J Sign Process Syst
DOI 10.1007/s11265-008-0263-x

Design Trade-offs in Customized On-chip Crossbar Schedulers

Jae Young Hur · Stephan Wong · Todor Stefanov

Received: 31 October 2007 / Revised: 5 June 2008 / Accepted: 16 August 2008

Abstract In this paper, we present a design and an analysis of customized crossbar schedulers for reconfigurable on-chip crossbar networks. In order to alleviate the scalability problem in a conventional crossbar network, we propose adaptive schedulers on customized crossbar ports. Specifically, we present a scheduler with a weighted round robin arbitration scheme that takes into account the bandwidth requirements of specific applications. In addition, we propose the sharing of schedulers among multiple ports in order to reduce the implementation cost. The proposed schedulers arbitrate on-demand (at design time) interconnects and adhere to the link bandwidth requirements, where physical topologies are identical to logical topologies for given applications. Considering conventional crossbar schedulers as reference designs, a comparative performance analysis is conducted. The hardware scheduler modules are parameterized. Experiments with practical applications show that our custom schedulers occupy up to 83% less area, and maintain better performance compared to the reference schedulers.

J. Y. Hur (✉) · S. Wong · T. Stefanov
Computer Engineering Lab., TU Delft, The Netherlands
e-mail: J.Y.Hur@ewi.tudelft.nl
URL: http://ce.et.tudelft.nl

S. Wong
e-mail: J.S.S.M.Wong@tudelft.nl

T. Stefanov
Leiden Embedded Research Center, Leiden University,
Leiden, The Netherlands
e-mail: stefanov@liacs.nl
URL: http://www.liacs.nl

Keywords Interconnection architectures · Schedulers · Network topology · Reconfigurable hardware

1 Introduction

It is a well-known fact that a crossbar network provides high network performance and minimum network congestion. The non-blocking nature of communication and the relatively simple implementation makes the crossbar popular as an internet switch (Cisco Systems, Inc., http://www.cisco.com). A typical crossbar consists of a scheduler and a switch fabric. The scheduler plays a key role in achieving high network performance and becomes more important as the size of the network increases. A commercial crossbar scheduler typically accommodates an arbiter per input/output port and each arbiter concurrently arbitrates the incoming packets [1]. In these 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 [1], 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 interconnects inside the scheduler module. In addition, the crossbar scheduler is an important basic building block for modern networks-on-chip (NoC) [2]. The scheduler in an on-chip router accommodates all-to-all connections. In many cases, such schedulers contain a single central arbiter that sequentially arbitrates a single request at a time, while data transmission can be

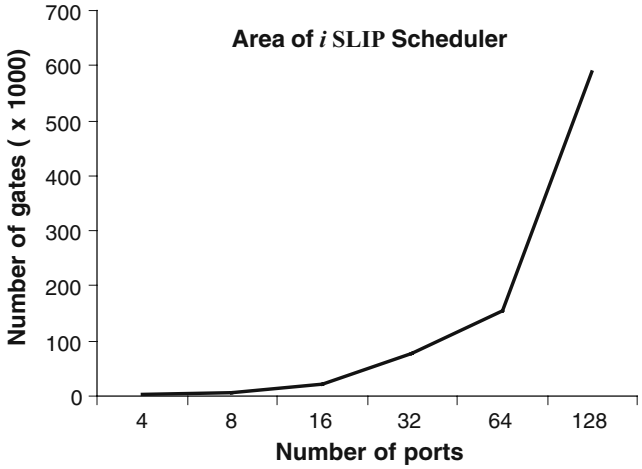


Figure 1 Area of crossbar scheduler [1].

parallel. Due to this, the arbitration latency increases as the number of crossbar ports increases. This work alleviates these problems by establishing on-demand topologies using a crossbar in a reconfigurable multiprocessor system-on-chip platform. This work is motivated by the following observations:

- The logical topology and traffic information can be derived from the parallel application specification.
- Communication patterns of different applications represent different logical topologies. Figure 2 depicts task graphs of realistic applications taken from [21–24], where numbers on links represent the traffic loads (or required bandwidth). The numbers between braces in the depicted topologies indicate the number of nodes and the number of required links. As an example, MJPEG{6,14} indicates that the MJPEG application requires 6 nodes and 14 links. As depicted in Fig. 2, logical topologies are application-specific and require only a small portion of all-to-all communications. It can be noted that we focus on communication performance. Detailed descriptions of the computation nodes in Fig. 2 are not relevant in this work.
- Single applications can be specified differently as observed in the MJPEG [Fig. 2(7) and (8)] and MPEG4 specifications [Fig. 2(1) and (5)]. Moreover, the traffic load is differently distributed for different communication links as depicted in Fig. 2 (see the numbers on the links).
- More than 70% of modern reconfigurable hardware, such as field-programmable gate arrays (FPGAs), is preoccupied by millions of wire segments. As an example, Virtex-II Pro device has nine metal layers to enhance the routing flexibility and for a designer to realize on-demand reconfigurability.

In this paper, we present a systematic design, an analysis, and an implementation of novel application-specific crossbar schedulers. Our custom schedulers arbitrate only the necessary interconnects instead of all-to-all interconnects. In addition, our weighted round-robin scheduler can be adapted to the traffic requirements known at design time. The presented

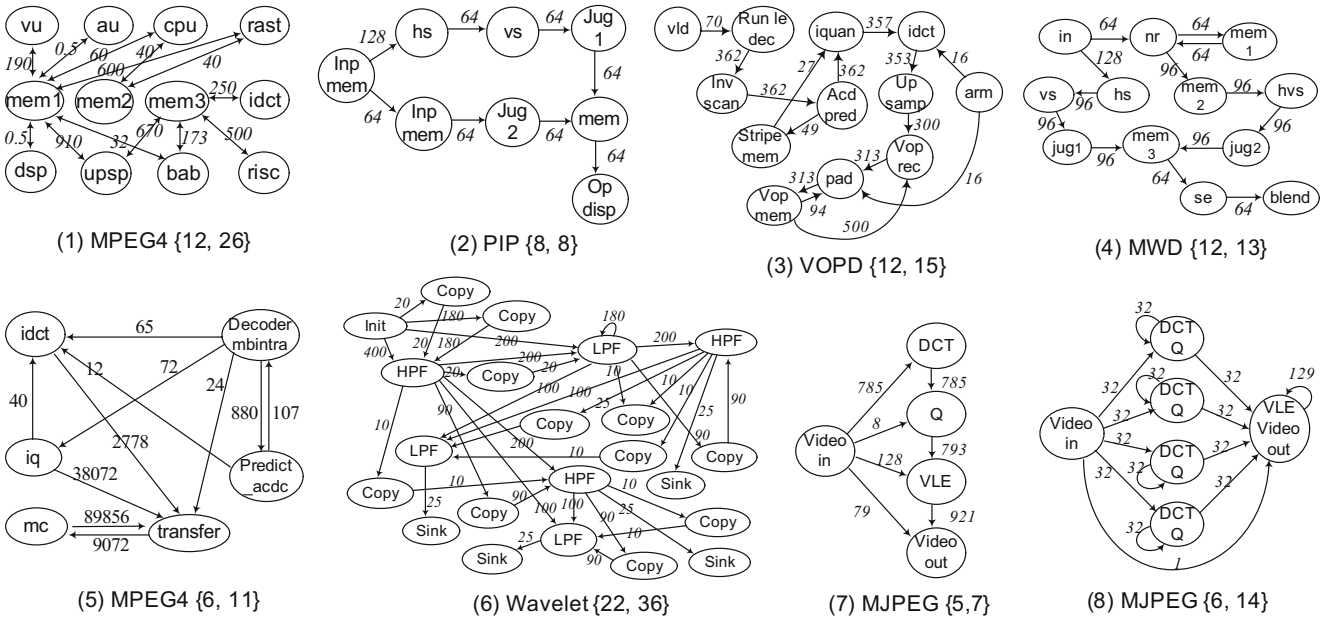


Figure 2 Logical topologies with different traffic load distribution in practical applications [21–24].

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

- We propose a weighted round robin custom parallel scheduler (WCPS). The presented scheduler is implemented with on-demand interconnects, where the physical topologies are identical to the logical topologies for an application. Experiments on realistic applications indicate that our WCPS scheme performs better than reference schedulers with moderate area overheads.
- We propose a custom parallel scheduler with shared arbitration scheme (SCPS). Experiments show that the implementation cost can be reduced with moderate performance overheads, when the number of links per port is sufficiently large.

The organization of this paper is as follows. In Section 2, related work is presented. The proposed scheduler designs are described in Section 3. In Section 4, we present the performance evaluation. The hardware implementation results are presented in Section 5. Finally, conclusions are drawn in Section 6.

2 Related Work

Our work is based on the general approach for on-demand reconfigurable networks [3, 4]. In this paper, we present a custom crossbar scheduler utilizing on-demand reconfigurable interconnects.

Numerous NoCs targeting ASICs (surveyed in [2]) employ rigid underlying physical networks. Typically, packet routers constitute tiled NoC architectures. Each packet router accommodates a crossbar switch fabric and a scheduler with internally all-to-all physical interconnects. Our scheduler is different from the schedulers in ASIC-targeted NoC routers, since on-demand topology is established in our scheduler by utilizing the reconfigurability of a reconfigurable hardware. NoCs targeting FPGAs [5–8] employ fixed topologies defined at design-time. In [5–8], packet switched networks are presented and each router contains a full crossbar fabric. The topology in this related work is defined by the interconnections between routers or switches. The scheduler in [7] accommodates an arbiter per port, which is similar to our approach. In [7], 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 [8], in which a topology adaptive parameterized network component is presented. While the crossbar interconnects inside a router of [7, 8] are still all-to-all, the physical topology of our crossbar network is identical to the on-demand topology of the application. In addition, the topology of our work is defined by the direct interconnection between processors. In other words, our custom crossbar constitutes a system network.

In [9], the crossbar network is implemented utilizing a modern partial and/or dynamic reconfiguration technology. In [9], a large-sized (928×928 bits) crossbar network utilizing native programmable interconnects and look-up tables (LUTs) is presented. However, all-to-all crossbars in [9] are not scalable, especially for large-sized networks. Our work differs from [9] in that our network is customized in order to alleviate the scalability problem. In addition, our network is implemented in a fully synthesizable VHDL. Our implemented network is technology-independent, though we target FPGAs in this work.

Recently, a couple of application-specific NoC designs were proposed. In [10], a multi-hop router network customization is presented, whereas our network is single-hop based. In [11, 12], an internal STbus crossbar customization is presented. Our work is similar to [11, 12] in that the crossbar is customized. In [13], a design method to generate a crossbar is presented, in which an application is traced (in simulation) and the graph clustering technique is used to generate the crossbar network. Our design method differs from [10–12] in that our custom crossbar is generated, where the physical topology is *identical* to the on-demand topology of an application. Our arbitration scheme can also be configured with regards to the traffic demand, where the physical topology inside the scheduler is again identical to the required topology. In [14], a design method to synthesize an application-specific bus matrix is presented. Our work is similar to [14] in that the application-specific interconnects are established between masters and slaves. Furthermore, our work is similar in that a shared arbitration method is presented. However, our design method differs in that the on-demand topology information and the traffic bandwidth information are extracted from the application specification step. On the other hand, in [14], a simulation-based traffic trace is conducted to synthesize the application-specific topology, which requires hours of design space exploration time. Finally, a performance comparison between the full crossbar and the partial bus matrix is not presented.

Finally, our crossbar network is similar to the virtual output queues (VOQ) in that the physical FIFOs are established at the input ports. The VOQ-based switch such as *i*SLIP [1] scheduler is not popular in the multi-hop NoC due to the high cost. We highlight that

our crossbar interconnects are single-hop, not multi-hop NoC. In other words, a crossbar (with single-hop latency) constitutes a system network in the multi-processor system on chip platform. Our application-specific network combines high performance of a VOQ-based crossbar and the low cost of customized interconnects. In [15], a fast round-robin crossbar scheduler is designed and implemented for high performance computer networks. In the conventional (weighted or priority-based) round-robin schedulers and [15], all-to-all interconnects are established because the traffic pattern is unknown. These conventional full crossbar schedulers accommodate the circuitry to arbitrate all possible requests from all ports. The major difference with our work is that our scheduler is application-specific. In the presented design flow, only necessary interconnects are systematically established based on the traffic patterns that an application exhibits. Moreover, the arbitration is only performed over actually established interconnects, instead of conventional all-to-all interconnects. 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. Accordingly, the cost is reduced by systematically establishing only necessary FIFOs.

3 Customized On-chip Crossbar Scheduler

As mentioned earlier, our objective is the design of customized reconfigurable crossbar schedulers in order to reduce the area compared to a fully parallel scheduler and increase the performance compared to a sequential scheduler. The same crossbar scheduler is required to dynamically generate the control signals to configure the switch fabric within a crossbar.

3.1 Design Flow

Our scheduler has been developed to be integrated as modular communication components in the ESPAM tool chain as depicted in Fig. 3. Details of the ESPAM design methodology can be found in [16–18]. In ESPAM, three input specifications are required, namely application/mapping/platform specification in XML. An application is specified as a Kahn Process Network (KPN). In this work, the KPN is considered as a programming model. A KPN is a network of concurrent processes that communicate over unbounded FIFO channels and synchronize by a blocking read on an empty FIFO. The KPN is a suitable model of

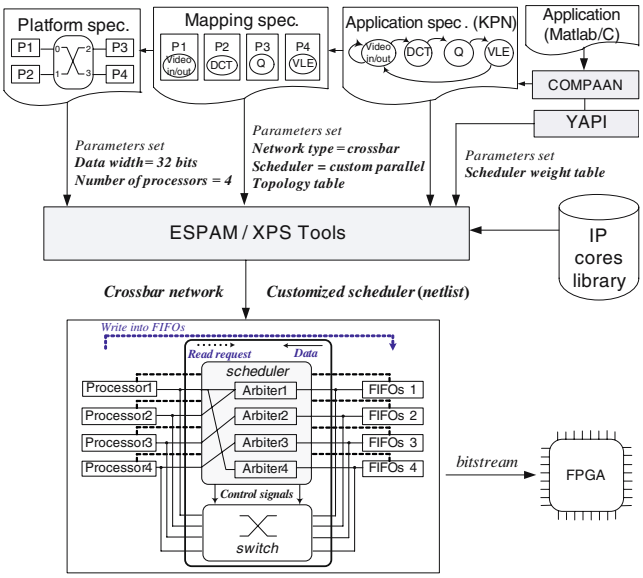


Figure 3 ESPAM design flow [16–18].

computation on top of the presented crossbar interconnection network, due to the following facts:

- The synchronization scheme is relatively simple. Processors synchronize only with the full/empty status of the hardware FIFOs. Subsequently, a parallel programmer does not need to explicitly handle the synchronization, since the synchronization is inherently supported by the hardware FIFOs.
- The system eliminates the traditional head of the line (HOL) blocking problem that all packets must wait behind the contended packet in the input queue. This is due to the fact that a physical FIFO is established for a *logical channel* in a dedicated manner.

A KPN specification is automatically generated from a sequential Matlab program using the COMPAAN tool [19]. We define the network topology in the KPN specification as the *logical topology* for an application, which consists of tasks and logical channels. Exemplary logical topologies are depicted in Fig. 2. Single task or multiple tasks are assigned to a physical processor in the mapping specification. The network type and the port mapping information are specified in the platform specification. Figure 3 depicts how custom crossbars can be implemented from a four-node MJPEG application specification. In the platform specification, four processors are port-mapped on a crossbar. From the mapping and platform specifications, port-mapped network topology is extracted as a static parameter and passed to ESPAM. We define the extracted port-mapped network topology as the *on-demand topology*

that an application requires, which consists of processors and physical links. Additionally, the traffic bandwidth information is obtained from the YAPI tool [20]. Subsequently, ESPAM refines the abstract platform model to an elaborate parameterized RTL (hardware) and C/C++ (software) models, which are inputs to the commercial Xilinx Platform Studio (XPS) tool. The XPS tool generates the on-demand netlist (depicted in Fig. 3) with regard to the parameters passed from the input specifications. Our system model is based on the *local write, remote read* scheme. The processor issues the request that designates the target port index and target FIFO index. Subsequently, the FIFO responds with a data stream to the processor. The crossbar network transfers the *requests* (from processors to FIFOs) and the *data* (from FIFOs to processors). The directions of requests (from left to right) and the direction of the data transfers (from right to left) are depicted in Fig. 3. Finally, a bitstream is generated for the FPGA prototype board to check the functionality and the performance.

3.2 Reference Scheduling Schemes

We consider a sequential scheduler (SQS) and a fully parallel scheduler (FPS) as references to compare with our custom schedulers. Figure 4 depicts the behavior of the SQS, FPS, and our custom schedulers for the MJPEG application in Fig. 2(8). Figure 4(1) depicts the logical topology (or data flow graph) with 6 nodes and 14 channels. In our model of computation, each communication channel is mapped onto the FIFOs labeled by $\mathbf{F}_1 \sim \mathbf{F}_{14}$. Figure 4(2) depicts the system model. The physical system consists of 6 nodes and 14 links (or 14 FIFOs). The ith node is connected to processor port p'_i and FIFO port p_i. For example, in the sixth node, processor $\mathbf{P}_6$ is connected to the processor port p'_6 and FIFO $\mathbf{F}_{14}$ is connected to the FIFO port p_6, as depicted in Fig. 4(2). A FIFO index corresponds to the channel index, as depicted in Fig. 4(1). The crossbar network transfers the *requests* (from processors to FIFOs) and the *data* (from FIFOs to processors). For example, a bold line in Fig. 4(2) represents that processor $\mathbf{P}_6$ sends the *request* signal to FIFO $\mathbf{F}_{11}$. Subsequently, FIFO $\mathbf{F}_{11}$ sends data to processor $\mathbf{P}_6$. Note that data in FIFO $\mathbf{F}_{11}$ is from processor $\mathbf{P}_4$. Figure 4(3) depicts the bipartite graph based on the system model. The thick and thin lines depict all possible requests according to the topology in Fig. 4(1). The thick lines represent an example request pattern. In this example, four processors request to four FIFOs. The numbers on the link represent the relative amount of traffic. For example, the link between p'_6 and p_1 is 5 times less utilized than the link between p'_4 and p_1.

Figures 4(4)∼(8) depict the cycle-by-cycle behavior of five different schedulers for the request patterns (bold links) in Fig. 4(3). In order to establish the link between the processor and the target FIFO, three steps are required. First, a processor *requests* to an arbiter. Second, the arbiter *grants* the request when the target port is idle and the round-robin pointer points to the requesting processor. Third, the request is *accepted* when the target FIFO contains data. These operations are denoted by R, G, and A. The round-robin pointer is denoted by the oval. For the sake of simplicity, the data is assumed to be requested by a processor in the first cycle. The arbiter is assumed to perform a circular (weighted) round-robin arbitration in the order of p'_1, p'_2, p'_3, and so on. After the request is granted, 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 in Fig. 4(4)∼(8) represent actual data transmission, which is assumed to take 5 cycles.

Figure 4(4) depicts the behavior of a SQS, where one port is arbitrated at a time. A crossbar contains a central arbiter that sequentially grants a single request at a time, while data transmission can be parallel. A request is served after a request in the previous port index is arbitrated. Subsequently, 48 cycles are required to serve those requests, as depicted in Fig. 4(4). Figure 4(5) depicts FPS, where homogeneous arbiters are located in each port. Unlike the SQS, multiple requests can be arbitrated in parallel. Each arbiter checks for all ports whether there is a request or not. Consequently, 40 cycles are required in total, as depicted in Fig. 4(5). Our FPS implementation is similar to the *i*SLIP scheduler [1], in that circular round-robin pointer is updated when the request is granted. Similar to *i*SLIP scheduler, all-to-all interconnects are established. The *i*SLIP scheduler is designed for the input-queued packet switch. Our FPS differs from the *i*SLIP scheduler in that the FPS has been implemented for on-chip multiprocessor systems with distributed memories. Additionally, while the *i*SLIP scheduler [1] requires two stages of arbiter arrays, our FPS requires a single stage of the arbiter array. As Fig. 4(5) depicts, FPS performs better than SQS, since concurrent requests can be served in parallel.

3.3 Round Robin Custom Scheduler

The round-robin custom parallel scheduler (CPS) scheme is presented in [21]. The CPS is similar to the FPS in that the scheduler consists of arbiter arrays. In the CPS, however, the round-robin pointer update operation is performed only for on-demand interconnects.

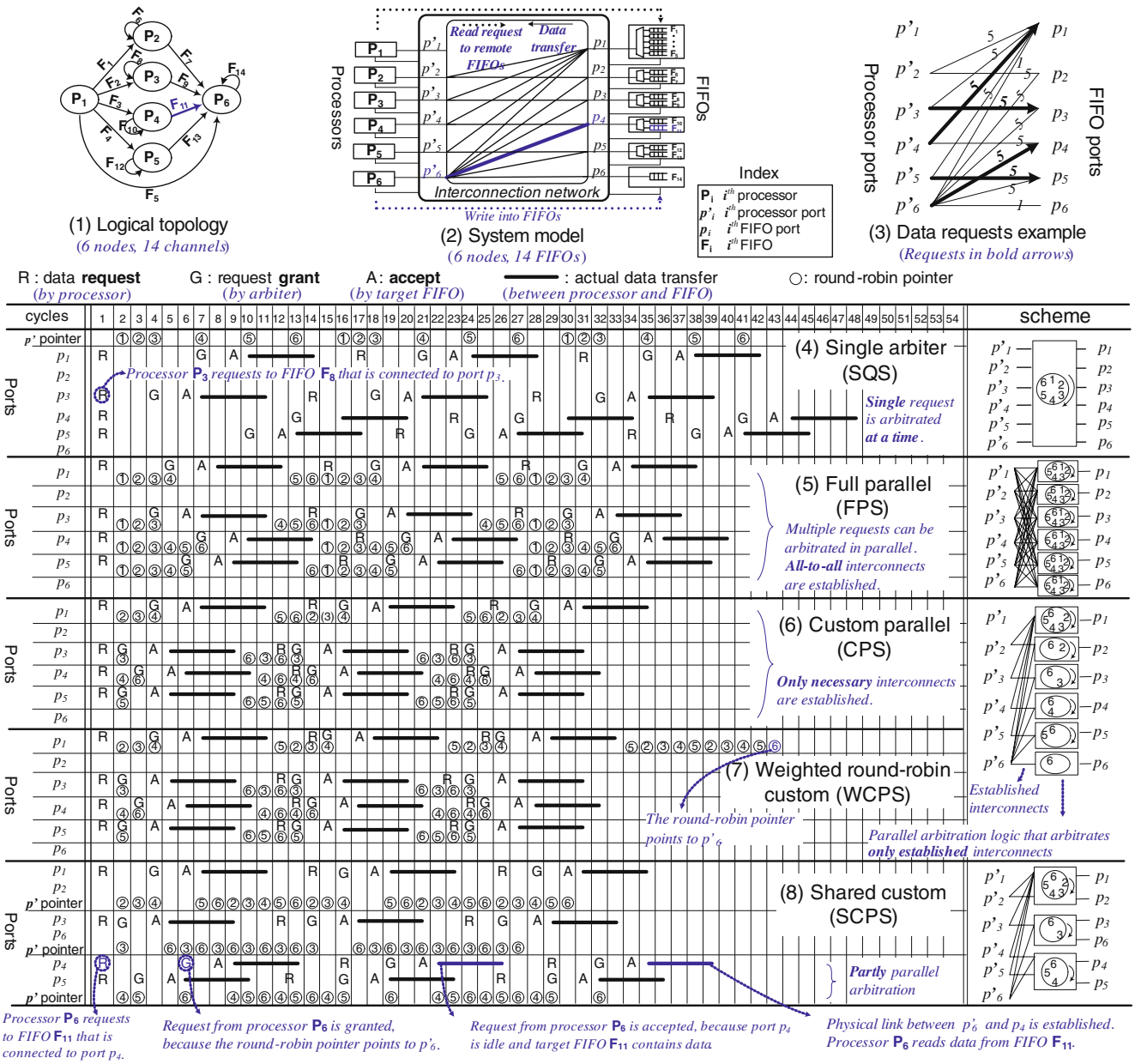


Figure 4 Different scheduling schemes.

We exploit the fact that the application is specified by a directed graph, in which each task has possibly a different number of connected channels. As a design technique, each arbiter is parameterized with regard to the on-demand logical topology. Application-specific and differently sized arbiters ensure that the topology of the physical interconnects is identical to the on-demand topology specified by the application partitioning. Given the on-demand 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. *Grant*: If there is a request in the round-robin pointer and the target FIFO port is idle, the request is validated.
3. *Accept*: The target FIFO status is checked. If the target FIFO is not empty, the request is accepted and the physical link is established. The round-robin pointer is updated to the one that appears next in the round-robin schedule, where the round-robin schedule is determined by *the on-demand topology for an application.*

If the pointed request is a *Clear_Request*, the link is cleared. If there is no request, the round-robin pointer

is incremented. Figure 4(6) depicts our example scenario for the CPS. Each arbiter checks if there is a request for required links. As an example, p_1 has five probable requests from $p'_2 \sim p'_6$. Therefore, the CPS arbiter at p_1 services five links. Similarly, p_5 has two probable requests from p'_5 and p'_6. Therefore, the CPS arbiter at p_5 services two links. For comparison, the FPS arbiter [Fig. 4(5)] services 6 links at each port. As Fig. 4(6) indicates, 35 cycles are required to serve the request patterns in our example. 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. Area reduction also can be expected, since only on-demand links are physically established. Moreover, only one link is often connected to an arbiter, indicating that the arbiter performs only handshaking operation and implementation cost can be reduced. 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 5.

3.4 Weighted Round Robin Scheduler

In the previous section, we presented a topologically customized scheduler (CPS) scheme. The CPS performs better and provides lower cost for an on-demand topology, compared to reference schedulers [21]. In the CPS scheme, an arbitration is performed in a simple round-robin fashion and the differently utilized traffic bandwidth is not considered. In this section, we propose a weighted round-robin custom scheduling scheme (WCPS) in order to adapt the scheduler to the traffic requirements.

3.4.1 Weighted Round Robin Custom Scheduler (WCPS)

Our proposed WCPS scheme is similar to the CPS in that the round-robin pointer is updated only for on-demand interconnects. Similarly, the topology of the physical interconnects is also identical to the on-demand topology specified in the ESPAM design flow. The difference between WCPS and CPS is that the pointer update operation in WCPS is performed with regard to weights. The weight refers to the relative number of utilized tokens (or packets), which corresponds to traffic bandwidth requirements. In this context, we define the weight by the *relative number of requests* for tokens utilized in a communication link. A *token* refers to a set of data words, which is our primitive communication unit. We exploit the fact that the weights can be determined by application profiling. In this work, we use the YAPI tool [20] (see Fig. 3). As a design technique, each arbiter is parameterized with respect to the on-demand topology together with a weight distribution. By using differently sized arbiters and application-specific weight information, we can increase the network performance. This is due to the fact that the scheduler checks links with a high probability of requests more frequently. The main idea of the WCPS to match physical network bandwidth to logical traffic bandwidth. Only when the traffic is uniformly distributed, our WCPS is identical to CPS. The novelty of our WCPS lies in that the arbitration is performed over only actually established on-demand interconnects. This means that no physical circuitry is established for unnecessary interconnects.

It can be noted that our scheme of assigning weights is static. We are motivated by the fact that the traffic information can be statically derived from the application specification. The traffic variation is not significant especially in data-streaming applications, since the traffic pattern is rather regular and periodic. The presented design flow rapidly instantiates on-demand interconnects and scheduler IPs, based on the extracted traffic information. In this work, we assume that the token (or packet) sizes for the applications in Fig. 2 are similar (or same). Though the token size can be different in the ESPAM design flow, the token sizes are similar for the applications considered in this work. As an example, YAPI [20] profiling result indicates that 97% of the Wavelet traffic and 86% of the MJPEG traffic contain identical-sized tokens.

3.4.2 WCPS Example

In Fig. 5(1), weights for each link are depicted as an example. The number on the channels refers to the relative number of requests for tokens. As an example, the weight of the links from $p'_2 \sim p'5$ to p_1 is 5, respectively. The weight of the link from p'_6 to p_1 is 1. This means that the links from $p'_2 \sim p'_5$ to p_1 are five times more utilized than the link from p'_6 to p_1. Therefore, the WCPS arbiter at p_1 checks $p'_2 \sim p'_5$ five times more often than p'_6.

We implement WCPS using the *weight table* which contains a list with the sequence of processor ports that an arbiter checks. Figure 5(2) depicts the weight table for port p_1. The sequence is permutated such that the same processor port index is sparsely distributed. We use a weight counter for each processor port index to handle the permutation. As an example, an arbitration sequence for port p_1 is depicted in Fig. 5(3). In this example, the maximum weight is 5, indicating that there are five sub-rounds. Initially, the weight counters are

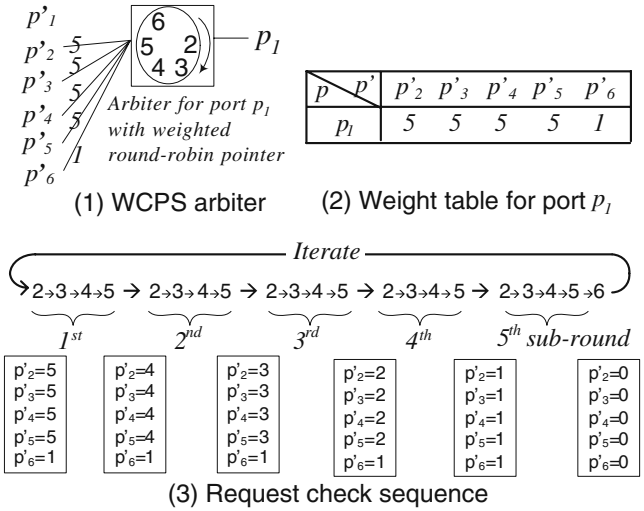


Figure 5 Weighted round robin custom parallel scheduler.

set by the weight table, as depicted in the rectangle in Fig. 5(3). In each sub-round, the arbiter searches processor ports, which have the same counter values. Subsequently, the weight counter value is decremented. When all the weight counter values are zero, or all the sub-rounds are finished, then the counter values are set by the initial weight table. The process is iterated, similarly to a round robin approach.

Figure 4(7) depicts an example scenario for the WCPS. The weight from $p'_2 \sim p'_6$ to p_1 is five times greater than the weight from p'_6 to p_1. In this case, 33 cycles are required to serve the request patterns. For comparison, the CPS requires 35 cycles. The WCPS requires less cycles than the CPS, because the arbitration time at port p_1 is reduced. In most cases, WCPS performs better than CPS, since the arbitration is adaptively performed with regard to the traffic information. WCPS occupies more area than CPS, since the internal logic is relatively more complex. The area overhead issue is also discussed in Section 5.

3.5 Shared Arbiters for Custom Crossbars

In our (W)CPS scheme, an arbiter is accommodated at each crossbar port in order to perform parallel arbitration. In addition, we considered that a single task is mapped onto a single processor. However, the number of tasks (or processes) is often greater than the number of ports (or processors). In this case, it is required to map multiple tasks onto a single processor. Subsequently, the number of links per node increases and the implementation cost of (W)CPS increases accordingly. In this section, we propose a novel shared arbitration scheme in order to provide a design trade-off between performance and cost.

3.5.1 Shared Custom Parallel Scheduler (SCPS)

In this work, we propose a custom scheduler with the shared arbitration (SCPS) scheme. In SCPS, multiple arbiters are shared, such that each arbiter sequentially performs arbitration while multiple arbiters operate in parallel. Our SCPS scheme combines the advantages of the low cost in SQS and increased performance in CPS. Figure 6 depicts an example of CPS and SCPS for a six-node MJPEG application. Figure 6(1) depicts CPS, where an arbiter is established per port. Figure 6(2) depicts SCPS, where arbiters for port (p_1, p_2), (p_3, p_6), and (p_4, p_5) are shared. The traffic requirement is also depicted in Fig. 6, derived from the YAPI tool [20]. λ is the total incoming traffic bandwidth or an arrival rate to the $p1$ port. We aim to reduce the implementation cost, by sharing the arbiter resources. Figure 6 indicates that the number of arbiters in SCPS is 3, while the number of arbiters in CPS is 6. This means the implementation cost can be reduced. Our SCPS is also constructed over on-demand interconnects. Compared to CPS, the network performance can be likely decreased. This is due to the fact that a shared arbiter is accommodated for multiple ports and the round-robin search space can be increased for the shared arbiters.

Figure 4(8) depicts the example scenario for the SCPS, where 3 sequential arbiters operate in parallel. As Fig. 4(8) shows, 39 cycles are required to serve the request patterns. For comparison, the CPS requires 35 cycles and the WCPS requires 33 cycles.

3.5.2 Clustering Method

In this section, we present a method to cluster arbiters. Our aim is to minimize the performance degradation by balancing the traffic bandwidth, compared to CPS.

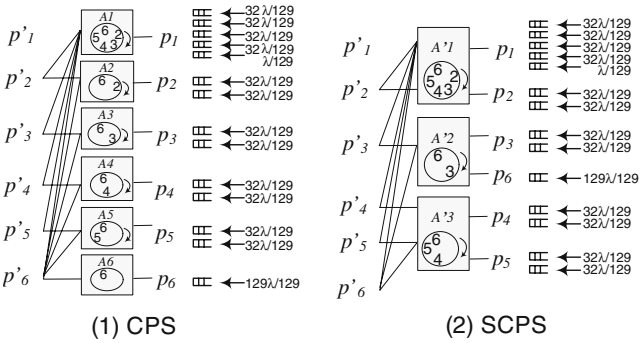


Figure 6 Shared custom parallel scheduler.

In this work, we consider clusters of two arbiters only. Given a CPS scheme, our method can be described as follows:

1. *Calculate cost function*: Calculate the individual cost using a cost function for arbiter A_j, where $1 \leq j \leq p$. p is the number of CPS arbiters.
2. *Sort*: Sort arbiters in an increasing order, based on the derived cost function.
3. *Iterate clustering*: Iteratively cluster arbiters with the lowest and the highest cost function, in the sorted list.

The cost function is a relative metric to represent the utilization of an arbiter. We define the cost function for the arbiter A_j as the following:

$$C\{A_j\} = \frac{\text{links}_j}{2} \times \text{U}_j, \tag{1}$$

where $C\{A_j\}$ refers to the cost function of a CPS arbiter in the jth port. $links_j$ refers to the number of physical links which are connected to the arbiter A_j. $\frac{\text{links}_j}{2}$ corresponds to the relative arbitration latency, as described in the next section. U_j refers to the summation of the traffic for the arbiter A_j and corresponds to relative arbiter utilization. The individual channel utilization for U_j is depicted in Fig. 6. As an example, U_1 is derived by $\frac{32+32+32+32+1}{129}$ [see Fig. 6(1)]. We multiply U_j by $\frac{\text{links}_j}{2}$ in order to reflect the actual utilization of an arbiter. As an example in Fig. 6, we cluster arbiters in the following way:

1. *Calculate cost function*: We calculate the cost function for each arbiter. As an example, $links_1 = 5$ and $\text{U}_1 = \frac{32+32+32+32+1}{129}$ for arbiter $A1$. Therefore $\text{C\{A1\}} = \frac{5}{2} \times (\frac{32+32+32+32+1}{129}) = 2.5$. Similarly, $\text{C\{A2\}} = \text{C\{A3\}} = \text{C\{A4\}} = \text{C\{A5\}} = 0.496$, and $\text{C\{A6\}} = 0.5$.
2. *Sort*: Arbiters are sorted in an increasing order. We obtain ({A2},{A3},{A4},{A5},{A6},{A1})
3. *Iterate clustering*: We cluster {A1,A2}, {A3,A6}, and {A4,A5}.

In this way, a highly utilized arbiter and a less utilized arbiter can be clustered. Also, more than 2 arbiters (or variable-sized arbiters) can be shared in a similar way, from the sorted list.

4 Performance Analysis

In this section, we present comparative performance evaluations for different schedulers. In Section 4.1, we define the performance metric. Different crossbar schedulers are compared in terms of the service rates. In Section 4.2, we conduct the queueing modeling and compare the entire network performance for considered applications.

4.1 Scheduler Performance

4.1.1 Performance Metric

In order to compare the relative scheduler performance for given applications, we consider the following performance metric for different schedulers:

$$M_{\text{scheduler}} = \sum_{i=1}^{N} \frac{\mu_i \times \mathbf{u}_i}{N}, \tag{2}$$

where $M_{\text{scheduler}}$ is a performance metric for each scheduler and refers to the relative service rate in tokens/s. μ_i is the service rate of the arbiter for the ith logical channel. N is the total number of logical channels. $\mathbf{u}_i$ refers to the normalized channel utilization for the ith channel. The $\mathbf{u}_i$ is multiplied by the arbiter service rate μ_i, in order to reflect the actual amount of traffic for an application. The service rate μ_i can be modeled as:

$$\mu_i = (T_{\text{arbit}} + T_{\text{transmit}})^{-1}, \tag{3}$$

where T_{arbit} is an arbitration latency to establish a link. T_{transmit} is the actual data transmission latency after the link is established. T_{transmit} can be derived as $\frac{\text{Num\_Word}}{\text{Clk}_{\text{sys}}}$, where Num_Word refers to the number of data words, or the token size. Clk_{sys} refers to the system clock frequency.

4.1.2 Crossbar Schedulers

In this subsection, we present a performance comparison between five different schedulers. We can fairly compare different scheduling schemes, since only the arbitration latencies are different. T_{arbit} in Eq. 3 for different crossbar schedulers can be approximated as follows:

$$T_{\text{arbit\_SQS}} = k_1 \frac{\left\lfloor \frac{\#\text{ports}}{2} \right\rfloor \times C_{\text{hand}}}{\text{Clk}_{\text{sys}}} \tag{4a}$$

$$T_{\text{arbit\_FPS}} = k_2 \frac{\left\lfloor \frac{\#\text{ports}}{2} \right\rfloor + C_{\text{hand}}}{\text{Clk}_{\text{sys}}} \tag{4b}$$

$$T_{\text{arbit\_CPS}} = k_3 \frac{\left\lfloor \frac{\#\text{links}}{2} \right\rfloor + C_{\text{hand}}}{\text{Clk}_{\text{sys}}} \tag{4c}$$

$$T_{\text{arbit\_WCPS}} = k_4 \frac{\lfloor \frac{\#\text{links}}{2} \rfloor \left(1 - \frac{W_{\text{std}}}{W_{\max}}\right) + C_{\text{hand}}}{\text{Clk}_{\text{sys}}} \quad (4d)$$

$$T_{\text{arbit\_SCPS}} = k_5 \frac{\lfloor \frac{\#\text{links}}{2} \rfloor \{(\alpha\, C_{\text{hand}}) + (1-\alpha)\, C_{\text{hand}}\}}{\text{Clk}_{\text{sys}}}, \quad (4e)$$

- $T_{\text{arbit\_SQS}}$ refers to the arbitration latency (in seconds) for the sequential scheduler (SQS). k_1 is the scaling factor 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 a single arbiter in the system. Only after the current processor port is served, the next port is served. C_{hand} refers to the handshaking latency in number of cycles. Therefore, we model these sequential operations by multiplying the handshaking latency C_{hand} by $\lfloor \frac{\#\text{ports}}{2} \rfloor$.
- $T_{\text{arbit\_FPS}}$ refers to the arbitration latency in the fully parallel scheduler (FPS). 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. However, multiple requests can be concurrently served. Therefore, we model these parallel arbitration operations, by adding $\lfloor \frac{\#\text{ports}}{2} \rfloor$ and the C_{hand}.
- $T_{\text{arbit\_CPS}}$ refers to the arbitration latency for the custom parallel scheduler (CPS). Similarly to the FPS, we model the arbitration latency by adding the request check latency and the C_{hand}. However, the request check latency is modeled by $\lfloor \frac{\#\text{links}}{2} \rfloor$, since the actual arbitration is performed for the only established physical links, instead of all links. #links is equal or less than #ports. Therefore, it is obvious that $T_{\text{arbit\_CPS}}$ is less than $T_{\text{arbit\_SQS}}$ and $T_{\text{arbit\_FPS}}$. Only if the required topology is all-to-all, then the $T_{\text{arbit\_CPS}}$ is equal to $T_{\text{arbit\_FPS}}$.
- $T_{\text{arbit\_WCPS}}$ refers to the arbitration latency for the weighted round-robin scheduler (WCPS). In Eq. 4d, W_{std} refers to the standard deviation of weights for connected links in an arbiter. $W_{\max}$ refers to the maximum weight in the connected links to an arbiter. We divide by $W_{\max}$ in order to normalize the weight. Compared to the round-robin CPS, the arbitration latency can be reduced. As an example, in Fig. 4(3), $\frac{W_{\text{std}}}{W_{\max}}$ for port p_1 is $\frac{1.8}{5}$, or 0.36. We used the term $\frac{W_{\text{std}}}{W_{\max}}$ to statistically model the reduced arbitration latency, due to the following facts. First, the higher weight deviation means that the variation of link bandwidths (that an application requires) is high. Second, our scheduler allocates higher bandwidth (or time slots) to the links that require high bandwidth. Third, the traffic pattern of our streamed multimedia applications is highly regular and periodic. As an example, the MJPEG application is performed block-wise, where each block contains 8×8 image pixels. Due to the regularity of the traffic pattern, the statistical model can be simplified as shown in Eq. 4d. It must be noted that the W_{std} is calculated for only connected links. In other words, the WCPS arbiter at each port checks for only the necessary ports with different weights.
- $T_{\text{arbit\_SCPS}}$ refers to the arbitration latency for the shared custom parallel scheduler (SCPS). In Eq. 4e, α is 1 if the number of arbiters is 1. In this case, SCPS is the same as SQS. α is 0 if the number of arbiters is greater than 1. In this case, the SCPS operates in a same way as CPS. The difference between CPS and SCPS is that the *#links* of a SCPS arbiter is in many cases greater than the *#links* of CPS arbiter. For example, as depicted in Fig. 6(2), *#links* for $(A'1, A'2, A'3)$ are (5,2,3). For comparison, as depicted in Fig. 6(1), *#links* for $(A1 \sim A6)$ are (5,2,2,2,2,1).

4.1.3 MJPEG Example

We consider the six-node MJPEG application, depicted in Fig. 2(8). The different scheduling schemes are given in Fig. 7. The service rates for each scheme are derived as follows. We assume that the schedulers operate at 100MHz and $k_1 \sim k_5$ are 1. C_{hand} is assumed to be 2 cycles per token, since each of the request and the acknowledgement requires 1 cycle, respectively. To obtain T_{transmit}, it is assumed that each word takes 1 cycle to transmit.

- *SQS*: The service rate μ_s per token for the centralized SQS arbiter is derived as follows. Considering the single-word token communications, or Num_Word = 1, T_{transmit} is $\frac{1\,\text{cycle}}{100\ \text{MHz}}$ (in seconds). Since C_{hand} is two cycles and number of ports is 6, $T_{\text{arbit\_SQS}}$ is obtained from Eq. 4a and it is $\frac{(\lfloor \frac{6}{2} \rfloor \times 2)\ \text{cycles}}{100\ \text{MHz}}$. By substituting $T_{\text{arbit\_SQS}}$ to Eq. 3, μ_s can be derived by $\frac{100\ \text{MHz}}{(\lfloor \frac{6}{2} \rfloor \times 2+1)\text{cycles}} = 14.3 \times 10^6$ tokens/s. The performance metric M_{SQS} is derived from Eq. 2. As Fig. 7(1) depicts, there are 14 channels, or $N = 14$. The relative channel utilization $\mathbf{u}$ is also depicted in Fig. 7(1). As a result, M_{SQS} is derived by $\frac{(14.3 \times 10^6)(\frac{12 \times 32 + 1 + 129}{129})}{14}$ or 4.1×10^6 tokens/s, from Eq. 2. The derived M_{SQS} indicates the relative service rate for the MJPEG traffic.

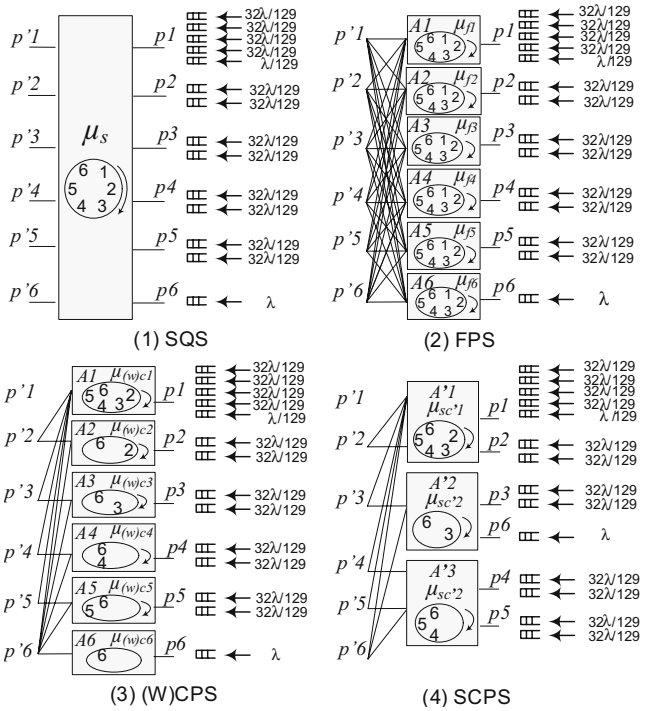


Figure 7 Different schedulers.

- *FPS*: Similarly, a service rate μ_f for each FPS arbiter can be derived by $\frac{100\text{ MHz}}{(\lfloor\frac{6}{2}\rfloor+2+1)\text{cycles}} = 16.7\times 10^6$ tokens/s [see Fig. 7(2)]. M_{FPS} is derived by $\frac{(16.7\times10^6)(\frac{12\times32+1+129}{129})}{14}$ or 4.8×10^6 tokens/s, from Eq. 2.
- *CPS*: A service rate μ_c for each CPS arbiter is determined by the topology [see Fig. 7(3)]. μ_{c1} for an arbiter A_1 is $\frac{100\text{ MHz}}{(\lfloor\frac{5}{2}\rfloor+2+1)\text{cycles}} = 20\times10^6$ tokens/s. Similarly, μ_{c2}, μ_{c3}, μ_{c4}, μ_{c5} is $\frac{100\text{ MHz}}{(\lfloor\frac{2}{2}\rfloor+2+1)\text{cycles}} = 25\times 10^6$ tokens/s. μ_{c6} is 33×10^6 tokens/s. As a result, M_{CPS} is 7.3×10^6 tokens/s and is derived by the following equation:

$$M_{CPS} = \frac{(20\times10^6)\left(\frac{32\times4+1}{129}\right)+(25\times10^6)\left(\frac{32\times8}{129}\right)+(33\times10^6)\left(\frac{129}{129}\right)}{14}.$$

- *WCPS*: A service rate μ_{wc} for each WCPS arbiter is determined by the topology as well as the weight distribution [see Fig. 7(3)]. The weight distribution is depicted in Fig. 2(8). As an example, the standard deviation W_{std} of (32, 32, 32, 32, 1) is 13.9 for arbiter A_1 in *Video_in* node. The maximum weight $W_{\max}$ is 32. Subsequently, $\frac{W_{\text{std}}}{W_{\max}}$ is calculated by $\frac{13.9}{32}$, or 0.43. Therefore, μ_{wc1} for arbiter A_1 is $\frac{100\text{ MHz}}{(\lfloor\frac{5}{2}\rfloor(1-0.43)+2+1)\text{cycles}} = 24\times10^6$ tokens/s. For comparison, the service rate in CPS for arbiter A_1 is 20×10^6 tokens/s. This means that the service rate of WCPS is 20% higher than the service rate of CPS for arbiter A_1. On the other hand, the weight distribution is uniform for arbiters $A_2 \sim A_5$. For the ports $p_2 \sim p_6$, the standard deviation W_{std} is 0, indicating that the arbitration is performed in the same way as in the round-robin CPS. In case of p_6, only single link is established. Subsequently, the service rates, $\mu_{wc2} \sim \mu_{wc6}$ are the same for both CPS and WCPS. M_{WCPS} is 7.6×10^6 tokens/s and is derived by the following equation:

$$M_{\text{WCPS}} = \frac{(23\times10^6)\left(\frac{32\times4+1}{129}\right)+(25\times10^6)\left(\frac{32\times8}{129}\right)+(33\times10^6)\left(\frac{129}{129}\right)}{14}.$$

- *SCPS*: A service rate μ_{sc} for each SCPS arbiter can be derived in a similar way as CPS [see Fig. 7(4)]. As described earlier, the difference between CPS and SCPS is the number of links that each arbiter handles. $\mu_{sc'1}$ for an arbiter A'_1 (for ports p_1, p_2) is $\frac{100\text{ MHz}}{(\lfloor\frac{5}{2}\rfloor+2+1)\text{cycles}} = 20\times10^6$ tokens/s. Similarly, $\mu_{sc'2} = \mu_{sc'3} = 25\times10^6$ tokens/s. M_{SCPS} is 6.6×10^6 tokens/s and is derived by the following equation:

$$M_{\text{SCPS}} = \frac{(20\times10^6)\left(\frac{32\times6+1}{129}\right)+(25\times10^6)\left(\frac{32\times6+129}{129}\right)}{14}.$$

Similarly, the performance metric for different schedulers with different applications is derived, as depicted in Fig. 8. The application task graphs of the 12-node MPEG4, the eight-node PIP, the 12-node VOPD, the 12-node MWD were taken from [22]. The task graph of the six-node MPEG4 specification was taken from [23]. The task graphs of the MJPEG and Wavelet applications were taken from [21]. When the token size is 1-word, our WCPS provides better service rate by factor of 3 compared to SQS and 2 compared to FPS. Our WCPS performs 14% better than CPS for MPEG4 and Wavelet applications. The performance improvement of WCPS over CPS is relatively small for other applications due to the following reasons. First, the topology in practical applications is simple in the sense that the average number of links per node is only 1.6. In many cases, only a single link is connected to the crossbar port, indicating that no arbitration is necessary for those ports. The SCPS is 2 times better than SQS and 1.7 times better than FPS. When the token size is 1-word, the CPS provides on average 22% better service rate than SCPS. This performance degradation of SCPS over CPS is due to the sharing arbiter resources.

As Fig. 8(2) shows, when the token size is 64-words, our WCPS performs still better than the reference schedulers, while the improvement is significantly less than the case of the small-sized tokens. This is due

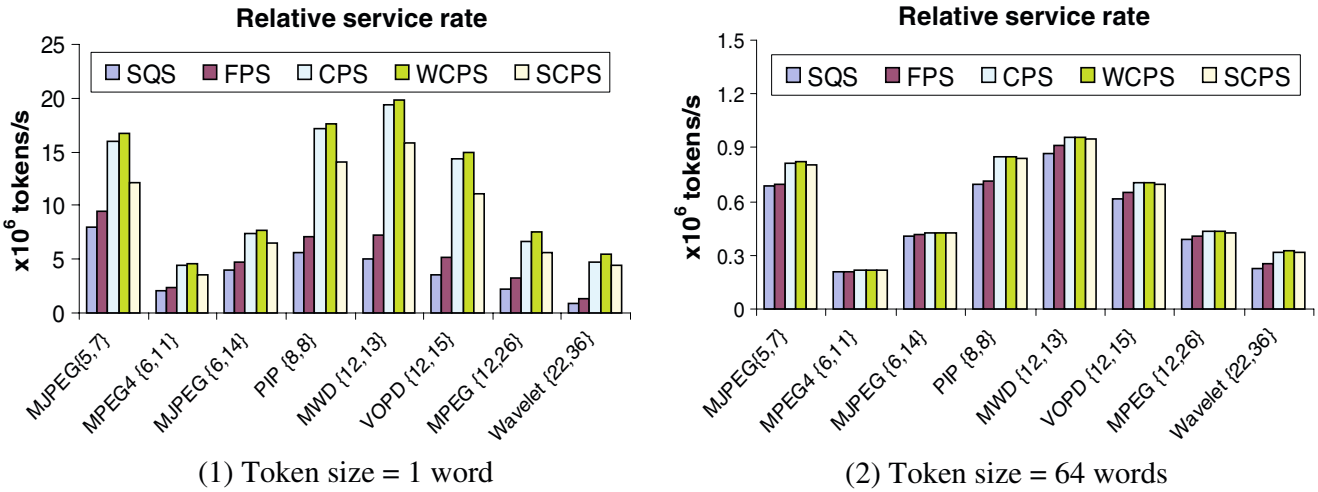


Figure 8 Relative service rate for different schedulers.

to the fact that the token transmission latency is a dominant term to determine the service rate.

4.2 Network Performance

In the previous section, the comparative performance of different scheduler modules is presented. In this section, the entire network performance is compared in terms of latency and throughput.

4.2.1 Queueing Modeling

We formulated a network performance model to compare the relative latency and throughput. Our analysis is based on the queuing model [25], since the queuing model provides a reasonable fit to the reality with relatively simple formulation. Based on the general queuing model, the following assumptions are made:

- The system network conforms to the Jackson model [25]. Each queue behaves as an independent single server and the total network latency can be modeled as the combination of each service latency.
- 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 the service time distribution is exponential.
- 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 [25] and can be utilized as a modeling method in general, since most of NoC-based systems accommodate buffers (or queues) for the communication. The queueing model can be suitably applied to our system due to the following facts:

- The KPN model and the actual system are indeed a network of queues.
- The incoming data stream pattern is statistically random.
- A token in the FIFO is independently served by a single scheduler (or server) at each crossbar port. In addition, the physical queue size is sufficiently large. As an example, we implement a FIFO using (multiple) embedded block RAMs, where single block RAM primitive in our target device can accommodate 512 32-bit words, which is large enough.

Consequently, the general network latency can be modeled as:

$$T_{\text{network}} = \frac{1}{\lambda} \sum_{i=1}^{N} \frac{\lambda_i}{\mu_i - \lambda_i}, \tag{5}$$

where T_{network} is the total latency of the system network. N 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 token rate to the ith queue. μ_i is the service rate of the arbiter for the ith queue.

4.2.2 Case Studies

We studied four cases with two different network sizes and two different token sizes. First, we consider the six-node MJPEG application, depicted in Fig. 2(8) which requires a small network size. The port-mapped system

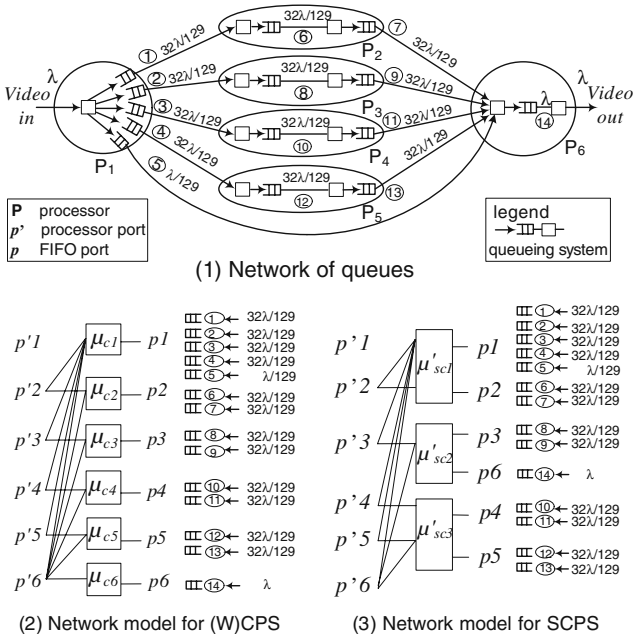


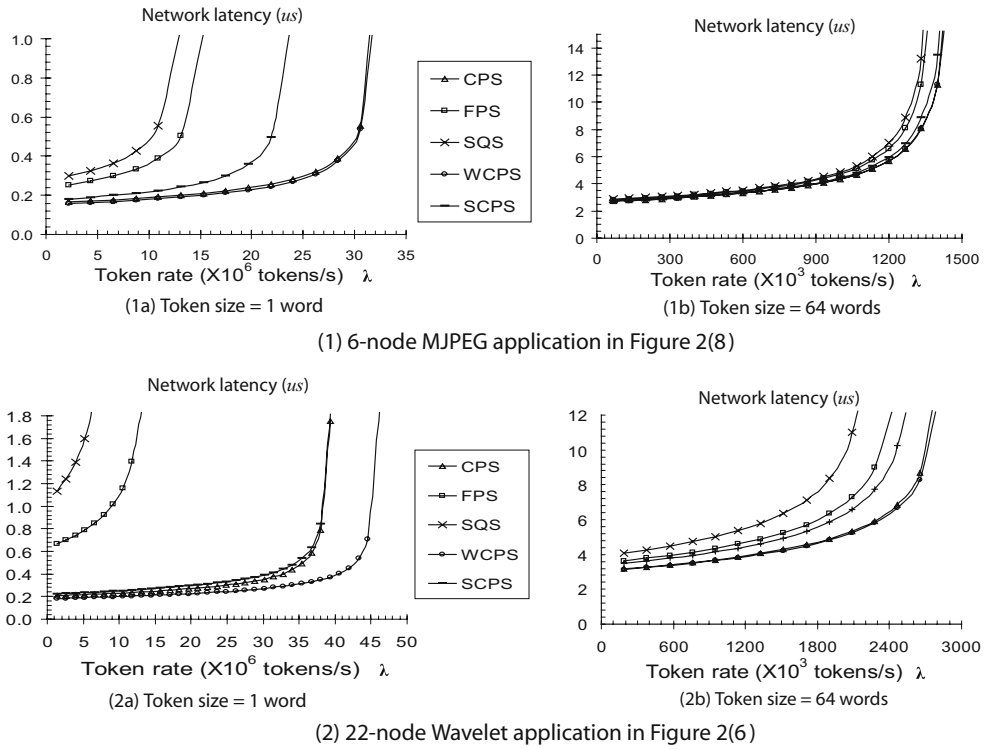
Figure 9 A case study.

model is depicted in Fig. 9(1). Considering processor $\mathbf{P}_1$ as a streamed data source, $\mathbf{P}_1$ generates the data in a rate of λ (tokens/s). A token rate in each queue is derived from the YAPI profiler [20], as depicted in Fig. 9(1). Figure 9(2) and (3) depict the network model for (W)CPS and SCPS. The service rate μ for each scheduler is derived in the same way as in Section 4.1.3 with Eq. 3. The total network latency can be derived by substituting the service rate μ to Eq. 5.

As a result, the network latencies are derived and depicted in Fig. 10(1a). The performance analysis indicates that the WCPS reduces latency by at least 44% 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, the network with our WCPS saturates at the bandwidth of 32×10^6 tokens/s and the network with SQS saturates at the bandwidth of 13×10^6 tokens/s. Therefore, WCPS is 2.5× better than SQS in terms of throughput. Similarly, our WCPS is 2× better than FPS in terms of throughput. Compared to CPS, WCPS provides marginal performance improvement. This is due to the fact that the weight variation is not high, as depicted in Fig. 9(2). Compared to SCPS, WCPS reduces the latency by at least 10% and provides 25% better throughput.

Second, we consider the MJPEG case study for a large token size with Num_Word = 64. The network performance can be derived in the similar way as in the previous case, while only the service rate differs. T_{transmit} is derived by $\frac{64\,\text{cycles}}{100\,\text{MHz}}$, because the Num_Word is 64. As a result, Fig. 10(1b) shows that the WCPS performs at least 5% better than SQS and 3.3% better than FPS for all ranges. The performance improvement

Figure 10 Network performance.



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}.

Third, we consider the 22-node Wavelet application depicted in Fig. 2(6), which requires a large-sized network and a single-word token size. As the number of crossbar ports increases, T_{arbit} for SQS and FPS proportionally increases. However, T_{arbit} for (W)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 22 ports. Figure 10(2a) depicts the network latency for single-word token transactions. The network latency of CPS is reduced by at least 84% compared to SQS and at least 73% compared to FPS. The WCPS provides 24% higher throughput than CPS.

Finally, Fig. 10(2b) depicts the network latency for 64-word token transactions in the Wavelet application. CPS performs at least 22% better than SQS and 13% better than FPS. It can be suggested that the presented CPS, WCPS, SCPS schemes are more beneficial for small sized tokens communicated over large networks.

5 Implementation Results

The aforementioned scheduler modules were implemented in VHDL to integrate the presented network components in the ESPAM design environment [16–18]. The presented schedulers are implemented with parameterized arbiter arrays. The scheduler modules are generic in terms of data width, number of ports, on-demand topologies, and a traffic weight table. The switch module in [24] is used as a common interconnects fabric and the communication controller in [17] is used as a common network interface. The functionality of the network is verified by VHDL simulations. In order to fairly compare different schedulers, we implemented the schedulers such that $k_1 \sim k_5 = 1$ for Eq. 4. Implementation details can be found in [21]. The exemplary behaviors of the implemented schedulers are depicted in Fig. 4.

The implemented scheduler modules are compared in terms of area utilization. We experimented with different task graph topologies of realistic applications, depicted in Fig. 2. The implemented schedulers 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 Fig. 11(1). Our CPS requires on average 83% less area compared to the FPS. The CPS requires on average 28% more area compared to the network with SQS. We consider that the area overhead of our CPS over SQS is relatively less significant, since our target xc2vp20 device contains 9,280 slices and chip-wise overhead of CPS over SQS is on average 3%. 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 1.6 and *does not increase* as the number of nodes increases. WCPS occupies on average 13% more area than CPS. The SCPS occupies 14% more area than CPS. As described earlier, this is due to the fact that in many cases only one link is connected to arbiters in CPS scheme. Due to this, the

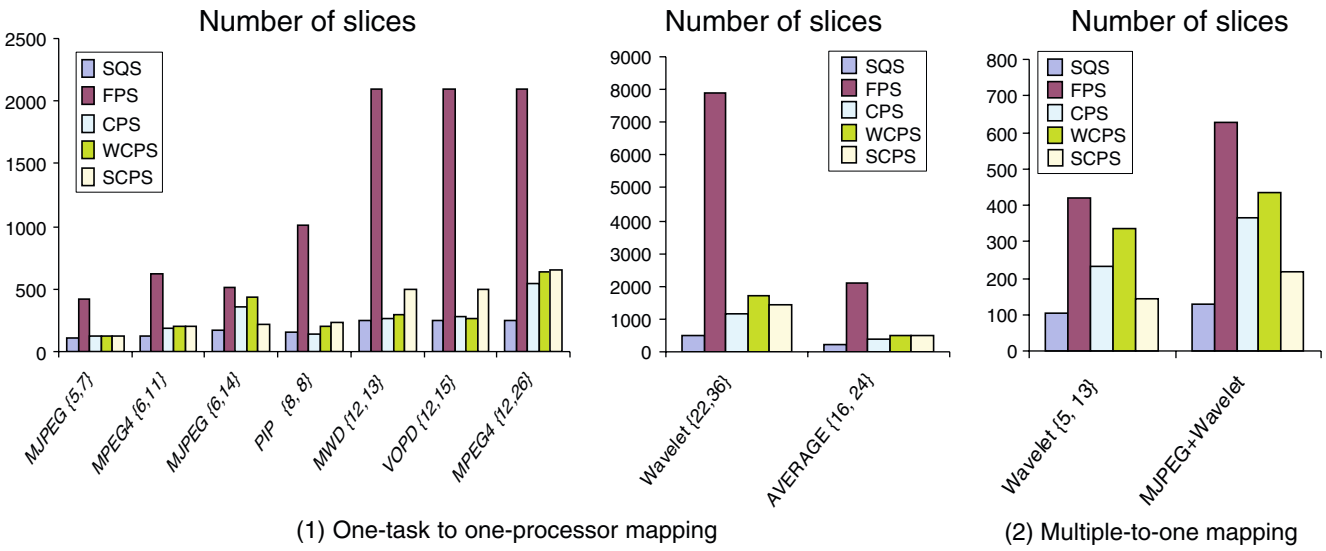


Figure 11 Experimental results.

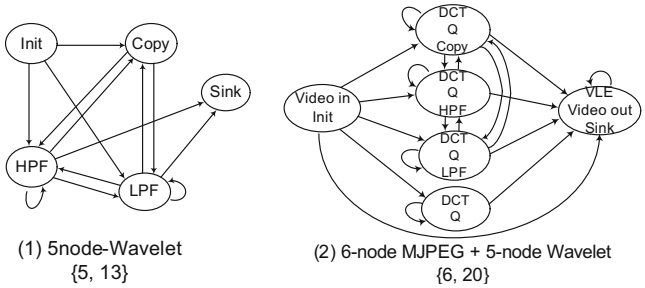


Figure 12 Topology after multiple tasks are mapped onto a processor.

CPS arbiter logic can be greatly simplified, especially when a single task is mapped onto a processor. In addition, the single SCPS arbiter contains relatively more complex decoding logic than single CPS arbiter.

The experiment shows that CPS scheme is suitable when single task is mapped onto a physical processor. In many cases, however, the number of tasks is often greater than the number of physical processors for given applications. Therefore, multiple tasks are required to be mapped onto a physical processor because a target device is limited in size. Subsequently, the required number of links per port increases. Figure 12(1) depicts the five-node representation for the 22-node Wavelet application. Figure 12(1) is derived by clustering the same tasks onto a single processor. As a result, the number of links per node is $\frac{13}{5}$, or 2.6. Note that the number of links per node is $\frac{36}{22}$, or 1.6 when a single task is mapped onto one processor. Figure 12(2) depicts a topology of a synthetic application that combines the six-node MJPEG and the five-node Wavelet specification. In this case, the number of links per node is $\frac{20}{6}$, or 3.3. In both cases, the number of links per node increases when compared to a one-to-one mapping. We implemented different schedulers for these cases and compared the area cost. As depicted in Fig. 11(2), the SCPS scheme requires 70% less area than the CPS scheme. Figure 13 depicts the performance metric defined in Section 4.1 for the five-node Wavelet representation. As Fig. 13(1) depicts, the WCPS provides 13% better service rate for 1-word token sizes. The WCPS scheme also occupies 20% more area than the CPS, whereas the chip-wise overhead is 1%, for five-node Wavelet representation. The experimental result suggests that when the number of links per port increases, the SCPS can be beneficial, providing a design trade-offs between cost and performance.

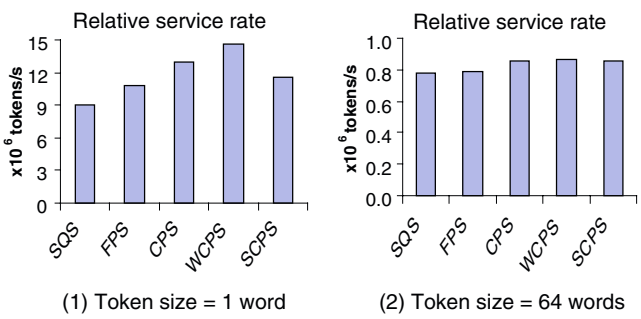


Figure 13 Performance for five-node wavelet specification.

6 Conclusions

The performance of a parallel application increases when the underlying network can be matched to the on-demand logical topology and adhered to the bandwidth requirements of an application. In this paper, we presented crossbar schedulers designed for reconfigurable on-chip-networks capable of adjusting themselves to custom topologies and custom bandwidth requirements. We conducted a queueing analysis to determine the overall network performance. From the performance evaluation and implementation, the following conclusions can be drawn:

- The presented custom schedulers (CPS, WCPS, SCPS) provide better performance than conventional schedulers, especially for small-sized tokens communicated over large-size network. In addition, our schedulers efficiently utilize the on-chip resources by adapting themselves to the logical topologies.
- When the tasks are one-to-one mapped onto processors, the custom parallel scheduler (CPS) performs significantly better than SQS, FPS, SCPS.
- The weighted round-robin custom scheduler (WCPS) can increase the network performance, by assigning different network bandwidth to different traffic requirements. The WCPS is beneficial when the traffic load is not evenly distributed.
- The shared custom scheduler (SCPS) provides a design trade-offs for cost and performance when compared to the mentioned schedulers. The SCPS can be beneficial when the number of links per port increases.
- The sequential scheduler (SQS) provides adequate performance and cost when the token size is large and the size of a network is fairly small.

Our schedulers were implemented using parameterized arbiter arrays. By utilizing the on-demand

topology and traffic requirements as design parameters, the schedulers were adapted to given applications without modifying the network implementation. We showed that our schedulers perform better and occupy significantly less area than conventional fully parallel schedulers. Our schedulers perform significantly better and occupy moderately more area than sequential schedulers.

References

1. Mckeown, N. (1999). The *i*SLIP scheduling algorithm for input-queued switches. *IEEE/ACM Transaction on Networking* (*TON*), *7*(2), 188–201, April.
2. Bjerregaard, T., & Mahadevan, S. (2006). A survey of research and practices of network-on-chip. *ACM Computing Surveys, 38*(1), 1–51, March.
3. Vassiliadis, S., & Sourdis, I. (2006). FLUX networks: Interconnects on demand. In *Proceedings of international conference on computer systems architectures modelling and simulation* (*IC-SAMOS'06*) (pp. 160–167), July.
4. Vassiliadis, S., & Sourdis, I. (2007). FLUX interconnection networks on demand. *Journal of Systems Architecture, 53*(10), 777–793, October.
5. Moraes, F., Calazans, N., Mello, A., Möller, L., & Ost, L. (2004). HERMES: An infrastructure for low area overhead packet-switching netwoks on chip. *Integration, The VLSI Journal, 38*(1), 69–93, October.
6. Marescaux, T., Nollet, V., Mignolet, J.-Y., Moffat, A. B. W., Avasare, P., Coene, P., et al. (2004). Run-time support for heterogeneous multitasking on reconfigurable SoCs. *Integration, The VLSI Journal, 38*(1), 107–130.
7. Sethuraman, B., Bhattacharya, P., Khan, J., & Vemuri, R. (2005). LiPaR: A lightweight parallel router for FPGA based networks on chip. In *Proceedings of the great lakes symposium on VLSI* (*GLSVLSI'05*) (pp. 452–457), April.
8. Bartic, T. A., Mignolet, J.-Y., Nollet, V., Marescaux, T., Verkest, D., Vernalde, S., et al. (2005). Topology adaptive network-on-chip design and implementation. *IEE Proceedings. Computers and Digital Techniques, 152*(4), 467–472, July.
9. Brebner, G., & Levi, D. (2003). Networking on chip with platform FPGAs. In *Proceedings of the IEEE international conference on field-programmable technology* (*FPT'03*) (pp. 13–20), December.
10. Srinivasan, K., & Chatha, K. S. (2005) A low complexity heuristic for design of custom network-on-chip architectures. In *Proceedings of international conference on design, automation and test in Europe* (*DATE'06*) (pp. 130–135), March.
11. Murali, S., & Micheli, G. D. (2005) An application-specific design methodology for STbus crossbar generation. In *Proceedings of international conference on design, automation and test in Europe* (*DATE'05*) (pp. 1176–1181), March.
12. Loghi, M., Angiolini, F., Bertozzi, D., Benini, L., & Zafalon, R. (2004). Analyzing on-chip communication in a MPSoC environment. In *Proceedings of international conference on design, automation and test in Europe* (*DATE'04*) (pp. 752–757), February.
13. Murali, S., Benini, L., & Micheli, G. D. (2007). An application-specific design methodology for on-chip crossbar generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 26*(7), 1283–1296, July.
14. Pasricha, S., Dutt, N., & Ben-Romdhane, M., (2006). Constraint-driven bus matrix synthesis for MPSoC. In *Proceedings of 11th Asia and South Pacific design automation conference* (*ASP-DAC'06*) (pp. 30–35), January.
15. Gupta, P., & McKeown, N. (1999). Designing and implementing a fast crossbar scheduler. *IEEE Micro, 19*(1), 203–289, January–February.
16. Nikolov, H., Stefanov, T., & Deprettere, E. (2006). Multi-processor system design with ESPAM. In *Proceedings of the 4th IEEE/ACM/IFIP international conference on HW/SW codesign and system synthesis* (*CODES-ISSS'06*) (pp. 211–216), October.
17. Nikolov, H., Stefanov, T., & Deprettere, E. (2006). Efficient automated synthesis, programming, and implementation of multi-processor platforms on FPGA chips. In *Proceedings of 16th international conference on field programmable logic and applications* (*FPL'06*) (pp. 323–328), August.
18. Nikolov, H., Stefanov, T., & Deprettere, E. (2008). Systematic and automated multi-processor system design, programming, and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits, 27*(3), 542–555, March.
19. Kienhuis, B., Rijpkema, E., & Deprettere, E. (2000). COMPAAN: Deriving process networks from Matlab for embedded signal processing architectures. In *Proceedings of 8th international workshop on hardware/software codesign* (*CODES'2000*) (pp. 13–17), May.
20. de Kock, E. A., Essink, G., Smits, W. J. M., van der Wolf, P., Brunel, J.-Y., Kruijtzer, W. M., et al. (2000). YAPI: Application modeling for signal processing systems. In *Proceedings of the 37th design automation conference* (*DAC'00*) (pp. 402–405), June.
21. Hur, J. Y., Stefanov, T., Wong, S., & Vassiliadis, S. (2007). Customizing reconfigurable on-chip crossbar scheduler. In *Proceedings of IEEE 18th international conference on application-specific systems, architectures and processors* (*ASAP'07*) (pp. 210–215), July.
22. Bertozzi, D., Jalabert, A., Murali, S., Tamhankar, R., Stergiou, S., Benini, L., et al. (2005). NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Transactions on Parallel and Distributed Systems, 16*(2), 113–129, February.
23. Chang, K.-C., Shen, J.-S., & Chen, T.-F. (2006). Evaluation and design trade-offs between circuit-switched and packet-switched NOCs for application-specific SOCs. In *Proceedings of the 43th design automation conference* (*DAC'06*) (pp. 143–148), July.
24. Hur, J. Y., Stefanov, T., Wong, S., & Vassiliadis, S. (2007). Systematic customization of on-chip crossbar interconnects. In *Proceedings of international workshop on applied reconfigurable computing* (*ARC'07*) (pp. 61–72), March.
25. Baldwin, R. O., Davis IV, N. J., Midkiff, S. F., & Kobza, J. E. (2003). Queueing network analysis: Concepts, terminology, and methods. *The Journal of Systems and Software, 66*(2), 99–117, May.

Jae Young Hur received the B.S. degree in electronics engineering from Cheju National University, Cheju, South Korea in 1995. He received the M.S. degrees in electronics engineering from Sogang University, Seoul, South Korea in 1998 and from Munich University of Technology, Munich, Germany in 2002. From 1999 to 2000, he was a semiconductor engineer in Samsung Electronics, Ltd., South Korea. Since November 2003, he has been with the computer engineering laboratory, Delft University of Technology, The Netherlands, where he is currently a research assistant (PhD student). His research interests include embedded systems design, networks-on-chip, VLSI design, and reconfigurable computing.

Stephan Wong received his PhD in Computer Engineering from the Electrical Engineering, Mathematics and Computer Science faculty of the Delft University of Technology (TU Delft), The Netherlands, in December 2002. He is currently working as an assistant professor at the Computer Engineering Laboratory at the Delft University of Technology (TU Delft), The Netherlands. He has considerable experience in the design of embedded reconfigurable media processors. He has worked also on microcoded FPGA complex instruction engines and the modeling of parallel processor communication networks. His research interests include embedded systems, multimedia processors, complex instruction set architectures, reconfigurable and parallel processing, microcoded machines, and distributed/grid processing.

Todor Stefanov received the Dipl.Ing. and M.S. degrees in computer engineering from The Technical University of Sofia, Sofia, Bulgaria, in 1998 and the Ph.D. degree in computer science from Leiden University, Leiden, The Netherlands, in 2004. From 1998 to May 2000, he was a Research and Development Engineer with Innovative Micro Systems, Ltd., Sofia. From June 2000 to August 2007, he was with the Leiden Institute of Advanced Computer Science, Leiden University, where he was a Research Assistant (PhD student) and a PostDoc Researcher at the Leiden Embedded Research Center. From September 2007 to August 2008, he was a Senior Researcher at the Computer Engineering Lab, Delft University of Technology, Delft, The Netherlands. Since September 1, 2008, Todor Stefanov has been an Assistant Professor with the Leiden Institute of Advanced Computer Science, Leiden University where he performs research at the Leiden Embedded Research Center. His research interests include several aspects of embedded systems design, with particular emphasis on system-level design automation, multiprocessor systems-on-chip design, and hardware/software codesign.