S4NOC: a Minimalistic Network-on-Chip for Real-Time Multicores

Martin Schoeberl
masca@dtu.dk
Technical University of Denmark
Lyngby, Denmark

Luca Pezzarossa
lpez@dtu.dk
Technical University of Denmark
Lyngby, Denmark

Jens Sparsø
jspa@dtu.dk
Technical University of Denmark
Lyngby, Denmark

ABSTRACT

Message passing using a network-on-chip (NoC) is an efficient way to provide core-to-core communication on a multicore processor. However, many NoCs use routers and network interfaces that are optimized for the average case. Therefore, it is hard to bound the worst-case latency of a message or the bandwidth. Furthermore, often large buffers are used in the routers and network interfaces, which require a considerable amount of area.

This paper presents a statically scheduled NoC that uses time-division multiplexing at the links, the routers, and the network interfaces. Static scheduled traffic allows computing upper bounds for end-to-end latencies of messages, which is a requirement for building multicore real-time systems. Furthermore, this static scheduled NoC needs no additional buffers, except pipeline registers, and the resulting resource requirement is low.

KEYWORDS

real-time systems, network-on-chip, time-predictable computer architecture

ACM Reference Format:
Martin Schoeberl, Luca Pezzarossa, and Jens Sparsø. 2019. S4NOC: a Minimalistic Network-on-Chip for Real-Time Multicores. In *12th International Workshop on Network on Chip Architectures (NoCArc '19), October 13, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3356045.3360714

1 INTRODUCTION

When multicore systems are used for real-time systems to increase performance, we need a time-predictable way for tasks executing on different cores to communicate. Using shared data in main memory, backed up by several levels of caches, and cache coherence protocols between core-local caches is hardly a time-predictable solution. A better solution for real-time systems is to use message passing. For message passing between cores a network-on-chip (NoC) can provide the technology to move bits between cores, without leaving the chip. However, most NoC designs are optimized for average case performance and are hardly a fit for real-time systems, where the worst case timing is of primary importance.

This paper presents a NoC, called S4NOC, that is optimized for real-time systems. It provides static upper bounds for the time a message needs to be transferred from one core to another core. To enable time-predictable on-chip communication, S4NOC uses time-division multiplexing (TDM) arbitration of all resources: the links, the routers, and the network interface (NI). TDM arbitration has two main advantages: (1) The fixed schedule allows to compute upper bounds for message latency and lower bounds for the bandwidth. (2) The fixed arbitration avoids flow control and additional buffers, resulting in low resource requirements.

The TDM schedule is precomputed and fixed to provide virtual channels between all processor cores. A single slot in the TDM schedule represents each channel, e.g., in a 3×3 NoC, each core has 8 channels to the other 8 cores. Therefore, the TDM schedule, at each core, contains 8 time slots when the NI can inject data into the NoC. A fixed schedule is cheap to implement, as the schedule table can be implemented in read-only memory.

The S4NOC supports packets of single word granularity. The routers contain the routing information. Therefore, the only additional information routed with the data is a single valid bit. The NI contains send and receive buffers, where each buffer element consists of the data word and the destination and sender address. The destination address is represented by the slot number when the NI shall inject the data word into the NoC. The sender address is represented by the slot number when the data word arrived at the receivers NI.

We have argued to use TDM arbitration to build a time-predictable NoC [12]. Our S4NOC design provides a complete solution of a NoC with TDM arbitration, including a network interface design and an implementation in a multicore processor in an FPGA.

The contributions of this paper are: (1) a NoC router and a network interface design that are optimized for usage in real-time systems; (2) an implementation that has a low resource requirements; and (3) an evaluation in a multicore processor.

This paper is organized in 5 sections: The following section presents related work. Section 3 presents the architecture of our S4NOC design. Section 4 evaluates the design with respect to hardware resources and possible bandwidth and latency. Section 5 concludes.

2 RELATED WORK

The Æthereal [4] NoC uses TDM where slots are reserved to allow a block of data to pass through the NoC router without waiting or blocking traffic. We conform to the TDM approach of Æthereal, but present a simpler NoC in this paper.

The PaterNoster NoC [8] is a relatively simple NoC. PaterNoster avoids flow control and complexity in the routers by restricting a packet to single standalone flits. The NI of PaterNoster is as well a simple design to support single word packets. The NI is connected to the memory stage of a RISC-V processor [7]. Our NoC uses a similar architecture and employs just single word packets. However,

NoCArc '19, October 13, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6949-7/19/10…$15.00
https://doi.org/10.1145/3356045.3360714

we use statically scheduled TDM arbitration to bound the maximum latency for packets and avoid any buffering in the routers. Our NI is mapped into an address and can be accessed by load and store instructions. Furthermore, by avoiding a full lookup in the receive buffer, the S4NOC NI is more than a factor of 10 smaller than the PaterNoster NI.

The OpenSoC Fabric [3] is an open-source NoC generator written in Chisel. It is intended to provide a system-on-chip for large-scale design exploration. The NoC itself is a state-of-the-art design with wormhole routing, credits for flow control, and virtual channels. Currently, the interface to the NoC is a ready/valid interface receiving either packets or flits. An extension with a NI and the use of a standard AXI4 interface is planned. The open-source implementation of OpenSoC allows us to compare our TDM based design against a state-of-the-art router design to show how cheap a TDM router (and NI) can be. A single OpenSoC router (in the default configuration) is as large as our complete 3×3 NoC including the NIs and open core protocol (OCP) [1] interfaces.

The Argo NoC [6] is another NoC that uses TDM based arbitration of resources. Compared to Æthereal, Argo also uses the same TDM schedule in the NI [15] 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 global asynchronous, local synchronous systems with an asynchronous router design and mesochronous (same clock source, but variable upwards bounded skew allowed) NIs.

The Hoplite architecture [5] uses routers without buffers, a unidirectional torus, and single flit packages that include the destination address. On an arbitration conflict, Hoplite uses deflection as a resolution mechanism. This design results in very small hardware usage, but it cannot provide real-time guarantees. HopliteRT [16] is an extension to Hoplite to provide real-time guarantees. Hoplite is modified to prioritize deflections and perform traffic shaping at the network interface to provide guarantees on end-to-end latencies for packets. All versions of Hoplite do not include any NI and use generated traffic patterns to evaluate the design. In contrast, our design contains NIs connected to real processor cores. We evaluate our design with programs executing on the cores.

The one-way shared memory [10] project uses the S4NOC to implement a special form of distributed shared memory. Each core contains a local on-chip memory that is connected to the NoC. Blocks within those local memories are constantly copied to other cores. As this transfer is only in one direction (similar to push messages), it is called one-way shared memory. The one-way shared memory is also a design with low resource requirement, but the programming interface is very different from our NI.

The time-predictable distributed shared memory [9] uses two instances of the S4NOC: (1) for write traffic to remote scratchpad memories and read requests and (2) to return the read data. The S4NOC has also been used to explore minimal network interfaces [13].

3 THE S4NOC ARCHITECTURE

The S4NOC is a statically scheduled, TDM arbitrated NoC intended for real-time systems. The S4NOC routes single words of 32 bits.[1]

[1] 32-bit is the default configuration, we explore wider links in the evaluation section.

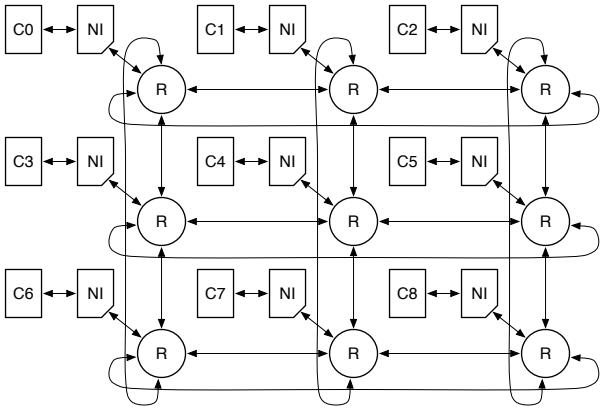

Figure 1: A 3 x 3 multicore connected by a bi-torus NoC.

Therefore, a packet, a flit, or a word all reference to the same amount of data. Besides the data, each packet contains a single valid bit.

As all traffic is statically scheduled, there are no conflicts on any shared resource, such as links or crossbars. The schedule is computed offline and stored in the routers. Without conflicts, there is no need to provide buffering in the routers, flow control between routers, or credit-based flow control between the NIs.

Figure 1 shows a typical configuration of our NoC, 9 cores are organized in a 3×3 bidirectional torus network topology. The bidirectional torus topology minimizes the number of hops for a packet to travel. The figure shows a normal torus for simplicity, while the practical layout will be a folded torus to avoid the long wraparound wire. Furthermore, the bidirectional torus topology enables the use of symmetric schedules, as explained later.

3.1 The Schedule

All resources (links, crossbars) of the NoC are shared according to a static TDM scheduling computed off-line [2]. Such schedule is based on a fixed period TDM round which is repeated over and over. The slot number (at the injection point) defines the destination address and the slot number at the receiving end identifies the sender address. These two numbers define a virtual circuit. The used schedule provides one virtual circuit from each core to every other core (all-to-all), as this is the most general case. One might consider this a waste of resource, but as shown later, the resource requirement for a NoC with such a static schedule is very low and the hardware simplicity allows to "waste" bandwidth at a very low cost.

An all-to-all core communication graph for N cores has $N \times N - 1$ virtual circuits, and for a 3×3 multicore, it corresponds to 72 virtual circuits. The scheduler can satisfy these requirements with a 10 slot TDM schedule. This schedule is only 2 slots more than the minimum theoretical period needed by the 8 outgoing and 8 incoming virtual circuits.

3.2 The Router

Figure 2 shows the S4NOC router. The router is characterized by the five input ports and five output ports (local, north, east, south, and west). A crossbar (X Bar) connects each of the input ports to one of the output ports. A register is placed at the output port and,

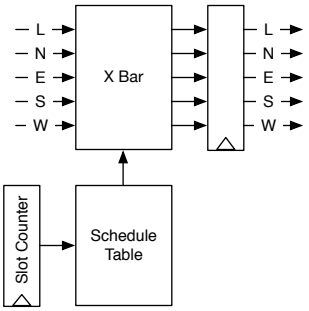

Figure 2: The S4NOC router with TDM driven static schedule.

except for this single stage pipeline register, no additional buffering is offered at the router level since the carefully scheduled traffic will not allow two or more inputs to compete for the same output port.

The crossbar provides all possible connections from input ports to output ports. Therefore, data coming from different input ports must target different output ports to be routed in the same cycle. The crossbar in an FPGA or standard cell ASIC implementation consists of a collection of 5 multiplexers (one for each output port) with 4 inputs. The inputs are the input ports except the one in which the output is routed, since routing a word back to the port where it came from is not allowed.

The connections in the crossbar are controlled by the content of the schedule table. The schedule table contains the scheduling information and is indexed by the TDM round slot counter, which runs synchronously in all the routers. At the end of one TDM round, the counter starts at the begin of the next round.

The router design is flexible and if more bandwidth is needed, the link width can be increased. If the delay of the wire and through the crossbar becomes a bottleneck, an additional pipeline register for the link, in front of the crossbar can be inserted. Furthermore, the schedule table can be shared between neighbor routers since the schedule is the same for all the routers, which can further reduce the hardware cost.

3.3 The Network Interface

Senders can identify a virtual circuit by the slot number in which its data is transmitted. Similarly, receivers can identify a virtual circuit by the slot number in which its data is received. The architecture of the NI is based on these observations since receive and send slot numbers are used to identify virtual circuits.

Figure 3 shows the architecture of the proposed NI. It consists of a processor interface block, a transmit (TX) FIFO, a receive (RX) FIFO, and a TDM counter. The TDM counter runs synchronously in all the NIs and all the routers. Its value identifies the current slot and is used to enable packets to be sent into the NoC according to schedule and to identify the sender based on the receiving slot.

3.3.1 Processor Interface. The NI is interfaced to the processor as a memory mapped peripheral device. The interface that we have implemented in this work is OCP. However, different interfaces, such as AXI, can be used.

The processor uses this interface to interact with the TX and RX FIFO. The processor can write into the TX FIFO and read from the

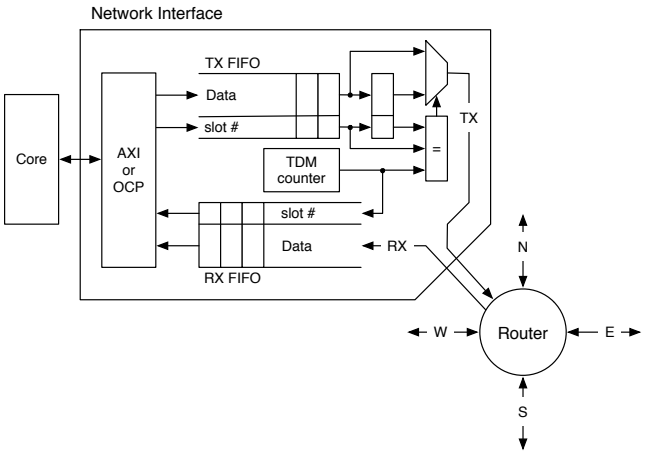

Figure 3: The network interface connects the processing core to the router.

RX FIFO. Also, it can also read status flags for the FIFOs status. The TX FIFO contains a flag showing if the buffer is empty, the RX FIFO contains a flag if there is some data available.

Each data word written to the TX FIFO needs also a slot number written to the FIFO to identify the receiver. As a small optimization we use part of the address bits to communicate the send slot, reducing the number of stores from two to one per data word.

3.3.2 TX FIFO. The TX FIFO stores data to be sent and the slot number to identify the virtual circuit and thus, the destination. When the TDM counter in the NI matches the slot number of an entry in the TX queue, the data is sent into the network.

As the TX FIFO in the sender NI is shared among all the outgoing virtual circuits, a simple FIFO implementation where only the head of the queue is compared to the TDM counter can produce head-of-queue blocking. A head-of-queue blockong happens when the destination of the data injected in the TX FIFO by the processor is not ordered according to the TDM schedule.

To prevent this, we included look-ahead logic that examines the first N entries at the head of the queue and allows to immediately send the content of one of these if the send slot matches the TDM counter. The parameter N is configurable, allowing a proper trade-off between performance degradation (due to head-of-queue blocking) and hardware cost.

Another approach is to insert the packets in software in the order according to the schedule. In this case, the worst-case waiting time for starting to send the data in the TX FIFO queue is one TDM round. If the destination of each data is ordered, when the head of the queue is sent, the rest of the data in the RX FIFO is also sent uninterruptedly.

In terms of implementation, we optimize the write into the NI by using the lower bits of the address to determine the send slot number. In this way, sending a single word would costs one load instruction to poll of the TX FIFO empty flag, and one store instruction to write the data and slot in the TX FIFO.

3.3.3 RX FIFO. The RX FIFO holds the packets that are received from the network. When a packet is received, the payload data is

written into the RX FIFO along with the slot number when it was received. The latter is used by the software to identify the virtual circuit and, thus, the sender. The NI provides the slot number when the packet arrived. Further dispatching of packets from different cores to different tasks running on the core is managed by the software.

In term of access time for our software implementation, one load instruction is needed to read the data available flag, and two further load instructions are then needed to read the data and the slot number. In the case where the sender is known, the load instruction for the slot table can be avoided, resulting in two load instructions to read the received data.

4 EVALUATION

For the evaluation of our NoC we use the open-source T-CREST multicore [11]. The T-CREST platform consists of a configurable number of Patmos [14] cores, connected to a shared main memory with a TDM based arbiter. The default configuration for T-CREST targets the slightly dated terasIC DE2-115 FPGA board. The board contains an Intel/Altera Cyclone IV FPGA (model EP4CE115). For synthesize we use the Quartus Prime 16.1 Lite Edition. All performance experiments are run in the FPGA.

4.1 Hardware Cost

Our aim was to design a lean NoC that even fits into a medium sized low-cost FPGA. We first evaluation the size of the individual components and then of complete NoC structures. Hardware requirement is reported in FPGA resources: in logic cell (LC) which contains a 4-bit lookup table, register bit, and on-chip memory. Our design does not use any embedded digital signal processing blocks. Maximum clock frequency is reported for the slow timing model at 1.2 V and 85 C.

4.1.1 The Router. The S4NOC router consists of a collection of multiplexers to build the crossbar, registers on the output ports, a schedule table, and a TDM slot counter. The default configuration is a link width of 32 bits plus a valid bit. To increase the bandwidth of our S4NOC, the link width is configurable.

For the evaluation of the router resource requirement we generate a 3×3 NoC configuration and use traffic generators to drive the local ports of the routers. For synthesize we use the default settings; we spent no extra effort to achieve a maximum clock frequency.

For any processor core, which is not a source (sensor) or a sink (actuator) only, we need at least one input channel and one output channel to produce useful work. In a 3×3 NoC the TDM schedule length is 10 clock cycles. Therefore, each 10 clock cycles one word enters the core and one word exits the core. In the ideal case of using just a single load and store instruction there are only 8 clock cycles left to produce some useful work with the data. Therefore, we consider this design being computation bound and not communication bound.

However, if we really need higher bandwidth, e.g., for bursty data, we can increase the link width. Table 1 shows the resource requirement and maximum clock frequency of our S4NOC routers with different link width.

The resource requirement grows less than linear with the number of bits, as the schedule table and schedule counter remain the same. The decrease of the maximum frequency results from the shared select signal of those many multiplexers. This can be improved by introducing pipelining in the schedule table, if needed. However, up to a 128-bit link width, the router is faster than the Patmos processor. Therefore, we did not introduce this pipelining.

Table 1: Resource requirement of S4NOC routers compared with other designs.

Router	LC	Register	fmax (MHz)
32-bit S4NOC	253	173	236
64-bit S4NOC	476	333	157
128-bit S4NOC	990	653	94
256-bit S4NOC	1951	1293	52
Argo	932	565	-
PaterNoster	1899	1297	-
OpenSoC	3752	1551	-
Patmos core	9437	4384	79

When we compare the S4NOC router with the Argo router, we see the cost of three pipeline stages and the combinational logic for the header processing in the Argo router. When considering the LCs, a 128-bit S4NOC router is as expensive as an Argo router. The register usage of the Argo router is between the 64-bit and the 128-bit S4NOC router, which is expected as the Argo router consists of three 32-bit pipeline stages.

Similarly to S4NOC and Argo, the PaterNoster NoC is available in open-source, which allows us to synthesize it for the same FPGA. We synthesized the PaterNoster node in a 2×2 configuration to fit into the same FPGA for the comparison. We can observe that the S4NOC router is about 7.5 times smaller than the PaterNoster router. The additional resource requirement in the PaterNoster router comes from the corner buffer in each router, which holds packets that want to switch from the x-direction ring to the y-direction ring.

The table also presents the resource requirement for a single router of the OpenSoC NoC [3]. For this result, we generated the Verilog code for the default configuration, which is a 2×2 mesh with routing based on virtual channels and one local port.

Comparing against the PaterNoster router and the OpenSoC router, which represents a standard router design, we see how resource efficient our TDM based NoC router is. We can *waste clock cycles* for a static TDM schedule at a very low cost. We conclude that a TDM based NoC is an efficient solution when the bandwidth per resource requirement is considered.

When we consider a link width of 128 bits, a 32-bit processor needs 4 load instructions for the input channel and 4 store instructions for the output channel. At a 10 clock cycle schedule length, this can only be sustained in a heavily unrolled loop.

To set all those router numbers in relation we report the resource requirement of the Patmos processor pipeline, which is part of the T-CREST platform. All routers are smaller than this processor core. A 32-bit S4NOC router requires about 2.7 % of the processor pipeline, which we consider a reasonable cost for a communication infrastructure.

4.1.2 The Network Interface. Table 2 shows the resource requirement of different configurations of the S4NOC NI and two other NIs.

Table 2: Resource requirement of different S4NOC NI configurations.

Component	LC	Register	Memory
S4NOC 1 buffer	121	80	0
S4NOC 2 buffers	195	152	0
S4NOC 4 buffers	342	296	0
S4NOC 8 buffers	638	584	0
Argo	849	361	1.3 KB
PaterNoster	6131	2249	0

Table 3: Resource requirement of a complete S4NOC configurations.

NoC	LC	Register	Memory
3×3 S4NOC with 1 buffer	3455	2438	0
3×3 S4NOC with 2 buffers	4097	3086	0
3×3 S4NOC with 4 buffers	5423	4382	0
3×3 S4NOC with 8 buffers	8084	6974	0
3×3 Argo NoC	15177	8342	12.1 KB
Single Patmos core	9437	4384	22 KB

The numbers have been collected when synthesizing a 3×3 NoC with the traffic generator. For the S4NOC NI we vary the size of the TX and RX FIFOs.

The memory requirement only refers to the memory used in the NoC (e.g., for schedule tables, etc.). The size for the local memory in the Argo NIs is configurable and therefore not shown in the table.

The S4NOC NI is considerable smaller than the NIs of the Argo or PaterNoster NoCs. However, the Argo and PaterNoster NIs provide additional functionality. The Argo NI contains a DMA for send and receive data movement and manages the schedule tables (Argo uses source routing). The PaterNoster NI uses a content addressable memory to demultiplex different receive channels, so that the processor can read a dedicated channel. A later version of the PaterNoster NI [7] uses a single receive FIFO to avoid the large overhead of the receive channel demultiplexing in hardware.

When comparing an S4NOC node (router plus NI) with the size of a Patmos core, which requires 9437 LUTs and 4384 registers, we can see that we achieved our goal of a small NoC. The resource requirement of one NI and router, configured with 4 buffers, is around 6 % of the Patmos core.

4.1.3 Complete Multicores. Table 3 shows the complete resource requirement of different NoCs for 9 cores. These numbers are derived from building a complete multicore system with 9 Patmos cores and the 3×3 NoC and synthesizing it for the FPGA. This configuration is also used to run the benchmarks in the FPGA hardware.

The maximum clock frequency of the 3×3 configuration is 72 MHz. This critical path is in the processor pipeline and not in any part of the S4NOC router or NI.

The increase in the number of entries for the FIFO increase the size of the NoC, which is expected. However, the base cost for the S4NOC still dominates the different configurations. For larger buffers we shall use a FIFO design that uses on-chip memory instead of registers.

The Argo NoC [6] is available in open source. Therefore, we can obtain the results by synthesizing two configurations of the Argo NoC for the same FPGA. We can observe, that the S4NOC is about half the size of the Argo NoC. However, the Argo NoC provides additional functionality with a TDM shared DMA for the movement of data between a local memory and the local link of the router.

The biggest S4NOC is still smaller than a single Patmos core. The example multicore contains 9 Patmos cores. Therefore, the biggest configuration of the resource requirement of the S4NOC is still less than 10 % of the complete multicore.

Table 4: Maximum measured throughput, in clock cycles per word for one virtual circuit.

Configuration	Throughput (clock cycles per word)
Unconstraint sender	10.1
Single word handshake	23–48
Double buffer (4 words)	12.0

4.2 Performance

We evaluate the performance (bandwidth) of the S4NOC in a 3×3 configuration with the Patmos processors serving as sender and receiver nodes. For a 3×3 configuration the TDM schedule length (the TDM round) is 10 clock cycles. Therefore, we can send and receive one packet from any to any node every 10 clock cycles.

We explore a single virtual channel without any other traffic on the NoC. Due to strict isolation of virtual channels with the static TDM schedule, other traffic would not influence the measurement. No contention is possible, the traffic on this virtual channel is independent of any traffic on any other virtual channel.

For a baseline we send in a tight loop packets without any handshaking. We send 4096 times 16 words, one word at a time. In the best case we should be able to send (and receive) one word per 10 clock cycles. At the receiver side we check if we received all packets. Table 4 shows a throughput of one word every 10.1 clock cycles, which is around the theoretical maximum of one word per 10 clock cycles. The 0.1 clock cycle accounts for the startup cost of the benchmark.

If we cannot guarantee that the sender does not overrun the network bandwidth and/or the receiver cannot consume the packets at the rate as they are produced, we need to introduce handshaking. The simplest form of handshaking is sending one credit back per packet received. This form of stop and go handshake is simple, but inefficient. We setup an experiment where again the sender and receiver exchange packets in a tight loop. The sender sends one packet and waits for the credit package from the receiver. The receiver waits for a packet to receive and immediately sends a credit packet back. This ping/pong results in a lockstep performance of 23 to 48 clock cycles per packet exchanged, depending on the FIFO size (we explored 2 to 16 entries). More entries in a bubble FIFO increase the latency and therefore decrease the bandwidth with the stop and go handshake.

To overlap sending of data packets with sending back credit packets we setup following experiment. We use a double buffer of two times four words on the receiver side. The sender can, without receiving a credit, send four packets. After that the sender waits for a credit to send another four packets. The receiver will send one credit at the start for the second buffer and read out the first four words from the first buffer. The receiver repeats sending a credit and reading four words of data. With this configuration, even when using a return channel for credit handshaking, we achieve a throughput of one word every 12 clock cycles, which is just 20 % slower than the theoretical bandwidth without handshaking.

4.3 Worst-Case Message Latency

With a statically scheduled path for packets, the worst-case message latency can be relatively easily computed. From the write of a packet, it needs to propagate the NI, which in a bubble FIFO will be the number of FIFO entries in clock cycles. Then the packet waits for the transmit slot, which is in the worst case just missing the slot and waiting for a complete TDM round. Then the packet traverses the NoC with one clock cycle per hop. At the receiving side the packed has again to pass through the NI, which might be one clock cycle per FIFO entry, if a bubble FIFO is used. When the NIs use a memory/pointer based FIFO, the NI traverse time is reduced to a constant time of a few clock cycles, depending on the interface.

The worst case latency L_{max} with an NI latency l, a TDM schedule length s, and maximum h hops is:

$$L_{max} = 2 \times l + s + h \tag{1}$$

For our 3×3 example with a 2 buffer FIFO l is 2, s is 10, and h, including the local transition, is 3. Therefore, for this NoC $L_{max} = 2 \times 2 + 10 + 3 = 17$ clock cycles. For a larger message with n words the latency is dominated by the schedule length (which sets the bandwidth of the NoC):

$$L_{max} = 2 \times l + n \times s + h \tag{2}$$

4.4 Source Access

The source of the S4NOC, the benchmarks, and a README file explaining the build process and how to reproduce the results are available in open source. The source of the S4NOC and the NI is available as part of the Patmos project at https://github.com/t-crest/patmos. Detailed instructions how to run the experiments from this sections can be found at:
https://github.com/t-crest/patmos/tree/master/c/apps/s4noc.

5 CONCLUSION

Multicore systems used for real-time systems need a time-predictable way to communicate data between processing cores. We present a statically scheduled network-on-chip, which we call S4NOC. S4NOC uses time-division multiplexing for all resources and therefore provides complete isolation between individual virtual channels. This isolation enables to compute an upper bound of the message latency.

The hardware design of a statically scheduled NoC results in a low resource requirement. The implementation of the S4NOC requires about 6 % of the resources of a RISC processor pipeline per node. Although, time-division multiplexing is not work conserving, the S4NOC design provides more bandwidth than can be practically used by an application at the low hardware cost.

ACKNOWLEDGMENTS

The work was partially funded by the Danish Council for Independent Research | Technology and Production Sciences under the project PREDICT, contract no. 4184-00127A.

REFERENCES

[1] Accellera Systems Initiative. 2013. Open Core Protocol Specification, Release 3.0. Available at http://accellera.org/downloads/standards/ocp/.

[2] Florian Brandner and Martin Schoeberl. 2012. Static Routing in Symmetric Real-Time Network-on-Chips. In *Proceedings of the 20th International Conference on Real-Time and Network Systems (RTNS 2012)*. Pont a Mousson, France, 61–70. https://doi.org/10.1145/2392987.2392995

[3] Farzaf Fatollahi-Fard, David Donofrio, George Michelogiannakis, and John Shalf. 2016. OpenSoC Fabric: On-chip network generator. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 194–203. https://doi.org/10.1109/ISPASS.2016.7482094

[4] Kees Goossens and Andreas Hansson. 2010. 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)*. 306 –311.

[5] Nachiket Kapre and Jan Gray. 2015. Hoplite: Building austere overlay NoCs for FPGAs. In *25th International Conference on Field Programmable Logic and Applications (FPL 2015)*. 1–8. https://doi.org/10.1109/FPL.2015.7293956

[6] Evangelia Kasapaki, Martin Schoeberl, Rasmus Bo Sørensen, Christian T. Müller, Kees Goossens, and Jens Sparsø. 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

[7] Jörg Mische, Martin Frieb, Alexander Stegmeier, and Theo Ungerer. 2019. PIMP My Many-Core: Pipeline-Integrated Message Passing. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS 2019)*.

[8] 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 (NoCArc '12)*. ACM, New York, NY, USA, 63–68. https://doi.org/10.1145/2401716.2401730

[9] Morten B. Petersen, Anthon V. Riber, Simon T. Andersen, and Martin Schoeberl. 2018. Time-Predictable Distributed Shared Memory for Multi-Core Processors. In *2018 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. 1–7. https://doi.org/10.1109/NORCHIP.2018.8573463

[10] Martin Schoeberl. 2018. One-Way Shared Memory. In *2018 Design, Automation and Test in Europe Conference Exhibition (DATE)*. 269–272. https://doi.org/10.23919/DATE.2018.8342017

[11] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. 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

[12] Martin Schoeberl, Florian Brandner, Jens Sparsø, 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

[13] Martin Schoeberl, Luca Pezzarossa, and Jens Sparsø. 2019. A minimal network interface for a simple network-on-chip. In *Architecture of Computing Systems - ARCS 2019*. Springer, 295–307. https://doi.org/10.1007/978-3-030-18656-2_22

[14] 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. https://doi.org/10.1007/s11241-018-9300-4

[15] Jens Sparsø, Evangelia Kasapaki, and Martin Schoeberl. 2013. An Area-efficient Network Interface for a TDM-based Network-on-Chip. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '13)*. EDA Consortium, San Jose, CA, USA, 1044–1047.

[16] Saud Wasly, Rodolfo Pellizzoni, and Nachiket Kapre. 2017. HopliteRT: An efficient FPGA NoC for real-time applications. In *2017 International Conference on Field Programmable Technology (ICFPT)*. 64–71. https://doi.org/10.1109/FPT.2017.8280122