Time-predictable Distributed Shared Memory for Multi-core Processors

Morten B. Petersen, Anthon V. Riber, Simon T. Andersen, and Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark

***Abstract*—Multi-core processors for real-time systems need to have a time-predictable way of communicating. The use of a single, external shared memory is the standard for multi-core processor communication. However, this solution is hardly time predictable. This paper presents a time-predictable solution for communication between cores, a distributed shared memory using a network-on-chip. The network-on-chip supports reading and writing data to and from distributed on-chip memory. This paper covers the implementation of time-predictable read requests on a network-on-chip. The network is implemented using statically scheduled, time-division multiplexing, enabling predictions for worst-case execution time. The implementation attempts to keep buffering as low as possible to obtain a small footprint. The solution has been implemented and successfully synthesized with a multi-core system on an FPGA. Finally, we show resource and performance measurements.**

***Index Terms*—NoC, Distributed memory, Real-time systems**

I. Introduction

Multi-core processors for real time applications need to have a time predictable way of communication. This paper presents a solution for time predictable communication for multi-core processors through distributed shared memory (DSM) using a network-on-chip (NoC). This solution builds on the ideas presented in [1], which presents an implementation of a one-way shared memory architecture using a statically scheduled network [2], [3]. As all this work is available in open source, we are able to build our version of DSM on top of this work.

The one-way memory [1] presents a solution for communicating between multiple cores using writes only. However, this results in some obvious limitations. Supporting reads in the solution presented in [1], will spoil the time predictable properties. Replying to a memory read request in the same cycle, as the core local to the network interface (NI) desires to write to memory to the core which sent the incoming read request, raises an issue. In these cases, if the schedule, used in [1] and [4], is not altered, the core would have to buffer one of its transmissions an additional schedule period, since only one message can be transmitted in the given time slot. This gives rise to a possible accumulation of read and write requests to be answered, and thus ruins time predictability.

This paper explores a possible solution for a time predictable network with support for both read and writes. Reading is enabled by adding a separate network to split the network traffic between writes and read-request responses. The main features of the provided solution are:

- The total address space is distributed across all NIs, allowing for fast memory accesses from cores to the address space local to their NI, avoiding potential bottlenecks from a single globally shared memory.
- A dual-ported memory in the NI supports concurrent access from a core (local access) as well as for handling external memory accesses.
- NIs supporting external write requests (a core writing to the memory in an external NI) as well as read requests (a core requesting memory in an address range located in an external NI).
- The system is using a statically, time-division multiplexing (TDM) based scheduled NoC [4], which allows the implementation to be built as a time-predictable architecture [5], suitable for time-critical real-time systems where the worst-case execution time must be bound and analyzable.
- A NoC and NI with low resource consumption.

II. Related Work

Communication between cores can be achieved either via cache coherence protocols that back up shared memory, or through explicit message passing with a NoC between core-local memories. For time-predictable on-chip communication, a NoC with TDM arbitration makes it possible to give bounds for the communication delay. Æthereal [6] 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. Like Æthereal, the Argo NoC [7] uses a TDM based NoC but also uses the same TDM schedule in the network interface [8]. We follow the TDM approach of Æthereal and Argo, but provide an even simpler network interface to the NoC.

The Paternoster NoC [9] avoids flow control and complexity in the routers by restricting a packet to single standalone flits. Paternoster connects the routers in a unidirectional torus, which results in only three ports in the routers. Flits are only inserted into the X ring when the slot is free. For the direction change to Y, a small FIFO buffer can hold the packet until a free slot arrives.

Our NoC uses a similar architecture and employs just single word packets. However, we use statically scheduled TDM-based arbitration to bound the maximum latency for packets and avoid any buffering in the routers. Furthermore, we support write and read requests by providing two NoCs: one for write and read request and one for the read response.

The Real-Time Capable Many-Core Model proposes many cores with a NoC with TDM-based arbitration [10]. The

978-1-5386-7656-1/18/$31.00 2018 IEEE

Reduced Complexity Many-Core architecture [11] proposes avoiding shared memory completely and supporting timing analysis by using a fine-grained message passing NoC. We agree with this preference for on-chip communication between local memories over shared memory communication, but support read and write operations.

The Hoplite architecture [12] 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. The Hoplite design is carefully handcrafted for Xilinx FPGAs to provide the lowest hardware consumption possible. While we praise the engineering effort, we think that the higher-level description of the architecture makes it more applicable for different use cases and technologies.

The multi-core processor Epiphany [13] uses a distributed memory architecture. Each core contains 32 KB of local memory that is mapped onto a global address space. The processors contain no caches. Access to the memory of a remote core takes place over a network-on-chip (NoC). The NoC is organized as a mesh, and favors writes over reads, as writes are posted such that the processor does not need to wait for the write to finish. Packets are single words and routing is performed in a single cycle per hop. However, conflicting packets can result in delays that are hard to bound. In contrast, we use TDM-based arbitration for our distributed memory.

Similar to our presented NoC, the Argo NoC [7] aims for time-predictability. It uses TDM-based arbitration in the routers. The Argo NoC also uses TDM-based DMA transfer of data from the local memory to the NoC. Argo uses source routing, which means the routing information is transmitted as a header word. For this header processing and the switching in the crossbar, the router is pipelined into three stages. By contrast, our simple router includes the routing information in the router itself and each simple multiplexer requires just a single pipeline register in the router. Another important difference is that Argo also supports global asynchronous, local synchronous systems with an asynchronous router design.

The one-way shared memory [1] is a communication architecture for multi-core systems using distributed shared memory, which implements functionality for communicating between nodes in the network using only writes. The basis of that implementation, such as the NoC and the scheduler, have been used in the solution presented in this paper.

The main differences between our work and all presented projects (except Epiphany) is the support of reads directly in the NI/NoC.

III. The S4NOC Network-on-Chip

We build our DSM on top of the S4NOC project [2] and the T-CREST multi-core platform [14]. S4NOC is now part of T-CREST and available in open source at GitHub: https://github.com/t-crest.

The S4NOC is a statically scheduled 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 the possibility of conflicts there is no need to provide buffering in the routers, flow control, or credit-based arbitration.

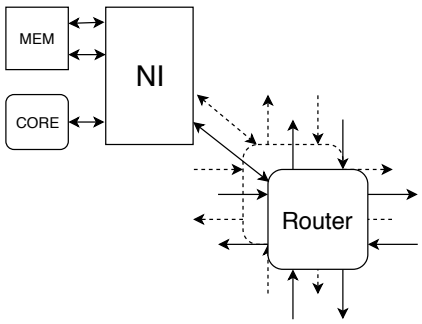

Fig. 1. Organization of a single tile. A dual-port memory is connected to the NI as well as the routers for the two separate networks, and a core.

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 multiplexer.

The default configuration of a S4NOC is a bidirectional torus, resulting in five output ports (north, east, south, west and local) and four inputs to the multiplexers. The default schedule is a one-to-all schedule where each core has a dedicated channel 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 very lean. One register per port, one 4:1 multiplexer per port, a counter for the TDM schedule, and a table for the schedule. The table for the schedule is generated at the hardware construction time.

IV. Distributed Shared memory

We propose a design that enables the sharing of memory situated locally in a tile, with all other tiles in the network. Cores can send memory access requests to its NI, wherein the NI will decide whether the request shall be delegated to the memory local to the NI, or a memory in an external NI. This delegation is done based on the address of the memory request. These requests can be either writes or reads.

A. Memory Organization

As shown in Figure 1, a dual-port memory is placed inside each tile. The dual-port memory enables two concurrent write- or read accesses to the memory. This allows support for handling memory requests from a core to its local memory in the same clock cycle as handling a read or write request from a remote core via the NoC. Given a suitable programming model, this organization can yield a substantial speedup in program execution time, relative to blindly using arbitrary addresses in the full shared memory space, given that cores are delegated to execute on memory local to their NI which has a one cycle memory access. Having a dual ported memory

TABLE I
DISTRIBUTION OF A 1024-WORD (10-BIT WIDE) ADDRESS SPACE ON A 2X2 NOC.

Address	NI
0x034	0
0x15f	1
0x229	2
0x359	3

Address space	NI
0x000 - 0x0ff	0
0x100 - 0x1ff	1
0x200 - 0x2ff	2
0x300 - 0x3ff	3

opens up for multiple writes and reads to the same address. In a write-write case to the same address, the core which owns the memory will have its value written and the other write request will be ignored. During a producer-consumer scenario, when a write and a read to the same address occurs simultaneously, the newly written value will be read.

The memory is sized and distributed in such a way to allow direct encoding of the receiving tile in the address space. An example of the memory distribution is as follows; Having a 2x2 NoC with a 1024-word large address space (10-bit wide), the lower 8 bits address a word within a *block*, whereas the upper 2 bits address the *location* of the given memory block within the network. Table I shows examples of this encoding.

The NI is able to determine if it needs to transmit a message request onto the network, based on the address of the memory request. The destination tile is addressed by sending the request out in the corresponding time slot. Each NI has a lookup table that maps destination addresses to time slots.

B. Network Structure and the Readback Network

Key to the idea of supporting time-predictable read- and write requests is having two separate NoCs. These two networks will be denoted as the *write* network and the *readback* network. The motivation for having two separate networks is to keep the schedule length as short as possible, and thus reduce the latency of memory accesses. As in [1], we use the TDM schedule developed in [15] and the router design from the S4NoC [4]. The TDM approach ensures a worst-case execution time for read and write requests, since the schedule is statically determined.

Each network will be responsible for handling separate kinds of communication:

- *Write network*: Handles transmission of memory writes from NIs to an address space external to the transmitting NI. Furthermore, handles the transmission of read requests from cores to an address space external to the transmitting core.
- *Readback network*: Handles transmission of read memory data from external NI to a requesting core, after a read request has been received.

The write network is able to support two functions, since a core in a given cycle is only able to send either a read- or a write request to its NI. The interface between a core and its NI follows the open core protocol [16]. After sending a request, a core is expected to stall until it receives a `valid` signal from its NI. The `valid` indicates either of the following; a *write* has been written to local memory or transmitted to the network, or that the value of its *read* request is available. From

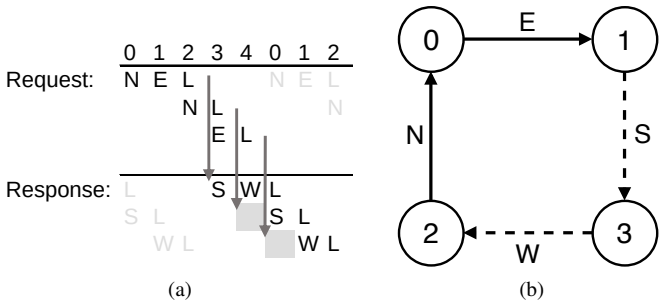

Fig. 2. Request: Write/read request network schedule, for the network responsible for writing and requesting reads to other NIs. Response: Readback network schedule, for the network responsible for returning data from a read request.
2a illustrates the schedules of a 2x2 write- and readback networks. Columns represent time slots. Rows hold a route each. Furthermore, 2a show the time shift between the two schedules. Grey boxes indicate buffered requests.
2b illustrates the concept of schedule inversion, where the shown routes are the topmost route of the Request and Response schedules. Solid lines represent the Request route and dashed the Response route.

a cores point of view, the interface for accessing both local and external memory are identical. However, the design inherently has a non-uniform memory access time, as the request depends on the access address as well as which time slot the request occurs in.

The phit width is dependent upon two factors: (1) the number of tiles in the network and (2) the size of the total address space. Furthermore, a valid bit and the data transmitted on a write is also required. In our case we support 32-bit words. Furthermore, a bit is reserved for indicating whether the transmitted message is a write or a read request. Finally, an address needs to be encoded into the message. As explained in section IV-A, since the top section of the address determines which NI should receive the request, only the address within a block needs to be transmitted, as the transmission slot is directly correlated to the upper part of the address. Thus, the phit width with 32-bit words can be calculated as follows:

$$w_{Write} = 1 + 1 + DW + \left\lceil \log_2 \left(\frac{memorysize}{n} \right) \right\rceil \quad (1)$$

With *n* being the number of tiles in the network, and DW the width of the data.

The static nature of the readback network allows a very lean implementation. No address information is required to be transmitted on the network, since the requesting core is expected to stall until a response is received from an external tile. Therefore, it is known that when receiving data on the readback network, that the received data is directly tied to the address at which the request was made. Having this, the phit width of the readback network will be:

$$w_{Readback} = 1 + DW \quad (2)$$

corresponding to a valid bit and, in this implementation, 32 bits of data.

C. Readback Network Schedule

With the given memory technology available, accessing the on-chip dual-port memory can be done in a constant-time. This

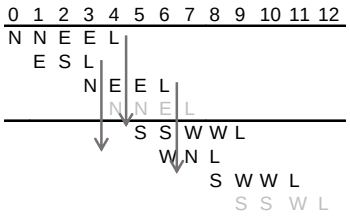

Fig. 3. Subpart of a default 4x4 schedule provided from [15]. The in-order constraint is in this case not obeyed and requires reordering.

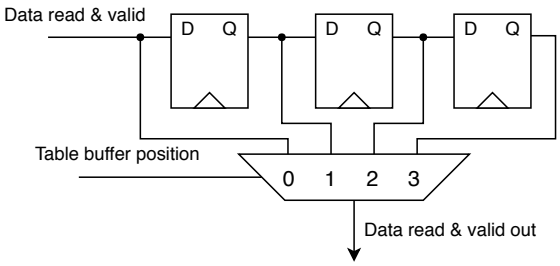

Fig. 4. Organization of the buffering circuitry for a schedule with 3 blank slots.

means that incoming requests can be handled simultaneously with local requests from the core. With this constant access time, the schedule for the *readback* network can be directly matched up with the schedule of the *write* network. This results in the possibility to implicitly transmit the read-request data to the core which transmitted the request, without having control circuitry in the NI dedicated to checking when the time slot to write the response on the *readback* network is present.

To generate a schedule for the readback network there is one major constraint which must be complied with; *The readback schedule must match up with the write schedule*. The implication of this constraint is, that the schedule must be able to transmit readback data in the same order as the requests are received (in-order).

As a solution, we introduce the concept of *schedule inversion*, where the following transformation is applied: `N→S, W→E, E→W, S→N`. Performing the inversion implies that responses to read requests will be routed to the transmitter of the read request through the readback network. Referring to figure 2b, it is seen that a read request sent from tile 2 to tile 1 is routed back from tile 1 to tile 2.

As inverting the schedule does not change the relative directions between the hops in a given timeslot. This assures that if a schedule generated for the write network with [15] is valid, this validity also applies to an inverted schedule.

Figure 2 shows the write- and readback schedule for a 2x2 NoC. From Figure 2 we see that this schedule follows the in-order schedule criteria, thus we can respond in-order and no hardware is needed to rearrange the readbacks. The in-order constraint is obeyed for the 2x2 and 3x3 schedule generated by the algorithm presented in [15].

The inversion of the write schedule does not always result in an in-order reply sequence. For instance, the 4x4 schedule generated by the S4NoC scheduler [15] introduces such an issue. Figure 3 illustrates the problem for when requests are not in order. The arrows show how the second schedule route is received before the first route. This leads to requests that are not received in the same order as they should be answered. We have made a valid 4x4 schedule by rearranging the routes of the generated schedule. This, however, comes at a price, which will be discussed in section VI.

D. Readback Network Time Shifting

With a valid schedule generated for the *readback* network, Figure 2 shows how the *readback* network schedule is shifted in relation to the *write* network schedule. The exact shift amount between the two schedules is specified by the length of the first schedule route in the *write* schedule. This is because we need to send a response to a request in the cycle after it was received. By shifting the second network by the length of the first schedule route, we see that a received request can be responded to immediately for the first route. As seen, the first route is 3 cycles long, and thus the *readback* schedule is shifted by 3 time slots. In practice, this variable time shifting is achieved by modifying the start value of the counter at which the router for the *readback* network uses to index into its schedule table. This solution is general wherein the same logic follows for 3x3 and larger schedules. With the time shifting, a precise match between the reception of a *readback* request is made with the transmission of the response. This is indicated by the arrows in figure 2.

E. Request Buffering

As indicated by the grey boxes in Figure 2, some requests need to be buffered before they can be transmitted to the *readback* network. This is indicated by the second arrow in figure 2, where the reception of the request from the transmitted core arrives one cycle before the transmission needs to occur. To accommodate for this, a shift register is implemented which has the request as input and has the same number of registers as there are leading blank spaces in the schedule. For the 2x2 and 3x3 networks there is only one leading blank space (one extra delay cycle between reception and transmission) and therefore only one register. The 4x4 network would require three registers, as there are three time slots where buffering is required. A multiplexer is then used to choose which request is currently being processed, as can be seen on figure 4. The multiplexer is controlled by a look up table generated at compile time, by analyzing the schedule. The lookup table dictates from where in the buffer data should be extracted in every time slot. The table is implemented as a ROM from where the address corresponding to the readback schedule time slot is accessed, and the data sent to the multiplexer. This is shown in Figure 4.

V. Results

The network uses the OCP interface for easy integration with multi-core platforms such as Patmos.

Using the T-CREST platform [14], the DSM has been attached to Patmos processors, synthesized, and implemented

TABLE II
RESOURCE CONSUMPTION OF THE DSM.

Component	LUTs	Registers
2x2 NI	145	84
2x2 Write network	1069	492
2x2 Read back network	534	397
2x2 Total	2188	1227
3x3 NI	200	85
3x3 Write network	3363	1845
3x3 Read back network	2386	1485
3x3 Total	7480	4085
4x4 NI	300	155
4x4 Write network	7472	3280
4x4 Read back network	6362	2640
4x4 Total	18549	8329

in an Altera DE2-115 FPGA, in 2x2 and 3x3 configurations. With this, C-language tests have been written and compiled for the Patmos processors. In these tests, a Patmos processor executes read- and writes to memory locations which maps to memory located in the other tiles of the DSM. Using the cycle counter in Patmos, latency of read- and writes can be measured, and from this, tests for measuring the bandwidth of our DSM performed.

In the T-CREST platform, each Patmos processor is connected to the main memory through TDM arbitration. It takes the main memory 21 cycles to deliver a 4 word burst, which yields a worst-case latency for accessing main memory being:

$$l_{wc} = n_{cores} \times 21 \tag{3}$$

In the context of the T-CREST platform, the DSM can therefore be seen as a high-speed solution for shared-memory inter-core communication.

A. Resource Consumption

Table II shows the resource usage for the 2x2, 3x3 and 4x4 networks. The consumption was found using Quartus Prime 16.1 for an Altera DE2-115 board that contains the Altera/Intel Cyclone IV FPGA. The NI usage is for a single NI and should thus be multiplied by the size of the network to get the total consumption. The shown consumption is excluding the actual memory, as this is dependent on the chosen size and implementation. All the networks assumed an address width of 10 bits. The individual NIs increase in size with larger configurations, as the look-up tables increase. The read back network is always smaller than the write network, as it carries less data. However, for larger networks the size difference diminishes. In comparison, it is noted that a single Patmos core uses approximately 8000 LUTs and 4200 registers. Therefore, an entire 3x3 network takes the same space as a single Patmos core.

The maximum frequency of the network has also been found using dummy cores, which are faster than the Patmos cores. For the Cyclone IV at the slow speed grade we observe the worst-case maximum frequency between 190 and 170 MHz for the 2x2, 3x3 and 4x4 networks. When running with a 3x3 Patmos setup the maximum frequency fell to 78 MHz, where the critical path was within the Patmos cores.

B. Bandwidth and Latencies

As we are targeting real-time systems the latencies need to be analyzable. Using the TDM based NoC this is easily doable. The lowest latency will be to the cores local memory, which is single cycle. Real-time systems care about the worst-case. We can analyze this for both a read and write request. As it is a statically scheduled TDM, we know exactly how long an NI will maximally wait to send a message. The maximal wait time is the length of the schedule (*period*) minus 1, if we just missed the time slot. Passing the message through the network will at most take the length of the longest schedule (*route*). Therefore, the worst-case latency for a write request will be *schedule period* - 1 + *route*. For a read request we need to include the response path, which is the same as the sent path, and the maximal buffering time in the node (*buffer*), which is equal to the number of blanks. Therefore, we get a worst-case latency for a read request of *schedule period* - 1 + *route* $\times$ 2 + *buffer*. For the 2x2, 3x3 and 4x4 network the *schedule period* is 5, 10 and 20 cycles, the longest route is 3, 4 and 5 cycles and the buffer is 1, 1 and 3 cycles.

The maximum bandwidth that can be achieved in the network occurs when cores execute write commands. To achieve maximum bandwidth, the cores must execute the writes such that they target external memories in the same sequence as defined by the schedule.

For instance, if in a 2x2 NoC the tiles are denoted as the set $\{N0 = NW, N1 = NE, N2 = SW, N3 = SE\}$, for $N0$, the optimal write sequence which follows the schedule of figure 2 will be $(NI3, NI2, NI1)$.

If this constraint is followed, a core can in a TDM round transmit $n_{nodes} - 1$ writes. Therefore, the maximum write bandwidth of the entire network can be stated as:

$$bw_{wr,max} = \frac{n \times (n-1)}{period} \times word \tag{4}$$

which for a 2x2 network is 2.4 words/cycle, for a 3x3 is 7.2 words/cycle and for a 4x4 is 12 words/cycle.

The theoretical results have been verified with measurements for a 2x2 and 3x3 network through C tests running on Patmos. The network was integrated using an OCP interface between the processor and the NI. For a 2x2 and 3x3 networks the measured write- and read minimum and maximum latencies are shown in Table III and IV. Using this setup, a write bandwidth of 2.396 words/cycle has been reached, where the discrepancy between the theoretical maximum bandwidth of 2.4 words/cycle and measured bandwidth is a result of the overhead related to the cores needing to align their write instructions with the schedule, as well as a slight measurement overhead.

As with the best-case scenario for writes, the best-case bandwidth for read requests is achieved if read-requests are transmitted in accordance to the schedule - ie. when a read-request response is received, the next read should be issued to the core which matches up to the timeslot that follows from receiving the response. As core stall time due to read requests varies with the schedule, an expression describing best-case

TABLE III
MEASURED MIN AND MAX MEMORY ACCESS CLOCK CYCLES IN A 2X2 NOC

	Min:	Max:
Core 0 → Core 0		
Write:	1	1
Read:	1	1
Core 0 → Core {1,2}		
Write:	1	5
Read:	6	10
Core 0 → Core 3		
Write:	1	5
Read:	7	11

TABLE IV
MEASURED MIN AND MAX MEMORY ACCESS CLOCK CYCLES IN A 3X3 NOC

	Min:	Max:
Core 0 → Core 0		
Write:	1	1
Read:	1	1
Core 0 → Core {1,2,3,6}		
Write:	1	10
Read:	6	15
Core 0 → Core {4,5,7,8}		
Write:	1	10
Read:	7	16

TABLE V
MEASURED NETWORK BANDWIDTH FOR READING

	Total bandwidth
2x2 NoC	0.788 words/cycle
3x3 NoC	0.886 words/cycle

bandwidth is non-trivial. This network is suited for real-time systems, which means that we care more about worst-case bandwidth. The worst-case read bandwidth can be expressed as:

$$bw_{rd,min} = (len_{TDMround} - 1 + 2 \cdot lrt + bc)^{-1} \quad (5)$$

With lrt = hops/cycles in longest route and bc = number of buffer cycles. A bandwidth measurement for the read network has been made, where all cores are set to continuously read from the same node. This test configuration does not resemble the best-case bandwidth, as timing of the schedule is not considered. However, this is a more realistic use case scenario due to spatial locality. This bandwidth has been measured for 2x2 and 3x3 NoCs, refer to tables III and IV. As we see, the bandwidth is not optimal, owing to the fact that the nodes must wait a substantial amount of time between receiving the response for the request and hitting the TDM slot for transmitting a new read to the target tile.

C. Source access

The entire solution is open source and freely available at `https://github.com/mortbopet/patmos-two-way-shared`
Project and related tests can be built via the makefile commands `make twoway` and `make twoway_test` in directory `patmos-two-way-shared/hardware`.

VI. FUTURE WORK

The schedule should obey the in-order constrain. As mentioned in section IV-C, the generated 4x4 schedule does not. The algorithm in [15] should be reevaluated, such that the in-order constraint is accounted for. We successfully rearranged the 4x4 schedule to comply with the constraint. The rearranged schedule can be found in the project `git` repository. As a side effect of the rearrangement the schedule is one clock cycle longer. As we can only have one occurrence of each direction in a given time slot while still complying to the in-order constraint, we will see that the schedule gets longer with the size of the NoC, compared to the schedule currently generated by the algorithm.

A perhaps more appealing solution is to alter the lookup table, used to control the multiplexer in the readback buffer system. It is possible to generate a lookup table that can support "reordering" such that the multiplexer picks out the appropriate data from the FIFO look-ahead buffer, even if the schedule does not comply with the in-order constraint. This implementation will decrease the hardware consumption, as the schedule length is shortened, and the lookup table takes the same space. This solution will avoid change of the schedule generator, and thereby avoid the longer schedules caused by the in-order constraint.

As it has been noted that our network can run at more than twice the clock speed of the Patmos processor, we would be able to run the NI and networks at double the frequency of the cores and have the OCP wrapper handle the different clock speeds and interfacing. A request would be clocked in, be processed and when ready, it would be stored and sent back to the node at the rising edge of the slow clock. This would result in a nearly halved latency for the long schedule routes and nearly a doubling of the bandwidth. It would not have any impact on the local reads and writes, as these are already handled in a single cycle.

VII. CONCLUSION

We have presented a solution for time-predictable communication for multi-core processors. The solution implements a DSM, with data transfer though a TDM based statically scheduled NoC. The design revolves around a network interface containing a dual-port memory for concurrent memory access to NI-local memory from both the core local to the NI as well as requests from external cores. The design relies on two parallel networks where the network traffic is split between write-messages and readback messages. Using inverted and time-shifted TDM schedules, it is possible to respond to read requests in a static manner, based on when the read requests are received by the NI, utilizing a minimum amount of buffering and control hardware.

The solution has been implemented and tested using the hardware construction language Chisel. Performance has been measured by integrating the system with the Patmos multi-core processor running C test programs. Resource consumption has been measured by implementing the design on an Altera DE2-115 FPGA, using Quartus Prime 16.1. Finally, we have briefly discussed further improvements of the design.

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 (http://predict.compute.dtu.dk/), contract no. 4184-00127A.

REFERENCES

[1] M. Schoeberl, "One-way shared memory," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 269–272, 2018.

[2] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki, "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)*, (Lyngby, Denmark), pp. 152–160, IEEE, May 2012.

[3] F. Brandner and M. Schoeberl, "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), pp. 61–70, November 2012.

[4] R. B. Sorensen, M. Schoeberl, and J. Sparso, "A light-weight statically scheduled network-on-chip," in *NORCHIP 2012*, 2012.

[5] M. Schoeberl, "Time-predictable computer architecture," *Eurasip Journal on Embedded Systems*, 2009.

[6] K. Goossens and A. Hansson, "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.

[7] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. T. Müller, K. Goossens, and J. Sparsø, "Argo: A real-time network-on-chip architecture with an efficient GALS implementation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, pp. 479–492, 2016.

[8] J. Sparsø, E. Kasapaki, and M. Schoeberl, "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, (San Jose, CA, USA), pp. 1044–1047, EDA Consortium, 2013.

[9] J. Mische and T. Ungerer, "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, (New York, NY, USA), pp. 63–68, ACM, 2012.

[10] S. Metzlaff, J. Mische, and T. Ungerer, "A real-time capable many-core model," in *Proceedings of 32nd IEEE Real-Time Systems Symposium: Work-in-Progress Session*, 2011.

[11] J. Mische, M. Frieb, A. Stegmeier, and T. Ungerer, "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* (J. Knoop, W. Karl, M. Schulz, K. Inoue, and T. Pionteck, eds.), (Cham), pp. 139–151, Springer International Publishing, 2017.

[12] N. Kapre and J. Gray, "Hoplite: Building austere overlay nocs for fpgas," in *25th International Conference on Field Programmable Logic and Applications (FPL 2015)*, pp. 1–8, Sept 2015.

[13] A. Olofsson, T. Nordström, and Z. ul Abdin, "Kickstarting high-performance energy-efficient manycore architectures with Epiphany," in *in Proc. Asilomar Conference on Signals, Systems and Computers* (M. B. Matthews, ed.), pp. 1719–1726, IEEE, 2014.

[14] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[15] F. Brandner and M. Schoeberl, "Static Routing in Symmetric Real-time Network-on-Chips," *Proceedings of the 20th International Conference on Real-time and Network Systems*, p. 61, 2012.

[16] Accellera Systems Initiative, "Open Core Protocol specification, release 3.0." Available at http://accellera.org/downloads/standards/ocp/, 2013.