Model-Based Actor Multiplexing with Application to Complex Communication Protocols

Christian Zebelein, Christian Haubelt
University of Rostock
christian.zebelein@uni-rostock.de

Joachim Falk, Tobias Schwarzer, Jürgen Teich
University of Erlangen-Nuremberg
joachim.falk@informatik.uni-erlangen.de

***Abstract*—We propose a dynamic scheduling approach for the concurrent execution of logical actor instances on a single synthesized actor instance. Based on a formal dataflow model of computation, the proposed approach can be applied to a wide range of applications in a model-based design flow. As case-study, we evaluate a bus-cycle-accurate SystemC RTL model based on an InfiniBand network adapter in a PCI Express system.**

I. Introduction and Related Work

Moving design flows for increasingly complex systems to a higher level of abstraction allows many forms of verification to be performed much earlier in the design process, thus reducing time to market, and lowering cost by discovering problems earlier. Here, executable High-Level Models (HLM) can be identified as a key factor, as they can be used to drive architectural exploration [1], hardware and software design flows, and integration & verification processes. In this context, *dataflow models* have successfully been used for high-level modeling in ESL tools [6]. In a dataflow model, concurrent actors communicate via packets (*tokens*) transmitted over channels. As actors naturally expose the parallelism contained in the application, they yield well to synthesis for platforms with multiple computing resources.

In the context of hardware synthesis, synthesizing each actor as a dedicated entity in order to exploit the available parallelism becomes infeasible if the number of actors becomes very large (cf. Fig. 1a). In this case, resource sharing across process boundaries is required in order to reduce the design area. However, existing high-level synthesis (HLS) tools [8] do not support resource sharing across process boundaries.

In [5], resources are shared between processes based on fixed time slots. While this approach permits static scheduling, time slots cannot be reused if the corresponding process is not active. Our approach is based on dynamic resource assignment, thereby improving resource utilization. In [9], resource sharing in the context of pipeline scheduling is described. However, the approach only addresses synchronous dataflow graphs [7]. Our approach targets more general, dynamic dataflow graphs.

We propose a concept to multiplex logical actor instances onto a single physical (synthesized) actor instance in order to reduce the design area compared to the parallel approach, while still providing for the concurrent execution of logical actor instances. To this end, we build on the concept of *reservation stations* as commonly found in out-of-order processor architectures: The logical actor instances are assigned to a small number of reservation stations based on the availability of tokens and free places on channels. Here, the concept of *colored tokens* and *colorless tokens* is introduced: While colored tokens target a specific logical actor instance, colorless tokens represent shared resources between logical actor instances. When assigned to a reservation station (RS), a logical actor instance becomes eligible for execution on the functional units (FUs) comprising the physical actor instance (cf. Fig. 1b).

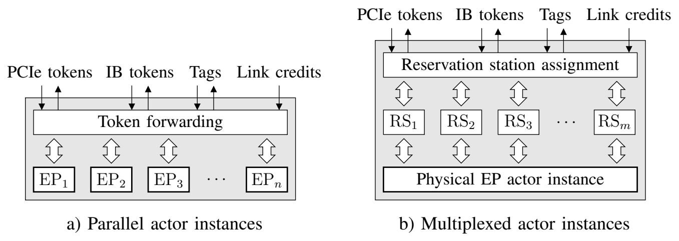


Fig. 1. Modeling approaches: a) parallel actor instances, and b) single physical instance with additional multiplexing logic. For InfiniBand, $n = 2^{24}$.

As additional multiplexing logic is required, the approach is advantageous only if an actor is instantiated more than once. For example, in the video processing domain, actors may be duplicated to model the color channels separately [3], thereby providing for a parallel execution on a suitable platform. In the networking domain, modeling the complex behavior of a single communication end-point (EP) (sockets in TCP/IP) and instantiating the resulting actor multiple times can also be beneficial [10]. We show that the approach can handle a large number of logical actor instances by means of a case-study based on an InfiniBand (IB) network adapter [4] in a PCI Express (PCIe) system.

II. System-Level Overview

The example used throughout the paper is based on an InfiniBand network adapter contained in a PCI Express system (cf. Fig. 2). PCIe employs point-to-point links to overcome the limitations of a shared bus. In a PCIe system, a *root complex device* connects the processor and memory subsystem to the PCIe switch fabric comprised of one or more *switch devices*. As a packet-based protocol, PCIe consists of three layers as known from the networking domain, namely the *transport layer*, the *data link layer*, and the *physical layer*.

The network adapter implements the InfiniBand *SEND* operation defined on the transport layer. With a SEND operation, the local end-point (EP) transmits data to a remote EP. To this end, each EP has an associated *send queue* (SQ) and a *receive queue* (RQ) where *work queue elements* (WQEs) are inserted by the user. A WQE specifies where to fetch and store the transmitted data. Each EP processes its posted WQEs, and, for each finished WQE, places a *completion queue element* (CQE) in an associated *completion queue* (CQ) polled by the user. The InfiniBand specification provides for a total number

978-3-9815370-2-4/DATE14/©2014 EDAA

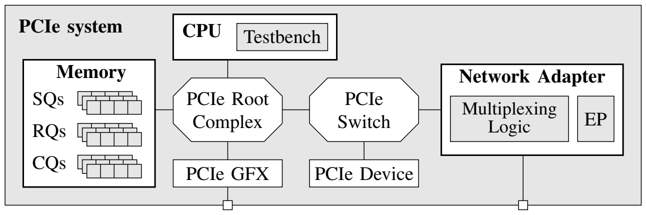


Fig. 2. System-level overview of the running example based on an InfiniBand network adapter in a PCI Express system.

of 2^{24} EPs. Note that an EP can simultaneously process WQEs from its SQ and its RQ. In our example, the SQs, RQs, and CQs are stored in main memory (cf. Fig. 2).

In order to reduce the complexity of the example, only the *unreliable connection* mode specified by InfiniBand [4] has been implemented. In this mode, the requester receives no acknowledgments for transmitted packets, and no guarantees concerning the packet order are given.

III. Scheduling

In this section, we describe our approach for the multiplexing of logical actor instances on just one (or a few) physical actor instances. Actors in our model consist of a set of *transitions* T. A transition $t \in T$ specifies the number of tokens and free places required on input and output ports, a *guard function* f_g, and an *action function* f_a, which is invoked only if enough tokens and free places are available, and the guard function evaluates to true.

Fig. 1b gives a more detailed view of the multiplexing process: The incoming and outgoing edges correspond to tokens from/to the PCIe switch fabric, tokens from/to the link interface, tags which are assigned to memory read requests issued by the model, and link credits consumed by the model before sending a link packet. Note that link credits are generated by the link interface.

First, the logical EP instances are assigned to reservation stations. When assigned to a RS, an EP becomes eligible for execution on the physical EP instance. In the following, the two stages are explained in more detail.

A. Token Color Assignment

First, a set of end-points $\{\mathrm{EP}_1, \ldots, \mathrm{EP}_n\}$ is multiplexed onto a set of available reservation stations $\{\mathrm{RS}_1, \ldots, \mathrm{RS}_m\}$ with $n > m$ in the general case. In our example, $n = 2^{24}$ as specified by InfiniBand, while $m \geq 6$ as will be seen later. Thus, a simple round-robin scheme in order to find an EP with an enabled transition cannot be applied in the general case. However, for our dataflow model, the availability of sufficient tokens and free places is a necessary condition for a transition to be enabled. Therefore, the EP selection process can be based solely on tokens available on input ports, and free places available on output ports.

Concerning input ports, two cases can be distinguished: On the one hand, an input port is a *colored port* if a specific target EP can be determined by analyzing the token stream. This process will be referred to as *token coloring* in the following. Only the target EP can consume its colored tokens. On the other hand, an input port is a *colorless port* if no target EP can be determined by analyzing the token stream. Colorless tokens can be consumed by any EP. For output ports, no such classification is required.

In our example, the PCIe and IB input ports are colored ports: For PCIe packets, the target EP can be determined from the read/write address specified by the packet. IB packets directly specify the target EP. Note that PCIe and IB packets are transmitted by means of multiple 32 bit tokens in our example. For example, in Fig. 3, three IB tokens have been assigned the color white, while the subsequent token has been assigned the color black.

The remaining input ports are colorless ports: Concerning the tags, any EP which wants to issue a PCIe read request must consume a tag token. However, it can consume *any* tag token. The tag is returned by the read response and is then used during token coloring to determine the target EP.

B. Reservation Station Assignment based on Colored Tokens

The colored tokens from a colored port p are dispatched to a smaller set of *colored fifos* C_p which are ultimately fed to the FUs of the physical actor instance. A *colored fifo* contains only tokens of the same color. Thus, in order to forward a token of color i to a colored fifo, two conditions must be met: EP_i must be assigned to a reservation station RS_j, and one of the channels $c \in C_p$ must either be empty, or already contain tokens of color i.

In order to prevent different colors from the same colored port from occupying all reservation stations, the number of colored fifos $|C_p|$ allocated for a colored port p should be less than the number m of reservation stations. Thus, $|C_p|$ will be referred to as *credits of* p in the following. An EP_i has *acquired* a credit for p if a colored channel $c \in C_p$ contains at least one token of color i.

For example, consider Fig. 3, where both RS_1 and RS_2 already have an EP assigned (namely $\mathrm{EP}_{\mathrm{black}}$ and $\mathrm{EP}_{\mathrm{white}}$). For each colored port, a single colored fifo is allocated, each of which is currently acquired by RS_1. Thus, while the black PCIe tokens can be forwarded to the only colored fifo for PCIe tokens, the white IB tokens cannot be forwarded until the remaining black token has been consumed from the only colored fifo for IB tokens.

If for a token of color i from port p, EP_i is not yet assigned to a reservation station, we wait 1) until a credit for p is available, and 2) any RS_j becomes *idle*. RS_j is *idle* if the assigned EP_v has no longer acquired any credits from colored input ports. If both conditions are met, we evict EP_v currently assigned to RS_j, and assign EP_i to RS_j. In order to efficiently re-activate EP_v, the corresponding EP number v is stored in an *initiative queue*. Basically, an initiative queue corresponds to an additional colored input port, but requires no colored fifos, as tokens stored on an initiative queue are solely used for scheduling purposes and are not forwarded to any FU.

Note that a minimum number of one colored fifo must be allocated for each colored input port. However, allocating more than one colored fifo allows tokens of different colors to be consumed by FUs in parallel, thereby enabling multiple logical EPs to be processed concurrently.

In the next section, the forwarding of tokens from colored fifos to FUs, as well as the forwarding of produced tokens to output ports is described.

C. Forwarding of Tokens to/from Functional Units

For each guard function f_g and action function f_a, at least one FU is instantiated (cf. Fig. 4). Thus, executing a guard or action function means executing the corresponding FU. In

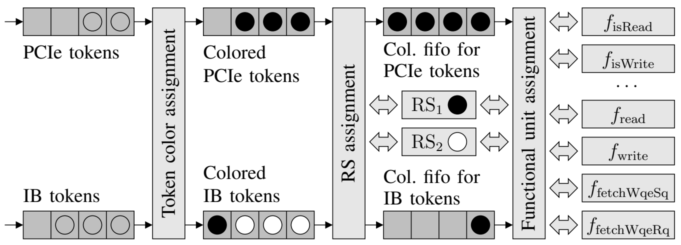


Fig. 3. Forwarding of colored tokens to colored fifos: both colored fifos are acquired by RS_1, and no credits (i.e., free colored fifos) remain.

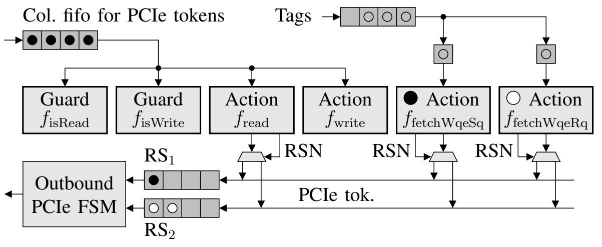

Col. fifo for PCIe tokens
Tags
Guard f_{isRead}
Guard f_{isWrite}
Action f_{read}
Action f_{write}
Action $f_{\mathrm{fetchWqeSq}}$
Action $f_{\mathrm{fetchWqeRq}}$
RSN
RSN
RSN
RS_1
Outbound
PCIe FSM
PCIe tok.
RS_2

Fig. 4. Token forwarding from colored fifos and colorless ports to FUs (upper part), and forwarding of tokens produced by FUs to output ports (lower part). Note that for IB packets, the colored fifo and dedicated output fifos are omitted.

order to provide for the concurrent execution of different EPs, each FU can be individually configured with the desired RS before executing the FU.

According to the underlying dataflow MoC, guard FUs may read actor variables and peek the first token from an input port. Actor variables are stored in the corresponding RS, and can be accessed by the FUs as desired. The tokens to peek are provided through the colored fifos and the colorless input ports. For example, the guard functions f_{isRead} and f_{isWrite} in Fig. 4 may peek the first (black) PCIe token from the (only) colored fifo allocated for PCIe tokens.

Action functions consume tokens. In fact, two action functions $f_a \neq f_b$ may consume tokens from the same port p in the general case. Consequently, an implementation must ensure that if f_a and f_b are executed in parallel, no token conflicts (as known from Petri nets) occur, i.e., that each action function can indeed consume its requested number of tokens.

For colored tokens, this condition is trivially satisfied, as we never execute multiple action FUs for the same RS in parallel in order to avoid data hazards w.r.t. the actor variables which are stored in a RS and accessed by the action FUs.

In contrast, colorless tokens can only be assigned to an action FU when it is about to be executed. Otherwise, deadlocks could be introduced into the model. Thus, in order to avoid token conflicts, the requested number of tokens must be reserved for f_a before f_b can be executed (and vice versa). Here, a simple token reservation scheme consists in allocating dedicated token buffers for f_a and f_b and copying the reserved tokens to the respective buffer before starting the other action FU. For example, in Fig. 4, one tag token has been reserved for each action FU $f_{\mathrm{fetchWqeSq}}$ and $f_{\mathrm{fetchWqeRq}}$.

Analogously, free places on an output port must be reserved for f_a before f_b can be started (and vice versa) if both action functions produce tokens on the same output port. To this end, we allocate a dedicated output fifo for each RS as shown in Fig. 4. Here, $f_{\mathrm{fetchWqeSq}}$ is currently producing black tokens on the output fifo allocated for RS_1, while $f_{\mathrm{fetchWqeRq}}$ is producing white tokens on the output fifo allocated for RS_2. Thus, interleaving of tokens of different colors is avoided at this point. Finally, the state machine for outbound tokens forwards tokens to the output port from the same output fifo until a complete token sequence (e.g., a complete PCIe packet) has been forwarded.

In the next section, we describe an algorithm for the assignment of reservation stations to FUs.

D. Functional Unit Assignment

Assume that EP_i is assigned to RS_j. In order for a transition to be enabled, enough tokens and free places must be available, and the guard function f_g must be evaluated. Consider again Fig. 3 and Fig. 4: Four PCIe tokens and one IB token are currently available for EP_{black}. In contrast, no PCIe or IB tokens are currently available for EP_{white}. Note that three colorless tag tokens are still available for any EP.

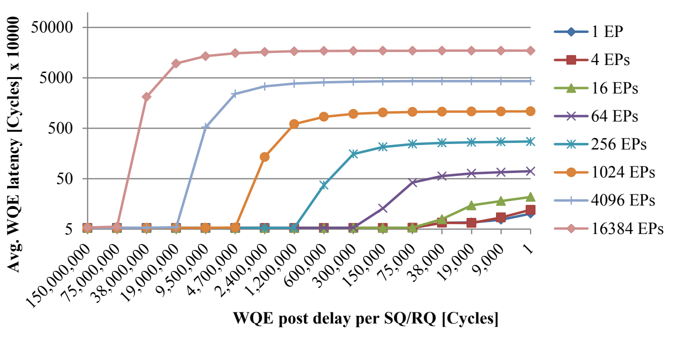
The guard FU can be executed for RS_j if it is not currently executed for any RS. After the execution of f_g is finished, the result is *cached* in RS_j. This prevents multiple executions of f_g for EP_i, and allows other transitions of EP_i to reuse the cached result of f_g.

If the cached result of f_g is $true$, the action function f_a can be finally executed. The corresponding action FU can be executed for RS_j if it is not currently executed for any RS. Executing f_a has the following consequences: 1) All guard results cached in RS_j are invalidated, as they must be re-evaluated after the action FU is finished. 2) All guard FUs which are currently executed for RS_j are reset, thereby freeing the FUs for different reservation stations. 3) RS_j is blocked from starting any guard or action FU until the current action FU is finished. 4) For other reservation stations $\mathrm{RS}_k \neq \mathrm{RS}_j$, cached guard results are invalidated and active guard FUs are reset, but only if the associated guard function may have peeked tokens that are now reserved for the action FU being executed. In our example, the set of invalidated guards is empty for all action functions.

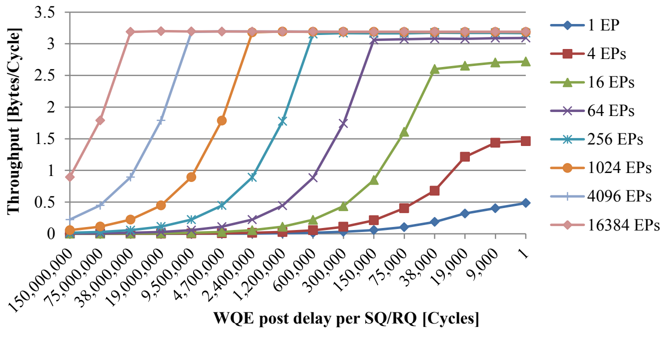
IV. Results

We show that the approach can multiplex a large number of logical actor instances without introducing deadlocks into the system by means of a bus-cycle-accurate SystemC [2] simulation. To this end, the functionality of a single EP has been implemented within the dataflow modeling framework presented in [11]. Subsequently, the multiplexing logic has been semi-automatically generated for the single EP actor by analyzing the transitions of the EP actor as described in the previous sections. At a glance, the only inputs required from the designer are the state machines for token color assignment (cf. Fig. 3) and for the forwarding of outbound token sequences (cf. Fig. 4). Two variants have been generated: 1) a functional implementation realized as a hierarchical actor within the modeling framework, and 2) a bus-cycle-accurate SystemC RTL model. While the former is used for functional verification, the latter is used to obtain performance estimations of the proposed multiplexing approach. In the following, only the SystemC model is discussed in more detail.

In order to obtain a bus-cycle accurate model, certain assumptions about the target platform have been made. Considering a Xilinx Virtex 6 FPGA platform, the various latencies are given in Table I. The compound EP state consists of the actor variables (104 bytes), the current actor mode (16 bits), and a bit vector which indicates the initiative queues where an EP is enqueued to avoid enqueuing the EP to the same



a) Throughput for $m = 12$ reservation stations.



b) Latency for $m = 12$ reservation stations.

Fig. 5. Results for simulation parameters given in Table I. In contrast to $m = 6$ reservation stations (not shown), peak throughput and latency are improved by a factor of approx. 2.6 for test cases with more than 16 EPs.

initiative queue more than once (4 bits). Thus, for 2^{24} EPs, a backing store of ca. 1.6 GB is required for the compound EP state, which is therefore assumed to be mapped to an off-chip memory. Note that the DDR3 RAM in the target platform has an interface data width of 512 bits, i.e., the compound EP state can be written back in two cycles. In order to hide the memory read latency, our implementation uses a direct-mapped cache for the compound EP state. Actor variables are assumed to be kept in registers of each RS. For each FU, a SystemC thread has been instantiated. This allows subsequent high-level synthesis tools to generate parallel modules for each FU.

The various parameters of the evaluated test cases are also given in Table I. For each test case, a certain number of EPs is initialized prior to the posting of WQEs. Then, WQEs are simultaneously posted to the SQ of an EP and to the RQ of its peer EP after the WQE post delay has elapsed for the SQ. A WQE is retired when the corresponding CQE has been generated by the associated EP. The simulation is finished when all WQEs have been retired.

The results are shown in Fig. 5: On the one hand, *throughput* denotes the ratio between the total number of bytes transferred between the EPs and the total number of cycles of the simulation. On the other hand, *avg. WQE latency* denotes the average number of cycles from the posting of a WQE to the receiving of the corresponding CQE. The following observations can be made: 1) The WQE latency does not approach infinity because we post only a limited number of 64 WQEs to each SQ/RQ. 2) A single EP cannot fully utilize the FUs even if the WQE post frequency is increased, resulting in a low throughput. 3) For the minimum number of reservation stations $m = 6$ (not shown), the throughput for 4 and 16 EPs is slightly better than the throughput for more EPs. As only one colored fifo is allocated per colored port in this case, EPs have to be evicted more often from a RS for more than 16 EPs. Eliminating this bottleneck by doubling the number of reservation stations improves both peak throughput and latency by a factor of ca. 2.6 for test cases with more than 16 EPs, because more logical EPs can now be executed concurrently by the FUs of the single physical EP instance. 3) The peak throughput can be sustained even for a large number of EPs in case of 12 reservation stations. 4) Increasing the number of FUs has only a negligible effect on the peak throughput in this case, as it already approaches the theoretical maximum throughput of 4 bytes/cycle for the 32 bit data path.

TABLE I. SIMULATION PARAMETERS

Parameter	**Value**
Number of active EPs	1, 4, 16, 64, 256, 1024, 4096, 16384
Number of WQEs per SQ/RQ	64
WQE size	1024 bytes
IB/PCIe MTU size	256 bytes (64 tokens)
WQE post delay per SQ/RQ	150,000,000 cycles – 1 cycle
Number of reservation stations	6, 12
Number of colored fifos per colored port	1, 4
Number of credits per initiative queue	1
Number of FUs per guard/action function	1
Size of compound EP state cache	1024 EPs
BRAM/FIFO write latency	1 cycle
BRAM/FIFO read latency	1 cycle
Off-chip mem. write latency (EP state)	1 cycle
Off-chip mem. read latency (EP state)	60 cycles
Main mem. write latency (PCIe)	1 cycle
Main mem. read latency (PCIe)	60 cycles
Link delay	20000 cycles

V. SUMMARY

We showed how logical actor instances can be multiplexed onto a single physical actor instance such that the logical actor instances can still be executed concurrently. Based on guarded actions, the proposed approach can be used for dynamic dataflow actors and is therefore applicable to a wide range of applications. We showed that a large number of logical actor instances can be multiplexed without introducing deadlocks into the system by means of an InfiniBand network adapter in a PCI Express system.

This work is supported in part by the German Science Foundation (DFG), HA 4463/3-2 and TE 163/18-2.

REFERENCES

[1] A. Gerstlauer et al. Abstract System-Level Models for Early Performance and Power Exploration. In *Proceedings of ASP-DAC*, pages 213–218, 2012.
[2] T. Grötker et al. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
[3] R. Gu et al. Exploring the Concurrency of an MPEG RVC Decoder Based on Dataflow Program Analysis. *IEEE TCSVT*, pages 1646–1657, 2009.
[4] ITA. *InfiniBand Specification*. InfiniBand Trade Association, 2013. http://www.infinibandta.org.
[5] C. Jäschke and R. Laur. Resource Constrained Modulo Scheduling With Global Resource Sharing. In *Proceedings of the ISSS*, pages 60–65, 1998.
[6] J. Keinert et al. SYSTEMCODESIGNER - An Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications. *ACM TODAES*, 14(1):1–23, 2009.
[7] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE TC*, pages 24–35, 1987.
[8] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt. An Overview of Today's High-Level Synthesis Tools. *DAES*, 2013.
[9] W. Sun et al. FPGA Pipeline Synthesis Design Exploration Using Module Selection and Resource Sharing. *IEEE TCAD*, pages 254–265, 2007.
[10] C. Zebelein, J. Falk, C. Haubelt, J. Teich, and R. Dorsch. Efficient High-Level Modeling in the Networking Domain. In *Proceedings of DATE*, 2010.
[11] C. Zebelein, C. Haubelt, J. Falk, T. Schwarzer, and J. Teich. Representing Mapping and Scheduling Decisions within Dataflow Graphs. In *Proceedings of FDL*, 2013.