

MIT Open Access Articles

InterPRET: a Time-predictable Multicore Processor

The MIT Faculty has made this article openly available. **Please share** how this access benefits you. Your story matters.

Citation: Jellum, Erling, Lin, Shaokai, Donovan, Peter, Jerad, Chadlia, Wang, Edward et al. 2023. "InterPRET: a Time-predictable Multicore Processor."

As Published: <https://doi.org/10.1145/3576914.3587497>

Publisher: ACM|Cyber-Physical Systems and Internet of Things Week 2023

Persistent URL: <https://hdl.handle.net/1721.1/150843>

Version: Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

Terms of use: Creative Commons Attribution



InterPRET: a Time-predictable Multicore Processor

Erling Rennemo Jellum
NTNU
Trondheim, Norway

Chadlia Jerad
University of Manouba
Manouba, Tunisia

Shaokai Lin
UC Berkeley
Berkeley, USA

Edward Wang
MIT
Boston, USA

Peter Donovan
UC Berkeley
Berkeley, USA

Marten Lohstroh
UC Berkeley
Berkeley, USA

Edward A. Lee
UC Berkeley
Berkeley, USA

Martin Schoeberl
DTU
Copenhagen, Denmark

ABSTRACT

With the end of Moore's law and the breakdown of Dennard scaling, multicore processors are the standard way to continue improving performance while reducing Size, Weight and Power (SWaP). However, this performance is typically achieved at the cost of repeatability and predictability. Precision-timed (PRET) architectures have been shown to deliver high performance without sacrificing predictability. In this paper, we introduce InterPRET: an architecture consisting of FlexPRET cores interconnected via the S4NOC network-on-chip. Both the processor cores and the network-on-chip are time-predictable, yielding an end-to-end time-predictable architecture suitable for real-time systems.

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures;**
Real-time system architecture.

ACM Reference Format:

Erling Rennemo Jellum, Shaokai Lin, Peter Donovan, Chadlia Jerad, Edward Wang, Marten Lohstroh, Edward A. Lee, and Martin Schoeberl. 2023. InterPRET: a Time-predictable Multicore Processor. In *Cyber-Physical Systems and Internet of Things Week 2023 (CPS-IoT Week Workshops '23)*, May 09–12, 2023, San Antonio, TX, USA. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3576914.3587497>

1 INTRODUCTION

To meet their timing requirements, real-time system engineers can follow at least three different strategies. The first and arguably the most common strategy is to use general-purpose computer architectures that are optimized for average-case performance. These architectures employ optimization techniques like memory caching, branch prediction, and out-of-order execution, which trades timing predictability for average-case performance. To ensure that all real-time deadlines are met, measurements of execution times to

find the worst-observed execution time are combined with over-provisioning of computing resources.

The second strategy is to implement the computation as a synchronous digital circuit, either on an FPGA or as an ASIC. This gives very precise control over timing at the granularity of the clock frequency driving the circuit. However, for modern real-time systems, which perform various computations on different workloads, implementations completely based on synchronous digital circuits are unrealistically complex and inefficient.

In this paper, we argue for a third strategy, which is to rely on timing-predictable computer architectures, also known as PRET machines [2]. PRET machines are stripped of optimization techniques that sacrifice timing predictability for average-case performance. To achieve performance similar to general-purpose processors [11], PRET machines often employ fine-grained multithreading using barrel register files. PRET machines offer such fine-grained control over timing that they are capable of implementing real-time interfaces in software. XMOS, the company behind the only commercially available PRET machine that we know of, refers to this capability as a “software-defined” System-on-Chip (SoC) [22].

As we reach the end of Dennard scaling [6], single-thread performance is no longer seeing much improvement. Computer architects are instead looking to multicore designs to deliver increased performance. This poses significant challenges for real-time systems designers. While multicore architectures may improve average-case performance, they often negatively impact worst-case execution time (WCET) or make it very difficult to reason about [15]. Communication between cores is typically implemented through shared memory. The cores typically do not share memory directly, but maintain a local cache of the shared memory, with a cache coherence protocol ensuring that data retrieved from local caches is consistent across caches. A consequence of this design is that a write in one core may invalidate a cache line in another, forcing a subsequent read to fetch a new cache line from a larger and slower memory. This makes it difficult to predict how long it takes retrieve a value from memory. In other words, it renders tasks *non-composable* [10] in the sense that tasks running on different cores can affect each other's timing, even if no data dependencies exist between the tasks. This is one of the reasons why real-time systems engineers resist the introduction of multicore architectures into safety-critical systems [3].



This work is licensed under a Creative Commons Attribution International 4.0 License.

CPS-IoT Week Workshops '23, May 09–12, 2023, San Antonio, TX, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0049-1/23/05.
<https://doi.org/10.1145/3576914.3587497>

A network-on-chip (NoC) is an alternative approach to inter-core communication based on *message passing*. Because message passing does not require a complex memory hierarchy, it is a better candidate for achieving composability. In general, we can distinguish between *event-triggered* and *time-triggered* NoCs. The former is more flexible and has a lower average-case latency. However, it is not composable. The latter is based on time-division multiplexing (TDM) where each communication link has statically allocated slots. This removes the potential for conflicts on shared resources and thus the need for buffering and complicated flow control.

In this paper, we introduce InterPRET (i.e., the **Inter**connected **PRET** machine), a multicore PRET machine based on message passing. InterPRET consists of a set of FlexPRET CPUs [24] interconnected by the time-predictable S4NOC [18, 19]. InterPRET is synthesized to an FPGA and evaluated in terms of resource utilization, timing predictability, and NoC throughput.

The paper is organized as follows. The following section presents related work. Section 3 provides background into FlexPRET and S4NOC. Section 4 presents the hardware architecture of InterPRET. Section 5 evaluates InterPRET, and Section 6 concludes the paper.

2 RELATED WORK

The XCore architecture from XMOS [13] is a commercially available multicore PRET machine. Like InterPRET, the XCore is based on fine-grained multithreading to remove variability in the execution time of instructions. It has a NoC, called XConnect, which uses credit-based control flow. The S4NOC, being based on a static schedule, does not need this type of flow control.

T-CREST [4] was a European research project that aimed to develop a new generation of time-predictable systems based on multicore processors. A T-CREST multicore consists of Patmos processor cores [20] connected via the Argo NoC [9]. The Argo NoC is a TDM-based NoC similar to S4NOC, but with the routing information in the header and a default packet length of three words. The main difference between T-CREST and InterPRET is that the T-CREST cores share external memory. Therefore, they compete for that memory, which is resolved by a time-division multiple access protocol. InterPRET uses local memories for instructions and data and communicates via the NoC only.

MERASA [5] is a time-predictable multicore architecture that guarantees temporal isolation of a single hard real-time task from up to three non-real-time tasks on each core. MERASA contains both scratchpad memories and a cache hierarchy. This enables both timing predictability and high performance. It is less flexible than the InterPRET, which can support multiple hard real-time tasks per core. Furthermore, the NoC-based communication makes InterPRET more scalable.

Vicuna [17] is a time-predictable vector processor for highly parallel real-time applications. Vicuna has a single-instruction multiple-data architecture. Vicuna implements the integer and fixed-point instructions of the RISC-V vector extension. As such, Vicuna is not a stand-alone processor and can only act as a co-processor. InterPRET, on the other hand, is in itself a complete system. A single core of the InterPRET could use Vicuna as a co-processor.

MultiPRET [7] is a multicore PRET machine based on the FlexPRET. Inter-core communication is achieved through shared memory over a TDMA bus. For configurations with tens of cores, the shared memory becomes a bottleneck and a NoC-based architecture, like the InterPRET, is preferable.

The Reduced Complexity Many-Core project from the University of Augsburg adapts RISC-V architectures and includes the PaterNoster [14] NoC. PaterNoster uses packets with single-word flits and a network interface with a single FIFO queue connected to the memory stage of a RISC-V processor. The RISC-V instruction set has been extended with transmit and receive instructions that block until a free slot is available or a packet is found, respectively. Our NoC uses a similar architecture with TDM-based scheduling and a network interface (NI) that can be accessed with normal load and store instructions.

3 BACKGROUND

The InterPRET architecture is based on two open-source projects: (1) The RISC-V PRET machine FlexPRET [24], and (2) the real-time NoC S4NOC [18]. In this section we provide background on them.

3.1 FlexPRET

The FlexPRET CPU architecture is the third generation of multi-threaded PRET machines designed at UC Berkeley [11] and was the result of Michael Zimmer's PhD thesis [24]. It is a five stage in-order processor implementing the base 32-bit RISC-V ISA (RV32I). FlexPRET has no branch predictor and uses fine-grained multithreading with a barrel register file to avoid pipeline bubbles. Compared to earlier-generation PRET machines, it has a more flexible thread scheduler which enables round-robin scheduling of threads configured as *hard-real time threads* (HRTTs) and opportunistic scheduling of threads configured as *soft real-time thread* (SRTTs).

The FlexPRET introduces three timing instructions for controlling the timing of each thread: (1) `delay_until` halts execution until some absolute point in the future; (2) `wait_until` halts until an interrupt, with the possibility of a timeout; and (3) `interrupt_expire` schedules an interrupt to occur at an absolute point in time.

3.2 The Network-on-Chip

The S4NOC [18] is designed to be used in real-time systems. It guarantees a maximum latency and a minimum bandwidth for all channels. S4NOC consists of a network of routers, connected with four ports in a bidirectional torus. The fifth port of the router is the local port and is connected to the network interface (NI). As shown in Figure 1, we connect a FlexPRET to the processor interface of each NI. The S4NOC connects each FlexPRET core (FPn) through a network interface (NI) to the local port of a router (R).

Access to shared resources in S4NOC, e.g., router and links, are statically scheduled based on TDM. Each slot in the schedule is a single clock cycle during which any given resource of the router (link, multiplexer, output register) may be used by a single packet. The packet moves from the injection point at the sender NI through the network of routers until the destination NI in a pipelined fashion, one hop each clock cycle.

The S4NOC routes packets from a sending core to a receiving core. A packet is a single word of 32 bits. As all traffic is statically

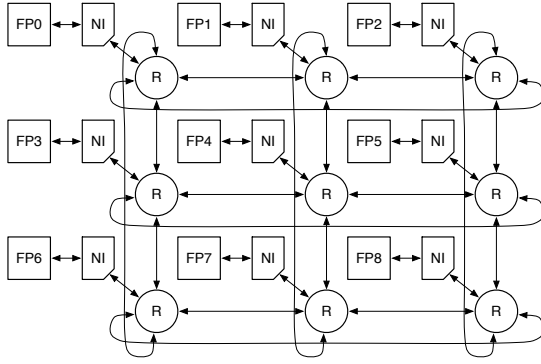


Figure 1: A bi-torus configuration of an S4NOC with FlexPRET cores FP_n , network interfaces NI, and routers R.

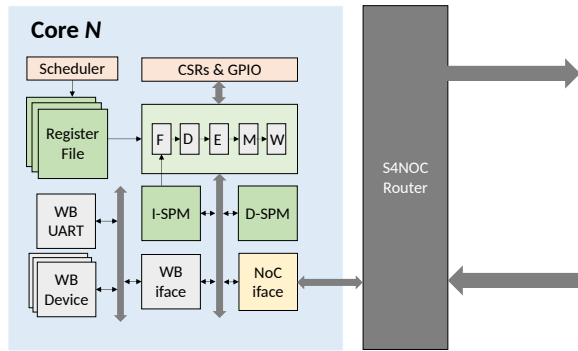


Figure 2: A single FlexPRET core part of the InterPRET

scheduled, there are no conflicts on any shared resources, such as links or crossbars. Without conflicts, there is no need for buffering in the routers, flow control between routers, or credit-based flow control between the NIs. The schedule is computed offline and stored in the routers. The schedule contains for each clock cycle the directions of the 5:4:1 multiplexers, which is 10 bits per clock cycle. Therefore, the size of the schedule table is negligible. An all-to-all core communication graph for N cores has $N \times (N - 1)$ virtual channels. As an example of a 3×3 multicore, the NoC has 72 virtual channels. The scheduler can implement these virtual channels with a 10-slot TDM schedule. This schedule is only 2 slots longer than the minimum theoretical period needed by the 8 outgoing and 8 incoming virtual channels at each core.

The NI is split into two components: a standard bus interface for the processor and the buffering to decouple the processor read/write interface and the NoC router's local port. The processor interface maps the ready/valid-based receive and transmit queues to memory-mapped IO ports. Those ports may block if there is no data to read or space to send. In addition to those two ports, the NI provides status bits to poll on those conditions to avoid blocking operations.

4 INTERPRET

InterPRET is a configurable multicore architecture. It consists of a number of FlexPRET cores interconnected by the S4NOC. This yields an end-to-end timing-predictable message-passing-based

```

1 void main0(void) { // Program executed on Core 0
2     uint32_t count = 0, release_time = START_TIME;
3     delay_until(release_time);
4     while(true) {
5         noc_send(1, ++count, NON_BLOCKING);
6         release_time += PERIOD;
7         delay_until(release_time + SEND_LATENCY);
8         noc_receive(&count, NON_BLOCKING);
9         release_time += PERIOD;
10        delay_until(release_time);
11    }
12 }
13 void main1(void) { // Program executed on Core 1
14     uint32_t count, release_time = START_TIME;
15     delay_until(release_time + SEND_LATENCY);
16     while(true) {
17         noc_receive(&count, NON_BLOCKING);
18         release_time += PERIOD;
19         delay_until(release_time);
20         noc_send(0, ++count, NON_BLOCKING);
21         release_time += PERIOD;
22         delay_until(release_time + SEND_LATENCY);
23     }
24 }
25 void main(void) { // Entry point for both cores
26     switch(read_coreid()) {
27         case 0: main0(); break;
28         case 1: main1(); break;
29     }
30 }

```

Listing 1: C code for PingPong on InterPRET

multicore. Figure 2 shows how a single core of the InterPRET is connected to the rest of the SoC through the NoC. Each FlexPRET core supports hardware multithreading through a barrel register file where each hardware thread has its own set of registers. The NoC interface is memory-mapped together with the data scratchpad memory (D-SPM), the instruction scratchpad memory (I-SPM), and the Wishbone interface.

The NoC is supported by a simple header files library that exposes an API to send data to other cores and receive data from them. The CPU is connected to the NI of the NoC, which contains a buffer that might be full or empty. A timeout is therefore passed to the send and receive functions.

Listing 1 shows an InterPRET program using the NoC library to facilitate inter-core communication. Both cores will store the same program in their I-SPM and start execution from the same `main`. Using a hardwired register, each core can jump to its respective main function, `main0` and `main1`. The program implements a ping-pong pattern where Core 0 sends a value to Core 1 and Core 1 responds with the same value. The non-blocking NoC API is used which means that for this program to work, the messages must have arrived by the time `noc_receive` is called (Lines 8 and 17). This is ensured by the calls to `delay_until` which time synchronize the two cores. This program will be further analyzed in Section 5.

4.1 Predictable Timing

Edwards et al. distinguish between repeatable timing and predictable timing [1]. A sequence of instructions is said to be *repeatable* if, given the same inputs, every correct execution of the

same sequence will lead to the same observable state. *Predictable* timing is a repeatable property that can be determined in finite time from a specification.

A single FlexPRET core achieves repeatable timing through fine-grained multithreading, which enables temporal isolation and constant instruction latencies, and the use of scratchpad memories. However, it is also the programmer's job to use the hardware resources appropriately to preserve this property. For example, the hardware threads are required to be scheduled at a constant rate (i.e., with a constant *scheduling frequency*).

FlexPRET can deliver repeatable behavior, but can it deliver predictable timing? Timing analyses typically involves two dimensions: the path problem, which is the challenge of finding the longest path in a program, and the state problem, which is the challenge of finding the least processor state that yields the longest execution time [21]. Since a HRTT are guaranteed a constant scheduling frequency and constant latencies of instructions [23, 24], the use of FlexPRET obviates the need to perform complex modeling of the microarchitecture. In other words, the timing analysis for FlexPRET is more tractable because it is solely concerned with the path problem.

The timing predictability of InterPRET can be attributed to its time-predictable subsystems. Given programs mapped to the FlexPRET cores and a set of inputs, it is possible to derive the execution times of programs running on the FlexPRET cores and the communication times of the S4NOC. These quantities can be used to further calculate end-to-end latencies between cores. Computing a *precise* end-to-end latency depends on a detailed analysis of the aforementioned path problem and an analysis of the TDM schedule. On the other hand, computing *upper bounds* on the end-to-end latencies is a simpler and more composable form of the timing analysis problem. The *worst-case* execution time of programs and the *worst-case* communication latency of the S4NOC can be derived *independently* to calculate the *worst-case* end-to-end latency.

4.2 Software-defined Devices

PRET machines like InterPRET support the implementation of timing critical functionalities, such as IO devices, purely in software. We call such programs Software-defined Devices or *SDDs* for short. We envision a suite of open-source SDDs that can be picked and placed on the different HRTTs of an InterPRET. This can enable using the InterPRET as a generic sensor interface with a portable solution to the synchronization problem [8].

This is in line with XMOS' vision of a software-defined SoC. Currently, we have implemented one such device, *sdd_uart*, which is a high-speed serial device. The *sdd_uart* runs in a single HRTT and uses the timing instruction *delay_until* to wake up at the right moment to read or write serial port data using a GPIO pin. When the thread is sleeping, other SRTTs can use the HRTT unused cycles.

The *sdd_uart* interacts with the rest of the InterPRET through two circular buffers. Data to be transmitted can be enqueued to the TX buffer, and received data from outside will be enqueued by the UART thread to the RX buffer.

4.3 Hardware Devices

There might be other reasons, besides timing predictability, to introduce devices in an SoC. For instance, to perform compute-intensive

	LUTs	LUTRAMs	FFs	BRAMs
FlexPRET	2526	160	2010	18
S4NOC	175	209	327	0
WB bus	22	0	38	0
WB UART	109	16	95	0

Table 1: Resource utilization for different components of the InterPRET, for a single core.

tasks like digital signal processing or cryptography. For these cases, using HRTTs might not be sufficient and a hardware accelerator might be required. The FlexPRET core is, for this purpose, extended with a Wishbone (WB) bus [16] to which hardware devices can be connected. The WB bus cannot be connected directly to the FlexPRETs bus interface because access latency to WB devices is not constant. The FlexPRET expects single-cycle writes and two-cycle reads with no stalling. The solution is to create a memory-mapped WB master device with predictable access time. The WB master will, in turn, be connected to the WB bus and communicate with the WB devices on behalf of the CPU.

4.4 Bootloader

To enable loading different programs into the instruction memory, a boot procedure is needed. The FlexPRET cores each have a small bootloader hardwired into the first section of the instruction memories. Each core also has a hardware UART device connected to the WB bus. The bootloader receives the application, wrapped in a simple protocol, over this UART device. The application is written, word-by-word, into the I-SPM. After complete reception of the program, the bootloader jumps to the beginning of the newly-written application and starts executing it. Each FlexPRET core of the InterPRET has its UART device connected to the same top-level pins and executes an identical bootloader. The result is that the same program is loaded into each I-SPM of all cores simultaneously.

5 EVALUATION

We evaluated our InterPRET implementation through benchmarks and case studies using the Zedboard FPGA, which has a XC7Z020 SoC FPGA with 630 KB on-chip memory and 13,300 logic slices.

5.1 Resource Utilization

To evaluate the resource utilization, we synthesize InterPRETs of various sizes targeting a XCKU035. Note that we target a different FPGA for resource utilization so that we can synthesize larger multicores. The FlexPRET cores of the InterPRET are configured with 16 KB I-SPM and D-SPM, the round-robin scheduler, four hardware threads, and without a hardware multiplier. The S4NOC is configured with a 32-bit data word size and NI buffer size of two.

Table 1 shows the resource utilization of the different components of an InterPRET. The numbers reported are the utilization for a single core of an InterPRET. The S4NOC is configured as a 2x2 grid and we divide the resource utilization by four. The NoC has low resource utilization compared to the FlexPRET, except for the LUTRAMs. These are used for the network interface buffers, which make up the backbone of the NoC.

	LUTs	LUTRAMs	FFs	BRAMs
4-core	11402 (1.0x)	1120 (1.0x)	9780 (1.0x)	74 (1.0x)
9-core	27346 (2.4x)	3412 (3.0x)	24271 (2.5x)	166.5 (2.3x)
16-core	50554 (4.4x)	8284 (7.4x)	47315 (4.8x)	296 (4.0x)

Table 2: Resource utilization for different InterPRET configurations.

Table 2 shows how the resource utilization of InterPRET scales with the number of cores. The usage of combinatorial logic (i.e., number of LUTs) and memory (BRAMs) scale linearly with the size of the InterPRET. That is to say, if the number of cores is doubled, so will roughly the utilization of these resources. However, the usage of distributed RAM (LUTRAMs) and registers (FFs) scales super-linearly. This is mainly due to buffering at each NI of the S4NOC. Each NI contains, besides shared input and output FIFOs, one decoupling buffer for each other node in the system. This means that a 2x2 configuration will have 4 NIs each with three buffers, while a 3x3 configuration will have 9 NIs each with eight buffers.

5.2 Latency, Jitter, and Bandwidth

To measure NoC sending and receiving latency, we implemented send and receive functions in assembly and called them from a simple C program to approximate a realistic amount of software overhead. In simulation, we measured communication latencies ranging from 35 to 44 cycles, inclusive, when sending data from core 0 to core 1, where InterPRET is configured with four cores and each core has a single HRTT that is scheduled in every clock cycle. Although we anticipate that in typical programs it will be infeasible to predict the communication latency exactly, it is straightforward to compute the jitter: The difference between the maximum latency and the minimum latency is 9 cycles because a 6-cycle busy-wait loop on the receiving side contributes 5 cycles of variability while the 5-cycle TDM schedule contributes 4 cycles of variability.

It is possible to achieve a high bandwidth without repeated software-level ready/valid checks if both the sender and receiver run in HRTTs that produce (respectively, consume) data at equal rates. In this case, the rate at which data can be sent over the NoC after initiating communication is limited only by the TDM schedule and by the rate at which software can produce the data. Because the TDM schedule is five cycles long on a 4-core InterPRET, a new word can be sent to each receiving core as frequently as every five clock cycles using an unrolled loop. Our implementation uses a main loop with an unrolled inner loop that sends 16 words, followed by a remainder loop. This allows it to send 500 words in 2744 clock cycles. This results in a bandwidth of 1 word per 5.49 clock cycles, which is close to the maximum NoC bandwidth of 1 word per 5 clock cycles for a single channel. On an FPGA running at 50 MHz, this translates to 291.5 Mbit/s, while the upper bound imposed by the TDM schedule is 320 Mbit/s.

Similar calculations apply to the other bandwidths reported in Table 3. Note that the maximum achievable bandwidth between any given pair of cores decreases as the number of cores increases. The TDM schedule on a 9-core InterPRET is 10 cycles long, yielding an

	20 words	100 words	500 words
4-core	165.8 Mbit/s	258.9 Mbit/s	291.5 Mbit/s
9-core	114.7 Mbit/s	148.3 Mbit/s	157.5 Mbit/s
16-core	67.5 Mbit/s	80.2 Mbit/s	83.4 Mbit/s

Table 3: Measured message-passing bandwidth for different InterPRET configurations.

upper bound of 160 Mbit/s, while the TDM schedule on a 16-core InterPRET is 19 cycles long, yielding an upper bound of 84.2 Mbit/s.

5.3 Case Study 1: Multicore PingPong

To demonstrate the time-predictability of InterPRET, we again turn to Listing 1 from Section 4. In this program, `noc_send()` and `noc_receive()` both take a timeout argument set to `NON_BLOCKING`. This means that the function will return immediately, with or without data. In order for this program to work the sender and receiver must be synchronized. Due to the timing predictability of FlexPRET this can easily be achieved without additional communication. The use of `delay_until()` enables both cores to synchronize using *time*. When receiving, the core will add an offset, `SEND_LATENCY`, to the call to `delay_until()`. This offset enables the sending core to wake up before the receiving core to account for the communication latency. As we shall see, this offset is statically computable due to the end-to-end time-predictability of the InterPRET.

In the following, we will compute a feasible value for `SEND_LATENCY`, L_{send} , ensuring no lost messages. The value can be expressed as

$$L_{send} = L_{max}^{tx} + L_{max}^{noc} - L_{min}^{rx} \quad (1)$$

where L_{max}^{tx} is the maximum latency at the sender side, i.e., the WCET from when `noc_send()` is called to when the message and destination address are enqueued to the NI buffer of the NoC. L_{max}^{noc} is the worst-case message latency between the NIs of two cores communicating through the NoC. Finally, L_{min}^{rx} is the minimum latency at the receiver side, i.e., the best case execution time from when `noc_receive` is called to when the NI buffer is read.

We can express L_{max}^{tx} as a function of c_{max}^{tx} , the worst-case number of *thread cycles*, N_t the number of HRTTs, and p the clock period of the CPU.

$$L_{max}^{tx} = c_{max}^{tx} \times p \times N_t \quad (2)$$

The thread cycles, c_{max}^{tx} , must not be confused with clock cycles. On the FlexPRET, all instructions consume a known number of thread cycles, typically 1. It means that the thread executing it only spends a single clock cycle on it. However, the total number of clock cycles spent will also depend on the frequency at which the thread is scheduled.

L_{min}^{rx} is expressed similarly. For a quad-core InterPRET with a round-robin scheduler and 4 HRTTs we get $c_{max}^{tx} = 40$ and $c_{min}^{rx} = 20$, yielding $L_{max}^{tx} = 3200ns$ and $L_{min}^{rx} = 1600ns$.

The worst case message latency between two cores, L_{max}^{noc} , is computed using the following formula extracted from [19]:

$$L_{max}^{noc} = (2 \times l + s + h) \times p \quad (3)$$

where l is the NI latency experienced at both ends of the communication, s is the TDM schedule length, and h is the maximum number

of hops over the NoC. For our quad-core configuration we have $h = 2$, $s = 5$, $l = 2$ and $p = 20$ ns, yielding $L_{max}^{noc} = 180$ ns.

Finally, we can compute $L_{send} = 1780$ ns based on Eq. 2. This value was confirmed experimentally on the FPGA.

5.4 Case Study 2: Software-defined UART

To demonstrate InterPRET's capabilities as a software-defined SoC, we developed `sdd_uart`, a software-defined UART. On the InterPRET, we use `sdd_uart` to implement a printing service on one of the HRTTs on Core 0. Other threads in the system can schedule the prints by enqueueing data to a circular buffer which is read by the `sdd_uart`-thread. If the thread is on another core, it can send the data to be printed via the NoC. A simple protocol is used where the number of bytes the thread wants to print is written to the buffer before the actual string of characters. The maximum achievable baud rate is limited by the *critical path* between driving the GPIO pin for two consecutive bits. We tested `sdd_uart` running on the FPGA clocked at 50MHz. It was connected to a PC and could reliably send data at baud rate of 270000.

6 CONCLUSION

We presented InterPRET, a multicore processor consisting of interconnected FlexPRET cores for time-predictable execution of real-time programs. To make the use of InterPRET practical, we believe that a time-aware message-passing programming model is key. As future work, we plan to make InterPRET programmable using Lingua Franca [12]—a reactor-oriented coordination language that provides deterministic-by-default concurrency and a semantic notion of time. The Chisel implementation of InterPRET is open source and available on GitHub¹.

ACKNOWLEDGMENTS

This work was supported by the Research Council of Norway (RCN) grant number 327538, Multi-Sensor Data Timing, Synchronization and Fusion for Intelligent Robots and grant number 223254, the center of excellence NTNU AMOS. It was also funded in part by the National Science Foundation (NSF), award #CNS-2233769 (Consistency vs. Availability in Cyber-Physical Systems) and the iCyPhy Research Center (Industrial Cyber-Physical Systems), supported by Denso, Siemens, and Toyota. The authors also thank Samuel Berkun, Efsane Soyer, and Matthew Chorlian for their contributions. Finally we thank the reviewers for their valuable feedback which helped us improve the quality of the paper

REFERENCES

- [1] Stephen A. Edwards, Sungjun Kim, Edward A. Lee, Isaac Liu, Hiren Patel, and Martin Schoeberl. 2009. A Disruptive Computer Design Idea: Architectures with Repeatable Timing. In *International Conference on Computer Design (ICCD)*. IEEE, 54–59. <https://doi.org/10.1109/ICCD.2009.5413177>
- [2] Stephen A. Edwards and Edward A. Lee. 2007. The Case for the Precision Timed (PRET) Machine. In *Design Automation Conference (DAC)*.
- [3] Jon Perez Cerrolaza et al. 2020. Multi-Core Devices for Safety-Critical Systems: A Survey. *ACM Comput. Surv.* 53, 4, Article 79 (aug 2020), 38 pages. <https://doi.org/10.1145/3398665>
- [4] Martin Schoeberl et al. 2015. T-CREST: Time-predictable Multi-Core Architecture for Embedded Systems. *Journal of Systems Architecture* 61, 9 (2015), 449–471. <https://doi.org/10.1016/j.sysarc.2015.04.002>
- [5] Paolieri et al. 2013. A Hard Real-Time Capable Multi-Core SMT Processor. *ACM Trans. Embed. Comput. Syst.* 12, 3, Article 79 (apr 2013), 26 pages. <https://doi.org/10.1145/2442116.2442129>
- [6] R.H. Dennard et al. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (1974), 256–268. <https://doi.org/10.1109/JSSC.1974.1050511>
- [7] Nicolas Hili, Alain Girault, and Eric Jenn. 2019. Worst-case reaction time optimization on deterministic multi-core architectures with synchronous languages. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 1–11.
- [8] Erling R. Jellum, Torleiv H. Bryne, Tor Arne Johansen, and Milica Orlandić. 2022. The Syncline Model - Analyzing the Impact of Time Synchronization in Sensor Fusion. In *2022 IEEE Conference on Control Technology and Applications (CCTA)*. 1446–1453. <https://doi.org/10.1109/CCTA49430.2022.9966179>
- [9] Evangelia Kasapaki, Martin Schoeberl, Rasmus Bo Sørensen, Christian T. Müller, Kees Goossens, and Jens Sparso. 2016. Argo: A Real-Time Network-on-Chip Architecture with an Efficient GALS Implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24 (2016), 479–492. <https://doi.org/10.1109/TVLSI.2015.2405614>
- [10] Hermann Kopetz. 1997. *Real-Time Systems : Design Principles for Distributed Embedded Applications*. Springer.
- [11] E. Lee, J. Reineke, and M. Zimmer. 2017. Abstract PRET Machines. In *2017 IEEE Real-Time Systems Symposium (RTSS)*. 1–11.
- [12] Marten Lohstroh, Martin Schoeberl, Andrés Goens, Armin Wasicek, Christopher Gill, Marjan Sirjani, and Edward A. Lee. 2019. Actors Revisited for Time-Critical Systems. In *Proceedings of the 56th Annual Design Automation Conference 2019 (Las Vegas, NV, USA) (DAC '19)*. ACM, New York, NY, USA, Article 152, 4 pages. <https://doi.org/10.1145/3316781.3323469>
- [13] David May. 2012. The X MOS architecture and XS1 chips. *IEEE Micro* 32, 6 (2012), 28–37.
- [14] Jörg Mische and Theo Ungerer. 2012. Low Power Flitwise Routing in an Unidirectional Torus with Minimal Buffering. In *Proceedings of the Fifth International Workshop on Network on Chip Architectures (Vancouver, British Columbia, Canada) (NoCArc '12)*. ACM, New York, NY, USA, 63–68. <https://doi.org/10.1145/2401716.2401730>
- [15] Jan Nowotzsch and Michael Paulitsch. 2012. Leveraging Multi-core Computing Architectures in Avionics. In *2012 Ninth European Dependable Computing Conference*. 132–143. <https://doi.org/10.1109/EDCC.2012.27>
- [16] OpenCores. 2019. WISHBONE Bus Specifications. <https://opencores.org/projects/wishbone>
- [17] Michael Platzter and Peter Puschner. 2021. Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 196)*, Björn B. Brandenburg (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1:1–1:18. <https://doi.org/10.4230/LIPIcs.ECRTS.2021.1>
- [18] Martin Schoeberl, Florian Brandner, Jens Sparso, and Evangelia Kasapaki. 2012. A Statically Scheduled Time-Division-Multiplexed Network-on-Chip for Real-Time Systems. In *Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS)*. IEEE, Lyngby, Denmark, 152–160. <https://doi.org/10.1109/NOCS.2012.25>
- [19] Martin Schoeberl, Luca Pezzarossa, and Jens Sparso. 2019. S4NOC: a Minimalistic Network-on-Chip for Real-Time Multicores. In *Proceedings of the 12th International Workshop on Network on Chip Architectures*. ACM, 8:1–8:6. <https://doi.org/10.1145/3356045.3360714>
- [20] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. 2018. Patmos: A Time-predictable Microprocessor. *Real-Time Systems* 54(2) (Apr 2018), 389–423.
- [21] Sanjit A. Seshia and Jonathan Kotker. 2011. GameTime: A toolkit for timing analysis of software. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.
- [22] X MOS. 2022. X MOS announces software-defined SoC platform now compatible with RISC-V. <https://www.xmos.ai/xmos-announces-software-defined-soc-platform-now-compatible-with-risc-v/>
- [23] Michael Zimmer. 2015. *Predictable Processors for Mixed-Criticality Systems and Precision-Timed I/O*. Ph. D. Dissertation. EECS Department, University of California, Berkeley. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-181.html>
- [24] Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. 2014. FlexPRET: A Processor Platform for Mixed-Criticality Systems. In *Real-Time and Embedded Technology and Application Symposium (RTAS)*. <http://chess.eecs.berkeley.edu/pubs/1048.html>

¹<https://www.github.com/lf-lang/interpret>