Time-predictable distributed shared on-chip memory

Petersen, Morten B.; Riber, Anthon Vincent; Andersen, Simon T.; Schoeberl, Martin

Published in:
Microprocessors and Microsystems

Link to article, DOI:
10.1016/j.micpro.2019.102896

Publication date:
2019

Document Version
Peer reviewed version

Link back to DTU Orbit

Citation (APA):
Petersen, M. B., Riber, A. V., Andersen, S. T., & Schoeberl, M. (2019). Time-predictable distributed shared on-chip memory. *Microprocessors and Microsystems*, *71*, Article 102896. https://doi.org/10.1016/j.micpro.2019.102896

Time-predictable Distributed Shared On-Chip Memory

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 static schedules, and 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*—Network-on-chip, Distributed memory, Real-time systems**

I. Introduction

Multi-core processors for real-time applications need to have a time-predictable way of communicating. Traditionally, Multi-core processors use a single shared main memory which is accessed through arbitration. However, using a shared main memory tend to scale poorly with increasing core count, resulting in long access latencies and poor bandwidth. This paper presents a solution for time-predictable communication for multi-core processors using a network-on-chip (NoC) based distributed shared memory (DSM). The presented solution aims to keep latency low and bandwidth high, as well as ensuring time-predictability. The presented solution uses the S4NOC [1] as the NoC, introduced in section III. The S4NOC uses the schedules presented in [2]. We present a network interface (NI) which provides transparent memory access to the DSM as well as supporting both reading and writing in a time-predictable manner. To keep memory access latency low and bandwidth high, we separate traffic between two NoCs: one network for transmitting write and read request, and one for returning read data. If only one network is used, with the schedule presented in [2], the following issue may arise: NI A has to answer a read request from NI B in the same cycle as it desires to write to NI B. This is not possible, as only one package can be transmitted per schedule time slot. In such cases, if the schedule [2] used in [1] is not altered, NI A would have to buffer one of its transmissions for an additional schedule period. This may lead to accumulation of read and write requests, thus spoiling time-predictability.

This paper explores a possible solution for a time-predictable distributed shared memory with support for both reading and writing. Reading is supported by adding a separate network to split the network traffic between transmitting writes and read requests and answering read requests.
The main features of the solution are:

- The total address space is distributed across all NIs, allowing for fast memory accesses from cores to the memory 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 support 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 uses a static TDM-based scheduled NoC [3], which allows the implementation to be built as a time-predictable architecture [4], 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

One-way memory [5] presents a time-predictable DSM solution for communicating between multiple cores using writes only. As with our solution, the one-way memory project uses the S4NOC [1] and the schedule presented in [2]. The S4NOC is a statically scheduled symmetric bi-torus NoC, which assigns a schedule time slot for a channel to each of the other tiles in the network. The symmetry of the network topology allows for the same schedule to be used in all routers. The one-way memory is designed as follows: each tile contains a transmit and a receive memory. The receive memory contains an entry from each other tile in the system while the transmit memory contains an entry to each other tile in the system. The system achieves memory coherency by continuously updating the contents of the transmit memories of each tile in all other tiles. The duration between writing to a transmit memory and this change to be propagated to all other receive memories in the system is fixed and therefore time-predictable. While being a novel example of a time-predictable distributed shared memory, implementation-specific knowledge is required to use the system as a method of communication in software. Our presented solution requires no implementation specific details, presenting itself as a contiguous address space which may be freely read by all tiles.

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 compute bounds for the communication delay. Æthereal [6] is such a NoC that uses TDM, where slots are reserved to allow a block of data to pass through the NoC router without waiting or blocking traffic. 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 read and write requests by providing two NoCs: one for read and write 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 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.

HopliteRT [13] 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.

HopliteBuf [14] extends Hoplite and introduces corner buffers in the routers, similar to the PaterNoster NoC, to store packets on congestion to completely avoid deflection routing. A static analysis tool provides bounds on the buffer sizes to avoid any stalling at the buffers and to provide in-order delivery of packets. HopliteBuf uses traffic shaping similar to HopliteRT, to provide real-time guarantees and limits on the corner buffers.

All versions of Hoplite omit an 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.

As HopliteRT and HopliteBuf provide upper bounds on the message latency, we can envision building a time-predictable DSM with those NoCs. We would need to double the NoCs for write and readback and add a modified version of our NI.

Hoplite and HopliteRT NoCs can result in out-of-order delivery of packets due to the deflection routing on congestion. There is no mechanism described for reestablishing the packet order at the receiver. HopliteBuf resolves this issue. With the TDM schedule in S4NOC we avoid out-of-order packets and can therefore rely on in-order delivery of several independent transmitted flits that resemble a longer message.

The multi-core processor Epiphany [15] 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 NoC. The NoC is organized as a mesh, and favors writes over reads, as writes are posted without the processor needing 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.

Like the NoC which we present here, the Argo NoC [7] aims for time-predictability. It uses TDM-based arbitration in the routers, and it also uses TDM-based direct memory access transfers 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 router 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 [5] is a communication architecture for multi-core systems using distributed shared memory, which implements functionality for communicating between tiles in the network using writes only. 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 the above projects (except Epiphany) is the direct support for reads in the NI/NoC.

Previous work in the field of NoC based DSM generally implements a shared memory as multiple tiles connected to NIs in the network. Monchiero et al. [16] presents a DSM with all memory tiles mirroring a single address space. This solution achieves latency improvements by placing shared memories near their utilizing PEs, as well as increasing the number of memory tiles in the system to reduce contention. Yuang et al. [17] presents a DSM with shared- and semaphore memories, used for communication and synchronization. The

Time slot:	0	1	2	3	4	0	1	2
Schedule:	N	E	L			N	E	L
			N	L				N
				E	L			

Fig. 1. 2x2 schedule used by the S4NOC. Columns represent time slots. Rows represents routes in the network.

shared memory tiles present a contiguous address space, similarly to this work.

Architecturally, this work distinguished itself in the placement of shared memories. Whereas [17] and [16] present the shared memories as being distinct tiles within the network, we place our segments of the shared memory within network interfaces. This provides UMA (Uniform Memory Access) for PE-local shared memory and NUMA (Non-Uniform Memory Access) for PE-external shared memories, allowing shared memories to be utilized as low-latency scratchpad memories. Furthermore, [17], [16] do not provide hard real-time guarantees.

III. The S4NOC Network-on-Chip

We base our DSM on the S4NOC project [1] and the T-CREST multi-core platform [18]. 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.

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 in- and output ports (north, east, south, west and local) in each router. The default schedule is a all-to-all schedule where each core has a dedicated channel to each other core. With a regular structure such as a bidirectional torus and an all-to-all schedule, it is possible to find one schedule that is executed in all routers [2]. That means it is the same for all routers. So if, for example, in a given clock cycle a word is routed from west to north, then this is done in all routers.

Implementing the S4NOC on an FPGA results in a low-area hardware implementation. 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 during synthesis.

We will now present an example of a network schedule using the 2x2 network schedule from Figure 1 as an reference. Each row of the schedule is a route which a package will traverse through the network. The columns are clock cycles, which will be referred to as time slots. In the first time slot (0) the first route starts. As it can be seen from Figure 1, the first hop of the first route is in the N (north) direction. This means that *every* router passes data from the input L (local) which is connected to the NI, to the north output of the router. In the following time slot (1) the next hop in the first route is E (east). Therefore, every router passes the data from input S (south) to the east output of the router. The reason that data is routed from from input S is that data was transmitted north in the previous time slot, so it is received through the south input of the receiving router. The same logic follows for all routes. In time slot 2, every router passes data from input W (west) to output L (local), which is the network interface. The next route starts in time slot 2. As this route starts with N, every router will pass data from input L to output N, concurrently with passing from W to L. The same logic applies for the entire schedule. When the schedule has gone through its period (5 cycles for the 2x2 schedule), it repeats. This method of transmission defined by schedules leads to the constraint that a given direction can only appear once in each time slot. Otherwise, two packages would have to share the same link in a single cycle.

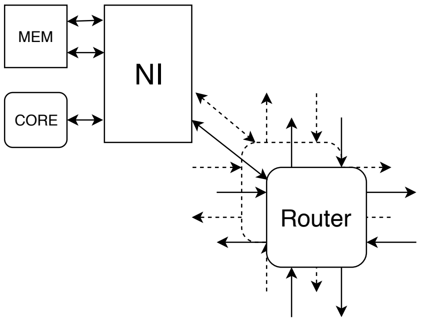


Fig. 2. Organization of a single tile. The NI is connected to a dual-port memory as well as to the routers for the two separate networks and a core.

IV. Distributed Shared On-Chip Memory

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

A. Memory Organization

As shown in Figure 2, a dual-port memory is placed inside each tile. The dual-port memory allows two concurrent write or read accesses to the memory. This provides 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. This organization may yield a substantial speedup in program execution time, compared to when using arbitrary addresses in a shared memory space, if cores are delegated to execute on memory local to their NI, which has a single cycle memory access time. Having a dual-ported memory allows for multiple writes or reads to the

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

Address space	NI
0x000 - 0x0FF	0
0x100 - 0x1FF	1
0x200 - 0x2FF	2
0x300 - 0x3FF	3

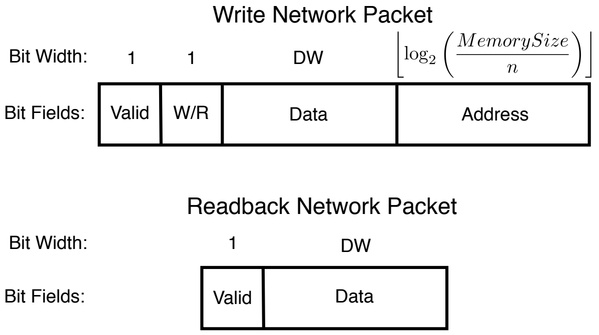


Fig. 3. Data fields of read and writeback network packets.

same address in a single cycle. 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. In a producer-consumer scenario, when a write and a read to the same address occur simultaneously, the newly written value will be read.

The memory is sized and distributed in such a way as to allow for the receiving tile to be specified as part of the address. An example of the memory distribution is as follows; Having a 2x2 NoC with a 1024-word address space (10-bit wide), the lower 8 bits address a word within the address space of a NI, while the upper 2 bits encode the tile within the network where the memory resides. Equations (1) and (IV-F) shows a generalization of the number of bits allocated to NI local address space and tile destination specification, given a size of the total distributed memory ($MemorySize$) and the number of tiles (n) in the system configuration. Table I illustrates how different memory addresses map to NIs within the distributed memory in the 2x2 case.

$$Bits_{LocalAddrSpace} = \left\lfloor \log_2\left(\frac{MemorySize}{n}\right)\right\rfloor \tag{1}$$

$$Bits_{DestinationNI} = \lceil \log_2(n) \rceil \tag{2}$$

The NI is able to determine if and when it needs to transmit a read request onto the network, based on the address provided in 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.

The focus of the presented solution is to reduce latency, and in this, worst case execution times for the system. An alternative, more space-efficient solution would be to rely on a single network and doubling the schedule length. In such a solution, the schedule would repeat itself twice, where the first half would be responsible for the transmission of writes and read requests and the second half for transmitting responses to read requests. This alternative solution would address the issue of buffering requests in [5] as described in Section I. While this alternative would be a valid solution if we focus on minimizing the footprint of the implementation, it would result in poorer worst-case execution times than the solution which we present.

As in [5], we use the TDM schedule developed in [1], [2] and the router design from the S4NOC [3]. The TDM approach ensures a worst-case execution time for read and write requests, since the schedule is statically determined.

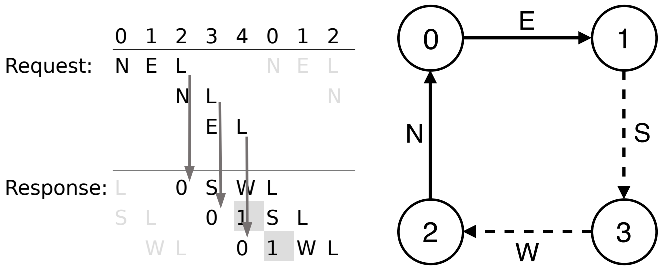
Each network will be responsible for handling separate kinds of communication:

- *Write network*: Handles transmission of memory writes from NIs to an address space external to the transmitting NI. It also 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 an 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 (OCP) [19]. After sending a request, a core is expected to stall until it receives a `valid` signal from its NI. The `valid` indicates either that (a) a *write* has been written to local memory or transmitted to the network, or (b) that the value of its *read* request is available. From a core's point of view, the interfaces for accessing 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 in which time slot the request occurs.

In the write network, the read or write address needs to be transmitted in the same packet as any data to be sent over the NoC. Figure 3 shows how the packet width of the write network varies according to: the number of tiles in the network (n), the size of the total address space ($MemorySize$) and the data width (DW). Furthermore, a `valid` bit and a bit indicating whether a transmitted message is a write or a read request are required. As explained in Section IV-A, since the most significant bits of the address determine 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.

The static nature of the readback network allows for a low area implementation. Given that a transmission onto the readback network is directly linked to when the read request



(a) (b)

Fig. 4. *Request*: Write/read request network schedule, the network responsible for writing and requesting reads to other NIs. *Response*: Readback network schedule, the network responsible for returning data from a read request.
4a illustrates the schedules of a pair of 2x2 write and readback networks. Columns represent time slots. Rows hold a route each. Grey boxes and numbers indicate buffered requests, see Figure 5.
4b illustrates the concept of schedule inversion, where the routes shown are the topmost route of the Request and Response schedules. Solid lines represent the Request route and dashed lines the Response route.

was received (and thus to which NI transmitted the request), the destination is implicit and does not need to be encoded. No address information needs to be encoded within the readback network, since the requesting core is expected to stall until a response is received from an external tile. When receiving data on the readback network, it is therefore known that that the received data is directly tied to the address to which the request was made. So the phit width of the readback network will be:

$$w_{Readback} = 1 + DW \tag{3}$$

corresponding to a `valid` bit and the data width.

C. Readback Network Schedule

With the memory technology available, accessing the on-chip dual-port memory can be done in constant time. This 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 of implicitly transmitting 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 transmit the response on the *readback* network is present. To ensure minimum memory read latency, *the readback schedule must match up with the write schedule*. Furthermore, it must be ensured that the readback schedule, like the write schedule, is a valid all-to-all schedule.

As a solution, we introduce the concept of *schedule inversion*, where the following transformation is applied to a schedule: `N→S, W→E, E→W, S→N`. This implies that responses to read requests will be routed to the transmitter of the read request backwards along the same route by which the request was transmitted. Referring to Figure 4b, it is seen that a read request sent from tile 2 to tile 1 is routed back from tile 1 to tile 2. Given that an inverted schedule maintains the relationship between routes, ie. that inverted routes do not interfere with other inverted routes, it is thus ensured that if a schedule generated for the write network with [20] is valid, this validity also applies to an inverted schedule.

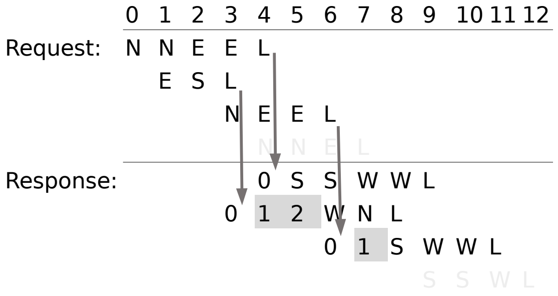


Fig. 5. Section of a 4x4 schedule taken from [20]. Shows that a read request can be received out of order with the corresponding transmissions onto the readback network, using a lookahead shift register. Read data buffering is explained in Section IV-E). Numbers indicate where read data is present within the shift register. Grey boxes indicate that the data is in a register within the shift register

D. Readback Network Time Shifting

With an inverted schedule generated for the *readback* network, Figures 4a and 5 show how the *readback* network schedule is shifted in relation to the *write* network schedule. The exact magnitude of the shift is specified by the length of the first schedule route in the *write* schedule, which will always be the longest route. By shifting the second network by the length of the first schedule route, it is seen that a received read request from the first route may be responded to immediately, ensuring minimum memory access latency. As seen in Figure 4a, 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 which the router for the *readback* network uses to index into its schedule table. This solution is general, so the same logic follows for 3x3 and larger schedules. With this time shifting, a precise match is made between the reception of a *readback* request and the transmission of the response. This is indicated by the arrows in Figure 4.

E. Request Buffering

As indicated by the grey boxes in Figure 4, some requests need to be buffered before they can be transmitted to the *readback* network. This is indicated by the second arrow in Figure 4, where the reception of the request from the transmitted core arrives one cycle before the transmission needs to occur. To accommodate this, a lookahead shift register is implemented which has the read response as input and has the same number of registers as there are leading grey boxes in the schedule. For the 2x2 and 3x3 networks there is only one leading grey box (one extra delay cycle between reception and transmission) and therefore only one register. A 4x4 network would require two registers, as there are two time slots where buffering is required.

The buffer shifts all registers every cycle, regardless of whether a new response have been read from the memory or not. A multiplexer is used to choose which response in the

TABLE II
RESOURCE CONSUMPTION OF THE DSM.

Component	LUTs	Registers
2x2 NI	190	97
2x2 Write network	1165	504
2x2 Read back network	535	396
2x2 Total	2460	1288
3x3 NI	251	102
3x3 Write network	2533	1800
3x3 Read back network	2389	1485
3x3 Total	7181	4203
4x4 NI	383	173
4x4 Write network	5798	3200
4x4 Read back network	5834	2640
4x4 Total	17760	8608
PaterNoster node	8030	3546
OpenSoC router	3752	1551
3 × 3 Argo NoC	15177	8342

shift register should be transmitted onto the readback network in a given schedule cycle. A table generated at compile time contains the multiplexer selection for every time slot. Each entry contains an index indicating where data should be read from the shift register and transmitted onto the readback network. This table is indexed with the shifted TDM cycle counter, whose output is used as the selector signal for the multiplexer.

F. Generating Tables for Control Circuitry

We have built our DSM around the NoC developed for the S4NOC [1], which uses the schedule generation algorithm presented in [20]. Key to our design is the ability to generate control circuitry for determining when to transmit packages onto the network. For each tile in the network, we are able to generate a lookup table by analyzing the static schedule in conjunction with a tiles position in the network in relation to all other tiles. This lookup table is indexed with the destination address, see eq. , and returns the timeslot where the route to the target tile starts. The technique of generating control circuitry by analyzing a tiles position in the network in relation to the static schedule, has also been used for implementing request buffering, described in section IV-E.

V. RESULTS

The network uses the OCP interface for easy integration with multi-core platforms such as Patmos or other processors. Using the T-CREST platform [18], the DSM has been attached to Patmos processors, synthesized, and implemented 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, latencies of reads and writes have been measured. From these, tests have been written for measuring the bandwidth of the DSM. Implementations larger than 3x3 configurations have been tested using simulations that mimic the processors. These simulations tested that all tiles can read and write to all other tiles in all possible time slots. The cycle accurate simulations have been used to determine the number of cycles to read and write to all other tiles in all possible time slots. These simulations have been successfully verified for networks up to an 8x8 configuration, while configurations up to 10x10 have been synthesized.

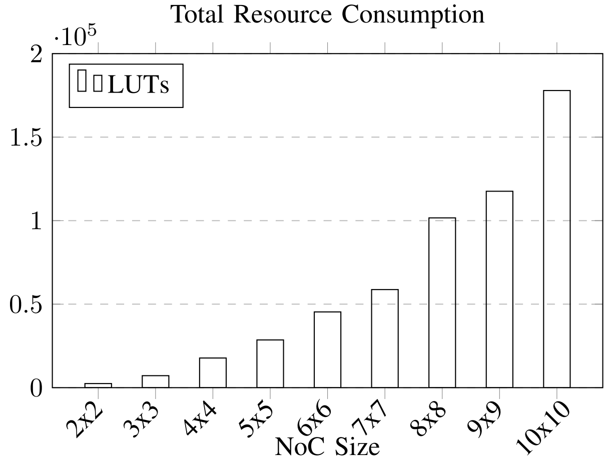


Fig. 6. Resource consumption for various network sizes with a 10 bit memory space.

A. Resource Consumption

Table II shows the resource consumption for the 2x2, 3x3, and 4x4 networks. Figure 6 shows the consumption for synthesized networks up to a 10x10 configuration. 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 shown in the table is for a single NI and should thus be multiplied by the size of the network to get the total consumption. The consumption shown excludes the actual memory, as this depends 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 for the routes increase. The readback network is always smaller than the write network in the number of registers, as it carries only the read data and no address. In comparison, a single Patmos core uses approximately 8000 LUTs and 4200 registers. Therefore, an entire 3x3 network requires the same amount of resources as a single Patmos core.

The maximum frequency of the network has 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 synthesizing with a 3x3 Patmos setup the maximum frequency is 78 MHz, where the critical path is within the Patmos cores. We therefore conclude that the NoC design with a single pipeline register per hop is a valid design point for FPGAs.

The last section of Table II shows other NoC components and a complete Argo NoC to compare with our DSM NoC. Similarly to S4NOC and Argo, the PaterNoster NoC and the OpenSoC NoC [21] are available in open-source, which allows us to synthesize those NoCs for the same FPGA. From the results, we can observe that a PaterNoster node (including an NI) and an OpenSoC router consume resources in the same range as a complete 3x3 configuration of our DSM. 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.

We observe that the Argo NoC requires double the amount of hardware in a 3x3 configuration. However, Argo contains an NI that provides hardware support for message passing and DMA handling.

B. Bandwidth and Latencies

As we are targeting real-time systems the maximum latency for a read or write must be analyzable. Using the TDM based NoC this is easily achievable. The lowest latency will be to the core's local memory, which requires a single cycle. For real-time systems we need to provide bounds for the worst-case. We can analyze those bounds 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 (*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 send path, and the maximal buffering time in the NI (*buffer*), which is equal to the number of grey boxes in schedule table (see Figures 4a and 5). Therefore, we get a worst-case latency for a read request of *schedule period* - 1 + *route* × 2 + *buffer*. For the 2x2, 3x3, and 4x4 networks respectively, the *schedule period* is 5, 10, and 19 cycles, the longest route is 3, 4, and 5 cycles and the buffer time is 1, 1, and 2 cycles.

This is a substantial latency reduction compared to the default memory configuration of the T-CREST platform. In the T-CREST platform, each Patmos processor is connected to the main memory through TDM arbitration. It takes 21 cycles to deliver a 4 word burst from main memory, which yields a worst-case latency for accessing main memory for a configuration of n cores of:

$$l_{wc} = n \times 21 \tag{4}$$

Given a 2x2 or 7x7 configuration this equates to write latencies of 84 or 1029 cycles, for a 4 word burst. For comparison our DSM shows, for 2x2 or 7x7 configurations, a single word write latency of 5 or 72 clock cycles. Worst-case read and write latencies for the DSM are shown in Table V.

The maximum bandwidth that can be achieved in the network occurs when cores execute write commands. To achieve maximum bandwidth, cores must execute writes such that they target external memories in a sequence defined by the schedule. For instance, if in a 2x2 NoC the tiles and their location are denoted by the associations $\{N0 \sim NW, N1 \sim NE, N2 \sim SW, N3 \sim SE\}$, for $N0$, the optimal write sequence which follows the schedule of Figure 4 will be $(N3, N2, N1)$. If this execution constraint is satisfied, a core can transmit $n - 1$ writes in a TDM round. The maximum write bandwidth of the entire network can therefore be stated as:

$$bw_{wr,max} = \frac{n \times (n-1)}{period} \times word \tag{5}$$

which for a 2x2 network is 2.4 words/cycle, for a 3x3 7.2 words/cycle and for a 4x4 12.63 words/cycle.

The theoretical results have been verified with measurements for a 2x2 and 3x3 network based on C tests running on Patmos on an FPGA. The network was integrated using an OCP interface between the processor and the NI. For a 2x2 and 3x3 network, the measured minimum and maximum latencies for writes and reads are shown in Tables III and IV.

For networks larger than 3x3, up to 7x7, latency has been measured through a testbench using the actual hardware. Due to the exponential increase in testbench simulation time, latency measurements for networks larger than 7x7 (up to 10x10) are model based. Table V shows the relationship between network size and maximum measured latency. This indicates that the networks tend to scale linearly with the number of tiles, which is a desirable feature. Furthermore, it is seen that the difference in read and write latencies is primarily dominated by the period length and that the relative difference between a write and a read becomes lower for larger networks. It is true for all sizes that the buffer is not used for the maximum read schedule, as the readback network has been matched for the longest latency. The Route values of Table V are for a single direction and so the worst case read values are all equal to the Period + 2 times the Route, while a write is only a single period, as this is the longest time the processor may wait for sending a packet. As seen earlier, the shared memory of the T-CREST platform, which uses a statically scheduled TDM arbiter for access, has a maximum write latency of $n \times 21$ cycles. Therefore, the DSM shows improved worst-case timings for memory access compared to the T-CREST shared main memory. For larger networks, congestion becomes an issue when using a shared main memory and as such may require alterations to keep the critical path low. This is much less of an issue when using the DSM. 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 with the schedule, ie., when a read-request response is received, the next read should be issued to the core which matches up to the time slot that follows from receiving the response. As core stall time due to read requests varies with the schedule, an expression describing best-case bandwidth is not trivial. However, this network is designed for real-time systems, which means that we care more about worst-case latency. The worst-case read latency

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
SCHEDULE PROPERTIES FOR INCREASING NOC SIZES, DENOTED IN CLOCK CYCLES. wc = WORST-CASE LATENCY

NoC size	Schedule period	Longest route	$Write_{wc}$	$Read_{wc}$
2x2	5	3	5	11
3x3	10	3	10	16
4x4	19	5	19	29
5x5	27	5	27	37
6x6	42	7	42	56
7x7	58	7	58	72
8x8	87	9	87	105
9x9	113	9	113	131
10x10	157	11	157	179

can be expressed as:

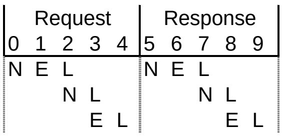
$$l_{rd,wc} = len_{TDMround} - 1 + 2 \cdot lrt + bc \tag{6}$$

With lrt = hops/cycles in the 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 tile. 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, see Tables III and IV.

As we see, the bandwidth is not optimal, owing to the fact that the NIs 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.

TABLE VI
MEASURED NETWORK BANDWIDTH FOR READING

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



Request					Response				
0	1	2	3	4	5	6	7	8	9
N	E	L			N	E	L		
		N	L				N	L	
			E	L				E	L

Fig. 7. Schedule for a 2x2 DSM using a single NoC.

C. Single NoC Comparison

As described in Section IV-B, an alternative and more area efficient solution for the problem presented is to utilize only a single network, doubling the schedule length and dedicating each half of the schedule to either reading or writing, see Figure 7. While this solution has not been implemented, we here present some considerations regarding such an alternative. We suspect that the worst-case read latency on a single-NoC DSM will be in the range of double the latency of the presented double-NoC solution, but do not present a definite answer. Such a solution may contain unexplored positive as well as negative influences on worst-case latency, i.e. possible schedule overlap and differences in buffer length. With respect to area utilization, we note that the readback network is considerably leaner than the write network, given the lack of address information within a readback packet (see Figure 3). As such, assuming that the presented work shows a theoretical halving of worst-case latency compared to a single NoC solution, a two-NoC DSM may present a substantial worst-case latency improvement without a major increase in area.

D. Source Access

The entire solution is open source and freely available at https://github.com/t-crest/patmos. The project and related tests can found in the `readme` at https://github.com/t-crest/patmos/tree/master/c/apps/twoway.

VI. Future work

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 NIs 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, would be stored and sent back to the requesting tile 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 distributed on-chip shared memory, with data transfer though a time-division multiplexed, statically scheduled network-on-chip. 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 read-back messages. Using an inverted and time-shifted schedule, it is possible to respond to read requests in a static manner, based on when the read requests are received by the network interface, 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.

Acknowledgement

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, 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.

[2] 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.

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

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

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

[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*, (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] S. Wasly, R. Pellizzoni, and N. Kapre, "Hoplitert: An efficient fpga noc for real-time applications," in *2017 International Conference on Field Programmable Technology (ICFPT)*, pp. 64–71, Dec 2017.

[14] T. Garg, S. Wasly, R. Pellizzoni, and N. Kapre, "Hoplitebuf: Fpga nocs with provably stall-free fifos," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, (New York, NY, USA), pp. 222–231, ACM, 2019.

[15] 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.

[16] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Exploration of distributed shared memory architectures for noc-based multiprocessors," in *2006 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pp. 144–151, July 2006.

[17] Zhang Yuang, Li Li, Yang Shengguang, Dong Lan, Lou Xiaoxiang, and Gao Minglun, "A scalable distributed memory architecture for network on chip," in *APCCAS 2008 - 2008 IEEE Asia Pacific Conference on Circuits and Systems*, pp. 1260–1263, Nov 2008.

[18] 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.

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

[20] 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.

[21] F. Fatollahi-Fard, D. Donofrio, G. Michelogiannakis, and J. Shalf, "Opensoc fabric: On-chip network generator," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 194–203, April 2016.