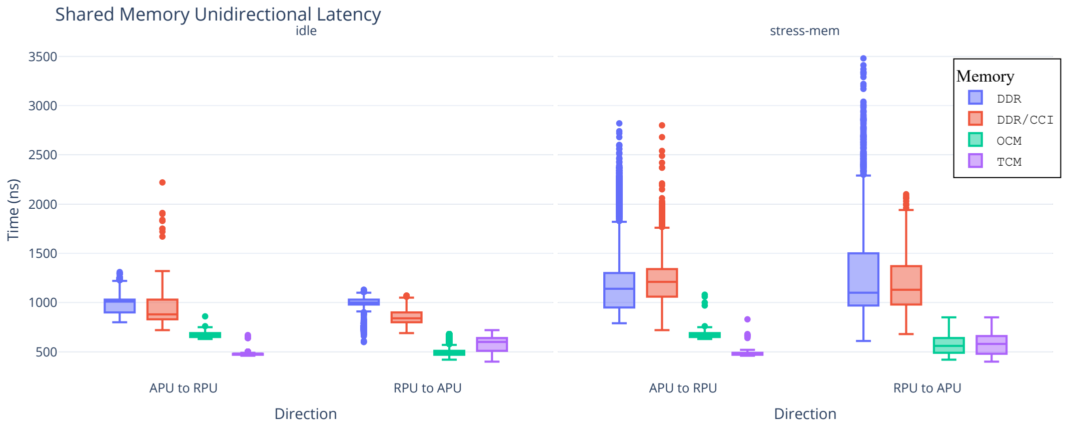


Figure 3 Shared memory unidirectional latency measured for four different memory types.

When a memory-intensive task is added on another APU core, the median latencies from the APU to the RPU increase to 1140 ns, 670 ns, 1210 ns, and 480 ns, respectively. In the opposite direction, the corresponding median latencies are 1100 ns, 560 ns, 1130 ns, and 580 ns, respectively. Additionally, there is a clear increase in the number of outliers in both directions when a memory-intensive task is added to the system. The effect is especially pronounced for the DDR and DDR/CCI cases. While the impact of the additional load could potentially be mitigated by adjusting QoS settings on the platform, we did not investigate these options as part of our testing.

We find that TCM and OCM show the fastest performance among the tested options, with TCM latencies being lower when going from APU to RPU, and OCM latencies being lower in the opposite direction. We also find these options are also the least sensitive to concurrent load. Finally, while the difference in latency between using TCM and OCM is quite small when going from RPU to APU, the latency advantage of the TCM in the opposite direction is more substantial, which suggests that the TCM is the most performant option overall.

3.2 Inter-Process Interrupt Latency

In this section, we describe the results of our benchmarks of inter-processor interrupt latency between the APU and the RPU. As in the shared memory latency benchmarks described in the previous section, we rely on the TTCs for our measurements: One core starts a timer and then immediately issues an inter-processor interrupt, at which point the other core will stop the timer in its interrupt handler. As previously, we perform these tests both on an idle system and with a memory-intensive task running on another APU core. The results of our benchmarks are shown in Figure 4.

With the other APU cores idle, we observe a median latency of 1970 ns going from the APU to the RPU and 1070 ns going in the other direction.

Adding our memory load does not substantially alter these values, however, it does result in a large increase in the number of outliers, especially when going from the APU to the RPU. This is significant as our solution is intended for low-latency real-time applications, where such increases in latency could lead to deadline misses and would not be tolerable.

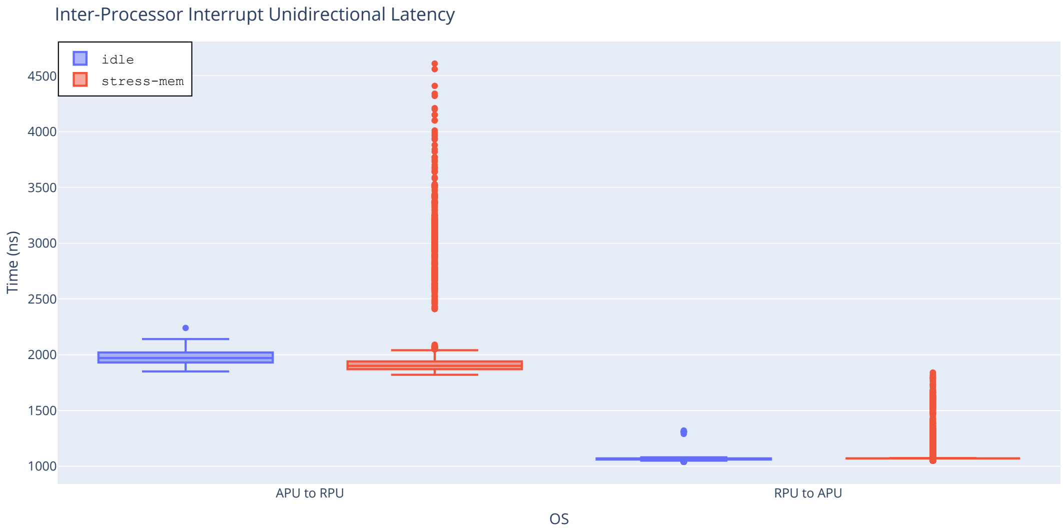


Figure 4 Inter-processor interrupt latencies.

Comparing these numbers to our previous results for shared memory latencies, we see that interrupt latencies are substantially higher in the median case, especially on an otherwise idle system. This holds true for all memories and configurations we tested, but the difference is especially pronounced when comparing the shared memory latency of the TCM to the inter-processor interrupt latencies we observed.

As outlined in Section 2, the primary goal of our solution is to optimize for latency. Based on these benchmark results, we therefore conclude that a solution relying only on shared memory for both signaling and data transfer is the most suitable as it allows for the lowest overall latency, even though this advantage comes at the cost of requiring busy-waiting on the receiving processing unit. We also conclude that, for scenarios where the amount of data is sufficiently small, and the required space in the TCM is available, the TCM should be the favored option, and we outline our solution building on this, and the performance for data transfer in the following.

3.2.1 Transferring Data via Tightly Coupled Memory (TCM)

Having chosen the TCM as the most performant option, we conduct additional testing to determine how long it takes to transfer data from the APU to the RPU in this way. While our previous tests only wrote or read a single byte, this test scenario involves one processor writing an array of varying size to the shared memory region, setting a flag to indicate that it is ready to be read, and the other processor reading the full array. The results of these tests are shown in Figure 5. Adding a memory load to the system increases data transfer latency substantially, as well as leading to an increasing number of outliers. Overall latency grows quickly for increasing data transfer sizes, showing that, while communication between processors is feasible even on time-critical paths, it is crucial to minimize the amount of data that needs to be sent or received. While up to 128 B have a limited impact, latency increases quickly after that point.

These results form the foundation of the inter-processor communication solution we describe in the following section.

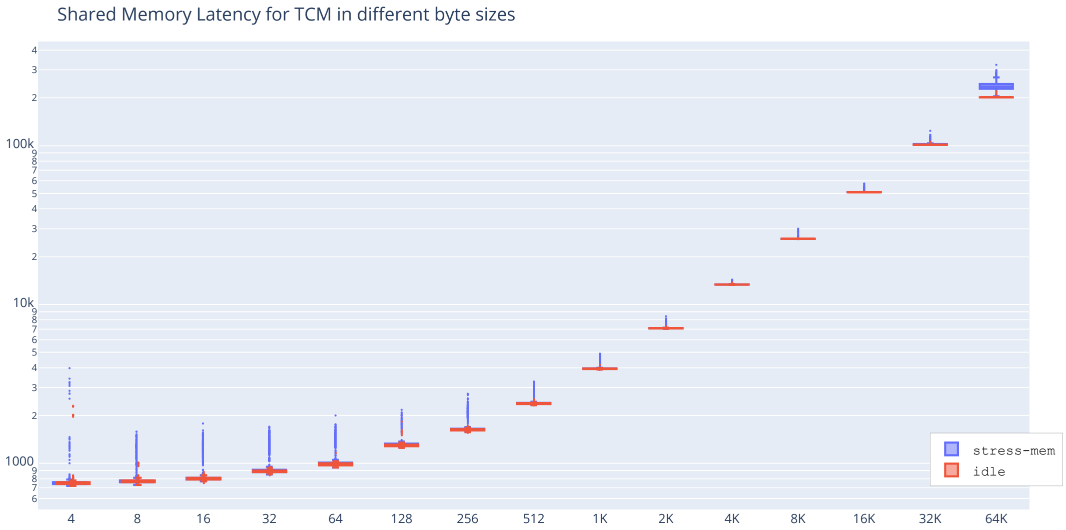


Figure 5 Latency for transferring data from the APU to the RPU via the TCM memory.

4 Inter-Core Communication Solution

As our benchmark results show, relying only on shared memory for signaling can achieve lower latencies than inter-processor interrupts. This comes at the cost of requiring the receiving side to busy-wait, thereby hurting overall efficiency since the receiving side is not able to perform useful work while waiting, and increasing CPU utilization. Nonetheless, we choose to build our solution only on shared memory, following the priorities described in Section 2.

As previously discussed in the same Section, we distinguish between two different types of inter-core communication: sporadic and periodic communication. We define periodic communication as communication that takes place during every execution of a cyclical task, for example, one processor sending an input for a computation to another and subsequently receiving its result. On the contrary, sporadic communication occurs only occasionally, for example, a task executing on one processor may update information that is used by a task running on another processor without any synchronization.

To facilitate an inter-core communication (ICC) between processors for these two communication types, we implement two concepts commonly used for inter-process communication: A mailbox and a double buffer. Figure 6 outlines the proposed inter-core communication API and its individual components.

The periodic communication is realized via a shared memory mailbox: A single-reader, single-writer mailbox that supports alternating reads and writes, i.e., for each write there must be a corresponding read prior to the next write. An in depth description is given in Section 4.1.

The sporadic communication is realized via a shared memory double buffer: A single-writer, multiple-reader buffer that may be read multiple times without restriction and can be updated independently of the reading process, provided that the update frequency remains lower than the read frequency. A detailed explanation is given in Section 4.2.

The ICC API is implemented as a header-only `C++` library, without any platform- or hardware-specific code. Users need to provide a shared memory region visible to both communicating cores and instantiate the shared memory structures on both sides. For

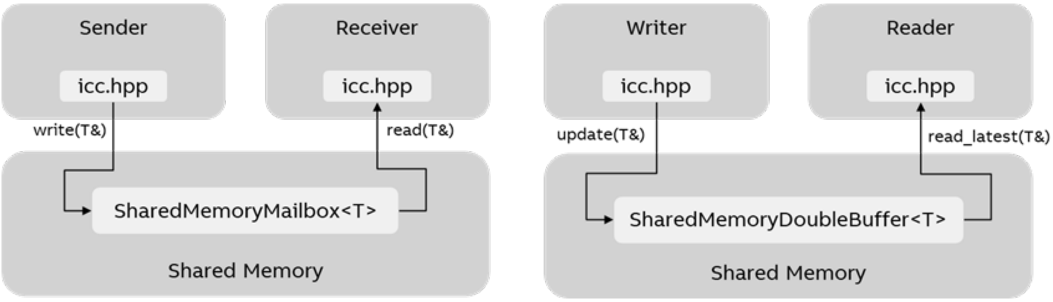
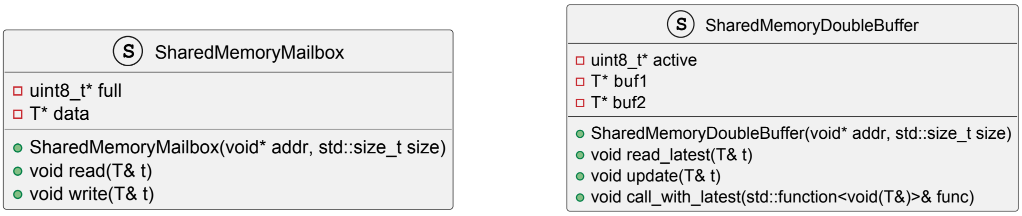


Figure 6 Overview of the proposed inter-core communication API.


S SharedMemoryMailbox
uint8_t* full
T* data
SharedMemoryMailbox(void* addr, std::size_t size)
void read(T& t)
void write(T& t)
S SharedMemoryDoubleBuffer
uint8_t* active
T* buf1
T* buf2
SharedMemoryDoubleBuffer(void* addr, std::size_t size)
void read_latest(T& t)
void update(T& t)
void call_with_latest(std::function<void(T&)>& func)

(a) Class diagram of the shared memory mailbox. **(b)** Class diagram of the double buffer.

Figure 7 The proposed data structures for inter-processor communication.

performance reasons, all data structures shared over this API should be trivially copyable and share the same layout on both sides. While adding serialization support is possible, and we have investigated a lightweight serialization extension to this solution, we omit a detailed discussion of this topic.

4.1 Shared Memory Mailbox

For cyclic communication, we propose to use a mailbox in shared memory. A slice of memory is allocated within a shared memory region, which is accessible to all participants. It serves as both a buffer, allowing messages to be written and read by various entities, and, by way of a status flag, a synchronization mechanism. The mailbox interface governs the interactions between the shared memory mailbox and the entities using it.

Our implementation of the shared memory mailbox uses a templated structure that is designed to store and transfer a single instance of a data type `<T>`, e.g., an integer value, in a shared memory segment. By employing `C++` templates, we ensure that the structure is capable of handling any data type required across various use cases. Figure 7a provides an overview of the shared memory mailbox, its variables, and functions. The `data` variable holds a pointer to the actual data of type `<T>` that will be exchanged via the mailbox. The `full` variable points to a flag which indicates whether the mailbox contains data. The constructor assigns addresses in the shared memory region to both of these, as well as ensuring that the shared memory region is sufficiently large and that `T` is trivially copyable. Note that `T` must have the same layout for all users of the mailbox, and this is not checked automatically.

To read data from the mailbox, readers can use the `read` function, providing a reference to an instance of `T` that will be updated with the data read from the mailbox. `read` is blocking, and is implemented using busy-waiting. If the mailbox is empty, the reader loops until the `full` flag is set and new data is available to read. After reading, the reader resets the flag to indicate that the mailbox is empty, allowing a writer to place new data there. Atomic operations are used to ensure sequential consistency and prevent data races or issues due to reordering.

To write data to the mailbox, writers can use the `write` function, providing a reference to an instance of `T` that will be copied to the mailbox. Like `read`, `write` is blocking: If the mailbox is already full, the writer loops until it is empty and new data can be placed there.

Our shared memory mailbox is a lightweight structure that allows communication between tasks on different processors with a simple, blocking API. It enables both data exchange and synchronization, and would be straightforward to extend with a non-blocking API (e.g., `bool try_write(T& t), bool try_read(T& t)`).

4.2 Shared Memory Double-Buffer

For sporadic communication, we propose to use a double buffer placed in shared memory. Like our shared memory mailbox, it only requires a slice of shared memory accessible to all participants. The double buffering technique is a widely used approach to ensure data consistency when reading and writing data concurrently in a shared memory environment. It ensures that the reader gets a complete and consistent view of the data, even if the writer is currently updating it. Figure 7b provides an overview of the shared memory double buffer, its variables, and functions. The shared memory double buffer contains a pointer`active` that indicates which of the two buffers is currently active, i.e., should be used for reading, and two buffers `buf1` and `buf2`, which are pointers to two instances of the type `<T>`. As for the mailbox, the constructor assigns addresses in the shared memory region to these pointers, checks that the region is sufficiently large, and that `T` is trivially copyable. As before, users must ensure that `T` additionally shares the same layout for all participants.

Readers can use the `read_latest` function to read the latest value from the double buffer. Unlike the mailbox's `read`, this is non-blocking, and will immediately update the provided instance of `T`. Reading does not change `active`, and the same value can be read multiple times.

To update the double buffer, writers can use the `update` function. This function checks the value of `active`, selects the currently inactive buffer, and overwrites it with the contents of the provided `T`, before updating `active` to point to the newly written value. Like `read_latest`, `update` is non-blocking. As in the mailbox, atomic operations are used to ensure sequential consistency.

It is important to note that this implementation allows for data races if updates happen at a higher frequency than reads: If a reader calls `read_latest`, and the writer calls `update` more than once while the reader is reading, it will affect the buffer used by the reader, which may lead to inconsistent data being read. In practice, this issue can be avoided by ensuring that the writing task τ_w has a longer task period than any reading task τ_r^i, and only updates the double buffer once per period. If this cannot be guaranteed, additional synchronization would be required, which may in turn lead to higher delays.

5 Related Work

In the rapidly evolving domain of Multi-Processor Systems-on-Chip (MPSoCs), extensive research has been conducted to enhance performance verification, communication architectures, task scheduling, and memory management. This section reviews key contributions, providing a comprehensive context for the solution proposed in this paper.

Inter-processor communication is a critical aspect of MPSoC design. Hung et al. [20] introduced MSG, an inter-core communication mechanism derived from MPI and MCAPI, showing improved performance on platforms like IBM CELL. Schwäricke et al. [34] used cache partitioning in a real-time communication framework, which is applicable to managing memory contention. These studies emphasize established protocols and cache management, yet our research distinguishes itself by using shared memory exclusively for signaling and data exchange, avoiding inter-processor interrupts to achieve sub-microsecond latencies essential for time-critical applications. Loghi et al. [25] provided insights into communication bottlenecks using SystemC simulations, while Nieto et al. [28] explored synchronization methods on the Zynq-UltraScale+, comparing techniques like InterProcessor Interrupts (IPIs) with shared memory. Our work builds on these by demonstrating that removing IPIs in favor of shared memory not only simplifies the communication model but also enhances latency performance, pivotal for real-time applications across heterogeneous units. Further studies by Mirosanlou et al. [26], Belloch et al. [5], Hassan et al. [18], and Nollet et al. [29] investigated communication strategies and low-latency interconnects relevant to shared memory systems.

Performance verification is crucial for MPSoCs. Richter et al. [32] and Castells-Rufas et al. [8] provided foundational methodologies for assessing communication performance. France-Pillois et al. [11] and Shanthi et al. [36] developed tools to optimize synchronization barriers, enhancing communication efficiency in real-time systems. Our study offers a streamlined solution with concrete benchmarking results, demonstrating the practical application of shared memory in achieving low-latency communication without traditional synchronization complexities.

Effective memory management underpins low-latency communication. Bansal et al. [4] and Carletti et al. [7] tackled cache coherency and memory contention, proposing techniques like cache coloring. Our approach simplifies memory access patterns by using shared memory, reducing contention and bypassing cache coherency challenges, thus enhancing predictability and efficiency. Hoornaert et al. [19], Hahn et al. [17], Kogel et al. [22], and Acacio et al. [1] focused on memory subsystem optimizations and architectural strategies for efficient memory access.

Optimizing task scheduling is vital for communication efficiency. Singh et al. [37] and Ali et al. [2] developed heuristics and methodologies for runtime mapping and energy efficiency. Spieck et al. [39] emphasized minimizing communication overhead. Our approach prioritizes achieving minimal communication latency over resource utilization, dedicating one processor to this task, aligning with the stringent requirements of real-time applications. Ristau et al. [33] and Guerin et al. [15] explored strategies that optimize resource usage and reduce communication latency, critical for efficient shared memory communication.

Real-time applications often operate within mixed criticality systems, which require predictable performance across varying criticality levels. Gracioli et al. [14] proposed frameworks that use dedicated memory interfaces and cache coloring to ensure timing predictability. Nowotsch et al. [30] focused on managing resource contention to maintain timing predictability. Our work uniquely addresses the latency needs of these applications through a shared memory-based model that avoids additional complexities, ensuring timing predictability in mixed criticality environments. Geyer et al. [13] and Kim et al. [21] provide additional context for managing real-time constraints.

Several studies, while not directly focused on shared memory communication, provide valuable insights into MPSoC design and optimization. Research on virtualization in mixed criticality systems [10, 9, 31], as well as security and energy efficiency in IoT applications [42, 16], highlight the importance of dynamic resource allocation and energy efficiency, complementing shared memory communication strategies. Advanced resource management techniques explored by Kumar et al. [23] and Frid et al. [12] contribute to optimizing performance and energy efficiency.

Further contributions include studies on programming models and software optimization, such as Sommer et al. [38], which examines parallel programming models for automotive applications, and Brown et al. [6], who introduced the Distributed Multiloop Language (DMLL) to optimize parallel applications. Our approach offers an easy-to-use, header-only C++ library that facilitates porting across different platforms, emphasizing practical applicability in real-time environments. Liu et al. [24] explores flexible inter-core communication methods in multicore processors, demonstrating low latency and high system performance, aligning with the objectives of optimizing shared memory communication. The work by Tsao et al. [41] and Nie et al. [27] on performance evaluation and design of multiprocessor embedded systems using shared memory further supports the development of efficient communication solutions. Anjum et al. [3] provides insights into the latest methodologies for optimizing inter-processor communication.

6 Conclusion

Deploying low-latency real-time applications across heterogeneous processors on modern MPSoCs poses a challenging problem, especially if communication between processors is required in a time-critical path.

We present benchmarking results and a proof-of-concept solution showing that, by avoiding inter-processor interrupts, relying only on shared memory for signaling and data exchange, and sacrificing efficient utilization of one of the processors to minimize latency, we can achieve signaling latencies and data transfer times in the low microsecond range on the Xilinx Zynq UltraScale+ platform.

We implement our solution in an easy-to-use, header-only C++ library. While our solution and implementation are straightforward to port to other hardware platforms, making the most effective use of it on another platform would entail additional benchmarking to select a suitable memory configuration.

References

1 Manuel E Acacio, Jose Gonzalez, Jose M Garcia, and Jose Duato. An architecture for high-performance scalable shared-memory multiprocessors exploiting on-chip integration. *IEEE Transactions on Parallel and Distributed Systems*, 15(8):755–768, 2004. `doi:10.1109/TPDS.2004.27`.

2 Haider Ali, Xiaojun Zhai, Umair Ullah Tariq, and Lu Liu. Energy efficient heuristic algorithm for task mapping on shared-memory heterogeneous mpsocs. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1099–1104. IEEE, 2018. `doi:10.1109/HPCC/SMARTCITY/DSS.2018.00183`.

3 Etsam Anjum and Jeffrey Hancock. Introduction to openamp library. `https://www.openampproject.org/docs/whitepapers/Introduction_to_OpenAMPlib_v1.1a.pdf`, 2023. Accessed: October 29, 2024.

4 Ayoosh Bansal, Rohan Tabish, Giovani Gracioli, Renato Mancuso, Rodolfo Pellizzoni, and Marco Caccamo. Evaluating memory subsystem of configurable heterogeneous mpsoc. *Proceedings of the Operating Systems Platforms for Embedded Real-Time applications*, 2018.

5 Jose A Belloch, Germán León, José M Badía, Almudena Lindoso, and Enrique San Millan. Evaluating the computational performance of the xilinx ultrascale+ eg heterogeneous mpsoc. *The Journal of Supercomputing*, 77:2124–2137, 2021. `doi:10.1007/S11227-020-03342-7`.

6 Kevin J Brown, HyoukJoong Lee, Tiark Rompf, Arvind K Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 194–205, 2016. `doi:10.1145/2854038.2854042`.

7 Lorenzo Carletti, Gianluca Brilli, Alessandro Capotondi, Paolo Valente, and Andrea Marongiu. The importance of worst-case memory contention analysis for heterogeneous socs. *arXiv preprint*, 2023. `doi:10.48550/arXiv.2309.12864`.

8 David Castells-Rufas, Jaume Joven, Sergi Risueño, Eduard Fernandez, Jordi Carrabina, Thomas William, and Hartmut Mix. Mpsoc performance analysis with virtual prototyping platforms. In *2010 39th International Conference on Parallel Processing Workshops*, pages 154–160. IEEE, 2010.

9 Marcello Cinque, Domenico Cotroneo, Luigi De Simone, and Stefano Rosiello. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. *Future Generation Computer Systems*, 129:315–330, 2022. `doi:10.1016/J.FUTURE.2021.12.002`.

10 Marcello Cinque, Gianmaria De Tommasi, Sara Dubbioso, and Daniele Ottaviano. Rpuguard: Real-time processing unit virtualization for mixed-criticality applications. In *2022 18th European Dependable Computing Conference (EDCC)*, pages 97–104. IEEE, 2022. `doi:10.1109/EDCC57035.2022.00025`.

11 Maxime France-Pillois, Jérôme Martin, and Frédéric Rousseau. A non-intrusive tool chain to optimize mpsoc end-to-end systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(2):1–22, 2021. `doi:10.1145/3445030`.

12 Nikolina Frid, Danko Ivošević, and Vlado Sruk. Performance estimation in heterogeneous mpsoc based on elementary operation cost. In *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1202–1205. IEEE, 2016.

13 Tobias Geyer, Nikolaos Oikonomou, Georgios Papafotiou, and Frederick D Kieferndorf. Model predictive pulse pattern control. *IEEE Transactions on Industry Applications*, 48(2):663–676, 2011.

14 Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing mixed criticality applications on modern heterogeneous mpsoc platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.

15 Xavier Guerin, Katalin Popovici, Wassim Youssef, Frederic Rousseau, and Ahmed Jerraya. Flexible application software generation for heterogeneous multi-processor system-on-chip. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 1, pages 279–286. IEEE, 2007.

16 Marisabel Guevara, Benjamin Lubin, and Benjamin C Lee. Navigating heterogeneous processors with market mechanisms. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 95–106. IEEE, 2013. `doi:10.1109/HPCA.2013.6522310`.

17 Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th international conference on real-time networks and systems*, pages 299–308, 2016. `doi:10.1145/2997465.2997471`.

18 Mohamed Hassan. Heterogeneous mpsocs for mixed criticality systems: Challenges and opportunities. *arXiv preprint*, 2017. `arXiv:1706.07429`.

19 Denis Hoornaert, Shahin Roozkhosh, and Renato Mancuso. A memory scheduling infrastructure for multi-core systems with re-programmable logic. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

20 Shih-Hao Hung, Wen-Long Yang, and Chia-Heng Tu. Designing and implementing a portable, efficient inter-core communication scheme for embedded multicore platforms. In *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 303–308. IEEE, 2010. `doi:10.1109/RTCSA.2010.17`.

21 Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154. IEEE, 2014. `doi:10.1109/RTAS.2014.6925998`.

22 Tim Kogel and Heinrich Meyr. Heterogeneous mp-soc: the solution to energy-efficient signal processing. In *Proceedings of the 41st annual Design Automation Conference*, pages 686–691, 2004. `doi:10.1145/996566.996754`.

23 Akash Kumar, Bart Mesman, Bart Theelen, Henk Corporaal, and Yajun Ha. Analyzing composability of applications on mpsoc platforms. *Journal of Systems Architecture*, 54(3-4):369–383, 2008. `doi:10.1016/J.SYSARC.2007.10.002`.

24 Cheng Liu, Lei Luo, Meng Li, Pinyuan Lei, Lirong Chen, and Kun Xiao. Inter-core communication mechanisms for microkernel operating system based on signal transmission and shared memory. In *2021 7th International Symposium on System and Software Reliability (ISSSR)*, pages 188–197. IEEE, 2021.

25 Mirko Loghi, Federico Angiolini, Davide Bertozzi, Luca Benini, and Roberto Zafalon. Analyzing on-chip communication in a mpsoc environment. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 2, pages 752–757. IEEE, 2004. `doi:10.1109/DATE.2004.1268966`.

26 Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Duetto: Latency guarantees at minimal performance cost. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1136–1141. IEEE, 2021. `doi:10.23919/DATE51398.2021.9474062`.

27 Yang Nie, Lili Jing, and Pengyu Zhao. Design and implementation of inter-core communication of embedded multiprocessor based on shared memory. *International Journal of Security and Its Applications*, 10(12):21–30, 2016.

28 Rubén Nieto, Edel Díaz, Raúl Mateos, and Álvaro Hernández. Evaluation of software inter-processor synchronization methods for the zynq-ultrascale+ architecture. In *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6. IEEE, 2020. `doi:10.1109/DCIS51330.2020.9268616`.

29 Vincent Nollet, Prabhat Avasare, J-Y Mignolet, and Diederik Verkest. Low cost task migration initiation in a heterogeneous mp-soc. In *Design, Automation and Test in Europe*, pages 252–253. IEEE, 2005.

30 Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In *2012 Ninth European Dependable Computing Conference*, pages 132–143. IEEE, 2012. `doi:10.1109/EDCC.2012.27`.

31 D Ottaviano, M Cinque, G Manduchi, and S Dubbioso. Virtualization of accelerators in embedded systems for mixed-criticality: Rpu exploitation for fusion diagnostics and control. *Fusion Engineering and Design*, 190:113518, 2023.

32 Kai Richter, Marek Jersak, and Rolf Ernst. A formal approach to mpsoc performance verification. *Computer*, 36(4):60–67, 2003. `doi:10.1109/MC.2003.1193230`.

33 Bastian Ristau and Gerhard Fettweis. Mapping and performance evaluation for heterogeneous mp-socs via packing. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 7th International Workshop, SAMOS 2007, Samos, Greece, July 16-19, 2007. Proceedings 7*, pages 117–126. Springer, 2007. `doi:10.1007/978-3-540-73625-7_14`.

34 Gero Schwäricke, Rohan Tabish, Rodolfo Pellizzoni, Renato Mancuso, Andrea Bastoni, Alexander Zuepke, and Marco Caccamo. A real-time virtio-based framework for predictable inter-vm communication. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 27–40. IEEE, 2021. `doi:10.1109/RTSS52674.2021.00015`.

35 Alejandro Serrano Cases, Juan M Reina, Jaume Abella Ferrer, Enrico Mezzetti, and Francisco Javier Cazorla Almeida. Leveraging hardware qos to control contention in the xilinx zynq ultrascale+ mpsoc. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021): July 5-9, 2021, Virtual Conference*, volume 196, pages 3–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ECRTS.2021.3`.

36 D Shanthi and R Amutha. Performance analysis of on-chip communication architecture in mpsoc. In *2011 International Conference on Emerging Trends in Electrical and Computer Technology*, pages 811–815. IEEE, 2011.

37 Amit Kumar Singh, Wu Jigang, Alok Prakash, and Thambipillai Srikanthan. Efficient heuristics for minimizing communication overhead in noc-based heterogeneous mpsoc platforms. In *2009 IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 55–60. IEEE, 2009. `doi:10.1109/RSP.2009.18`.

38 Lukas Sommer, Florian Stock, Leonardo Solis-Vasquez, and Andreas Koch. Using parallel programming models for automotive workloads on heterogeneous systems-a case study. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 17–21. IEEE, 2020. `doi:10.1109/PDP50117.2020.00010`.

39 Jan Spieck, Stefan Wildermann, and Jürgen Teich. A learning-based methodology for scenario-aware mapping of soft real-time applications onto heterogeneous mpsocs. *ACM Transactions on Design Automation of Electronic Systems*, 28(1):1–40, 2022. `doi:10.1145/3529230`.

40 STREAM: Sustainable Memory Bandwidth in High Performance Computers. `https://www.cs.virginia.edu/stream/`. Accessed: 2024-11-12.

41 Shiao-Li Tsao, Sung-Yuan Lee, et al. Performance evaluation of inter-processor communication for an embedded heterogeneous multi-core processor. *Journal of information science and engineering*, 28(3), 2012. URL: `http://www.iis.sinica.edu.tw/page/jise/2012/201205_07.html`.

42 Junlong Zhou, Jin Sun, Peijin Cong, Zhe Liu, Xiumin Zhou, Tongquan Wei, and Shiyan Hu. Security-critical energy-aware task scheduling for heterogeneous real-time mpsocs in iot. *IEEE Transactions on Services Computing*, 13(4):745–758, 2019. `doi:10.1109/TSC.2019.2963301`.