Customizing Reconfigurable On-Chip Crossbar Scheduler

Jae Young Hur[1], Todor Stefanov[2], Stephan Wong[1], and Stamatis Vassiliadis[1]
[1] Computer Engineering Lab., TU Delft, The Netherlands
http://ce.et.tudelft.nl
[2] Leiden Embedded Research Center, Leiden University, The Netherlands
http://www.liacs.nl

Abstract

We present a design of a customized crossbar scheduler for on-chip networks. The proposed scheduler arbitrates on-demand interconnects, where physical topologies are identical to logical topologies for given applications. Considering conventional fully parallel and sequential schedulers as reference designs, a comparative performance analysis is conducted. The hardware scheduler module is implemented with parameterized arbiter arrays. Experiments with practical applications show that the crossbar network with our custom scheduler realizes on-demand traffic patterns, occupies on average 52% less area, and maintains higher performance, compared to the crossbar network with a fully parallel scheduler. Additionally, our custom scheduler performs significantly better than the sequential scheduler with moderate area overheads for small-sized tokens communicated over large networks.

1 Introduction

It is a well-known fact that a crossbar network provides high performance, minimum network latency, and minimum network congestion. The non-blocking dedicated nature of communication and the relatively simple implementation makes the crossbar popular as an internet switch [1]. A typical crossbar consists of a scheduler and a switch fabric. A commercial crossbar typically accommodates an arbiter per port and each arbiter concurrently schedules the incoming packets [2]. The scheduler plays a key role in achieving high network performance and becomes more important as the size of the network increases. In the fully parallel schedulers, all-to-all connections are required to be accommodated since traffic patterns are in most cases unknown. Nevertheless, a major bottleneck of the fully parallel scheduler is the high cost due to the increasing amount of wires as the number of ports grows. Figure 1 depicts the area of the *i*SLIP crossbar scheduler [2], which is widely used for the commercial crossbar switches. As the number of ports increases, the area of the scheduler increases in an unscalable manner. This is mainly due to the all-to-all topology of the interconnects inside the scheduler module. In addition, the crossbar scheduler is an important basic building block for modern networks-on-chip (NoC) [3]. The scheduler in an on-chip router accommodates all-to-all connections. In many cases, the schedulers for NoCs are sequential. In other words, a single arbiter serves only a single port at a time. Consequently, performance degradation is the result especially for larger crossbars. This work alleviates these scalability problems by utilizing on-demand topologies in a NoC-based reconfigurable platform. This work is motivated by observations that communication patterns of different applications represent different logical topologies. In modern NoC platforms, the logical topology information can be derived from the parallel application specification. The applications in most cases require only a small portion of all-to-all communications. Figure 1(2) depicts realistic applications indicating that the required topologies are application-specific and much simpler than all-to-all topologies. Moreover, a single application can be specified differently as observed in the MJPEG specifications.

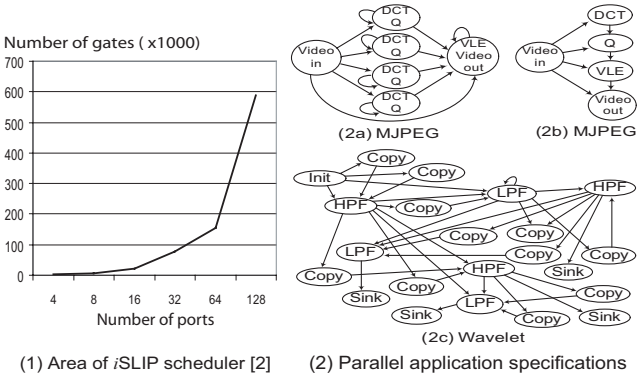

Figure 1. Motivational examples.

In this paper, we present a systematic design, an analysis, and an implementation of a novel application-specific crossbar scheduler. Our scheduler arbitrates only the necessary interconnects instead of all-to-all interconnects. The pre-

sented scheduler combines the high performance of a fully parallel scheduler and the reduced area of customized interconnects. The main contributions of this work are:

- We propose a custom scheduling scheme, where the physical topologies are identical to the logical topologies for an application.
- We perform a comparative queueing analysis for different scheduling schemes.
- Experiments on realistic applications show that the central crossbar network with our scheduler performs better and occupies 52% less area, compared to the crossbar network with a fully parallel scheduler.
- Experiments show that our scheduler performs significantly better with moderate area overheads, compared to a sequential scheduler.

The organization of this paper is as follows. In Section 2, related work is presented. Scheduler designs and the performance analysis are described in Sections 3. In Section 4, the hardware implementation and results are presented. Finally, conclusions are drawn in Section 5.

2 Related Work

Our work is based on the general approach for on-demand reconfigurable networks [4]. In this paper, we present a custom crossbar scheduler utilizing on-demand reconfigurable interconnects. Numerous NoCs targeting ASICs (surveyed in [3]) employ rigid underlying physical networks. Typically, packet routers constitute tiled NoC architectures and each packet router accommodates a crossbar switch fabric and a scheduler for internally all-to-all physical interconnects. Our scheduler is different from the schedulers in ASIC-targeted NoC routers, since our network topology is reconfigurable on demand and our scheduler utilizes the reconfigurability. NoCs targeting FPGAs (for example, [5][6][7]) employ fixed topologies defined at design-time. The topology is defined by the interconnections between routers and the crossbar inside the router also accommodates internally all-to-all physical interconnects. Our approach is different from these NoCs, since our centralized scheduler accommodates on-demand interconnects. The scheduler in [6] accommodates an arbiter per port, which is similar to our approach. In [6], single 2D-mesh packet router for an 8-bit flit occupies 352 slices and 10 block memories (BRAMs) in a Virtex-II Pro (xc2vp30) device. Our work is close to [7], in which a topology adaptive parameterized network component is presented. While the crossbar interconnects inside a router of [6][7] are still all-to-all, the physical topology of our crossbar interconnects is identical to the logical topology of the application. Finally, our custom scheduling scheme differs from traditional traffic-specific scheduling schemes, such as a weighted round-robin, in that our scheduler does not arbitrate unnecessary interconnects.

3 Customized On-Chip Crossbar Scheduler

As mentioned earlier, our objective is to systematically design on-demand reconfigurable crossbar scheduler in order to reduce the area compared to a fully parallel scheduler and increase the performance compared to a sequential scheduler. A crossbar scheduler is also required to dynamically generate the control signals to configure the switch fabrics. Our system is based on the Kahn Process Network (KPN) model of computation, where a KPN is a network of concurrent processes that communicate over unbounded FIFO channels and synchronize by a blocking read on an empty FIFO. However, the presented design techniques also can be utilized in other systems.

3.1 Reference Scheduling Schemes

We consider a conventional sequential scheduler (SQS) and a fully parallel scheduler (FPS) as references to compare our custom scheduler with. Figure 2 depicts the SQS, FPS, and the proposed custom scheduler for the application in Figure 1(2a). Figure 2(1) depicts a topology after a port-mapping and a corresponding system model. In our system, the crossbar network transfers the *requests* (from processors) and *data* (from FIFOs). Figure 2(2) depicts possible request patterns, where 4 processors request to 4 FIFOs as an example. For the sake of simplicity, the data is assumed to be requested in the first cycle. The arbiter is also assumed to perform a circular round-robin arbitration in the order of P1,P2,P3, and so on. After arbitration, a link between a processor and a FIFO port is established using a handshaking protocol, which is assumed to take 2 cycles. The bold lines represent actual data transmission, which is assumed to take 10 cycles. Figure 2(3) depicts the behavior of a typical SQS, where one FIFO port is arbitrated at a time by a single scheduler. A request is served after a request in the previous port index is arbitrated and/or the link is established. Subsequently, 24 cycles are required in total, as depicted in Figure 2(3). Figure 2(4) depicts our implemented FPS, where homogeneous arbiters are located in each port. Each arbiter checks for all ports whether there is a request or not. Consequently, all-to-all interconnects are established and 19 cycles are required in total, as depicted in Figure 2(4). Our FPS implementation is similar to *i*SLIP scheduler [2], in that circular round-robin pointer is updated when the request is granted. The round-robin pointer indicates the currently served processor port. The *i*SLIP scheduler is designed for the input-queued packet switch. However, our FPS differs from the *i*SLIP scheduler in the following ways. First, our FPS has been implemented for on-chip multiprocessor systems with distributed memories. Second, while the *i*SLIP scheduler [2] requires two stages of arbiter arrays, our FPS requires a single stage of arbiter arrays. As Figure 2(4) depicts, FPS performs better than SQS, since the concurrent requests can be served in parallel.

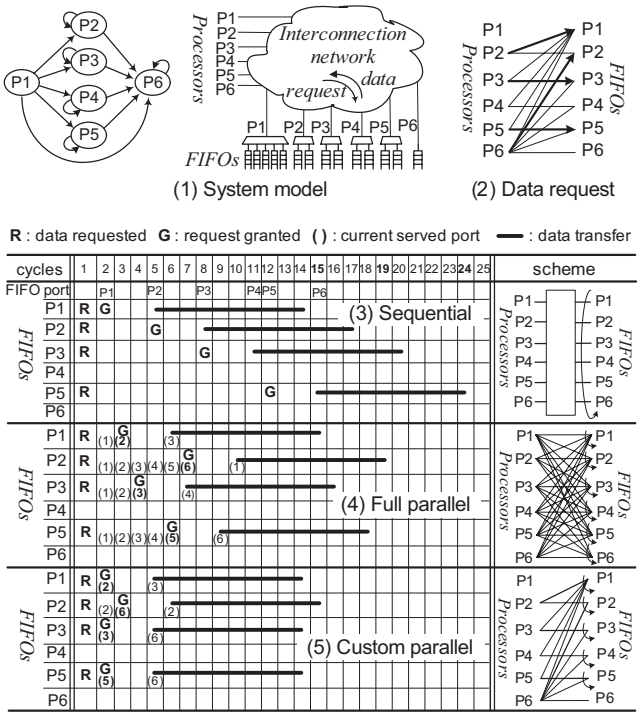

Figure 2. Different scheduling schemes.

3.2 Proposed Custom Scheduler

Our custom parallel scheduler (CPS) scheme is similar to the FPS in that the scheduler consists of arbiter arrays. In our CPS, however, the round-robin pointer update operation is performed only for on-demand interconnects. We exploit the fact that the application is specified by a point-to-point graph, in which each node has possibly a different number of connected links. As a design technique, each arbiter is parameterized with respect to the logical topology. Application-specific and differently sized arbiters ensure that the topology of the physical interconnects is identical to the logical topology specified by the application partitioning. Given the logical topologies from the application specifications, our CPS operates as follows:

1. *Request*: A processor issues a request, by designating the target FIFO port and FIFO index.
2. *Validate*: If there is a request in the round-robin pointer and the target FIFO port is idle, the request is validated.
3. *Establish*: The target FIFO status is checked. If the target FIFO is not empty, the request is accepted and the channel is established. The round-robin pointer is updated to the one that appears next in a round-robin schedule, where the round-robin schedule is determined by *the topology of an application.*

If the pointed request is a *Clear_Request*, the channel is cleared. If there is no request, the round-robin pointer is incremented. Figure 2(5) depicts the scenario of the CPS. Each arbiter checks if there is a request for required links. As an example, P2 has two probable requests in total, from P2 and P6. Therefore, the CPS arbiter at P2 searches for only two links. Note that an FPS arbiter at each port searches for 6 links. As Figure 2(5) indicates, only 15 cycles are required. In general, CPS performs better than FPS, since the request search space of CPS is a subset of the full search space of the FPS. Moreover, area reduction also can be expected, since on-demand links are physically established. Additionally, CPS performs significantly better than SQS, since the arbitration is performed in parallel. In many cases, CPS occupies more area than SQS. The area overhead issue is discussed in Section 4.

3.3 Performance Analysis

We have formulated a network delay model to compare the relative performance. Our analysis is based on the queuing model [11], since the queuing model provides a reasonable fit to the reality with relatively simple formulation. Based on the general queuing model, the following assumptions were made. First, our system network conforms to the Jackson model [11]. Each queue behaves as an independent single server and the total network latency can be modeled as the combination of each service latency. Second, each server is analyzed by an (M/M/1) queuing model. In other words, the incoming traffic obeys the Poisson distribution. The data arrivals occur randomly and are independent from one another. Additionally, the service time distribution is exponential. Third, if the server is idle, a data in the queue is served immediately. The queue size is adequately large to avoid the stall of the data flow. The Jackson's open queueing model is based on the network of queues [11] and can be suitably applied to our system as explained in the following facts. First, our KPN model and actual system are indeed a network of queues. Second, the incoming data stream pattern is statistically random. Third, a token in the FIFO is independently served by a single scheduler (or server) at each crossbar port. In this work, a *token* refers to a set of data words, which is a primitive communication unit. Consequently, the general network latency can be modeled as:

$$T_{network} = \frac{1}{\lambda} \sum_{i=1}^{M} \frac{\lambda_i}{\mu_i - \lambda_i}, \quad (1)$$

where $T_{network}$ is the total latency of the crossbar system network. M is the number of queueing systems. λ is the total incoming arrival token rate to the network (or outgoing rate from the network). λ_i is the incoming arrival rate to the i^{th} queue. μ_i is the service rate of the arbiter in the i^{th} queue.

$$\mu_i = (T_{\text{arbit}} + T_{\text{transmit}})^{-1}, \quad (2)$$

where T_{arbit} is the round-robin arbitration latency to establish a link. T_{transmit} is the actual data transmission la-

tency after the link is established. T_{transmit} can be derived as $\frac{Num\_Word}{Clk_{sys}}$, where Num_Word refers to the number of data words, or the token size. Clk_{sys} refers to the system clock frequency. We can fairly compare different scheduling schemes, since the arbitration latencies are only different. T_{arbit} for different schedulers can be approximated as follows:

$$T_{arbit\_SQS} = k_1 \left(\lfloor \frac{\#\text{ports}}{2} \rfloor \times T_{hand}\right) / (Clk_{sys}) \quad (3a)$$

$$T_{arbit\_FPS} = k_2 \left(\lfloor \frac{\#\text{ports}}{2} \rfloor + T_{hand}\right) / (Clk_{sys}) \quad (3b)$$

$$T_{arbit\_CPS} = k_3 \left(\lfloor \frac{\#\text{links}}{2} \rfloor + T_{hand}\right) / (Clk_{sys}), \quad (3c)$$

where T_{arbit_SQS} refers to the arbitration latency (in seconds) for SQS. k_1, k_2, k_3 are the scaling factors to calibrate the hardware implementation. The request check latency is modeled by $\lfloor \frac{\#\text{ports}}{2} \rfloor$ cycles. We divide by 2, since the circular round-robin pointer is statistically located in the middle of the search space. In the SQS, there is only one arbiter in the system. Only after the requested link is established using the handshaking protocol for the currently served port, the next port is served. Therefore, we model these sequential operations by multiplying the handshaking latency T_{hand} by $\lfloor \frac{\#\text{ports}}{2} \rfloor$. T_{arbit_FPS} refers to the arbitration latency for each port in the FPS. Since multiple requests can be concurrently served, we model these parallel operations by adding the T_{hand}. In the FPS, the arbiter at each port obliviously checks for all ports. The request check latency is modeled by $\lfloor \frac{\#\text{ports}}{2} \rfloor$ cycles, similarly to the SQS. T_{arbit_CPS} refers to the arbitration latency for the CPS. Similarly to the FPS, we model the single server latency by adding the T_{hand}. However, the request check latency is modeled by $\lfloor \frac{\#\text{links}}{2} \rfloor$, since the actual arbitration is performed for the only required links, instead of all links. #links is equal or less than #ports. Therefore, it is obvious that T_{arbit_CPS} is less than T_{arbit_SQS} and T_{arbit_FPS}. Only if the required topology is all-to-all, then the T_{arbit_CPS} is equal to T_{arbit_FPS}.

3.4 Case Studies

As a case study, we consider the MJPEG application in Figure 1(2a). The port-mapped system model is depicted in Figure 3(1). Considering $P1$ as a streamed data source, $P1$ generates the data in a rate of λ (tokens/s). A token rate in each queue is derived from the *Yapi* profiler [8], as depicted in Figure 3(1). Figure 3(2) depicts a scheduler model for CPS and total network latencies can be derived, as depicted in Figure 3(3). The service rates for different scheduling schemes are derived as follows. We assume that the system operates at 100MHz and k_1, k_2, k_3 are 1. T_{hand} is assumed to be 2 cycles, since each of the request and the acknowledgement requires 1 cycle, respectively. $T_{transmit}$ is assumed to be 1 cycle per word. The service rate μ_s for each

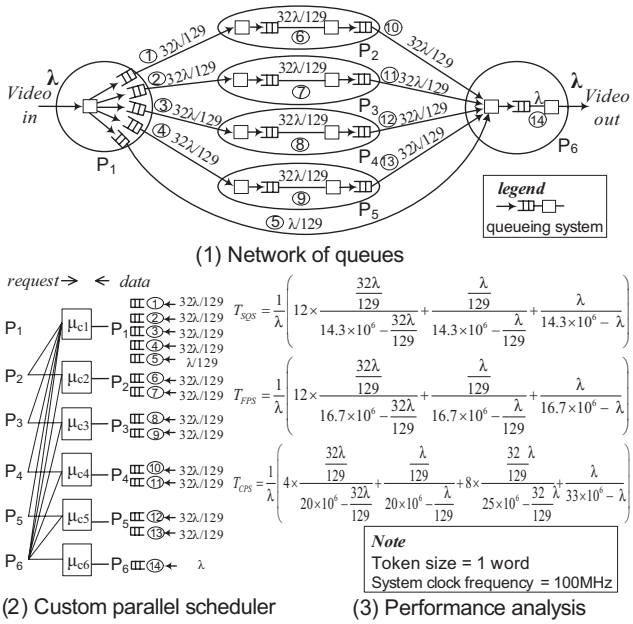

Figure 3. A case study.

port in the SQS is the same. Considering the single-word token communications, or Num_Word = 1, μ_s can be derived by $\frac{100MHz}{(3\times2+1)\text{cycles}} = 14.3\times10^6$ tokens/s from Equation (3a). Similarly, each service rate μ_p for the FPS can be derived by $\frac{100MHz}{(3+2+1)\text{cycles}} = 16.7\times10^6$ tokens/s from Equation (3b). Each service rate μ_c for the CPS is determined by the topology. μ_{c1} is $\frac{100MHz}{(2+2+1)\text{cycles}}$ = 20×10^6 tokens/s, since 5 links are established, or $\lfloor \frac{5}{2} \rfloor$ = 2. Similarly, μ_{c2}, μ_{c3}, μ_{c4}, μ_{c5} is $\frac{100MHz}{(1+2+1)\text{cycles}} = 25\times10^6$ tokens/s. Finally, μ_{c6} is $\frac{100MHz}{(0+2+1)\text{cycles}} = 33\times10^6$ tokens/s. Note that only a single link is necessary for port 6, indicating that no arbitration is necessary. As a result, the network system latencies are derived and depicted in Figure 4(1a). The performance analysis indicates that the CPS performs over 44% better than SQS and at least 34% better than the FPS for all token rate ranges. Also, the performance is better improved as the token rate increases. Moreover, our CPS is 3× better than SQS and 2× better than FPS in terms of throughput. Figure 4(1b) depicts the case study for a large token size with Num_Word=64. The CPS performs at least 5% better than SQS and 3.3% better than FPS for all ranges. The performance improvement is smaller than the case of single-word token transactions, since $T_{transmit}$ is a dominant factor for the network latency, compared to T_{arbit}.

Similarly, the network latencies for the 22-node Wavelet application, depicted in Figure 1(2), are derived. As the number of crossbar ports increases, T_{arbit} for SQS and FPS proportionally increases. However, T_{arbit} for CPS does not increase, since the average number of ports for the round-robin pointer is 1.6. In other words, on average 1.6 ports are only required to be arbitrated by an arbiter, instead of

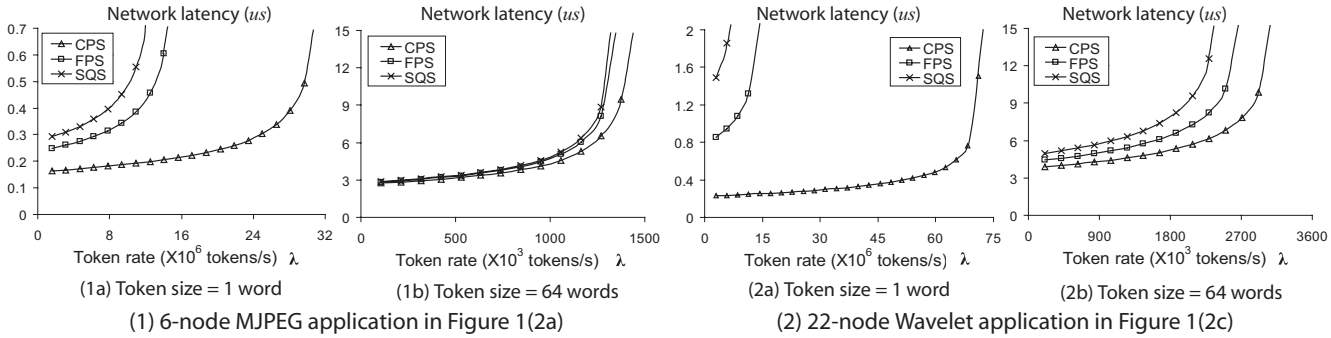

Figure 4. Network performance.

22. Figure 4(2a) depicts the network latency for single-word token transactions. The network latency is reduced by at least 84% compared to SQS and at least 73% compared to FPS. Our CPS also provides significantly higher throughput. Figure 4(2b) depicts the network latency for 64-word token transactions. CPS performs at least 22% better than SQS and 13% better than FPS. It can be suggested that our CPS scheme is more beneficial for small sized tokens communicated over large networks.

4 Implementation and Results

The aforementioned scheduler modules have been implemented in VHDL to integrate the presented network components in the ESPAM design environment [10]. The CPS module is implemented with parameterized arbiter arrays. The scheduler module is generic in terms of data width, number of ports, and logical topologies. The arbiter is implemented with a three-state finite state machine, as described in Section 3.2. The switch module in [9] is used as a common interconnects fabric and the communication controller in [10] is used as a common network interface. The functionality of the network is verified by VHDL simulations. From the implementation, $k_1 = k_2 = k_3 = 1$ have been obtained for Equation (3). The crossbar network with our scheduler has been implemented with the following specific steps. First, the schedule information is extracted from the application specifications. Figure 5(1) depicts how the schedule information is extracted for the MJPEG application in Figure 1(2a). Each FIFO port has possibly different set of request links, as depicted in Figure 5(1a). The schedule table in Figure 5(1b) shows the number of links and a list of ports from which the links are directed. As an example, the round-robin pointer in the arbiter A2 points to either P2 or P6, as depicted in Figure 5(1c), indicating that the data in FIFOs connected to P2 is transferred to either processor P2 or processor P6. The schedule table for the round-robin pointer is identical to the topology information. Second, given the schedule table, the arbiter generates two control signals, namely $CTRL\_PROC$ and $CTRL\_FIFO$. Figure 5(2) depicts that P6 reads from a remote memory in P2, as represented by the bold line. $CTRL\_PROC$ and $CTRL\_FIFO$ arbitrates the *requests* and *data transfers*, respectively. In case there is a request, the request is registered. The registered 32-bit request signal contains a target port and a target FIFO index. If the target port is idle and the designated FIFO contains data, $CTRL\_PROC$ signals are generated. $CTRL\_FIFO$ signal is generated by simply swapping the $CTRL\_PROC$. Third, control signals dynamically configure the switch fabrics. There are two types of multiplexors, namely processor-side multiplexors and FIFO-side multiplexors. Processor-side multiplexors are controlled by $CTRL\_FIFO$ signals and the FIFO-side multiplexors are controlled by $CTRL\_PROC$ signals. Once a link is established, a remote memory behaves as a local memory until the link is cleared.

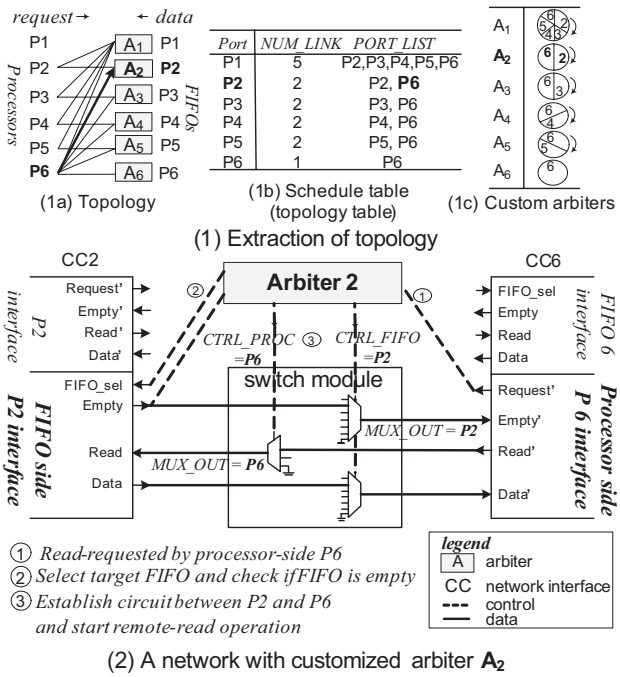

Figure 5. A customized crossbar network.

The implemented scheduler modules are integrated in the centralized crossbar network and are compared in terms of area utilization. We experimented with different task graph topologies of realistic applications, where our network provided the on-demand topologies. The application task graphs of MPEG4, PIP, MWD were taken from [12]. The task graphs of H.263 encoding, MP3 encoding, and MMS were taken from [13]. The task graphs of 802.11 MAC, TCP checksum, VOPD are taken from [14],[15],[16], respectively. The numbers between braces indicate the number of nodes and the number of required links. As an example, TCP_Chk{5,14} indicates that the crossbar for the TCP checksum application requires 5 nodes and 14 links. Assuming each node is associated with a single crossbar port, the implemented networks were synthesized using the Xilinx ISE 8.2 tool targeting the Virtex-II Pro (xc2vp20-7-896) FPGA and the areas were obtained and depicted in Figure 6. The network with our CPS requires on average 52% less area compared to the network with FPS. As an example, our centralized 5-node crossbar network employing the FPS occupies 437 slices, while the area is reduced to 187 slices when the crossbar network accommodates our CPS for the topology of the MJPEG application in Figure 1(2a). The network with our CPS requires on average 17% more area compared to the network with SQS. We consider that the area overhead of our CPS over SQS is less significant, since the xc2vp20 device we target contains 9280 slices and chip-wise overhead of CPS over SQS is on average 2.5%. As an example, a crossbar system network for 16-node VOPD application occupies 716 slices for CPS and 664 slices for SQS. The area of our network is not only dependent on the number of nodes that determine its size but also on the network topology. It is observed that the higher area reduction is obtained as the network size increases. This is due to the fact that the average number of links per node is 1.8 and does not increase as the number of nodes increases.

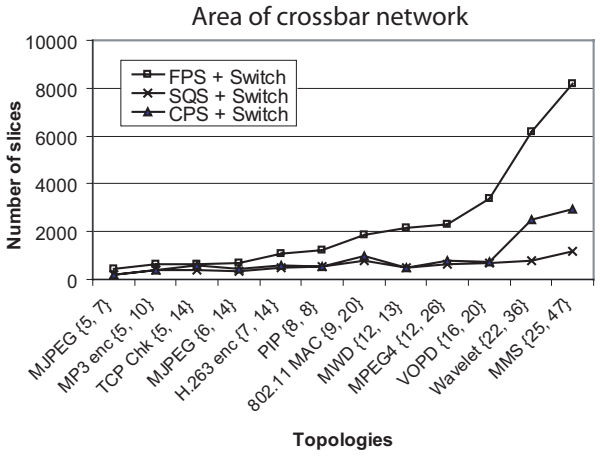

Figure 6. Experimental results.

5 Conclusions

In this paper, we presented a topologically customized crossbar scheduler designed for networks on chip. We showed that our scheduler can be implemented using parameterized arbiter arrays. By utilizing the topology as a parameter, the scheduler is adapted to given applications, without modifying the network implementation. Our customized network efficiently utilizes the bandwidth, by constructing on-demand topologies. We showed that our scheduler performs better and occupies significantly less area than conventional fully parallel schedulers. We showed that our scheduler performs significantly better for small-sized tokens communicated over large networks and occupies moderately more area than sequential schedulers.

Acknowledgement. This work was supported by the Dutch Science Foundation (STW) in the context of the Architecture, Programming, and Exploration of Networks-on-Chip based Embedded System Platforms (ARTEMISIA) project (number LES.6389).

References

[1] Cisco Systems, Inc., http://www.cisco.com.
[2] N. Mckeown, "The *i*SLIP scheduling algorithm for input-queued switches," IEEE/ACM Transaction on Networking, vol. 7, no. 2, pp. 188-201, Apr 1999.
[3] T. Bjerregaard and S. Mahadevan, "A Survey of Research and Practices of Network-on-chip," ACM Computing Surveys, vol. 38, no. 1, pp. 1–51, Mar 2006.
[4] S. Vassiliadis and I. Sourdis, "FLUX Networks: Interconnects on Demand," Proceedings of International Conference on Computer Systems Architectures Modelling and Simulation (IC-SAMOS'06), pp. 160–167, Jul 2006.
[5] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost, "HERMES: an Infrastructure for Low Area Overhead Packet-switching Netwoks on Chip," Integration, the VLSI Journal, vol. 38, no. 1, pp. 69–93, Oct 2004.
[6] B. Sethuraman, P. Bhattacharya, J. Khan, and R. Vemuri, "LiPaR: A Light-weight Parallel Router for FPGA-based Networks-on-Chip," Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI'05), pp. 452–457, Apr 2005.
[7] T.A. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins, "Topology adaptive network-on-chip design and implementation," IEE Proceedings of Computers & Digital Techniques, vol. 152, no. 4, pp. 467–472, Jul 2005.
[8] E.A. de Kock, G. Essink, W.J.M Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers, "YAPI: Application Modeling for Signal Processing Systems," Proceedings of the 37th Design Automation Conference (DAC'00), pp. 402–405, Jun 2000.
[9] J. Y. Hur, T. Stefanov, S. Wong, and S. Vassiliadis, "Systematic Customization of On-Chip Crossbar Interconnects," Proceedings of International Workshop on Applied Reconfigurable Computing (ARC'07), pp. 61–72, Mar 2007.
[10] H. Nikolov, T. Stefanov, and E. Deprettere, "Efficient Automated Synthesis, Programming, and Implementation of Multi-processor Platforms on FPGA Chips," Proceedings of 16th International Conference on Field Programmable Logic and Applications (FPL'06), pp. 323–328, Aug 2006.
[11] Rusty O. Baldwin, Nathaniel J. Davis IV, Scott F. Midkiff b, and John E. Kobza, "Queueing network analysis: concepts, terminology, and methods," The Journal of Systems and Software, vol. 66, no. 2, pp. 99–117, May 2003.
[12] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G.D. Micheli, "NoC Synthesis Flow for Customized Domain Specific Multi-processor Systems-on-Chip," IEEE Transactions on Parallel and Distributed Systems, vol. 16, no. 2, pp. 113–129, Feb 2005.
[13] J. Hu and R. Marculescu, "Energy-Aware Mapping for Tile-based NoC Architectures Under Performance Constraints," Proceedings of the 8th Asia and South Pacific Design Automation Conference (ASP-DAC'03), pp. 233–239, Jan 2003.
[14] K. Sekar, K. Lahiri, A. Raghunathan, and S. Dey, "FLEXBUS: A High-Performance System-on-Chip Communication Architecture with a Dynamically Configurable Topology," Proceedings of 42th International Conference on Design Automation Conference (DAC'05), pp. 571–574, Jun 2005.
[15] K. Lahiri, A. Raghunathan, G. Lakshminarayana and S. Dey, "Design of High-Performance System-On-Chips Using Communication Architecture Tuners," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 23, no. 5, pp. 620–636, May 2004.
[16] S. Murali and G.D. Micheli, "Bandwidth-Constrained Mapping of Cores onto NoC Architectures," Proceedings of International Conference on Design, Automation and Test in Europe (DATE'04), pp. 896–901, Feb 2004.