Multicore Models of Communication for Cyber-Physical Systems

Martin Schoeberl[1] [0000−0003−2366−382X]

Department of Applied Mathematics and Computer Science
Technical University of Denmark, Kgs. Lyngby, Denmark masca@dtu.dk

Abstract. Cyber-physical systems are systems where the environment interacts with computers (the cyber part) with real-time constraints. Emerging technologies, such as artificial intelligence and machine learning, call for ever-increasing processing power. However, for real-time systems, we need to prove statically that this processing demand can be performed within strict deadlines.
This paper explores a time-predictable multicore architecture for those demanding cyber-physical systems. We explore different models of communication between those multiple cores. We compare the message passing model on top of a network-on-chip with message passing on two forms of shared scratchpad memory.

Keywords: real-time systems · multicore communication · time-predictable computer architecture.

1 Introduction

Future cyber-physical systems may be in need of higher computing power. One way to increase computing power is to integrate multiple processing cores in a single chip to form a multicore processor. Cyber-physical systems often need to react to the environment within a guaranteed deadline. We call those systems real-time systems. If such a system is part of a safety-critical system, we need to guarantee that all deadlines are met. Such proof includes worst-case execution time (WCET) analysis of individual tasks, analysis of communication time, and schedulability analysis.

Multicore processors used in cyber-physical systems need to support time-predictable computation and communication. As communication via shared main memory supported by a cache coherence protocol is hardly time-predictable, we need other forms of core-to-core communication.

This paper explores different models of communication between processing cores and the hardware support for it. We present forms of shared on-chip memories, links between processor cores, and network-on-chip architectures. In this paper, we include only solutions that are time-predictable, except describing the baseline of a hardly time-predictable shared main memory with cache coherence.

Shared on-chip memories with a time-predictable arbitration, such as time-division multiplexing, provide an efficient solution for around a dozen cores. For more cores, a distributed communication architecture, such as a network-on-chip, is a better scaling solution.

In this paper, we use the term task as a notion of parts of a program that can execute concurrently. We avoid the term thread, as threads are usually associated with a single form of concurrency: communication via data in shared memory, protected by locks. Tasks need to communicate when working together as an application.

The contribution of this paper is a detailed overview of several communication architectures for a real-time multicore processor. The overview may also serve as a small survey of real-time multicore communication architectures. Furthermore, we picked several architectures and compared them with an evaluation of message passing. Our overall goal is to build time-predictable computer architecture [32] for future demanding cyber-physical systems. Initial ideas on models of communication for multicore processors have been presented in [39].

This paper is organized into 5 sections: Section 2 presents the software view of multicore communication. Section 3 is the main section, describing several hardware architectures to support time-predictable multicore communication. Section 4 evaluates several of the presented architectures with a message passing microbenchmark. Section 5 concludes.

2 The Software View

When multiple tasks shall work together towards completing work, they need to communicate in some form. This combination of tasks and forms of communication is also called the "model of computation." The Ptolemy II handbook [30] gives a good overview of those different forms. In the following sections, we focus on three example models of computation and communication.

2.1 Communicating Sequential Processes

One of the first approaches to establish message passing between tasks was Hoare's communicating sequential processes, CSP for short [15]. The CSP concept became popular enough that even a programming language, Occam [23], was developed to include CSP in the language.

Transputers [43,16], a unique form of processors, where developed to execute Occam programs. Transputers included hardware support for the Occam channels. The idea was to build massive parallel multiprocessors. However, in the mid of the '80s the performance increase of standard processors was still around 50 % per year [13], and there was no need for multiprocessor systems. A single task program is easier to develop and test. Dividing an algorithm into multiple tasks that communicate via channels is hard, and errors can lead to hard-to-debug blocking of tasks. Therefore, CSP and transputers did not become a success story.

2.2 Multithreading

Early forms of multiprogramming consisted of using individual programs that communicate. One form of communication was the usage of Unix pipes, where the output of one program is fed as input to another program. A Unix pipe represents a stream with

one writer and one reader process. Message passing can easily be built on top of such a stream. A tighter form of communication between programs was the creation of a shared memory space by the operating system. However, those multiple programs still run as individual processes protected from each other by the operating system.

To simplify multiprogramming, the concept of multiple threads of execution in a single process was developed. Those threads share memory and use data allocated on the heap for communication. Those data structures are usually protected by locks [14]. This programming paradigm became especially popular when Java supported threads and locks as part of the core language definition.

A runtime system can map multiple threads to multiple cores in a multicore processor. Also, the communication via shared objects is handled by a cache coherence protocol.

At the time of this writing, multithreading with shared data is the most popular approach to use with concurrent tasks. However, getting the locking of objects right for multithreaded programs is far from trivial. Locking also is a bottleneck for scaling programs for many cores. Therefore, the current trend is to explore message passing again in the form of actors.

2.3 Actors and Message Passing

The concept of actors is currently becoming popular through the Akka[1] toolkit. Akka is a library and runtime to support concurrent and distributed applications. The primary programming model for multiple tasks is actor-based. Akka is written in Scala but can be used from programs written in Java or Scala.

Actors in Akka are the tasks that communicate via message passing. In contrast to CSP, the message passing is asynchronous. Typical Akka programs avoid shared mutable data and locks to protect them.

However, non-constrained asynchronous message passing may lead to buffer overflow and is hardly time-predictable. Stricter forms of communication are, for example, synchronous data flow (SDF) [19]. An SDF actor fires (executes) when all input ports contain their fixed number of tokens. With the fixed number of tokens consumed and produced, buffers are bounded, and for a single core, a statically schedule for the actor firing can be computed.

Recent work extends actors for precise timing in cyber-physical systems [22]. The actors, called reactors in the paper, have strict rules on fire order and mutual exclusion of different reactions. Reactors include the notion of delays and deadlines. Delays allow for physical time to pass, and deadlines are a contract with the environment. WCET and schedulability analysis of reactions can be used to check if all deadlines can be met.

3 Communication Hardware

Message passing can be implemented on top of different communication infrastructures. In contrast, the concept of shared objects is usually implemented on top of cache-coherent, shared main memory only. Therefore, message passing is the more hardware

[1] Available at `https://akka.io/`

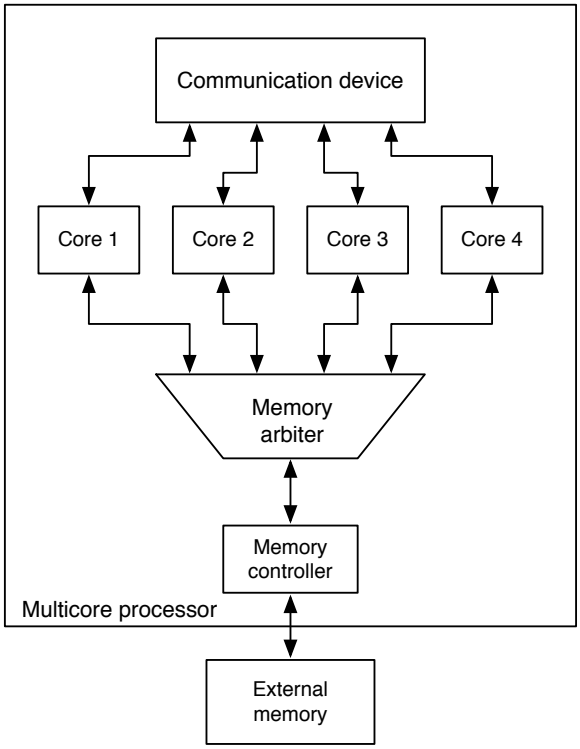

Fig. 1. A multicore processor with the cores connected to (1) an arbiter to the memory controller for the shared, external memory and (2) to the communication hardware.

friendly approach for communication. In the following sections, we discuss several different hardware mechanisms for communication between multiple cores on a chip multicore.

Figure 1 shows a multicore processor where the cores are connected to (1) external memory via a memory arbiter and (2) to a communication device. That communication device is the topic of this paper, and we discuss variations of it in the following sections.

3.1 Shared Main Memory

The state-of-the-art communication mechanism for multicore processors is shared main memory. Objects are allocated in the main memory, and the access to the objects is protected by locks. As access latency to main memory is in the range of hundreds of processor clock cycles, several levels of cache are introduced. It is not uncommon to include 3 levels of cache, where the 2nd and 3rd levels of cache are shared between the cores. The first level of cache is usually core local. Therefore, when sharing data, these local caches need to be kept coherent with a cache-coherent protocol. As this cache coherence protocol is an all-to-all communication, it scales only to a few tens of processor cores.

However, the main issue with shared memory backed up by a cache coherence protocol is that it is barely time-predictable. The WCET analysis of tasks needs to include an analysis of which memory blocks are in the cache and in which caches. WCET analysis is further complicated by the fact that on a multicore, we have true concurrency where individual tasks influence the occupancy of the shared caches. This problem would need a WCET analysis that includes all tasks in the system. We are not aware of any WCET analysis tool (except niche research experiments) that supports multiple tasks and multiple levels of caches, including the cache coherence protocol. The industry standard WCET tool aiT [12] supports single tasks only. We quote from AbsInt's website:[2]

> aiT computes an upper bound of the WCET of a task. A task must be a sequentially executed piece of code, i.e. there must not be any threads, parallelism, or external events. aiT assumes no interference from the outside. Effects of exceptions, interrupts, DRAM refreshes, input/output, timers and other processors or co-processors are not reflected in the predicted runtime and have to be considered separately, e.g. via quantitative analysis.

However, we are aware that realistic applications and their data are too large to fit in on-chip memory. Therefore, some code and data need to be loaded into external memory. To provide time-predictable access to external memory, we propose to use a time-division-multiplexing (TDM) arbiter for the memory accesses [35].

3.2 Network-on-Chip

Network-on-chip (NoC) technology [6] is an alternative to cache coherence based inter-core communication. A NoC is a distributed architecture, and therefore the provided bandwidth scales well with the number of cores. A NoC connects cores (also called processing elements in NoC literature) to a network of routers. In most cases, one router serves one core. The routers are connected in a network, where mesh and torus are the most common organizations.

A NoC itself does not yet provide a communication mechanism. Between the core and a router, the network interface (NI) provides an interface to the network. NoCs are used for a wide variety of traffics: serving cache coherence traffic, access to a memory controller and external memory, streaming between cores, message-passing between cores, and access to remote on-chip memories. The NI determines what kind of traffic is supported.

Many routers (and NIs) are optimized for the average case performance with buffers and dynamic arbitration at each router. Those NoCs are hardly time-predictable. For real-time systems, two mechanisms are popular: rate control at the injection site or TDM arbitration at the routers.

Rate control, also called traffic shaping, limits the number of packets injected into the NoC. Network calculus [4,5,18] is used to compute bounds on buffer sizes and

[2] `https://www.absint.com/ait/features.htm`

bounds on latencies. The Kalray multicore processor [7] is especially designed to support time-predictable message passing with rate control in the sender and no further flow control within the NoC [8].

With a static schedule performing TDM arbitration in the NoC routers, there is no traffic conflict, and the worst-case message latency can be statically computed. Æthereal [9] is such a NoC that uses TDM where slots are reserved to allow a block of data to pass through the NoC router without waiting or blocking traffic. Slot tables with routing information are contained in the routers, and no arbitration or link-to-link flow control is required. Instead, credit-based flow control is applied for end-to-end control, saving buffer space between links. aelite, a light version of Æthereal, only offers guaranteed services resulting in a simpler router design [11].

The Argo NoC [17] is another NoC that uses TDM based arbitration of resources. Compared to Æthereal, Argo also uses the same TDM schedule in the NI [42] to time-multiplex the NI resources. The Argo NI offers TDM-based DMA transfer of data from the local memory across the NoC and into the local memory of another core. Argo supports a global asynchronous, local synchronous system with an asynchronous router design and mesochronous (same clock source, but variable upwards bounded skew allowed) NIs.

While Æthereal uses TDM at the routers, it uses buffers with flow-control in the NIs. In contrast, the Argo NoC [17] uses TDM for the arbitration in the routers *and* at the NI [42], resulting in an end-to-end TDM schedule. S4NOC [37,36] is a TDM based NoC, simpler than Argo, with FIFO buffers as NI. We use S4NOC in the evaluation section.

The Real-Time Capable Many-Core Model proposes many cores with a static switched NoC with TDM-based arbitration [24]. The project also proposes avoiding shared memory altogether and supporting timing analysis by using a fine-grained message passing NoC [25].

Paukovits and Kopetz use a time-triggered NoC for the time-triggered system-on-chip (TTSoC) architecture [28]. The main difference to other NoC designs is the absolute time format, which is *not* directly related to the clock frequency. The macro tick is a power of two fraction of a second and the basis for the TDM slotting. The idea behind this time format is a good integration with off-chip versions of time-triggered networks.

When comparing TDM arbitration with rate control and network calculus [31], TDM arbitration results in shorter worst-case latencies while network calculus leads to higher bandwidth. However, using TDM for arbitration leads to simpler routers and network interfaces than supporting dynamic arbitration and buffering NoC.

3.3 Shared Scratchpad Memory

While NoCs can provide a high bandwidth communication path, their usage is more elaborated. I.e., messages need to be setup and explicitly sent to other cores. An alternative is to use on-chip memory, also called scratchpad memory (SPM), that is shared between several cores. For a small number of cores that memory can be shared by all the cores. However, with an increase in the number of cores, this solution does not scale. Therefore, several shared on-chip memories can be shared only by a subset of the cores. These subsets can be disjoint, as in the Kalray processor, to form clusters, which

are connected by a NoC. Alternative, these sets can overlap to provide a communications path between neighboring cores.

The Kalray manycore processor [7] is specially designed for time-critical computation. The processor is organized in 16 clusters of 16 cores. Each core within a cluster is connected to a shared SPM, consisting of 16 independent memory banks. By carefully selecting the allocation of data and access to the memory banks, access can be time-predictable [2].

We have implemented a shared SPM in the T-CREST processor [40]. We use TDM based arbitration, which results with a single cycle SPM in a maximum access time of n clock cycles for n cores. We found that a shared SPM scales up to nine cores when implemented in an FPGA. We use our shared SPM in the evaluation.

3.4 Scratchpad Memory with Ownership

Access latency to a shared SPM is a few clock cycles, way less than access to main memory. However, often, the SPM is not used by all cores, and the TDM arbitration wastes memory bandwidth. For example, in a producer/consumer setting, only a single core writes into the SPM and when finished a different core reads from the SPM. For this setup we introduce the notion of ownership [40]. A core *owns* an SPM for some time, uses it to compute write data into it, and then transfers the ownership to a core that consumes the data. When tasks agree on the ownership of the SPM, there is no need for arbitration. The core has exclusive access to the owned SPM with short (single cycle) access time. This mechanism allows fast transfer of bulk data.

For double-buffered communication and several communication channels, we introduce a pool of SPMs with ownership. Different cores can acquire SPMs out of this pool and after usage, either transfer the ownership to another core or put it back into the pool of free SPMs. This pool of SPMs scales up, similar to a shared SPM, to about nine cores in an FPGA. Beyond that number of cores, the SPM pools need to be clustered.

3.5 Distributed Shared On-Chip Memory

Combining core-local SPMs with a NoC leads to a distributed shared on-chip memory. Each core is attached to local memory and to a NoC that supports access to a local memory of a remote core. A standard solution for remote read and writes is to use two NoCs: one to support writes and read requests and a second to deliver the response for the reads.

Epiphany is a high-performance energy-efficient manycore processor [27] that uses distributed on-chip memory. Epiphany is intended as an accelerator processor for real-time embedded systems. Two versions, a 16-core chip, and a 64-core chip have been taped out. The multicore processor Epiphany uses a distributed memory architecture. Each core contains 32 KB of local memory that is mapped into a global address space. The processors contain no caches. Access to the memory of a remote core is performed over a NoC. The NoC is organized as a mesh and favors writes over reads, as writes are posted writes where the processor does not need to wait for the write to finish. Packets are single word long, and routing is performed in a single cycle per hop. A second NoC is dedicated for read responses and a third NoC supports off-chip traffic, e.g., with

a master processor and external shared memory. There is no documentation available on how the arbitration in the NoC routers is performed on a conflict. We explored the processor and measured considerable latency variations depending on the NoC load. Therefore, we cannot (yet) recommend it for applications with tight timing constraints.

We have implemented a distributed shared memory in the T-CREST multicore [29]. We use two instances of the S4NOC [36]: one is used to write to a remote SPM or transmit a read request, and the second is used to return the read result. The SPMs are mapped into different address ranges in the global address range, and the read or write address determines which SPM to access. As several remote read requests may arrive at one core in successive clock cycles, the read results (one per clock cycle) may queue up waiting for their slot to be sent on the read response NoC. In the worst case, this could be $n-1$ words for an n core system. To minimize the length of this queue, the TDM schedule for the return NoC is optimized and aligned to the read request schedule. As S4NOC uses TDM arbitration and a static schedule, we can provide guarantees on latency bounds for reads and writes. As reads need to travel a NoC twice, their latency is double the latency of writes.

Operating system support to virtualize SPMs on a distributed shared on-chip memory is presented in the ShaVe-ICE project [41]. Similar to Epiphany and our solution, each core contains a local SPM and is connected via a NoC. The operating system support is to manage the changing demand of threads for local memory by allocating and deallocating memory on the local or a remote SPM. When allocating on a remote core, the hop distance is taken into account for the allocation policy.

3.6 Direct Links and Memory Between Cores

Another way to structure communication between cores is to have direct links between neighboring cores, organized in a mesh or folded torus. The main benefit of such an organization is that it is a local link and fully supports two types of parallel applications: (1) applications organized in a computing pipeline or (2) physical simulations, such as finite element simulation where access to the neighbor elements is needed.

The link can be as simple as a FIFO queue or more sophisticated, like a dual-port memory between cores. Isaac Liu uses dual-port memories for a multicore organization of a precision timed machine in the evaluation of his Ph.D. thesis [20]. He implemented a real-time computational fluid dynamics simulator on a multicore PRET [21]. The cores use so-called *privately shared* SPMs between cores to provide point-to-point communication channels.

Although less flexible than a fully blown NoC, direct links may be implemented very efficiently and being, therefore, a practical solution. This form of communication has not yet received much attention when discussing multicore communication.

3.7 One-Way Shared Memory

A quite exotic form of on-chip communication is the so-called *one-way shared memory* [33]. The one-way memory uses the TDM scheduled S4NOC for communication but uses a very simple NI. Each core contains a core local memory connected to the NI. The main idea is that the NoC continuously copies data blocks between the core-local

memories. There is one communication channel between each pair of cores. The NoC reads from the senders' core-local memory and writes into the receiver's core-local memory. As this update is performed in one direction only, we call this architecture a *one-way memory*.

The routers have a fixed, pre-programmed schedule. For symmetric structures, such as the torus, all routers execute the same schedule [3]. One such schedule is one TDM round in which one word is transferred between each core. To transfer a memory block of n words, we need n TDM rounds.

The simplicity of the one-way memory paradigm results in very low resource usage. The resource consumption of the NoC and the NI, which implements the one-way memory, is lower than other NoC solutions. This simplicity, i.e., low logical element usage, can be translated either into lower power consumption or higher NoC bandwidth. Higher NoC bandwidth is achieved simply by duplicating the local core memory or using wider NoC router links.

3.8 Additional Hardware Support for Message Passing

The previous sections presented on-chip communication architectures that can be used for message passing. However, we can provide additional hardware to optimize the performance of message passing further.

To reduce the overhead of message passing, a tight integration of message passing instructions into the processor pipeline has been proposed [26]. A RISC-V processor has been extended with a send, receive, and source instructions to allow fast message passing of short messages over a NoC. Additionally, to optimize the checking for ready to send and receive messages available, four branch instructions have been added.

The NI of the Argo NoC [42] includes a local memory and a DMA machinery to transfer data from the local memory to the TDM based NoC. The DMA contains a table with entries of memory regions that shall be sent to different cores. Each virtual channel may have its entry in the table. A message is created in the local memory by the processor, an entry into the DMA table is programmed, and the DAM started. The message transfer happens in parallel to program execution on the processor core.

CSP uses messages not only for data transfer but also as synchronization points between tasks (called processes in CSP). The CSP rendezvous can be implemented by exchanging two messages. We extended a ring-based NoC on a multicore Java processor with explicit support for this synchronization [10]. As an optimization, the NoC supports a dedicated *Ack* command for the rendezvous.

4 Evaluation

We have built several of the proposed hardware solutions in the context of the T-CREST [34] multicore processor Patmos [38]. The hardware is described in Chisel [1] and available in open source at `https://github.com/t-crest/patmos`. As Patmos uses the open-core protocol to interface to IO devices and memory, all those multicore devices are implemented with this interface.

To enable wider adaption of our multicore hardware, we are currently in the process of extracting those devices into its own GitHub repository `https://github.com/schoeberl/soc-comm`. There we will use a simple interface definition and will provide bridges for the open-core protocol, Wishbone, and AXI.

4.1 Experimental Setup

We compare different solutions by evaluating them in an FPGA. The default configuration for T-CREST supports the Altera DE2-115 development board. The FPGA on this board, the Intel/Altera Cyclone IV EP4CE115 FPGA, is big enough to build a system with up to 9 cores. All experiments use the 9-core version of T-CREST. We have chosen the 9-core setup as this is a regular setup for a NoC (3×3 cores), and is the largest setup that fits in the FPGA used.

The Patmos cores are configured with a single-issue pipeline, an 8 KB method cache with 16 methods, a 4 KB write-through data cache, a 2 KB stack cache, a 1 KB instruction SPM and a 2 KB local SPM. External memory is 2 MB with an access time of 21 clock cycles for a burst of 4 32-bit words for a single core. For multicores, the main memory is TDM arbitrated, resulting in access time between 21 and $n \times 21$ clock cycles for n cores.

We use a shared SPM of 16 KB that is TDM arbitrated. The SPM with ownership is configured as a pool of 16 SPMs, each of 1 KB. We measured read access times to the SPM and the ownership SPM. For access to the TDM arbitrated SPM, we observe all possible access times, i.e., for the 9 core version between 3 and 10 clock cycles. We perform the same measurement with the SPM with ownership. As expected, we observe a constant access time of 1 clock cycle.

4.2 Benchmark

For the evaluation, we implement a producer and a consumer who exchange messages. As all presented solutions have no time interference from communication on other channels, it is enough to measure a single virtual channel. We measure throughput in clock cycles, to provide a measurement that is only dependent on the architecture and not on the achievable clock frequency in concrete technology. With a know maximum clock frequency, the maximum bandwidth in bytes per second can be easily computed.

For comparison with a NoC we use the S4NOC [36], configured for 9 cores. The resulting schedule length for the TDM scheduling of the NoC packets is 10 clock cycles for an all-to-all schedule. Therefore, the maximum bandwidth per virtual channel is 10 clock cycles per word. Note that this all-to-all configuration provides $8 \times 9 = 72$ channels, resulting in an overall bandwidth of 7.2 words per clock cycle.

The NIs for the S4NOC consist of FIFO buffers for the sender and receiver. The sender FIFO contains entries for 32-bit data and the send slot number as a representation of the destination address. The receive FIFO includes the read data and the receive slot number as a representation of the sending core. We use small FIFOs built out of registers.

Table 1. Measured throughput, in clock cycles per word for one channel.

Configuration	Message size (32-bit words)	Throughput (clock cycles per word)
Main memory	8	236.4
Main memory	16	212.9
Main memory	32	201.3
Main memory	64	195.7
Shared SPM	8	12.4
Shared SPM	16	11.1
Shared SPM	32	18.9
Shared SPM	64	18.6
SPM with ownership	8	5.7
SPM with ownership	16	4.9
SPM with ownership	32	9.9
SPM with ownership	64	9.5
S4NOC, unconstraint sender	-	10.1
S4NOC, with handshaking	-	12.0

4.3 Measured Throughput

Table 1 shows throughput in clock cycles per word of messages of different sizes on different communication devices. The long access time to shared main memory dominates the low throughput, showing the need for on-chip communication. For all memory-based devices the throughput increases with the message length, as the overhead of sending a message is less dominating. However, we observe an increase in the number of clock cycles between 16-word messages and 32-word messages. We explored the generated code and find that the compiler unrolls loops up to 16 iterations, explaining this anomaly. At 32 or more iterations, the compiler generates code for a standard loop. The throughput of the shared SPM is close to the limit of the access time of one word per 9 clock cycles. For the SPM with ownership, which has a guaranteed access latency of 1 clock cycle, the loop overhead of sending the data dominates.

For the NoC device, we performed two experiments. In the first experiment, we let the producer send as fast as possible without handshaking, assuming that the consumer is fast enough to cover the maximum throughput. As the TDM schedule of the 9 core NoC is 10 clock cycles per TDM round, the 10.1 clock cycles per word are close to the NoC limit. In the second experiment, we used a double buffer of 2 times 4 words (in the NI FIFO) and handshaking so that every 4 words are acknowledged by the receiver. This small buffer and the handshaking ensures that the sender will never overrun the receiver, but introduces an overhead of just 20 % compared to the theoretical maximum throughput.

Table 2. Resource requirement of different communication devices.

Device	LCs	registers	Memory
Shared SPM	654	490	16 KB
SPM with ownership	8694	77	16 KB
S4NOC	5517	4454	0 KB

4.4 Resource Consumption

Table 2 shows the resource requirements of the three different multicore communication devices for 9 cores. The resources are given in logic cells (LC) that contain a 4-bit lookup table, registers (D flip-flops), and on-chip memory. The shared SPM is relatively cheap, as it needs logic only for a simple TDM arbiter for 9 cores. The SPM with ownership contains a pool of 16 SPMs that are multiplexed for 9 cores and therefore need a considerable amount of combinational logic (high LC count). The S4NOC is in the resource requirements between the single SPM and the SPM with ownership but needs no on-chip memory. The relative high register count comes from the small FIFOs built out of registers. We can change the FIFO to use on-chip memories; two per node, one for send and one for receive.

The three solutions scale differently with respect to the maximum clocking frequency. As a baseline, the Patmos processor can be clocked at 80 MHz within this FPGA. The NoC is a distributed design and therefore scales best. The 3×3 S4NOC can be clocked faster than 200 MHz, clearly not being the bottleneck in the system. The single shared SPM has a single merge point and limits the system frequency to about 70 MHz. We assume one pipeline stage, which increases read access latency by one clock cycle, should be enough to increase the maximum clocking frequency to be higher than the 80 MHz of the processor cores. However, moving to a 4×4 organization of the single SPM may reduce the clocking frequency further. The SPM with ownership has the worst clock frequency of just 50 MHz. Adding one pipeline stage should help, but this would double the access latency from 1 to 2 clock cycles.

4.5 Discussion

When we look at the performance, the resource requirement, and the clock frequency, there is no clear winner between the three solutions. The cheapest solution is the shared SPM, but the access time in clock cycles for a producer-consumer workload is higher than at the other two solutions. The SPM with ownership has the highest throughput in clock cycles, but also the highest hardware demand and the lowest clock frequency. This solution should probably be clustered with fewer cores or fewer SPMs in the pool. The NoC solution is probably the sweet spot having medium resource requirement, throughput between the single SPM and the SPM with ownership, and, perhaps most important, scales well with a higher core count.

In summary, a combination of a NoC for the global traffic combined with locally clustered shared SPMs may be the right solution. This combination of a NoC and shared

SPMs is similar to the Kalray architecture, but we propose to have clusters that use a shared SPM overlap for a more flexible continuum for communication.

5 Conclusion

Multicore processors used in cyber-physical systems need to support time-predictable computation and communication. As communication via shared main memory supported by a cache coherence protocol is hardly time-predictable, we need other forms of core-to-core communication. In this paper, we explored different forms of hardware support for on-chip message passing between cores. Shared on-chip memories with a time-predictable arbitration, such as time-division multiplexing, provide an efficient solution for around a dozen cores. For more cores, a distributed communication architecture, such as a network-on-chip. is a better scaling solution. Also, hybrid solutions using shared memories in clusters, which are connected by a network-on-chip, are an option. The usage of multicore processors in safety-critical cyber-physical systems is not yet common. Future applications and experiments will tell which on-chip communication solution will be the most preferred one.

Acknowledgment

The work presented in this paper was partially funded by the Danish Council for Independent Research | Technology and Production Sciences under the project PREDICT[3] (no. 4184-00127A).

References

1. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avizienis, R., Wawrzynek, J., Asanovic, K.: Chisel: constructing hardware in a scala embedded language. In: The 49th Annual Design Automation Conference (DAC 2012). pp. 1216–1225. ACM, San Francisco, CA, USA (June 2012)
2. Becker, M., Dasari, D., Nicolic, B., Akesson, B., Nelis, V., Nolte, T.: Contention-free execution of automotive applications on a clustered many-core platform. In: 28th Euromicro Conference on Real-Time Systems (ECRTS). pp. 14–24 (July 2016). https://doi.org/10.1109/ECRTS.2016.14
3. Brandner, F., Schoeberl, M.: Static routing in symmetric real-time network-on-chips. In: Proceedings of the 20th International Conference on Real-Time and Network Systems (RTNS 2012). pp. 61–70. Pont a Mousson, France (November 2012). https://doi.org/10.1145/2392987.2392995
4. Cruz, R.L.: A calculus for network delay. I. Network elements in isolation. IEEE Transactions on Information Theory **37**(1), 114–131 (Jan 1991). https://doi.org/10.1109/18.61110
5. Cruz, R.L.: A calculus for network delay. II. Network analysis. IEEE Transactions on Information Theory **37**(1), 132–141 (Jan 1991). https://doi.org/10.1109/18.61110
6. Dally, W.J., Towles, B.: Route packets, not wires: On-chip interconnection networks. In: DAC. pp. 684–689. ACM (2001)

[3] `http://predict.compute.dtu.dk/`

7. Dupont de Dinechin, B., van Amstel, D., Poulhiès, M., Lager, G.: Time-critical computing on a single-chip massively parallel processor. In: Conference on Design, Automation and Test in Europe. pp. 97:1–97:6. DATE '14, European Design and Automation Association, 3001 Leuven, Belgium, Belgium (2014)
8. Dupont de Dinechin, B., Durand, Y., van Amstel, D., Ghiti, A.: Guaranteed services of the NoC of a manycore processor. In: International Workshop on Network on Chip Architectures (NoCArc). pp. 11–16. ACM, New York, NY, USA (Dec 2014). https://doi.org/10.1145/2685342.2685344
9. Goossens, K., Hansson, A.: The AEthereal network on chip after ten years: Goals, evolution, lessons, and future. In: Proceedings of the 47th ACM/IEEE Design Automation Conference (DAC 2010). pp. 306 –311 (2010)
10. Gruian, F., Schoeberl, M.: Hardware support for CSP on a Java chip-multiprocessor. Microprocessors and Microsystems **37**(4–5), 472–481 (2013). https://doi.org/10.1016/j.micpro.2012.08.004
11. Hansson, A., Subburaman, M., Goossens, K.: aelite: a flit-synchronous network on chip with composable and predictable services. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2009). pp. 250–255. Leuven, Belgium (2009)
12. Heckmann, R., Ferdinand, C.: Worst-case execution time prediction by static program analysis. Tech. rep., AbsInt Angewandte Informatik GmbH, [Online, last accessed November 2013]
13. Hennessy, J., Patterson, D.: Computer Architecture: A Quantitative Approach, 4th ed. Morgan Kaufmann Publishers (2006)
14. Hoare, C.A.R.: Monitors: An operating system structuring concept. Commun. ACM **17**(10), 549–557 (Oct 1974). https://doi.org/10.1145/355620.361161
15. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978). https://doi.org/10.1145/359576.359585
16. Homewood, M., May, D., Shepherd, D., Shepherd, R.: The ims t800 transputer. IEEE Micro **7**(5), 10–26 (1987). https://doi.org/10.1109/MM.1987.305012
17. Kasapaki, E., Schoeberl, M., Sørensen, R.B., Müller, C.T., Goossens, K., Sparsø, J.: Argo: A real-time network-on-chip architecture with an efficient GALS implementation. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **24**, 479–492 (2016). https://doi.org/10.1109/TVLSI.2015.2405614
18. Le Boudec, J.Y.: Application of network calculus to guaranteed service networks. IEEE Transactions on Information Theory **44**(3), 1087–1096 (May 1998). https://doi.org/10.1109/18.669170
19. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proceedings of the IEEE **75**(9), 1235–1245 (Sept 1987). https://doi.org/10.1109/PROC.1987.13876
20. Liu, I.: Precision Timed Machines. Ph.D. thesis, EECS Department, University of California, Berkeley (May 2012)
21. Liu, I., Reineke, J., Broman, D., Zimmer, M., Lee, E.A.: A PRET microarchitecture implementation with repeatable timing and competitive performance. In: Proceedings of IEEE International Conference on Computer Design (ICCD 2012) (October 2012)
22. Lohstroh, M., Schoeberl, M., Goens, A., Wasicek, A., Gill, C., Sirjani, M., Lee, E.A.: Actors revisited for time-critical systems. In: Proceedings of the 56th Annual Design Automation Conference 2019. pp. 152:1–152:4. DAC '19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3316781.3323469
23. May, D., Shepherd, R.: Occam and the transputer. In: Proc. of the IFIP WG 10.3 workshop on Concurrent languages in distributed systems: hardware supported implementation. pp. 19–33. Elsevier North-Holland, Inc., New York, NY, USA (1985)
24. Metzlaff, S., Mische, J., Ungerer, T.: A real-time capable many-core model. In: Proceedings of 32nd IEEE Real-Time Systems Symposium: Work-in-Progress Session (2011)

25. Mische, J., Frieb, M., Stegmeier, A., Ungerer, T.: Reduced complexity many-core: Timing predictability due to message-passing. In: Architecture of Computing Systems - ARCS 2017: 30th International Conference, Vienna, Austria, April 3–6, 2017, Proceedings. pp. 139–151. Springer International Publishing, Cham (2017)
26. Mische, J., Frieb, M., Stegmeier, A., Ungerer, T.: Pimp my many-core: Pipeline-integrated message passing. In: Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS 2019) (2019)
27. Olofsson, A., Nordström, T., ul Abdin, Z.: Kickstarting high-performance energy-efficient manycore architectures with Epiphany. In: Matthews, M.B. (ed.) in Proc. Asilomar Conference on Signals, Systems and Computers. pp. 1719–1726. IEEE (2014)
28. Paukovits, C., Kopetz, H.: Concepts of switching in the time-triggered network-on-chip. In: Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2008). pp. 120 –129 (August 2008). https://doi.org/10.1109/RTCSA.2008.18
29. Petersen, M.B., Riber, A.V., Andersen, S.T., Schoeberl, M.: Time-predictable distributed shared on-chip memory. Microprocessors and Microsystems (2019). https://doi.org/10.1016/j.micpro.2019.102896
30. Ptolemaeus, C. (ed.): System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014)
31. Puffitsch, W., Sørensen, R.B., Schoeberl, M.: Time-division multiplexing vs network calculus: A comparison. In: Proceedings of the 23th International Conference on Real-Time and Network Systems (RTNS 2015). Lille, France (November 2015). https://doi.org/10.1145/2834848.2834868
32. Schoeberl, M.: Time-predictable computer architecture. EURASIP Journal on Embedded Systems **vol. 2009, Article ID 758480**, 17 pages (2009). https://doi.org/10.1155/2009/758480
33. Schoeberl, M.: One-way shared memory. In: 2018 Design, Automation and Test in Europe Conference Exhibition (DATE). pp. 269–272 (March 2018). https://doi.org/10.23919/DATE.2018.8342017
34. Schoeberl, M., Abbaspour, S., Akesson, B., Audsley, N., Capasso, R., Garside, J., Goossens, K., Goossens, S., Hansen, S., Heckmann, R., Hepp, S., Huber, B., Jordan, A., Kasapaki, E., Knoop, J., Li, Y., Prokesch, D., Puffitsch, W., Puschner, P., Rocha, A., Silva, C., Sparsø, J., Tocchi, A.: T-CREST: Time-predictable multi-core architecture for embedded systems. Journal of Systems Architecture **61**(9), 449–471 (2015). https://doi.org/10.1016/j.sysarc.2015.04.002
35. Schoeberl, M., Chong, D.V., Puffitsch, W., Sparsø, J.: A time-predictable memory network-on-chip. In: Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014). pp. 53–62. Madrid, Spain (July 2014). https://doi.org/10.4230/OASIcs.WCET.2014.53
36. Schoeberl, M., Pezzarossa, L., Sparsø, J.: A minimal network interface for a simple network-on-chip. In: Architecture of Computing Systems - ARCS 2019. pp. 295–307. Springer (1 2019). https://doi.org/10.1007/978-3-030-18656-2_22
37. Schoeberl, M., Pezzarossa, L., Sparsø, J.: S4noc: a minimalistic network-on-chip for real-time multicores. In: 12th International Workshop on Network on Chip Architectures (NoCArc '19). ACM (October 2019). https://doi.org/10.1145/3356045.3360714
38. Schoeberl, M., Puffitsch, W., Hepp, S., Huber, B., Prokesch, D.: Patmos: A time-predictable microprocessor. Real-Time Systems **54(2)**, 389–423 (Apr 2018). https://doi.org/10.1007/s11241-018-9300-4
39. Schoeberl, M., Sørensen, R.B., Sparsø, J.: Models of communication for multicore processors. In: Proceedings of the 11th Workshop on Software Technologies for Embedded and

Ubiquitous Systems (SEUS 2015). pp. 44–51. IEEE, Auckland, New Zealand (April 2015). https://doi.org/10.1109/ISORCW.2015.57
40. Schoeberl, M., Strøm, T.B., Baris, O., Sparsø, J.: Scratchpad memories with ownership. In: 2019 Design, Automation and Test in Europe Conference Exhibition (DATE) (2019)
41. Shoushtari, M., Donyanavard, B., Bathen, L.A.D., Dutt, N.: Shave-ice: Sharing distributed virtualized spms in many-core embedded systems. ACM Trans. Embed. Comput. Syst. **17**(2), 47:1–47:25 (Feb 2018). https://doi.org/10.1145/3157667
42. Sparsø, J., Kasapaki, E., Schoeberl, M.: An area-efficient network interface for a TDM-based network-on-chip. In: Proceedings of the Conference on Design, Automation and Test in Europe. pp. 1044–1047. DATE '13, EDA Consortium, San Jose, CA, USA (2013)
43. Whitby-Strevens, C.: The transputer. SIGARCH Comput. Archit. News **13**(3), 292–300 (1985). https://doi.org/10.1145/327070.327269