Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Transaction-based online debug for NoC-based multiprocessor SoCs [☆]

Mehdi Dehbashi [c,*], Görschwin Fey [a,b]

[a] Institute of Computer Science, University of Bremen, Bremen, Germany
[b] Institute of Space Systems, German Aerospace Center (DLR), Bremen, Germany
[c] Infineon Technologies AG, Munich, Germany

ARTICLE INFO

Article history:
Available online 11 March 2015

Keywords:
Transaction-based online debug
System-on-Chip (SoC)
Network-on-Chip (NoC)

ABSTRACT

As complexity and size of Systems-on-Chip (SoC) grow, debugging becomes a bottleneck for designing IC products. In this paper, we present an approach for online debug of NoC-based multiprocessor SoCs. Our approach utilizes monitors and filters implemented in hardware. Monitors and filters observe and filter transactions at run-time. They are connected to a Debug Unit (DU). Transaction-based programmable Finite State Machines (FSMs) in the DU check assertions online to validate the correct relation of transactions at run-time. The experimental results show efficiency and performance of our approach.

1. Introduction

Modern high-performance Systems-on-Chip (SoC) include many IP cores such as processors and memories. Network-on-Chips (NoC) have been proposed as a scalable interconnect solution to integrate large multiprocessor SoCs [2] [3]. Having a large SoC with complex communication among its cores, achieving complete verification coverage at pre-silicon stage is almost impossible. Therefore in addition to electrical bugs, some design bugs may also appear in the final prototype of an SoC.

The idea of transaction-based communication-centric debug is introduced in [4] to debug complex SoCs which interact through concurrent interconnects such as NoC. The transactions are observed using monitors [5] and the debug control unit can control the execution of the SoC (stopping, single stepping, etc.). In [6], transactions are stored at run-time in a trace buffer using on-chip circuits. After an SoC run, the content of the trace buffer is read and analyzed offline with software. The analysis software tries to find certain patterns [7] in the extracted transactions that are defined by their *Transaction Debug Pattern Specification Language* (TDPSL). Because of limited size of a trace buffer, getting an execution trace of the transactions related to the time of bug activation is a challenging problem. To overcome this problem, the content of the trace buffer is utilized to backtrace the transactions along their execution paths [8]. The backtracing is performed in transaction-level states using *Bounded Model Checking* (BMC). However, backtracing needs formal pre-image computations which can blow up for large and complex designs [9]. To address this problem, we need to have online detection to stop the SoC close to the time of bug activation at the transaction level.

In this paper, we present a transaction-based debug infrastructure which can be used not only for online debug and online system recovery but also for interactive debug in which an external debug platform programs the FSMs and the filters according to the considered assertions at each round of debugging. Our hardware infrastructure contains monitors, filters, and a debug network including *Debug Units* (DU). Filters and DUs are programmed according to the transaction-based assertions defined by TDPSL. Transactions are monitored only at master interconnects. Slaves send information to masters. This redundant information is used to observe the elements of transactions online. No modification of the internal components of the NoC is required. At run-time the programmable FSMs in the DUs investigate the assertions online and detect an error. Upon detection of an error, the DU recovers the SoC by informing the masters which have participated in the observed error. Then, the corresponding masters start the recovery process at run-time. Also we identify the requirements which a debug infrastructure has to fulfill in order to perform transaction-based online debug.

The main contributions of this paper are as follows:

– Introducing a debugging infrastructure to transaction-based online debug of NoC-based SoCs without modifying the internal components of the corresponding NoC (non-intrusive to the NoC).

☆ Mehdi Dehbashi did this work as part of his PhD in the University of Bremen. This work has been supported in part by the University of Bremen's Graduate School SyDe, funded by the German Excellence Initiative and in part by the German Research Foundation (DFG, grant no. FE 797/6-1). This paper is an extended version of [1].

* Corresponding author.
E-mail address: fey@informatik.uni-bremen.de (G. Fey).

– Analyzing and finding transaction-based debug patterns at-speed using debug units including programmable filters and FSMs.

– Presenting an ordering mechanism in the routers of the debug network to order the transactions online.

– Online system recovery without stopping and interrupting the NoC.

The experimental results show the efficiency of our approach using different assertion patterns defined by TDPSL such as race, deadlock, and livelock. An NoC-based SoC using a mesh network is setup in the Nirgam NoC simulator [10] to evaluate our approach. Also we show the effectiveness of the proposed online recovery in the experimental results.

The remainder of this paper is organized as follows. Related work is discussed in Section 2. Section 3 introduces preliminary information on transactions and TDPSL. Our debug method including hardware and software parts is explained in Section 4. The debug patterns and their corresponding FSMs are explained in Section 5. This section also presents experimental results on an NoC-based SoC. The last section concludes the work.

2. Related work

Previous work also considered infrastructures for SoC debug. The existing debug infrastructures for complex SoCs are reviewed in [11]. These infrastructures support debugging such that the internal nodes become observable and controllable from the outside. The work in [12] presents a Design-for-Debug (DfD) technique for NoC-based SoCs. The technique enables data transfer between a debugger and a *Core-Under-Debug* (CUD) through the available NoC to facilitate debugging. A debug platform to support concurrent debug access to the CUDs and the NoC in a unified architecture is proposed in [13,14]. This platform is realized by introducing core-level debug probes in between the CUDs and their network interfaces and a system-level debug agent. The work in [15] proposes a ring-based NoC architecture to debug SoCs. The NoC is used to send back the information observed by monitors to the debugger. A *Non-Uniform Debugging Architecture* (NUDA) is proposed in [16] to debug many-core systems. A NUDA node in each cluster has three main parts: nanoprocessor, memory and communication. The NUDAs are distributed across a set of hierarchical clusters and are connected to each other through a ring interconnection. Then the address space is monitored using non-uniform protocols for race detection. Monitoring the address space without abstraction consumes a large storage and increases the latency of the error detection.

NoC test and diagnosis is the main focus in most of the previous work. Packet address driven test configurations are utilized in [17] to test and to diagnose regular mesh-like NoCs using a functional fault model. Then, link faults are diagnosed using test results and a diagnosis tree. The system test is modeled at the transaction level in [18] in order to facilitate test design space exploration, as well as the validation of test strategies and schedules. Interconnect faults in Torus NoCs are detected and diagnosed using BIST structures in [19]. Afterwards, the NoC is repaired by activating alternative paths for faulty links. In [20] an NoC with a faulty router or a broken link is repaired using spare routers. The inherent structural redundancy of the NoC architecture is exploited in a cooperative way to detect the faults using BIST [21]. Also diagnosis units in switches are utilized to localize a fault. In the diagnosis unit there are different comparators to compare data from all the possible pairs of switch input ports. A comprehensive defect diagnosis for NoCs is proposed in [22]. The approach uses an end-to-end error symptom collection mechanism [23] to localize datapath faults and a distributed counting and timeout-based technique to localize faulty control components [22]. The work in [24] diagnoses the NoC switch faults using hardware redundancy in each switch and a high level fault model. These approaches focus only on electrical bugs in the components of an NoC. However we consider design bugs which influence communications in an NoC-based SoC.

The work in [4] proposes a communication-centric debug approach. The approach focuses on the communication and the synchronization between the IP cores. Their approach uses not only monitors on the IP interconnects but also monitors on the internal components of an NoC such as routers. Debug data is read-out using scan chains and *Test Access Ports* (TAP). In our work, we do not transfer the debug data out of the SoC. The debug data is analyzed online using debug units. Also we monitor only the master interconnects without modifying the internal components of an NoC. The work in [6] uses trace buffers to store the transactions at run-time. The content of the trace buffer is analyzed offline in order to form transactions and to find debug patterns. Their approach monitors the bus to store the events in the trace buffer. However, we present an approach to debug NoC-based SoCs. We form the transactions online using distributed monitors and debug units. Also the debug patterns are found at run-time.

3. Preliminaries

3.1. Transaction

In this section we shortly explain the transaction elements from [25,6]. Each transaction includes a request and a response. Masters request and slaves respond. Each transaction has four basic elements: *Start of Request (SoRq)*, *End of Request (EoRq)*, *Start of Response (SoRp)*, and *End of Response (EoRp)*. In TLM, SoRq corresponds to putting the request in the channel by the master. EoRq is getting the request by the slave. SoRp corresponds to putting the response in the channel by the slave. EoRp is getting the response from the channel by the master. Also there are two additional elements which are called: *Request Error* (ErrRq) and *Response Error* (ErrRp). These elements handle error conditions and correspond to any kind of error that causes a request or a response to fail.

3.2. Transaction Debug Pattern Specification Language (TDPSL)

TPDSL has three layers: Boolean layer, temporal layer, and verification layer [6]. The Boolean layer includes *trans_exp* which represents *the basic elements of transactions*. The *trans_exp* format is as follows:

$trans\_type\ (master, slave, type, address, tag)$

Field *trans_type* can be any transaction element mentioned in Section 3.1 as well as the *Start of Transaction* (SoTr) and the *End of Transaction* (EoTr) which are similar to SoRq and EoRp respectively. Fields *master* and *slave* specify the ID of master and slave. Field *type* can be *Rd* or *Wr*. Field *address* indicates the slave address symbolically as SAME, SEQ, and OTHER. Field *tag* indicates the transaction number and is only used for buses that allow non-blocking requests and out-of-order responses [6]. In our paper, we show a transaction without considering the field *tag*.

The motivation to use symbols for the address field is to abstract and to compress the address bits. In this case, only the compact address information is stored or sent via network for debugging. The symbols can be defined with respect to the application and the granularity of debugging. SAME specifies that in the current transaction, slave address is same as the address in the previous transaction for this slave. SEQ specifies that in the current transaction, slave address has one word difference with the previous address for this slave. OTHER specifies that in the current transaction, the slave address in neither SAME nor SEQ.

For example transaction *EoTr* $(m1, s2, Rd, -)$ represents the end of a read transaction from master $m1$ to slave $s2$ with any address. The symbol "$-$" indicates that we leave the corresponding field as don't care.

The properties in terms of transaction sequences are defined at the temporal layer. Different operators are utilized at this layer such as concatenation operator $(;)$, fusion operator $(:)$, or operator $(-)$, and operator $(\&)$, and repetition operators [6]. In the verification layer, the *assert* statement is defined. Also a *filter* can be defined which specifies a filter over the execution path for the evaluation of the assertion statement.

Following is an example of a simple assertion in TDPSL:

$assert\ never$
$EoTr\ (m2, s1, Wr, -);\ SoTr\ (m1, s1, Rd, -)$

This assertion specifies that start of a read transaction from master $m1$ to slave $s1$ must never be directly after the end of a write transaction from master $m2$ to slave $s1$.

4. Debug method

Transaction level online debug aims at improving the observability and the controllability of the system. Whenever transactions conform to certain debug patterns, an error is detected. In this case, the *Debug Unit* (DU) sends the debug packets to the SoC nodes in order to control the network and to recover from the error state.

We have some requirements to enable transaction-based online debugging:

1. Our debug infrastructure has to be able to collect the elements of each transaction at run-time.
2. We have to be able to order the transactions online.
3. We need to assert debug patterns, i.e., the relation of transactions, at run-time.

If a debug infrastructure fulfills the three mentioned requirements, it can be used for transaction-based online debug.

In the following, we explain our debug infrastructure fulfilling the above mentioned requirements. To collect all elements of each transaction in a system based on NoC (first requirement), we need distributed monitors and *Debug Redundant Information* (DRI). Monitors and DRI are explained and discussed in Sections 4.1 and 4.2, respectively.

The transaction ordering mechanism in the *Debug Units* (DU) is responsible to order transactions (Section 4.4) fulfilling the second requirement. A DU is the main part of the debug infrastructure which searches for certain debug patterns in the received transactions. We use a tree-based debug network structure in which all monitors have a short distance to DUs. In the debug network, the transactions are ordered using DUs. The ordered transactions are transferred on each link of the debug network from bottom to top such that the ordered transactions can be utilized in each level of the debug network for hierarchical and assertion-based debug.

FSMs in DUs are utilized to investigate transaction-based assertions at run-time to fulfill the third requirement (Section 4.6). The filters in DUs and in monitors help FSMs by dropping unrelated transactions (Section 4.5).

Fig. 1 shows the hardware infrastructure of our approach for an SoC including four IPs, 2 masters and 2 slaves. The debug infrastructure has the following parts: monitors, filters, DU, and DRI.

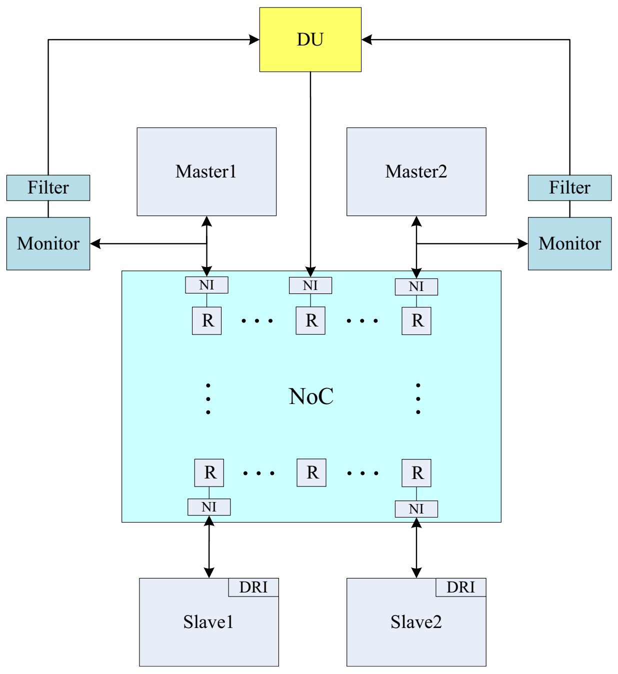


Fig. 1. Debug infrastructure.

The internal structure of a DU has also three main parts: transaction ordering, filter, and FSM. Each IP in Fig. 1 is connected via a *Network Interface* (NI) to the NoC.

Fig. 2 shows the tree-based debug infrastructure. The lowest level of the infrastructure includes monitors and filters. Monitors are connected to *Master Interconnects* (MI) to observe the transactions. A *Central Debug Unit* (CDU) is only at the top level. The other levels have *Local Debug Units* (LDU). LDU and CDU structures are explained in Section 4.3.

4.1. Monitor

Monitors extract the basic elements of a transaction as mentioned in Section 3.2. They observe master interconnects to enable transaction-based debug [4]. In a packet-based protocol in an NoC, we can immediately extract the elements *master*, *slave*, and *type* by observing the master interconnects. But to extract the element *address* as SAME, SEQ, and OTHER, which is a comparison of the

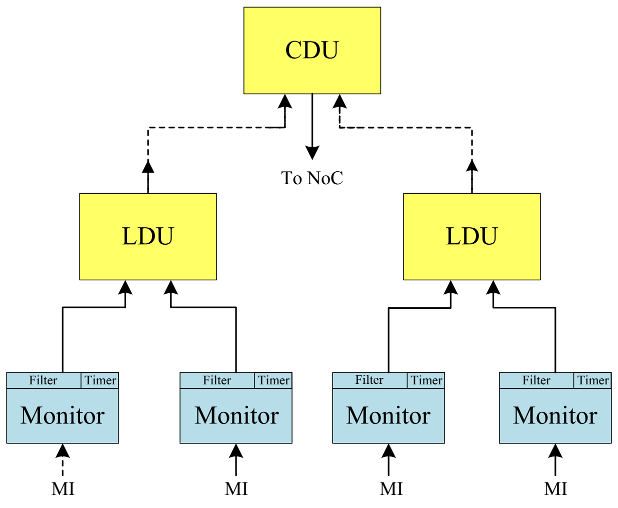


Fig. 2. Tree-based debug infrastructure.

slave address in the current transaction and the previous transaction for the corresponding slave, we need some DRI. The next section explains the DRI.

A monitor in our infrastructure observes a master interconnect and signals a matching transaction expression explained in Section 3.2 as an output.

Each monitor includes also a timer. The timer is used to attach a timestamp to each observed transaction. The timestamp attached to a packet is utilized at DUs to order the transactions arriving from the left and right input links of DUs. As the transactions are consumed online using FSMs, large timestamps are not required. Timestamps only need to distinguish the order of transactions arriving at DUs.

In an SoC with asynchronous IPs, the CDU sends synchronization packets to monitors. The timers in monitors are synchronized according to the synchronization packets. As only the CDU sends the packets from the top level to the bottom level in the tree-based debug network, the delay of synchronization packets arriving at monitors are predetermined. In this case, the time in monitors is synchronized with the time in the CDU by incrementing the CDU time included in the synchronization packet with the delay of synchronization packet.

Another approach to synchronize the transactions is using the *relative timestampes*. In this approach, the time of each transaction is calculated in comparison to the time of the front transaction in the debug network. Then, this relative timestamp is attached to the corresponding transaction. To use this approach, some timers are required in LDUs and CDU.

4.2. Debug Redundant Information (DRI)

DRI is used to extract and to transfer the element *address* of a transaction. We can form the element *address* using slaves (slave-based approach) or using debug units (DU-based approach). In the following we discuss these two approaches.

In a slave-based approach, the element *address* is formed in the slaves and is sent as redundant information to masters through the NoC. Because the element *address* is a comparison of the address of the current transaction with the address of the previous transaction for the corresponding slave, this comparison can be simply done in each slave.

The slave should send two bits redundant information to masters. These two bits specify the symbols SAME, SEQ, and OTHER. We can also use more symbols for the slave address to have more accurate data depending on the applications running on the SoC. We use the slave-based approach to detect deadlock and livelock.

The DRI section in each slave in Fig. 1 compares the slave address of the current transaction with the slave address of the previous transaction in the corresponding slave. Then the DRI section selects a symbol (SAME, SEQ, or OTHER) and adds this symbol in the response packet as two redundant bits. On the master side these two bits are read by monitors to constitute a complete transaction expression. These two redundant bits are used only in monitors. The master applications should ignore these two redundant bits.

In this case, we can have the address information for the corresponding slave only in the *EoTr*. The element *address* is not available in *SoTr*. To have this information, we should wait to receive an *EoTr* by the corresponding slave.

The second approach to form the element *address* uses debug units, i.e., DU-based approach. In this approach, the slave addresses are observed by monitors and sent to the DUs. In the DU, there is one address register for each slave. The address registers keep the address of the previous transaction for each slave independently. When a new transaction is performed, the content of the address register related to the corresponding slave is compared to the new transaction address. Then the symbols SAME, SEQ, and OTHER are derived and the address register is updated to keep the slave address in the latest transaction for the corresponding slave. We use the DU-based approach to detect races.

In the DU-based approach, the element address is available for both *SoTr* and *EoTr*. The DU-based approach needs more memory in the debug units storing slave addresses. Also it needs more bandwidth for the debug network to transfer slave addresses to DUs. The advantage of this approach is being non-intrusive to the SoC.

4.3. Debug Unit (DU)

A debug unit can be an LDU or a CDU. A CDU is used at the top level in the tree-based debug network (Fig. 2). An LDU is used in other levels of the debug network. The structure of an LDU is suitable to build a tree-based network. An LDU has three ports (Fig. 3(a)): top port T, left port L, and right port R. The right and left ports transfer the data observed by monitors to the top level. Also the synchronization packets sent by the CDU are transferred from the top port to the left and right ports reaching timers. The CDU controls the traffic of the packets sent from the top level to the leaves in the debug network. In this case, we use only one buffer in each LDU to transfer synchronization packets.

The packets arriving at the inputs of the right and left ports are stored in the right and left FIFOs. Then the transaction ordering selects a transaction packet such that the transactions are ordered based on their timestamps. The filter does not allow a transaction to be forwarded if the transaction is not related to the considered assertions. The related transactions regarding the considered assertions are used in the FSM to investigate the assertions. If an assertion fails, an error message is sent to the CDU.

The CDU is used at the top level of the debug network. The CDU has two additional tasks: synchronizing the timers and handling error cases (Fig. 3(b)). Synchronization is performed by part *Synch* in Fig. 3(b). When there is an error, the error handler in the CDU manages the network by sending some debug packets to other nodes in the SoC. The CDU is connected to the NoC communicating with other nodes in the network. In this case, the CDU can send the error state to all nodes or some special nodes in the network in order to collect more accurate debug information or to recover the SoC from the error state.

Fig. 4 shows the CDU procedure to recover the SoC from an error. At the step of pattern detection, the CDU checks the debug patterns at run-time. If the CDU detects an error, the second step is started. In this step, the CDU sends a recovery packet to the

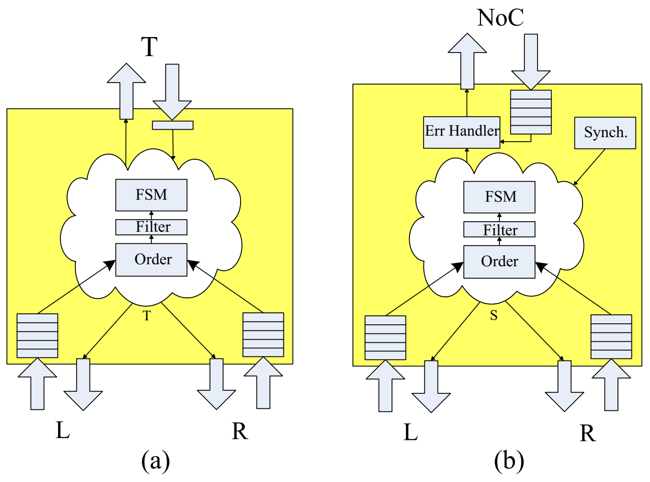


Fig. 3. (a) LDU structure and (b) CDU structure.

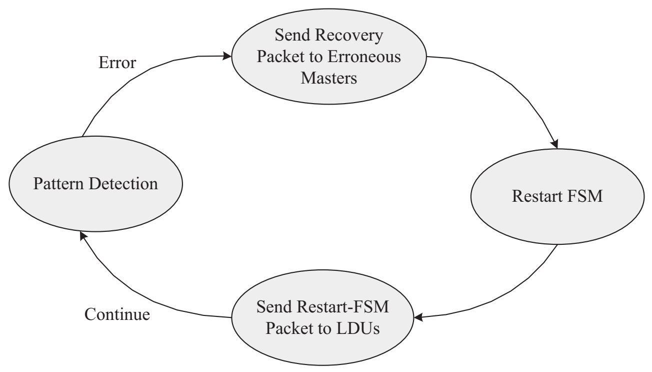

Send Recovery Packet to Erroneous Masters
Error
Restart FSM
Pattern Detection
Send Restart-FSM Packet to LDUs
Continue

Fig. 4. CDU procedure to recover the SoC.

masters which have contributed to the observed error. A recovery packet contains an error type and additional information helping the masters to start a recovery process. In the third step, the CDU restarts its own FSM. Then, the CDU sends a restart packet to the LDUs restarting the LDU-FSMs. Afterwards, the procedure continues with the step of pattern detection.

Fig. 5 shows an example for a master recovery process in the case of a software deadlock. In this case, when a master receives a recovery packet from the CDU with the error type deadlock, the master releases the locked resources. Then the master waits for a random time and proceeds its main function again. With this procedure, the system is recovered online from the error state without stopping and interrupting the NoC.

DUs include FSMs to investigate transaction-based assertions at run-time. To check an assertion in an efficient way, we need to program both LDU-FSMs and CDU-FSM. Distributed online assertion checking can be performed through programming the FSMs in different levels.

4.4. Transaction ordering

When there is more transaction traffic on one link than another link in the debug network, some early-generated transactions are accumulated and buffered in the FIFO of the corresponding DU. This case may also occur when the bandwidth of the debug network is less than the bandwidth of the NoC. When there are some transactions in the FIFO of the left link which have been generated earlier than the transaction available in the FIFO of the right link, the transaction of the right link has to wait until the transactions with smaller timestamps on the other link have been transferred. By comparing the timestamp of a packet in the left FIFO and the right FIFO, the packets are ordered based on their generation time.

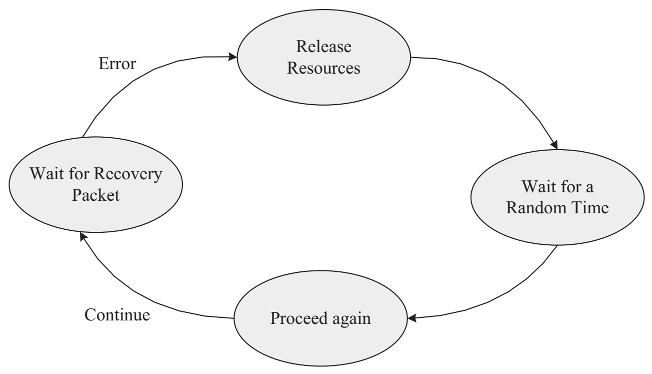


Fig. 5. An example for master recovery thread in the case of a software deadlock.

The length of the timestamp depends on the worst case delay of the debug network. A timestamp should only be able to distinguish the packets based on the time in which they have been generated or sampled. In Fig. 3(a) and (b), the part *Order* in DU compares the timestamps of a packet in the right and the left FIFOs and selects a packet which has a smaller timestamp.

The size of the left and right FIFOs may influence the accuracy of the debug pattern detection because if the FIFO becomes full, some transactions are lost. In this work, we assume that the size of the FIFOs is sufficient to process the transactions.

4.5. Filter

A filter is located in monitors and DUs. A filter is used to filter unrelated transactions in a trace. In this way, the debug unit receives only the related transactions for the assertion statements. Filtering can be done over all parameters of a transaction expression, i.e., *trans_type*, *master*, *slave*, *type*, and *address*. The filter is programmable according to the main assertion statements.

Fig. 6 shows the filter structure. A filter has five fields: *trans_type*, *master*, *slave*, *type*, and *address*. In the fields *master* and *slave*, multiple IDs can be stored. When a transaction is received from the ordering section, the received transaction is compared to all fields of the filter. If at least one element of the received transaction is equal to its corresponding field in the filter, the transaction is considered as unrelated transaction and is not passed to the FSM. Otherwise, the transaction is transferred to the FSM as a related transaction.

If a special field in the filter is not used, the corresponding field should be programmed such that its value becomes always different from the corresponding transaction element. In this case, the output of the inequality operation for that field in Fig. 6 becomes always 1 which is a non-controlling value for the AND gate. In this case, the value of the corresponding field in the filter is called *don't care value*.

4.6. Debug FSM

Debug FSMs are programmable FSMs which are utilized in debug units to investigate the assertions online. Debug FSMs include local FSMs and global FSMs verifying local assertions and global assertions. Debug units can be programmed to implement distributed FSMs validating online assertions in different levels. To do this, first the transaction-based assertions should be analyzed based on their locality in the corresponding SoC. We need to know which task is running on which IP. Accordingly, the filters should be programmed and the assertions should be distributed among debug units (LDUs and CDU).

To implement programmable FSMs, we use lookup-table memory [26] which can be programmed in every debug round according to the new assertions. Fig. 7 shows the structure of an FSM using lookup-table memory. As shown in Fig. 7, in this method the current state bits and inputs are connected to the address bus of a memory. The next state is taken from the memory output. The correct next states have to be stored in each location of the memory to ensure the correct operation [26].

The number of bits for the current state depends on the total number of states in an FSM. For input bits we can connect all elements of a transaction directly to the input lines of the lookup-table memory. In this case, if element *trans_type* has 1 bit (*SoTr* and *EoTr*), element *master* has 2 bits (4 IDs), element *slave* has 2 bits (4 IDs), element *type* has 1 bit (*Wr* and *rd*), and element *address* has 2 bits (SAME, SEQ, and OTHER), totally we need 8 bits for the input part of the FSM. To decrease the number of input bits, we can firstly encode the transactions. Then we connect the encoded

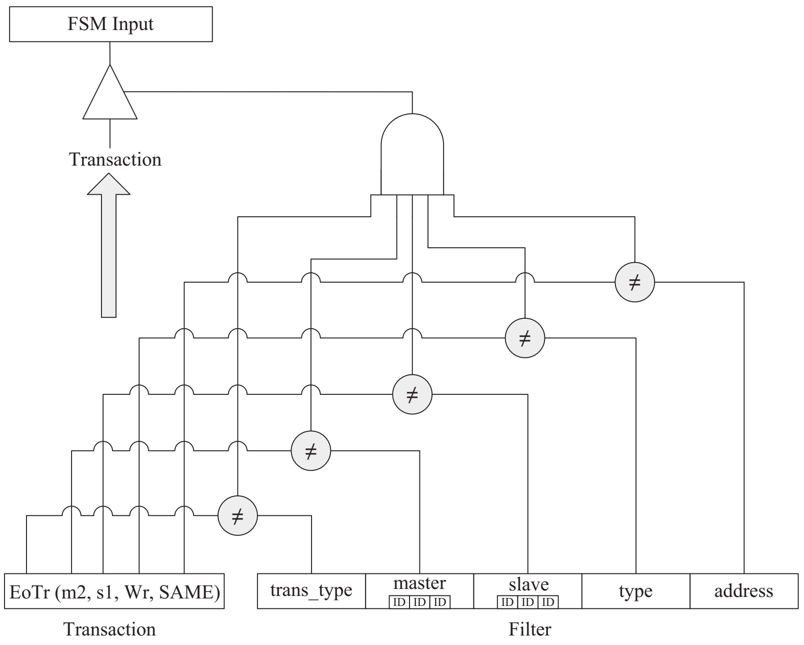


Fig. 6. Filter structure.

Address		Data
Current State	Input	Next State
Start	Tr1	A
Start	Oth	Start
A	Tr1	A
A	$Tr2_1$	B
A	Oth	Start
B	Tr1	A
B	Tr3	C
B	Oth	Start
C	Tr1	A
C	$Tr4_1$	D
C	Oth	Start
D	Tr1	A
D	$Tr5_1$	E
D	$Tr6_1$	F
D	Oth	Start
E	Tr1	A
E	$Tr6_1$	Err
E	Oth	Start
F	Tr1	A
F	$Tr5_1$	Err
F	Oth	Start
Err	-	Err

Fig. 7. FSM implementation using lookup-table memory.

transaction to the input of the memory. As shown in Fig. 8, to encode a received transaction, we use some programmable *transaction patterns*. In each transaction pattern, all elements of one transaction are specified and stored. A received transaction is compared to the transaction patterns. If it is equal to one transaction pattern, the output of the corresponding AND gate becomes 1. Then an encoder converts 2^n bits input to n bits output. In Fig. 8, the encoder has 4 bits input and 2 bits output. The output of the encoder is connected to the input of the lookup-table memory. The goal of the encoder is to reduce the size of lookup-table memory, while the goal of the filter is to discard a group of unrelated transactions.

In this paper, the size of the lookup-table memory, i.e., the number of address bits and data bits in Fig. 7, and the number of transaction patterns in Fig. 8 are used to measure the area overhead of the assertion-based FSMs.

The worst case size of the FSM memory can be estimated based on the worst case number of transactions t and the worst case number of states s. In this case, the input of the address requires $\lceil \log_2 t \rceil$ bits and the current state of the address requires $\lceil \log_2 s \rceil$ bits. Totally the address of the FSM memory requires $\lceil \log_2 s \rceil + \lceil \log_2 t \rceil$ bits (Fig. 7). The data of the FSM memory requires $\lceil \log_2 s \rceil$ bits. Totally a memory with $(\lceil \log_2 s \rceil + \lceil \log_2 t \rceil)^2 * \lceil \log_2 s \rceil$ bits is required.

In Fig. 8, field M is a mask bit for the address field. When M is 1, the address is considered as don't care, and the corresponding input of the AND gate becomes 1. One mask bit can be utilized for each field of a transaction pattern. However in Fig. 8 we show a mask bit only for the address field.

A complex assertion including many master and slave IDs may increase the size of the FSM. To handle this case, a complex assertion should be divided into different parts such that each part is checked by its respective LDU. Main FSM structure can be implemented in the CDU. The main FSM has some sub-FSMs. Each sub-FSM is implemented in its respective LDU. The LDUs send the state of the sub-FSMs to the CDU updating the state of the main FSM.

4.7. Design decisions and limitations of the approach

We use a tree-based network to connect the DUs. In a tree-based network, all of the leaves have a short distance to the CDU. Therefore, the detection latency of an error is short as all transactions can arrive at the CDU in a short time. The latency of the debug network depends on the traffic of transactions and the number of hops (levels) from the leaves to the CDU. If the number of monitors is m, the tree debug network has $l = \lceil \log_2 m \rceil$ levels. In Fig. 2, the number of monitors (the number of masters) is $m = 4$ and the debug network has 2 levels. Using a tree-based debug network

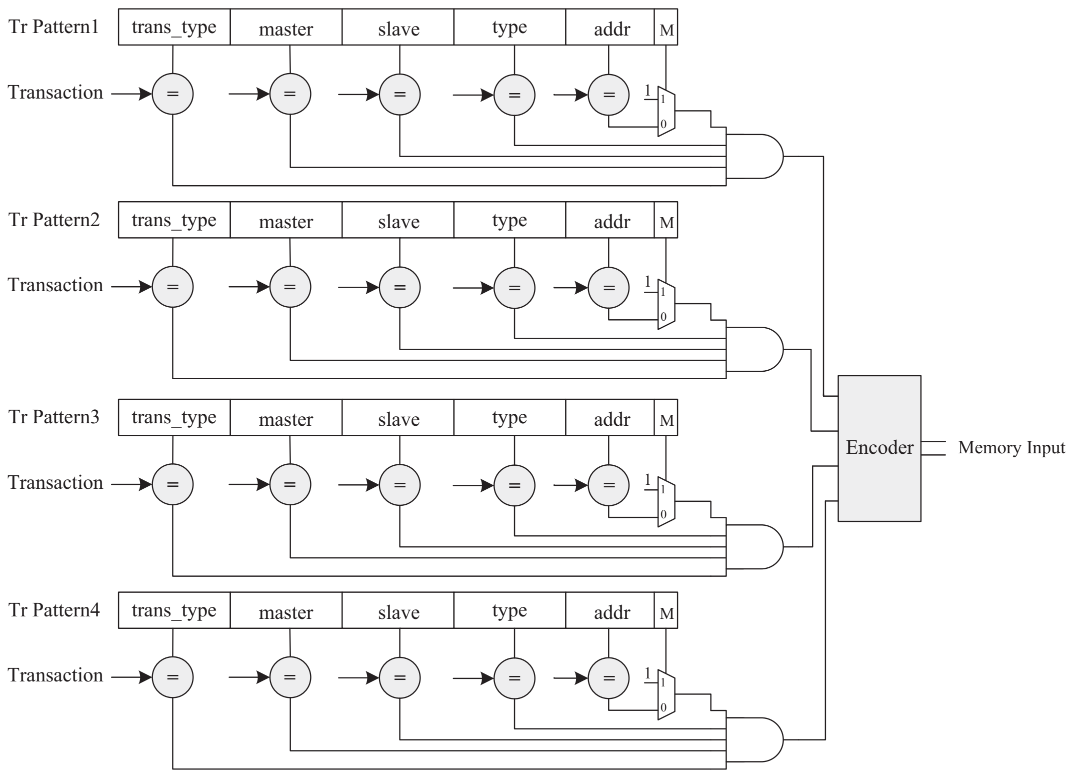


Fig. 8. Encoding of transaction patterns.

needs a large number of DUs (routers) when the number of masters increases. When the number of masters m is a power of two (2^n), the number of DUs in a tree-based network is $\#DU = \sum_{i=1}^{l} m/2^i = m - 1$. In each level i of the debug network, the number of DUs is $m/2^i$.

For the topology of the debug network, other alternatives can be used in order to decrease the hardware cost. One of the topologies which has been used for a debug network is ring topology [16,15]. Using a ring topology for our approach decreases the number of DUs (routers) in the debug network. In the ring network, the number of required DUs is $\#DU = \lceil m/2 \rceil + 1$. We assume that each LDU can be connected to at most two monitors. The LDUs are also connected to each other in a serial manner. A CDU closes the ring. One disadvantage of the ring topology is that some LDUs have a close distance to the CDU while some other LDUs have a far distance from the CDU. The ring debug network requires some special approaches to balance the traffic in the network [15]. In the ring network, the CDU has to wait until all of the required transactions from all the LDUs have arrived. Afterwards, the CDU can check the assertions. In the ring topology, the worst case number of hops from an LDU to the CDU is $l = \lceil \#LDUs/2 \rceil = \lceil m/4 \rceil$ which influences the latency of error detection in the CDU. In this case, as the CDU has to wait for the transactions of the longest path (distance), the CDU needs larger FIFOs storing all the transactions which have arrived through the shorter paths.

Another challenge is ordering the transactions of the FIFO by the CDU based on the timestamps. In [16], a nanoprocessor is used in order to investigate the transactions and their timestamps. However, in the tree network, the transactions are automatically ordered by a simple hardware mechanism from the bottom level to the top level such that at every level of the debug network, the ordered transactions are available.

The selection of the topology of the debug network is a trade-off between hardware cost and diagnosis latency. Using a hybrid topology may alleviate hardware costs and may help achieving a reasonable diagnosis latency. A tree network is suitable for hierarchical debugging while a ring network is suitable for local debugging. The many-core systems can be divided into different clusters [16]. In each cluster, a ring sub-network can be used to evaluate the local assertions. The local ring sub-networks are connected to each other using a tree topology evaluating the global assertions.

The functionality of filters in tree-based debug networks has some limitations. In the debug network, each LDU has two subnetworks, i.e., right subnetwork and left subnetwork. In this case, a filter in an LDU can only be applied on the transactions which arrive from the masters being at the lower levels of the subnetworks of the corresponding LDU. The LDUs in the same level cannot communicate with each other. Therefore, each LDU can only check the assertions which are related to the masters falling in the leaves of the corresponding LDU. Only the CDU can have a comprehensive assertion checking the transactions of all masters.

In order to use our approach, first the address space of slaves has to be categorized in an offline step. In this step, symbols are assigned to each relevant address range. Also, the ID of each IP relevant for the assertion of interest is determined in the offline step. The IDs are required in order to write a transaction expression.

In our approach, an FSM is derived from assertions. Then, the FSMs are distributed among LDUs. Each LDU includes the FSMs which are relevant to the masters being at its lower levels. A DU can have multiple FSMs running in parallel. Parallel FSMs also help checking assertions for different address spaces. In this case, each FSM can have its own filter. The following experiments show the application of our approach and indicate the hardware overhead required.

A hardware fault in the NoC may cause an EoTr failing to reach the corresponding master. In this case, the CDU detects an error. But our approach cannot recover the NoC from the error. The recovery example of Fig. 5 is suitable for software deadlocks. In the case of a hardware fault in the network which leads to a wrong routing and consequently results in a deadlock; the recovery algorithm has to select a new route avoiding the faulty route. In this case, the recovery algorithm can be enhanced using a diagnosis approach such as [22] to localize faulty components in the network.

Our approach detects the errors which are related to the transactions between the cores. But it does not detect the deadlocks which may happen between different threads of a single core.

5. Implementation

For the experiments we setup a 3×3 mesh network in the Nirgam NoC simulator [10]. Nirgam is a cycle-accurate simulator which is implemented in SystemC language. We have simulated the system for one million cycles. During the run-time of the SoC, our debug infrastructure asserts the debug FSMs which are mentioned in the next sections. We have implemented dining philosophers [6] and a random application [8] as example applications. In the random application, each master waits for a random time. Then the master selects a random list of slaves as resources. If the master can lock all the required resources, the processing is started. Afterwards, the resources are released and the procedure is repeated. If the master cannot lock all the required resources, the master waits for a random time and tries again [8].

In our experimental setup, the SoC has four masters (philosophers) and four slaves (chopsticks) which are divided into two groups communicating in parallel. Each group has two masters and two slaves (first group: $m1$, $s1$, $m2$, $s2$. second group: $m3$, $s3$, $m4$, $s4$). Four monitors are used to observe the master interconnects. Also two LDUs and one CDU constitute the debug network. In the following sections, we discuss debug patterns for race, deadlock, and livelock as an example. In each section, we also discuss the hardware and storage costs in terms of the size of lookup-table memory and the number of required transaction patterns. In the slave-based approach in which the address symbol is transferred by each slave, the bandwidth of the debug network is 8 bits/cycle, as a transaction has 8 bits: 1 bit for element *trans_type* (*SoTr* and *EoTr*), 2 bits for element *master* (4 IDs), 2 bits for element *slave* (4 IDs), 1 bit for element *type* (*Wr* and *Rd*) and 2 bits for element *address* (SAME, SEQ and OTHER). In the DU-based approach, the debug network needs a higher bandwidth, as the slave address is transferred by the debug network (Section 4.2). In this case, the bandwidth of the debug network has to be $(6 + \#Address\_Bits)/cycle$.

If the filter is implemented with one ID in the field *master* and one ID in the field *slave* (Fig. 6), the size of the filter is same as the size of a transaction, i.e. 8 bits. In our experiments we considered two IDs in the field *master* and two IDs in the field *slave*. As each additional ID requires 2 bits, the filter requires $8 + 2 + 2 = 12$ bits. In addition, the filter requires 12 XOR gates with two inputs for the inequality operations and 1 AND gate with twelve inputs.

The size of the encoder structure depends on the maximum number of transaction patterns. In our experiments the maximum number of transaction patterns is 8. The size of a transaction pattern is the size of a transaction (8 bits) plus 1 mask bit as shown in Fig. 8. Therefore, the encoder structure requires $8 * 9 = 72$ bits. In addition, the encoder structure requires 8 multiplexers, 8 AND gates with eight inputs, 64 XNORs with two inputs and one 8-to-3 encoder. For the experiments we used a FIFO with the capacity of 5 transactions. The size of the FIFO is $5 * 8 = 40$ bits.

5.1. Debug pattern for race

A race may occur when one write transaction to the same place occurs during the previous write. In TDPSL this case is written as follows:

$$
\begin{array}{l}
assert\ never\{\\
SoTr(m1, s1, Wr, -);\ SoTr(m2, s1, Wr, SAME);\\
EoTr(m1, s1, Wr, SAME)\\
\}filter(*, *, *)
\end{array}
$$

Filtering is done on the three first parameters of transaction expressions. Sign $*$ in the filter means only the related transaction types, masters, and slaves should be considered. Therefore the transactions related to slave $s2$ are omitted. Also all transactions related to the second group, i.e. group of master $m3$ and master $m4$, have to be omitted. In our infrastructure, the filters are programmed such that the transactions related to slave $s2$, master $m3$ and master $m4$ are filtered online. The field *slave* in the filter of Fig. 6 has to include the slave ID 2. The field *master* has to include the master ID 3 and 4. Other fields in the filter have to have don't care values.

As explained in Section 4.2, the DRI can be transferred using the slave-based approach or the DU-based approach. As in the race assertion we need the element *address* in *SoTr*, therefore we can only use the DU-based approach to investigate this assertion. In the slave-based approach, we can have the element *address* only for *EoTr*. If we use the slave-based approach, we need to change the race assertion such that only *EoTr*s includes element *address*. But this case causes some latency in the detection of the assertion violation.

To build the FSM, first we sign each transaction with a unique number. For example, transaction $SoTr(m1, s1, Wr, -)$ is written as $T1$. The address field in each transaction can be $-$, SAME, SEQ or OTHER. We use an index for each transaction in order to distinguish different cases in the field *address*. We use index 1–3 for SAME, SEQ, and OTHER, respectively. For example $SoTr(m1, s1, Wr, SAME)$ is written as $T1_1$. Transactions $T1_1$, $T1_2$, and $T1_3$ are a subclass of transaction $T1$. In this case, the race assertion is written as $T1\,T2_1\,T3_1$.

The race FSM is implemented using four states which need 2 bits from the address line of the lookup-table memory. Also there are three transactions T_1, $T2_1$, and $T3_1$. To program a transaction pattern in Fig. 8 by $T1$, the mask bit of the corresponding transaction pattern is set to consider the address field as don't care.

By using the encoding structure of Fig. 8, three transactions are encoded into 2 bits as input bits of the memory. Totally the race assertion needs 4 bits (2 bits current states + 2 bits input) for the memory address and 2 bits for the memory data. Also 3 transaction patterns are required (Race Pattern 1 in Table 1).

To increase the verification coverage of the FSM, we need to have a more comprehensive pattern. In the following, we write an improved race pattern to cover more race conditions happening on slave $s1 : T_1\,T2_1\,T3_1\,|T2\,T1_1\,T4_1$:

$$
\begin{array}{l}
assert\ never\{\\
SoTr(m1, s1, Wr, -);\ SoTr(m2, s1, Wr, SAME);\\
EoTr(m1, s1, Wr, SAME)\\
|SoTr(m2, s1, Wr, -);\ SoTr(m1, s1, Wr, SAME);\\
EoTr(m2, s1, Wr, SAME)\\
\}filter(*, *, *)
\end{array}
$$

Table 1
Area overhead.

Debug Pattern	Lookup Table Size (#Bits)		#Tr Patterns
	Address	Data	
Race Pattern 1	4	2	3
Race Pattern 2	6	3	8
Deadlock Pattern 1	6	3	6
Deadlock Pattern 2	7	4	6
Livelock Pattern 1	7	4	6
Livelock Pattern 2	7	4	6

In the first part of the assertion, the pattern checks a race condition in which master $m1$ starts a race. The second part of the assertion specifies a race condition in which master $m2$ starts a race. Six states are used to check this assertion online. For the improved race assertion, we need 3 bits for the FSM states. We need the following 8 transaction patterns: $T1_1$, $T1_2$, $T1_3$, $T2_1$, $T2_2$, $T2_3$, $T3_1$, and $T4_1$. Eight transaction patterns are encoded into 3 bits. Therefore the size of the memory address is 6 bits (Race Pattern 2 in Table 1).

5.2. Debug pattern for deadlock and livelock

When some masters are waiting for other masters to release shared resources, deadlock happens. Here we show the case of two masters and two slaves as an example. Each slave has a semaphore which specifies its access permission. When semaphore is 0, the slave is free. When semaphore is 1, the slave is locked. Each master should first lock required slaves, then it can start its process using the corresponding slaves as resources. To lock a slave, a master has to first read the semaphore of the corresponding slave. If the semaphore is 0, then the master can write 1 to the semaphore to lock the corresponding slave. Therefore to lock a slave, a master needs two transactions, i.e., one read transaction and one write transaction. If the semaphore is 1, i.e., the slave is already locked, then the master should wait until the corresponding slave becomes released (in the application of dining philosophers). Both masters have access to the semaphore of each slave. Accessing a semaphore is equivalent to accessing the same address by different masters.

A simple deadlock scenario for two masters and two slaves is as follows [6]: (1) Master1 locks the first semaphore. (2) Master2 locks the second semaphore. (3) Master1 waits for the second semaphore. (4) Master2 waits for the first semaphore. (5) Steps 3 and 4 are repeated. This deadlock condition is written in TDPSL as follows [6]:

$assert\ never\{$
$EoTr(m1, s1, Rd, -);\ EoTr(m1, s1, Wr, SAME);$
$EoTr(m2, s2, Rd, -);\ EoTr(m2, s2, Wr, SAME);$
$\{EoTr(m1, s2, Rd, SAME);\ EoTr(m2, s1, Rd, SAME)$
$|EoTr(m2, s1, Rd, SAME);\ EoTr(m1, s2, Rd, SAME)$
$\}[+]$
$filter(*, *, *)$

This assertion is written for applications in which each master first locks the slave with the same ID. For example master $m1$ first locks slave $s1$. If it is successful, then it locks slave $s2$. To implement this assertion by our debug infrastructure, the filters are programmed such that transactions *SoTr* are filtered online as unrelated transactions. Also all transactions related to the second group, i.e. group of master $m3$ and master $m4$, are filtered. In the filter structure of Fig. 6, the field *trans_type* is programmed to have *SoTr*. The field *master* has to include the master ID 3 and 4. Other fields are programmed to have don't care values.

We abstract the deadline assertion to $T1T2_1\ T3T4_1$ $\{T5_1T6_1 | T6_1T5_1\}$. Fig. 7 shows the FSM related to the deadlock assertion. In the deadlock assertion, there is a repetition operator $+$ which means the transaction sequence between the corresponding two brackets $\{\ldots\}$ may be repeated one or more times. For the sake of simplicity, we show the case of one repetition in the FSM. The operator $+$ can be implemented using a counter in the FSM which checks how many times a special transaction sequence is repeated. In Fig. 7, states $(Start, A, B, C, D)$ check the transaction sequence $T1T2_1\ T3T4_1$. The transaction sequences $T5_1\ T6_1$ and $T6_1\ T5_1$ are verified by states (D, E, Err) and (D, F, Err). The FSM works correctly in the presence of suitable filters. In this case, only the related transactions are investigated by the FSM.

The deadlock FSM requires 3 bits to indicate eight states and requires 6 transaction patterns: $T1$, $T2_1$, $T3$, $T4_1$, $T5_1$, and $T6_1$. The transaction patterns are encoded into 3 bits. Therefore the address line has 6 bits $(3 + 3)$. The size of memory data is 3 bits (Deadlock Pattern 1 in Table 1).

In the mentioned deadlock assertion, first the lock process from master $m1$ is checked, then the lock process from master $m2$. To illustrate this case better, we denote a read transaction (write transaction) from master m_x to slave s_x as Rxy (Wxy). In the previous deadlock assertion only the sequence $(R11, W11, R22, W22)$ is checked for the lock process. To increase the verification coverage of the deadlock assertion we check the following sequences for the lock process: $(R11, W11, R22, W22)$, $(R11, R22, W11, W22)$, $(R11, R22, W22, W11)$, $(R22, W22, R11, W11)$, $(R22, R11, W22, W11)$, $(R22, R11, W11, W22)$.

The improved deadlock assertion requires 4 bits to implement thirteen states. Also it uses 6 transactions (3 bits input). Thus the size of address and data are 7 bits $(4 + 3)$ and 4 bits (Deadlock Pattern 2 in Table 1).

A livelock is similar to a deadlock where two or more processes proceed accessing shared resources which are already locked. But in the case of a livelock, they release the locked resources permitting the other processes to continue. A simple livelock scenario for two masters and two slaves is as follows [6]: (1) Master1 locks the first semaphore. (2) Master2 locks the second semaphore. (3) Master1 waits for the second semaphore. (4) Master2 waits for the first semaphore. (5) Master1 unlocks the first semaphore. (6) Master2 unlocks the second semaphore. (7) Steps 1–6 are repeated.

To implement the livelock FSM, the deadlock pattern 1 and the deadlock pattern 2 are enhanced to check the steps 5 and 6. Table 1 shows the area overhead of the livelock pattern.

Our debug infrastructure is programmed according to race, deadlock, and livelock debug patterns and detects the occurrence of each debug pattern at run-time. Table 2 shows the number of times a debug pattern is detected during the simulation time of one million cycles. In the random application, the race pattern is detected 62 times. Also at run-time, this information is sent to the corresponding masters (Fig. 4). In the application of dining philosophers, the deadlock pattern is detected one time. In this case, after the first deadlock detection, the group of deadlocked masters cannot proceed with their process anymore.

As explained in Section 4.3 (Fig. 5), the masters can start a recovery process after the CDU has sent them the error state. Table 3 indicates the effect of using the recovery process in the masters. Without recovery process in the application of dining philosophers, the masters in one group can eat only 6 times. After that a deadlock happens and the masters cannot pick up their required chopsticks. However, the recovery process of Fig. 5 causes that the masters can continue their main process even if they get into a deadlock. In this case, a deadlock happens 77 times. But in each time, the CDU detects the deadlock at run-time and triggers the recovery process in the masters to recover the deadlocked network. Consequently, the masters can eat more often (3276 times) as shown in Table 3.

The recovery example of Fig. 5 is suitable for software deadlocks. In the case of a hardware fault in the network which leads to a wrong routing and consequently results in a deadlock, the recovery algorithm has to select a new route avoiding the faulty

Table 2
Number of debug patterns detected for each application.

	Rand. application	Din. application
Race Pattern 2	62	0
Deadlock Pattern 2	0	1
Livelock Pattern 2	0	0

Table 3
Effect of online recovery.

	Without recovery	With recovery
#Eating	6	3276
#Resolved Deadlock	0	77

route. In this case, the recovery algorithm can be enhanced using a diagnosis approach such as [22] to localize faulty components in the network.

6. Conclusion

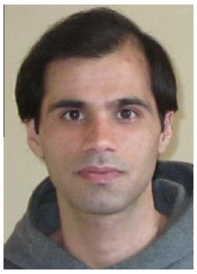
We introduced an approach to online debug for NoC-based multiprocessor SoCs. Our approach contains a hardware infrastructure, debug redundant information, and FSMs. Monitors, filters, and debug units are considered in our debug hardware infrastructure. This infrastructure allows us to investigate and to debug the behavior of an NoC-based SoC at run-time. Filters and FSMs are programmed according to the transaction-based assertions defined by TDPSL. In the experimental results, we investigated the efficiency of our approach for the debug patterns race, deadlock, and livelock. Our debug infrastructure is used not only for online debug and online system recovery but also for interactive debug in which an external debug platform programs the FSMs and the filters according to the considered assertions at each round of debugging.

Acknowledgements

Mehdi Dehbashi did this work as part of his PhD in the University of Bremen. This work has been supported in part by the University of Bremen's Graduate School SyDe, funded by the German Excellence Initiative and in part by the German Research Foundation (DFG, Grant No. FE 797/6-1). This paper is an extended version of [1].

References

[1] M. Dehbashi, G. Fey, Transaction-based online debug for NoC-based multiprocessor SoCs, in: Euromicro Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2014, pp. 400–404.
[2] L. Benini, G.D. Micheli, Networks on chips: a new SoC paradigm, IEEE Comput. 35 (1) (2002) 70–78.
[3] P.P. Pande, C. Grecu, A. Ivanov, R.A. Saleh, G.D. Micheli, Design, synthesis, and test of networks on chips, IEEE Des. Test Comput. 22 (5) (2005) 404–413.
[4] K. Goossens, B. Vermeulen, R. van Steeden, M.T. Bennebroek, Transaction-based communication-centric debug, in: International Symposium on Networks-on-Chips (NOCS), 2007, pp. 95–106.
[5] B. Vermeulen, K. Goossens, A network-on-chip monitoring infrastructure for communication-centric debug of embedded multi-processor SoCs, in: International Symposium on VLSI Design, Automation and Test (VLSI-DAT), 2009, pp. 183–186.
[6] A.M. Gharehbaghi, M. Fujita, Transaction-based debugging of system-on-chips with patterns, in: Int'l Conf. on Comp. Design, 2009, pp. 186–192.
[7] W. Ecker, V. Esen, M. Hull, T. Steininger, M. Velten, Requirements and concepts for transaction level assertions, in: Int'l Conf. on Comp. Design, 2006, pp. 286–293.
[8] A.M. Gharehbaghi, M. Fujita, Transaction-based post-silicon debug of many-core system-on-chips, in: Int'l Symp. on Quality Electronic Design, 2012, pp. 702–708.
[9] F.M. de Paula, A. Nahir, Z. Nevo, A. Orni, A.J. Hu, TAB-backspace: unlimited-length trace buffers with zero additional on-chip overhead, in: Design Automation Conf., 2011, pp. 411–416.
[10] NIRGAM NoC Simulator, 2013. <http://nirgam.ecs.soton.ac.uk/>.
[11] A. Hopkins, K. McDonald-Maier, Debug support for complex systems on-chip: a review, IEE Comput. Digit. Tech. 153 (4) (2006) 197–207.
[12] H. Yi, S. Park, S. Kundu, On-chip support for NoC-based SoC debugging, IEEE Trans. Circ. Syst. 57-I (7) (2010) 1608–1617.
[13] S. Tang, Q. Xu, A multi-core debug platform for NoC-based systems, in: Design, Automation and Test in Europe, 2007, pp. 870–875.
[14] S. Tang, Q. Xu, A debug probe for concurrently debugging multiple embedded cores and inter-core transactions in NoC-based systems, in: ASP Design Automation Conf., 2008, pp. 416–421.
[15] V. Todorov, A. Ghiribaldi, H. Reinig, D. Bertozzi, U. Schlichtmann, Non-intrusive trace and debug NoC architecture with accurate timestamping for GALS SoCs, in: International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2012, pp. 181–186.
[16] C.-N. Wen, S.-H. Chou, C.-C. Chen, T.-F. Chen, NUDA: a non-uniform debugging architecture and nonintrusive race detection for many-core systems, IEEE Trans. Comput. 61 (2) (2012) 199–212.
[17] J. Raik, V. Govind, R. Ubar, Design-for-testability-based external test and diagnosis of mesh-like network-on-a-chips, IET Comput. Digit. Tech. 3 (5) (2009) 476–486.
[18] M.A. Kochte, C.G. Zoellin, M.E. Imhof, R.S. Khaligh, M. Radetzki, H.-J. Wunderlich, S.D. Carlo, P. Prinetto, Test exploration and validation using transaction level models, in: Design, Automation and Test in Europe, 2009, pp. 1250–1253.
[19] C. Concatto, P. Almeida, F.L. Kastensmidt, É. F. Cota, M. Lubaszewski, M. Herve, Improving yield of torus NoCs through fault-diagnosis-and-repair of interconnect faults, in: IEEE International On-Line Testing Symposium (IOLTS), 2009, pp. 61–66.
[20] Y.-C. Chang, C.-T. Chiu, S.-Y. Lin, C.-K. Liu, On the design and analysis of fault tolerant NoC architecture using spare routers, in: ASP Design Automation Conf., 2011, pp. 431–436.
[21] A. Strano, C.G. Requena, D. Ludovici, M. Favalli, M.E. Gómez, D. Bertozzi, Exploiting network-on-chip structural redundancy for a cooperative and scalable built-in self-test architecture, in: Design, Automation and Test in Europe, 2011, pp. 661–666.
[22] A. Ghofrani, R. Parikh, S. Shamshiri, A. DeOrio, K.-T. Cheng, V. Bertacco, Comprehensive online defect diagnosis in on-chip networks, in: VLSI Test Symp., 2012, pp. 44–49.
[23] S. Shamshiri, A. Ghofrani, K.-T. Cheng, End-to-end error correction and online diagnosis for on-chip networks, in: Int'l Test Conf., 2011, pp. 1–10.
[24] A. Alaghi, N. Karimi, M. Sedghi, Z. Navabi, Online NoC switch fault detection and diagnosis using a high level fault mode, in: IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2007, pp. 21–30.
[25] OSCI TLM-2.0 Language Reference Manual, 2013. <http://www.systemc.org>.
[26] J.D. Carpinelli, Computer Systems Organization & Architecture, Addison-Wesley Boston, San Francisco, New York, 2001.


Mehdi Dehbashi received his M.Sc. in computer engineering from Sharif University of Technology, Tehran, Iran, in 2007, and his Ph.D. in computer science from the University of Bremen, Bremen, Germany, in 2013. He works currently in Infineon Technologies AG, Munich, Germany. His research interests include computer aided design for circuits and systems, dependable embedded systems design, and distributed embedded systems.

Görschwin Fey received his diploma in computer science form the Martin-Luther Universität, Halle-Wittenberg, Germany, in 2001, and his doctoral degree in computer science from the University of Bremen, Bremen, Germany, in 2006.
Since 2012 he is leading the Group of Reliable Embedded Systems at the University of Bremen and heads the Department of Avionics Systems at the Institute for Space Systems of the German Aerospace Center, Bremen. His research interests are in testing and formal verification of circuits and systems.