A Minimal Network Interface for a Simple Network-on-Chip

Martin Schoeberl[1], Luca Pezzarossa[1], and Jens Sparsø[1]

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

Abstract. Network-on-chip implementations are typically complex in the design of the routers and the network interfaces. The resource consumption of such routers and network interfaces approaches the size of an in-order processor pipeline. For the job of just moving data between processors, this may be considered too much overhead. This paper presents a lightweight network-on-chip solution. We build on the S4NOC for the router design and add a minimal network interface. The presented architecture supports the transfer of single words between all processor cores. Furthermore, as we use time-division multiplexing of the router and link resources, the latency of such transfers is upper bounded. Therefore, this network-on-chip can be used for real-time systems. The router and network interface together consume around 6 % of the resources of a RISC processor pipeline.

Keywords: Network-on-Chip · Network Interface · Real-Time Systems · Multicore Processor · Communication.

1 Introduction

With the move to multicore processors to increase performance (both average case and worst case), the emerging question is how those multiple cores communicate to execute a distributed workload. One of the main aims is to keep the communication on-chip to avoid the time and energy cost of moving bits off-chip to and from shared main memory. For this, on-chip communication networks-on-chip (NoC) architectures have emerged.

The research field of NoC architecture and implementation is large and diverse. While some general understanding of router designs have evolved (possibly because routers implement well defined and limited functionality), the architecture and implementation of network interfaces (NIs) is more diverse, complex, and difficult to compare.

NIs can be optimized for quite different uses. We identify five different uses of NoCs: (1) supporting cache coherence protocols, (2) single word memory accesses to a different core or input/output device, (3) access to a shared external memory, (4) supporting message passing between cores, and (5) supporting streaming data. Depending on the types of traffic supported, the NoCs and in particular the NIs providing the interface to the NoC may be rather diverse. We see a tendency to implement more support functionality in hardware, e.g., end-to-end flow control and buffer handling with DMA support. In combination with different packet and bus interfaces this results in a large variety of NI designs that are often quite advanced and expensive.

This paper presents a minimal NI that directly supports the synchronous data flow model of computation [7]. It supports sending of single word packets from a sender to a receiver. The resulting NI is lightweight and consumes a fraction of resource compared to other NIs. The resource consumption of the NI and the router is around 6 % of the resources of the Patmos processor, which we use in the evaluation. Support for message passing or streaming data can be added in software on top of the service that the NI provides. If needed, flow control can also be handled in software.

As a starting point, we use a simple NoC, the S4NOC [13], that is available in open source.[1] S4NOC uses time-division multiplexing (TDM) of the link and router resources. The tool to generate TDM schedules [3] is also available in open source. We extend the S4NOC with a simple NI with first-in-first-out (FIFO) buffers and connect it to the T-CREST multicore platform [12], similar to the one-way memory [11] project.

The proposed NoC and NI are optimized for the real-time domain. To enable static worst-case execution time analysis of tasks, the computing platform and the communication needs to be time-predictable. The S4NOC was designed to be time-predictable. Therefore, our NI extension aims to be time predictable as well.

The contributions of this paper are: (1) a reestablishing of a minimalistic NoC using static TDM arbitration and simple routers and (2) a minimal NI that supports message passing between processing cores on top of the low-cost NoC. Furthermore, we present a benchmarking framework for NoCs that support data flow applications.

This paper is organized in 6 sections: Section 2 presents related work. Section 3 provides background on the S4NOC architecture that we use to build upon. Section 4 presents the minimal NI as a fit for the low-cost S4NOC architecture. Section 5 evaluates the NI design for the S4NOC. Section 6 concludes.

2 Related Work

For time-predictable on-chip communication, a NoC with TDM arbitration allows for bounding the communication delay. Æthereal [5] is one such 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. We conform to the TDM approach of Æthereal, but present a simpler NI in this paper. In comparison with the aelite, which is one variant of the Æthereal family of NoCs, the S4NOC, including our proposed NI, is considerably smaller. For a 2x2 NoC, the S4NOC uses 1183 4-input LUTs and 1110 flip-flops. In contrast, aelite uses 7665 6-input LUTs and 15444 flip-flops [16].

The PaterNoster NoC [10] avoids flow control and complexity in the routers by restricting a packet to single standalone flits. The NI of PaterNoster is a simple design to support single word packets. The NI is connected to the memory stage of a RISC-V processor [9]. The RISC-V instruction set has been extended with a transmit instruction that blocks until a free slot is available in the NoC and a receive instruction that explores all input buffers in parallel to find a packet for a source address. If no packet is available, the pipeline blocks. Our NoC uses a similar architecture, but we use TDM based

[1] The original design is available in VHDL at `https://github.com/t-crest/s4noc`, while a rewrite in Chisel [2] has been made available at `https://github.com/schoeberl/one-way-shared-memory`.

scheduling. Our NI is mapped into an address and can be accessed by normal load and store instructions. Furthermore, by avoiding a full lookup in the receive buffer, our NI is more than a factor of 10 smaller than the PaterNoster NI.

The OpenSoC Fabric [4] 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 is planned. A single OpenSoC router (in the default configuration) is as large as our complete 3x3 NoC including the NIs and open core protocol (OCP) interfaces.

Similar to Æthereal, the Argo NoC [6] uses a TDM based NoC, but also uses the same TDM schedule in the NI [15]. The Argo NI and NoC offer time-predictable transfer of data from a core local memory across the NoC and into a local memory of another core. This TDM-based DMA mechanism is part of the NI, and as a result, data is transferred without any buffering or (credit based) flow control. In comparison with the NI presented in this paper, the Argo NI is substantially larger, as the use of DMA-driven data transfer results in a correspondingly higher throughput across the NoC when larger blocks of data are transferred.

The one-way shared memory [11] project uses the S4NOC to implement a special form of distributed shared memory. Each core contains a local on-chip memory where blocks within those local memories are constantly copied to other cores. The one-way shared memory is also a design with low resource consumption, but the programming interface is very different from our NI.

3 The S4NOC Design

Our work builds on top of the S4NOC NoC design [13] by adding a minimal NI. Therefore, we provide here background information on the S4NOC design. The S4NOC implementation in Chisel does not contain a NI but is just used for a one-way shared memory [11]. Therefore, we add a NI to the S4NOC with the same design philosophy of building a lightweight NoC.

The S4NOC is a statically scheduled, time-division multiplexed (TDM) NoC intended for real-time systems. As all traffic is statically scheduled, there are no conflicts on any shared resource, such as links or multiplexers. Without conflicts, there is no need to provide buffering in the routers, flow control between routers, or credit-based flow control between the NIs.

A static schedule for the TDM NoC is precomputed and results in a TDM round with individual TDM slots. For single word packets, the TDM slot is a single clock cycle. Each core can send one word to every other core in one TDM round. The slot number identifies the virtual circuit to the receiving core. The TDM round repeats for further packets.

The original design supports single word packets and single cycle hops between routers. The routers contain one output register per port and a multiplexer in front of that register. The schedule is stored in the router and drives the multiplexers for the five output ports.

The default configuration of the S4NOC is a bidirectional torus, resulting in five output ports (north, east, south, west and local) and four inputs to the multiplexers, which form the crossbar. The default schedule is a one-to-all schedule where each core has a dedicated virtual circuit to each other core. With such a regular structure of a bidirectional torus and an all-to-all schedule, it is possible to find one schedule that is executed in all routers [3]. That means it is the same for all routers, e.g., if at one clock cycle a word is routed from west to north, it is done in all routers.

The resulting hardware is lean. One register per port, one 4:1 multiplexer per port, a counter for the TDM schedule, and a table for the schedule. With Chisel, the table for the schedule is computed at the hardware generation time.

4 The Minimal Network Interface

Figure 1 shows an overview of a 9-core processor organized in a 3×3 grid. All cores are connected via a NI to the network of routers. The NoC topology is a bidirectional torus. The bidirectional torus minimizes the number of hops for a packet to travel. The corresponding all-to-all core communication graph for N cores has $N \times (N-1)$ virtual circuits. For a 3×3 multicore, this results in 72 virtual circuits, which can be served by a 10 slot TDM schedule [3] for the NoC. This is only 2 slots more than what is needed by the 8 outgoing and 8 incoming virtual circuits. This short TDM schedule is possible due to the high bandwidth provided by the 36 links connecting the 3×3 multicore.

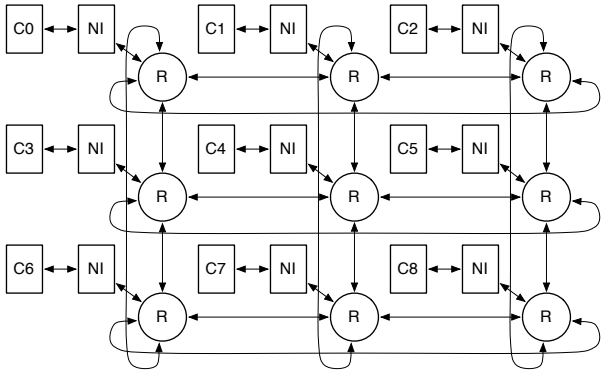

Fig. 1. A 3x3 multicore connected by a bi-torus NoC.

A straightforward implementation of a NI could use separate FIFOs for each virtual circuit endpoint; in the 9 core example, this would be 8 FIFOs for transmitting data and 8 FIFOs for receiving data. The result would be a relatively large design and a design that scales poorly with a growing number of cores.

In our design, the same functionality is implemented by a combination of hardware and software. By exploiting the TDM scheduling used in the routers and by sacrificing a small amount of bandwidth, we have been able to design a NI that has only a single

FIFO for transmission of data and a single FIFO for reception of data. The result is a small NI design, as shown in Figure 2.

A virtual circuit can be identified at the senders end by the slot number in which its data is transmitted and at the receivers end by the slot number when its data is received. The slot number is stored in the transmit FIFO along with the data to be transmitted. The slot number of the element at the head of the transmit FIFO is compared against the TDM slot counter and the data is sent at the scheduled point of time.

From the view of the processor, the NI is a peripheral device mapped into the address space of the processor. It consists of a transmit and receive buffer and two flags for the status of those buffers. The transmit buffer contains a flag showing if the buffer is empty, the receive buffer contains a flag if there is some data available. The sender and receiver have to poll these flags.

Figure 2 shows the NI in detail. The NI contains two FIFO buffers: one receive (RX) FIFO and one transmit (TX) FIFO. On the processor side, those buffers are connected as an IO device via the OCP [1] interface. On the NoC side, the buffers are connected to the local port (L) of the router. The TDM slot counter compares the current count with the slot number of the packet at the head of the TX FIFO and inserts it into the NoC if equal. On the receiving side, the NI takes a valid packet from the local port and inserts it, together with the value of the TDM slot counter, into the RX FIFO.

The data word and the slot number are the basic interfaces to the NI. To transmit a word from core A to core B, at core A the sender needs to know which slot number belongs to the virtual circuit from A to B. The mapping between the slot number and the virtual circuit is derived from the static TDM schedule. At the receiving end, core B reads the data and the receiving slot number when the packet has arrived. The slot number when a word is received indentifies the source node. Therefore, there is no further information needed in the packet or in the NI to determine the source or destination of a packet.

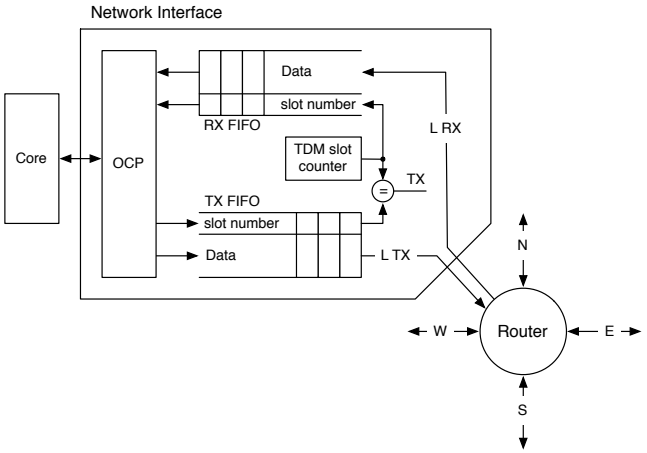

Fig. 2. One processing node consisting of a core, our NI, and a router.

At the sending side, we optimize the write into the NI by using the lower bits of the address to determine the send slot number. E.g., when the processor writes the data word to BASE_ADDRESS + 3, it requests a send in time slot 3. With the polling of the TX FIFO empty flag, sending a single word needs at least one load and one store instruction.

When a packet is received from the network the payload data is written into the RX FIFO along with the slot number when it was received, which identifies the sender. Before reading the RX FIFO, the core must first read the data available flag to ensure there is data to read. And based on this, the software can identify the virtual circuit and, thus, the sender. The software is in charge to dispatch packets received from different cores to different tasks waiting for the packets. The NI only provides the virtual circuit number in form of the slot number when the packet arrived.

On the receive side, we need two load instruction to read the data and to determine the receiving slot number. Including the polling for data available this results in a minimum of three load instructions. However, if the sender is known, we can avoid reading the receive slot number, resulting in two instructions per word, as at the sending part.

As the TX FIFO in the sender NI is shared among all the outgoing virtual circuits, only the head of the queue can be sent into the switched structure of the NoC. This can produce head-of-queue blocking 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, the software inserts the packets 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. Once the head of the queue is sent, the rest of the data in the RX FIFO is sent uninterruptedly, since the destination of each data is ordered.

Having a dedicated TX FIFO per outgoing virtual circuit would remove the head-of-queue blocking and the initial waiting for the TDM slot for the data at the head of the queue. In our approach, we trade a minor reduction in performance (waiting for the head-of-queue TDM slot and ordering in software) for a minimal and simple architecture.

The NI design (and TDM arbitration) might waste bandwidth. However, the key parameter is what bandwidth can be achieved at what hardware cost. If our design is small, we can waste bandwidth at a very low cost.

5 Evaluation

In this section, we evaluate and discuss the presented NI/NoC architecture in terms of performance and hardware cost. As part of the evaluation, we present the custom micro-benchmark framework based on the data flow model of computation that we developed and used to characterize the NI/NoC performance.

The results are produced using Intel/Altera Quartus Prime (v16.1) targeting the Intel/Altera Cyclone IV FPGA (model EP4CE115) which is used on the DE2-115 board. Specifically, performance results are obtained by running the benchmarks on a 3-by-3 multicore platform implemented on the same FPGA using the Patmos [14] processors as cores.

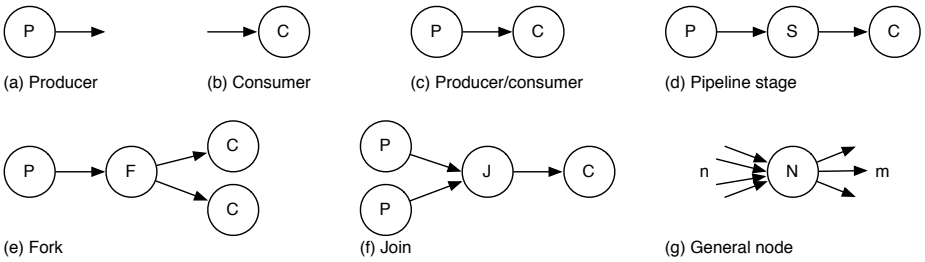

Fig. 3. Elementary structures that can be used to model data flow applications. Structure (c) to (f) are used as benchmarks.

5.1 Benchmarking Method

Our NI is intended to support message passing between processor cores. Therefore, we introduce a benchmarking framework inspired by the synchronous data flow model of computation [7]. In this model of computation, data are processed by a statically ordered sequence of actors. When an actor receives enough input tokens (data units), it starts the computation to produce output tokens to be sent to the next actors.

The benchmarks consist of a selection of elementary structures that can be used to model data flow applications. The actors are running on different nodes of the platform and the NoC supports the communication channels between them. In other words, the elementary structures can be considered as the building blocks of any data flow applications.

Figure 3 shows the elementary structures, where the ones of Figures 3(c - f) are directly used as benchmarks. The elementary structures are as follows: (a) A producer, with a single output channel, that can produce at a pre-determined rate. (b) An eager consumer, with a single input channel, that can receive as fast as possible. (c) A producer directly connected to a consumer. This benchmark is used to measure the pure NoC throughput between two actors placed in different nodes. (d) A pipeline stage, with one input and one output channels. This benchmark is used to characterize the overhead of the pipeline stage node. (e) A fork stage, with one input and two or more output channels. This benchmark is used to characterize the overhead of the fork node. (f) A join stage, with two or more input and one output channels. This benchmark is used to characterize the overhead of the join node. (g) The general case node, where an actor has n input channels and m output channels. The above classifications are specializations of this general node.

5.2 Performance

The maximum bandwidth offered by the NoC depends on the TDM schedule. The following analysis assumes a schedule that implements a fully connected core communication graph where each processor core has a (virtual) circuit towards all other processors. The maximum bandwidth on a virtual circuit corresponds to one word per TDM round. The TDM round for the 3×3 platform used for the experiments is 10 clock cycles.

Table 1. Maximum measured throughput, in clock cycles per word, for the four micro benchmarks used in the evaluation.

Benchmark	Throughput (clock cycles per word)
Producer/consumer	10.1
Pipelined stage	10.1
Fork	23.1
Join	25.1

To evaluate the performance of the NoC/NI architecture, we measure the bandwidth between actors (processor cores) for the elementary structures presented earlier. We assume a time-triggered system without any form of flow control. In the experiments, we increase the transmission rate of the producer until the consumer is saturated (i.e., just before it would start to miss packets/tokens).

Table 1 presents the measured maximum throughput, expressed in clock cycles per word per channel, for the four elementary structures used in the evaluation. For the first two benchmarks, the measured throughput coincides with the maximum theoretical one of one word per TDM round since all the actors involved are faster than the TDM round. For the fork and join test cases, the throughput is lower. This can be explained by observing that the fork and the join actors have to perform more operations before being able to send a token to the next actors.

If flow-control is introduced in form of credits sent back from the receiver to the sender, the maximum measurable throughput is reduced. Due to more software overhead, the latency of individual words is increased. Hardware support for flow-control would result in a shorter latency. We implemented a version of the producer/consumer example with flow control using a single credit, and in this case the throughput is 23.0 clock cycles per word (as opposed to 10.1 for the time triggered organization).

All the results presented and discussed above are obtained using a FIFO queue of 4 words. Further buffering is managed in software. The sending and the receiving operations consist of two nested for-loops. The outer loop iterates every time an entire buffer is sent or received by the inner loop, which iterates for every word of a buffer. Figure 4 shows the maximum measured throughput for the four benchmarks for buffer sizes from 1 to 64 words.

For all the graphs we observe a similar course or pattern: a decrease to a minimum followed by an increase to a maximum and finally stabilization to a value between the minimum and the maximum. This can be explained by the effect of the loop unrolling executed by the compiler on the inner loop. The minimum occurs when the compiler completely unrolls the loop, while the maximum occurs when the number of loop iterations is too large for the loop to be unrolled.

5.3 Hardware Cost

The resource consumption is given in 4-input look-up tables (LUT), flip-flops (DFF), and memory consumption in bytes. The memory consumption only refers to the memory

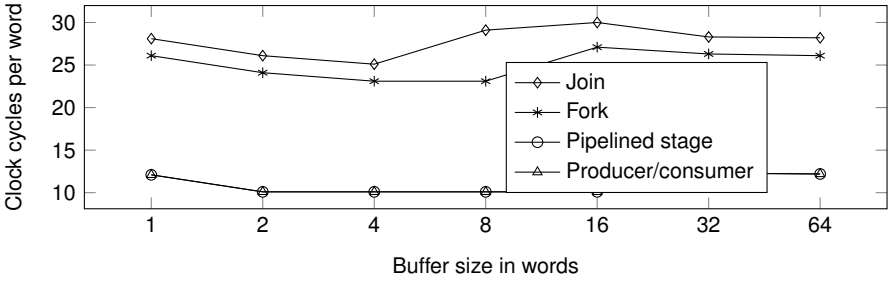

Fig. 4. Maximum measured throughput, in clock cycles per transferred word, for the four micro-benchmark for different buffer sizes. The graphs for the pipelined stage and the producer/consumer benchmarks fall on top of each other.

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. Maximum clock frequency is reported for the slow timing model at 1.2 V and 85 C.

Table 2. Resource consumption, maximum frequency, and length of the TDM schedule of different configurations of the S4NOC.

Configuration	LUT	DFF	fmax (MHz)	Sched. length
$2 \times 2 = 4$	1784	1596	235.8	5
$3 \times 3 = 9$	5351	4221	236.1	10
$4 \times 4 = 16$	10761	7568	221.0	19
$5 \times 5 = 25$	17732	11825	216.6	27
$6 \times 6 = 36$	29136	17172	188.6	42
$7 \times 7 = 49$	36783	23373	195.5	58
$8 \times 8 = 64$	55423	30784	183.2	87
$9 \times 9 = 81$	68079	38961	172.8	113
$10 \times 10 = 100$	94540	48500	150.8	157

Table 2 shows the hardware resource consumption of the S4NOC (NI and routers) in different configurations. We generate those synthesize results with simple traffic generators (instead of the OCP interface) that drive the local ports and merge the outputs of the local ports to FPGA pins. We also provide the maximum clock frequency and the length of the TDM schedule in the table.

We observe a slightly higher than linear increase of the resource usage with the increase in the number of nodes. This is a result of the larger schedule tables in the routers for larger NoCs. Furthermore, we observe a decrease in the maximum clocking frequency as the number of nodes increases. However, the maximum frequency is still higher than the maximum frequency of the Patmos core, which is below 80 MHz in the used FPGA.

Table 3 shows the hardware resource consumption of the S4NOC using the presented NI with the OCP interface and other NoCs. The first group of entries in Table 3 shows the resource consumption of a single S4NOC node including the router and the NI for a configuration with 4 buffers in the FIFOs. The resource consumption is further split into the router and NI components. The resource numbers have been collected from a 3×3 configuration, where we took the median value of the resource consumption of the 9 nodes. 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 next group of entries in Table 3 report the results for a single node of the Argo NoC [6]. The Argo NoC is available in open source. Therefore, we can obtain the results by synthesizing two configurations of the Argo NoC for the same FPGA.

The next group set of result in Table 3 is for the PaterNoster node for a 2×2 configuration. Similarly to S4NOC and Argo, the PaterNoster NoC is available in open-source, which allows us to synthesize it for the same FPGA. From the results, we can observe that the S4NOC node is more than 10 times smaller than the PaterNoster node. The PaterNoster NI is relatively large, as it contains a fully associative receive buffer to be able to read from any channel independently of the receiving order.

Table 3. Resource consumption of different components of the S4NOC compared with other designs.

Component	LUT	DFF	Memory
S4NOC node	602	453	0
router	266	165	0
network interface	336	288	0
Argo node	1750	926	1.3 KB
router	932	565	0
network interface	849	361	1.3 KB
PaterNoster node	8030	3546	0
router	1899	1297	0
network interface	6131	2249	0
OpenSoC router	3752	1551	0.8 KB
3×3 S4NOC	5423	4382	0
3×3 Argo NoC	15177	8342	12.1 KB

The table also presents the results for a single router of the OpenSoC NoC [4]. 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. From the results, we can observe that the size of a single OpenSoC router is as large as the entire 3×3 S4NOC with a single buffer.

The next group shows resource consumptions of complete 3×3 NoCs. The S4NOC is around 3 times smaller than the Argo NoC. At this cost, the Argo NoC provides hardware support for message passing and DMA handling.

When comparing an S4NOC node with the size of a Patmos core, which consumes 9437 LUTs and 4384 registers, we can see that we achieved our goal of a small NoC. The resource consumption of one NI and router is around 6 % of the Patmos core. When comparing our NoC with a leaner RISC core, such as the RISC-V implementation that is part of the Real-Time Capable Many-Core Model [8] and consumes 5375 LUTs and 1557 registers, our NoC is still in the range of 11 % of that RISC pipeline.

5.4 Source Access

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`.

6 Conclusion

State-of-the-art network-on-chip implementations tend to provide a lot of functionality in hardware. This results in complex design of the routers and the network interfaces. The resource consumption of such routers and network interfaces approache the size of a simple processor pipeline.

The paper presents a design at the other end of the spectrum: a lightweight network-on-chip solution with a minimal network interface that supports the transmission of single word packets between processor cores. The resulting design consumes about 6 % of the resources of a RISC processor pipeline per node. Furthermore, as we use time-division multiplexing of the router and link resources, the latency of the communication is upper bounded and we can use this network-on-chip for real-time systems.

Acknowledgment

We would like to thank Constantina Ioannou for bringing up the idea of simply using a FIFO as a network interface.

The work presented in this paper was partially funded by the Danish Council for Independent Research | Technology and Production Sciences under the project PREDICT (`http://predict.compute.dtu.dk/`), contract no. 4184-00127A.

References

1. Accellera Systems Initiative: Open Core Protocol specification, release 3.0. Available at `http://accellera.org/downloads/standards/ocp/` (2013)
2. 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)

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. Fatollahi-Fard, F., Donofrio, D., Michelogiannakis, G., Shalf, J.: Opensoc fabric: On-chip network generator. In: 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 194–203 (April 2016). https://doi.org/10.1109/ISPASS.2016.7482094
5. 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)
6. 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
7. 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
8. 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)
9. 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)
10. Mische, J., Ungerer, T.: Low power flitwise routing in an unidirectional torus with minimal buffering. In: Proceedings of the Fifth International Workshop on Network on Chip Architectures. pp. 63–68. NoCArc '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2401716.2401730
11. 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
12. 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
13. Schoeberl, M., Brandner, F., Sparsø, J., Kasapaki, E.: 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). pp. 152–160. IEEE, Lyngby, Denmark (May 2012). https://doi.org/10.1109/NOCS.2012.25
14. 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
15. 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)
16. Stefan, R.A., Molnos, A., Goossens, K.: dAElite: A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-Up. IEEE Transactions on Computers **63**(3), 583–594 (2014). https://doi.org/10.1109/TC.2012.117