Transaction-Based Communication-Centric Debug

Kees Goossens[1,2], Bart Vermeulen[1], Remco van Steeden[3], Martijn Bennebroek[4]
[1] Research, NXP Semiconductors, The Netherlands, {first.lastname}@nxp.com
[2] Computer Engineering, Technical University Delft, The Netherlands
[3] Testable Design and Test of Integrated Systems, Technical University of Twente, The Netherlands
[4] Research, Philips, The Netherlands, Martijn.Bennebroek@nxp.com

***Abstract*— The behaviour of systems on chip (SOC) is complex because they contain multiple processors that interact through concurrent interconnects, such as networks on chip (NOC). Debugging such SOCs is hard. Based on a classification of debug scope and granularity, we propose that debugging should be communication-centric and based on transactions. Communication-centric debug focusses on the communication and the synchronisation between the IP blocks, which are implemented by the interconnect using transactions.**

We define and implement a modular debug architecture, based on NOC, monitors, and a dedicated high-speed event-distribution broadcast interconnect. The manufacturing-test scan chains and IEEE1149.1 test access ports (TAP) are re-used for configuration and debug data read-out.

Our debug architecture requires only small changes to the functional architecture. The additional area cost is limited to the monitors and the event distribution interconnect, which are 4.5% of the NOC area, or less than 0.2% of the SOC area. The debug architecture runs at NOC functional speed and reacts very quickly to debug events to stop the SOC close in time to the condition that raised the event. The speed at which data is retrieved from the SOC after stopping using the TAP is 10 MHz.

We prove our concepts and architecture with a gate-level implementation that includes the NOC, event distribution interconnect, and clock, reset, and TAP controllers. We include gate-level signal traces illustrating debug at message and transaction levels.

I. Introduction

Today's high-performance systems on chip (SOC) contain many intellectual property (IP) blocks, such as memories, dedicated hardware blocks, and programmable processors. Applications are implemented by a number of concurrent computations threads running on the (programmable) IP blocks. The threads communicate through the SOC interconnect.

In the past, the SOC interconnect was single-threaded. As a result, all computation threads were effectively serialised by processing one transaction at a time, offering a simple linear view on the SOC. Today, however, SOC interconnects, such as multi-layer Amba [1] and networks on chip (NOC) [2], [3], support multiple concurrent transactions. As a result, a single thread of control no longer exists in the interconnect, and transactions between different computation threads are not constrained to any particular order in time.

Because SOCs comprise multiple programmable processors that interact through a concurrent programmable interconnect their behaviour is very complex, and designing right-first-time SOC hardware and software has become difficult [4], [5]. In this paper we address the challenge of *debugging* SOCs. Debugging involves observing the SOC in its target environment and controlling its execution (stopping, single stepping, etc.) to efficiently and effectively locate the root cause of any undesired behaviour. We propose a new debug methodology that is centred on communication and based on transactions. We discuss these aspects in turn.

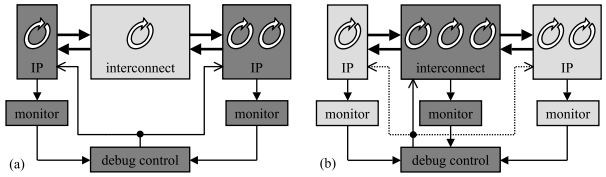

Fig. 1. Computation-centric debug (a) vs. communication-centric debug (b).

A. Communication-centric debug

Monitoring and debugging computation, especially of a single processor, is a mature area for which tools already exist [6], [7]. However, debug of multiple (programmable) IP blocks on a SOC is an emerging research field. Debugging IP blocks by themselves is not enough, because the complexity of the SOC increasingly resides in the interactions between the IP blocks [4], [8]. Therefore, debug must be conducted at a higher system level, where the computation threads and communication threads interact. Because the interconnect implements the communication, and hence the synchronisation between the IP blocks it is the natural locus for system-level debug, where all transactions can be observed concurrently, (partially) ordered, or fully sequentialised.

Figure 1(a) illustrates conventional computation-centric debug, where IP blocks with their threads are monitored, triggering events. Based on these events the debug control can stop and inspect the state of the IP blocks. Instead, we propose to focus on monitoring the communication between the IP blocks (Figure 1(b)), where the debug control manages the interaction between the IP blocks by controlling the interconnect. (Of course, the IP blocks can still be monitored and controlled too, as shown by the dashed lines.)

B. Transaction-based debug.

Processor software is typically debugged at either the source code, or the processor instruction level. The latter is the lowest level that is still meaningful for the programmer. Processor

Proceedings of the First International Symposium on Networks-on-Chip (NOCS'07)
0-7695-2773-6/07 $20.00 © 2007 IEEE

instructions constitute a natural abstraction level between the *software and the processor hardware*.

But (software) threads also communicate with each other via the interconnect, using transactions. Transactions are the result of processor instructions (such as load and store) that cause activity on the interconnect and other IP blocks such as memories. *Transactions* therefore are a natural interface between *computation and communication*. The instruction abstraction is important for integrated hardware/software debug, while the concept of a transaction is fundamental for system-level debug (i.e. multiple IP cores). This is confirmed by the use of transaction-level modelling (TLM) for interconnects and SOCs. Transactions allow us to recuperate a globally consistent view on the SOC, as described next.

Consistent SOC *view.* A globally consistent view on both SOC hardware and software is desirable, but has been hard to achieve for the following reasons. First, software debug takes place at instruction level or above, while hardware debug is typically performed at the level of clock cycles. Experience shows that it is hard to correlate the information at these levels. Second, as Figure 2(a) illustrates, there may be no globally consistent state at any point in time (clock cycle). Due to e.g GALS, SOCs are no longer fully synchronous, which gives rise to non-determinism at the level of clock cycles. IP2 and IP3 are in different clock domains that may never have simultaneous clock edges. Moreover, cross-clock-domain synchronisation may be non-deterministic, such that there may not be a single point in time at which the entire SOC (chip) is in a global state that can be correlated with pre-silicon simulation models and test benches [9]. Finally, there may be no point in time where all processors have finished their instructions, as illustrated in Figure 2(a).

Transaction-based debug. By abstracting the view on the SOC from clock cycles to transactions, the dark lines in Figure 2(b) may be interpreted as "transaction cycles" instead of clock cycles. Alternatively, the transaction cycles can be *enforced* as shown in Figure 2(c), by inserting idle cycles or stretching the clocks. In the example, the entire SOC advances in lock step at the level of transactions. This transaction trace can be correlated to processor instructions, is deterministic, is globally consistent, and can be faster than simulation at the clock or instruction level. The notion of consistency will be further elaborated in Sections II and III.

C. Main contributions

This paper addresses the problem of debugging of complex multi-processor SOCs with concurrent interconnects.

We introduce the new *concept of communication-centric transaction-based debugging*. Based on a model of transactions and of multi-hop interconnects, we introduce a detailed *classification* of debug scope and granularity. Processor and interconnect monitoring and debug coincide at a few points: the instruction/flit level, which is the smallest grain of control; the *message level* where interactions between master and slave IP blocks are visible, and the *transaction level*, where master behaviour alone is traced.

We further define a *general debug architecture*, based on monitors, IEEE1149.1 test access port (TAP), and several debug interconnects. We apply the debug architecture to the Æthereal NOC [10], monitors [11], [12], and the debug infrastructure of [13]. In particular, we distribute events raised by monitors using a fast dedicated broadcast interconnect. It stops transaction valid/accept handshakes between the network interfaces (NI) and IP blocks, after which the manufacturing-test scan chains are accessed using the TAP controller to read out the NOC and IP state.

Our debug architecture requires no changes to IP, routers, or NI kernels, and very few changes to NI shells. Given that the scan chains and TAP controller are already present in the SOC the additional *area cost* is limited to the monitors and the event distribution interconnect, which are only 4.5% of the NOC area.

The debug architecture *reacts very quickly to debug events* (it runs at NOC functional speed) to stop the SOC close in time to the condition that raised the event, and to not lose data. The speed at which data is retrieved from the SOC after stopping is acceptable (10 MHz test clock). To scan out around 43000 registers in our example NOC takes approximately 4 milliseconds.

Finally, three example *gate-level signal traces* are included of a debug example at message and transaction levels, to illustrate that the debug concepts and architecture function.

In the remainder of this paper we first introduce models for transactions and interconnects (Section II). In Section III we identify and classify scopes and levels of abstraction in the architecture at which debug can take place. We define a general debug architecture in Section IV, which is refined in Section V. We present our results in Section VI and related work in Section VII, and conclude in Section VIII.

II. Transaction and Interconnect Models

In this section we define the general model for transactions, and an interconnect model tuned for NOCs. The models serve as input for the classification of debug scopes and abstractions of Section III.

A. Transaction Model

IP blocks interact on *ports*, either directly or using an interconnect, and use *transactions*. A transaction is initiated by a *master* port, by sending a *request* that is executed by the IP block attached to the receiving *slave* port. (For brevity, in the remainder for "master" read "master port," and for "slave" read "slave port.") The execution may result in a *response* that the slave sends to the master. A *message* is a request or a response. Typical requests include read and write commands; read data and write acknowledgements are typical reponses. Both communication protocols based on distributed shared memory (e.g. DTL [14], AXI [15], and OCP [16]) and message passing (no responses) fit within this model.

The master and slave are connected by an intermediate interconnect, as shown in Figure 3. Between every pair of blocks, a request is passed from the *initiator* to the *target*; the

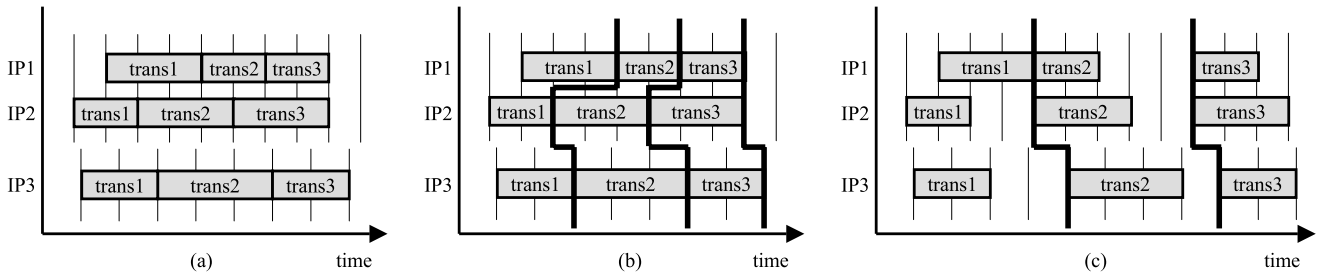

Fig. 2. Globally consistent state of a SOC, is absent at clock-cycle level (a), but is possible at the transaction level (b) and (c).

response travels in the opposite direction. Hence, the master is the first initiator and the slave the last target. Requests and responses can be transferred independently, using a valid-accept handshake.[1] The initiator offers a request to the target by driving the *valid* signal high. The target in turn indicates that it has accepted the request by driving the *accept* signal high. For responses, the roles of initiator and target are reversed. A transaction *starts* when the master signals that the request is valid, and is *in progress* until the transaction *completes*, which is when the master has accepted the response. Depending on the protocol, transactions may either be *split* (the request and response handshakes are independent, e.g. AXI) or not (request and response handshakes coincide, e.g. APB). In addition, split transactions can either be *pipelined* (i.e. allow multiple outstanding requests, e.g. AXI), or not.

The transaction model describes the interactions between the IP blocks and the interconnect. Next, we turn our attention to the details of transaction transportation.

B. Interconnect Model

In this section we describe a general multi-hop interconnect model. It covers connection-oriented NOCs such as Mango [17], Nostrum [18], Æthereal [10], and FAUST [19], and connection-less NOCs [20], [21], [22], but also hierarchical busses [23], [15].

A NOC configuration (or *use case*) [24] is a set of *connections* [25]. A configuration is either explicitly declared in connection-oriented NOCs or implicitly defined by the master-slave communication pattern in connection-less NOCs. Transactions between a single master and one or more slaves take place on a connection. All transactions on a single connection are ordered. This means that the order of the requests offered to the NOC by the master, and the order of the requests offered to each slave by the NOC, are equal to the order of

[1]To be more precise, the request and response can be subdivided in *signal groups*; for e.g. AXI, the request comprises the command group and the write data group, and the reponse comprises the read data group. The valid-accept handshake then typically takes places per word and independently for each signal group. Depending on the protocol there may be restrictions on interleaving, ordering, and pipelining. For example, the write data may come before the write command in AXI, but not in AHB. AXI allows message interleaving, where DTL and AHB do not. In this paper, to simplify presentation, we use the handshake at the level of messages instead of signal groups. Our concepts apply unchanged also to signal groups.

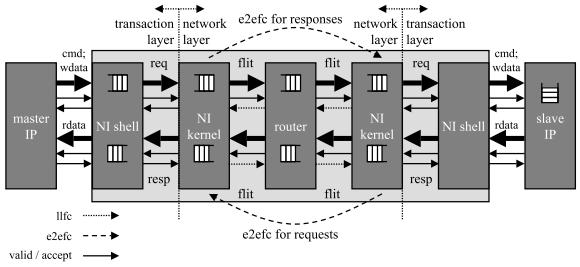

Fig. 3. Architecture model.

the responses offered to the master by the NOC.[2] (We assume that on a connection, the slave offers responses in the same order as it accepted the requests, regardless of the precise order in which it executes them.) There is no execution order specified between transactions of different slaves on the same connection, nor between transactions on different connections. In the former case, executions can be forced to be in order by disallowing multiple outstanding transactions to different slaves. In the latter case, however, no ordering can be enforced due to the concurrent nature (distributed arbitration) of NOCs.

Multi-hop interconnects are composed internally of multiple stages ("hops"), as shown in Figure 3. NOCs are mostly *packet* based, which means that messages (requests or responses) are internally transported in (almost always) smaller packets. Routers transport packets, and network interfaces (NIs) convert messages to and from packets. Inside the kernel and router network packets are usually split in smaller units, called *flits*, which consist of a fixed number of *words*.

A NI is often split into separate components, the NI kernel and the NI shell [26]. The kernel and routers perform the OSI [27] network layer functions, i.e. move data from one NI kernel to another. The shells implement the OSI transport layer, which in the present context could be called the transaction layer. The shells convert IP transactions received on master or slave NI ports into serial data for the kernel. All knowledge regarding IP protocols (and possible conversions) resides in the shells. As a result, kernels are highly re-used, even across IP-block protocols, and shells are re-usable in systems that

[2]A number of protocols require in-order reads and in-order writes, but allow re-ordering between them. This does not essentially change the argument.

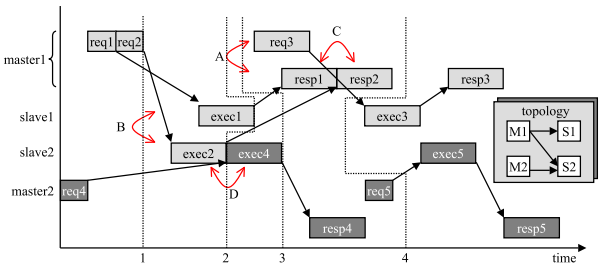

Fig. 4. Transaction ordering and interleaving.

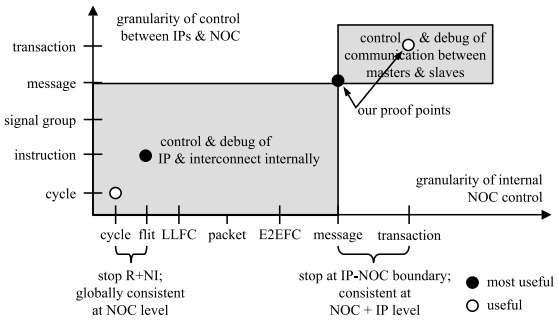

Fig. 5. Scope and granularity of debugging.

utilize the same IP-block protocols.

Packet boundaries are always aligned with flit boundaries. Most NOCs also align messages and packets, i.e. where a message header is always immediately preceded by a packet header, at the cost of packetisation efficiency [26]. Sometimes message and packet headers are even merged to increase efficiency. In both cases, kernel and shell functions are combined at the expense of easy re-use. Æthereal does *not* align message and packet boundaries, which complicates debug [12].

To avoid buffer overflows and possible data loss in the NI kernels and routers, NOCs use *link-level flow control* (LLFC): an initiator (router or NI kernel) sends a flit to the target (router or NI kernel) only if there is space in the target buffer. To avoid data loss and deadlock, *end-to-end flow control* (E2EFC) is an independent mechanism that ensures that connection buffers in the NI kernels do not overflow. That is, data only leaves a sender's NI kernel buffer when there is space in the receiver's NI kernel buffer.

All NOCs offer a best-effort service (BE) and a number of NOCs also offer a guaranteed service (GS) [17], [18], [10], [19]. LLFC is used by all NOCs except Nostrum for BE. E2EFC is used by SPIN for BE, and by Æthereal and FAUST for BE & GS. Note that Æthereal and Nostrum do not use LLFC for GS, because contention is guaranteed to be absent.

In the next section, we classify debug activities based on the transaction and interconnect models. First, we illustrate the transaction model.

C. Example

Figure 4 shows an example of the concepts introduced above. Master 1 uses split pipelined transactions because it sends request 2 before response 1 has arrived. Requests and responses may overlap; e.g. response 1 and request 3 in Figure 4[A]. Requests 1-3 of master 1 are executed by different slaves: 1, 2, and 1, respectively. Slave 2 executes request 2 *before* slave 1 executes request 1 (Figure 4[B]), even though the latter was accepted earlier by the NOC. This can occur when request 1 took longer to arrive than request 2 (e.g. due to a longer path, more congestion, less reserved bandwidth, or lower QoS), or when slave 1 is slower than slave 2. Whatever the execution order, responses are always offered to the master in the correct order (Figure 4[C]). However, note that transactions of different connections are unordered. For example, request 2 of master 1 and request 4 of master 2 are executed and completed in the reverse order of acceptance by the NOC, cf. Figure 4[D]. Finally, note that master 2 does not use split transactions: transaction 4 completes before transaction 5 is started.

III. SOC AND NOC DEBUG

In this section we define various scopes of debug: SOC, IP, and NOC. Then we define the temporal granularity at which we can debug, e.g. clock cycles, flits, messages, and transactions. We show that communication-centric debug based on transactions offers a good combination of spatial and temporal localisation of SOC problems.

A. Debug scope

Until now, SOCs have been debugged predominantly by concentrating on the computation, through monitoring and debugging of the programmable processors. However, SOC complexity is shifting from computation on a single processor to the interaction between the multiple processors. In addition, interconnects are increasingly complex, and no longer impose a single thread of control through transaction serialisation. The scope of debug therefore extends beyond the traditional debugging of processors, to include debugging of the interconnect, and system-level debug that concentrates on the interactions between IP blocks. Therefore, acting on events from the monitors, we control the interaction between IP blocks through the interconnect (Figure 1(b), instead the internal behaviour of the IP blocks (Figure 1(a)).

B. Debug granularity

In the following we restrict our discussion to NOCs and other multi-hop interconnects. We can distinguish various levels of granularity at which a NOC can be debugged. The horizontal axis in Figure 5 shows a number of levels at which a NOC can be monitored and potentially stopped. First, we discuss each of the cycle, flit, link-level flow control (LLFC), packet, end-to-end flow control (E2EFC), message, and transaction levels in turn. Then we discuss the vertical axis of processor debug granularity.

Stopping a NOC at the *clock cycle or flit granularity* halts all BE and GS data in the NOC in situ, i.e. in the NIs and routers. Both stop methods function well in synchronous NOCs, and

the latter also in asynchronous NOCs, where NIs and routers handshake at least at the level of flits. The former method can be achieved by gating the NOC clock, the latter method by masking the flit handshake between routers (forcing the flit valid signal to '0').

Stopping a NOC at the level of LLFC can be implemented by masking the LLFC credit counters to 0. As a result NIs and routers do not send data anymore because it seems as if all receiving buffers are full. The flit granularity coincides with the LLFC granularity when all traffic types use LLFC. This is not the case for Æthereal and Nostrum that do not use LLFC for GS traffic. Stopping LLFC stops all BE traffic in situ, but all GS traffic continues to be transported.

Stopping a NOC at the granularity of *packets* is only useful in two degenerate cases. First, if LLFC is performed at packet level (store and forward), reducing it to LLFC granularity. Second, if packets and messages coincide, when it corresponds to the message level, described below.

Stopping a NOC at the granularity of E2EFC by masking E2EFC allows all packets that are in the router network to continue to their destination NIs, but prohibits packets from leaving the NIs. This flushes the router network, and all useful state remains in the NIs. However, E2EFC is often performed at the granularity of words, and masking the E2EFC can therefore result in messages being split over the sender and receiver NIs. NOCs that perform the local E2EFC [28] at the level of messages do not suffer from this problem.

By stopping a NOC at the level of *messages*, i.e. requests and responses, we intend that packets, LLFC, and E2EFC continue to operate. The NOC finishes message handshakes that are *in progress* with the IP blocks. This can happen at four places (cf. Figure 3): the request at the master, the request at the slave, the response at the slave, and the response at the master. This may in fact take a number of clock cycles (see footnote 1). For example, the NOC continues with master requests until all write data words have been accepted. This method may be implemented by masking the valid and/or accept of the request and response at the master and slave to '0'. We give more details in Section V-B.

The coarsest granularity of stopping is at the level of *transactions*, i.e. when all outstanding messages have completed. In this case, essentially the NOC and all slaves continue operating. This may be implemented by masking the request accept to '0' at the master NI ports: no new transactions are accepted by the NOC, and all outstanding transactions are completed. Figure 2(c) is an example of a trace where all IP blocks execute a single transaction at a time. However, this level is limited for two reasons. First, if an IP block uses pipelined transactions (e.g., master 1 in Figure 4), then there may be no time at which all transactions have completed because the IP block wants to issue a request before accepting a response. Second, the complexity of SOCs resides in the synchronisation of IP blocks, i.e. in the concurrency and interleaving of transactions. Message-level debug exposes the ordering of messages at *both masters and slaves*. But transaction-level debug exposes the ordering of transactions at the *masters only*, which could also be obtained from IP-block debug alone.

Therefore, because not all NOCs are synchronous, many NOCs do not have E2EFC, and not all NOCs use LLFC for all traffic classes, *the most generally useful* NOC *debug granularities are the flit level and the message level*. At the flit granularity the data is stopped in situ, i.e. in the routers & NIs, whereas at the message level, the router network is flushed and all messages gather in the NIs. The former is required to debug the NOC, and the latter is most suited to debug the SOC, i.e. the interactions between the IP blocks and the interconnect.

Processors: Although a fine-grained distinction can be made for processors, like we have done for NOCs, here we restrict ourselves to processor debug at the level of clock cycles, instructions, messages, and transactions (vertical axis in Figure 5). Instructions are the lowest level that a programmer can relate to, messages expose the interleaving of master and slave threads, whereas transactions show only the master view.

Conclusion: The important conclusion that can be drawn from Figure 5 is that *processor and interconnect monitoring and debug coincide at a few points* that are marked in Figure 5: the instruction/flit level, which is the smallest grain of control; and the message level where interactions between master and slave IP blocks are visible. The other points serve to debug either the NOC or processors internally, or debug only the master IP blocks.

C. Debug actions

To debug, relevant parts of the system must be monitored. Monitors generate an *event* whenever something interesting is observed. The debug infrastructure can react on events by sending information out of the chip, stopping (part of) the SOC, etc. When the SOC is stopped its state can be read out and/or changed, before continuing operation. Traditionally, it may be possible to continue by performing a single step on all or some of the clocks (or IP blocks), or by resuming full operation. Because we focus on the communication rather than the computation, we can enforce "single stepping" on any of the debug granularities described before. As an example, consider Figure 4. If the NOC stops accepting new messages from initiators and stops offering new messages to targets at "message time" 1, then requests 1, 2, and 4 have been accepted from the masters, but not yet been offered to the slaves. By allowing a single message step on both slaves we advance to message time 2. Another single step on the slaves executes request 4 and advances to time 3, and so on. In this paper, we define and implement the functionality that enables stopping, single-stepping, etc., but leave the use of this infrastructure for future work.

IV. General debug architecture

In this section we give an overview of the general communication-centric debug architecture. The details follow in the next section. Figure 6 shows the hardware debug architecture, with the functional NOC at the centre. For clarity only the request signals between master, NOC, and slave are shown. The response architecture is similar.

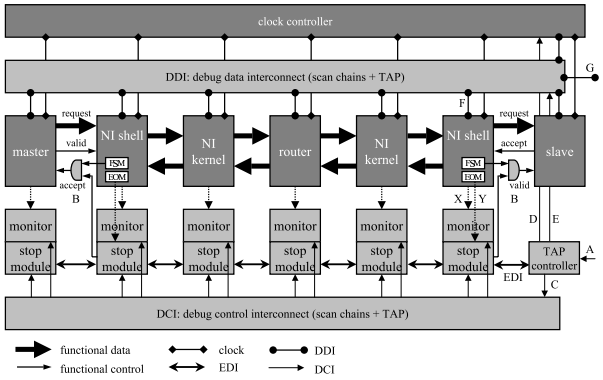

Fig. 6. Hardware debug architecture.

A. Components

Monitors [29], [11] can be attached to any SOC component or connection (wires) between components. Monitors extract information from the component they are attached to without changing the operation of the component, as suggested by the dashed arrows. Monitors can be programmed to trigger on certain (combinations of) events, such as a packet on a certain connection, a message of a given type within a given address range, and so on. When a monitor generates an event, its *stop module* [29] and the event distribution interconnect distribute the event to other debug components, which take appropriate further action.

Possible actions include generating an interrupt on a programmable processor in the SOC, generating an event on the off-chip trace or debug bus, stopping parts of the SOC (IP and/or interconnect), downloading or uploading information, etc. Many of these actions involve stopping the clocks of the IP blocks and/or the NOC. This is implemented by the clock controller and specific debug blocks such as the TAP controller, discussed in detail in Section V.

B. Interconnects

We can distinguish four interconnects in Figure 6: the functional interconnect, the event distribution interconnect (EDI), the debug data interconnect (DDI), and the debug control interconnect (DCI). In the remainder we assume that the functional interconnect is a NOC.

The *event distribution interconnect* (EDI), which comprises the stop modules, must quickly distribute events raised by monitors to all stop modules, clock controller, and other debug components. Events must be reacted on quickly enough to not lose the data that caused the monitor to trigger the event. For example, if the debug granularity is at the clock cycle level ("after a certain event, stop the NOC in the next clock cycle"), then the event must reach all NOC components and the clock controller within one clock cycle. If the granularity at which the NOC must be stopped is a message, the event must reach all stop modules before the message on which the monitor triggered has left the NOC (finished its handshake).

Downloading from or uploading of information to the stopped NOC and/or IP blocks takes place via the *debug data interconnect* (DDI). Programming of monitors and stop modules takes place via the *debug control interconnect* (DCI). Both can be implemented with a dedicated interconnect or by re-using an existing interconnect, such as the NOC or the manufacturing-test scan chains.

V. Message- and transaction-based debug

To prove the concepts described above, we implemented message-level debug, where, based on events generated by monitors, on-going message handshakes are finished before stopping the communication between NOC and IP blocks. We also implemented transaction-level debug, where no new transactions are accepted instead. Following this, the state of the NOC (and/or IP blocks) can be read out and/or modified via the manufacturing-test scan chains. Our experiments are based on the following:

- The Æthereal NOC.
- The monitors are attached to the routers.
- The EDI is a dedicated multi-hop broadcast interconnect, mirroring the topology of the NOC.
- The DDI and DCI re-use the manufacturing-test scan chains and are accessible via the *test access port* (TAP) on the chip pins.

Below, we provide further details on each of these.

A. NOC and monitors

In the context of debug, the Æthereal NOC is interesting (cf. Section II-B) because it does not use LLFC for its GS traffic, and because it uses E2EFC for both BE and GS traffic. Moreover, because message and packet boundaries are not necessarily aligned, it may be difficult for monitors to trigger on messages. A solution to this problem is given in [12] but in our experiments we use a simplified version of their monitors. We use the raw and connection-based modes, where it is possible to detect patterns in data passing on the links between the routers and/or NIs. Besides the 32 data wires, the pattern includes the two side band control bits that indicate word valid, flit QoS (BE/GS), packet header, and packet tail [26]. For example, all packets of a given connection are caught by matching on a packet header with the path from source to destination (rotated by the number of hops until now) and the destination queue identifier. The monitor raises an event to its stop monitor one cycle after data on the link matches a pattern. The match can occur on any word in a flit or message, which gives tight timing constraints, as discussed below. The dashed arrow [X] from the routers, NIs, and IP blocks indicate that monitoring is not intrusive, i.e. does not change the behaviour of the components that are observed. The monitors are programmed via the DCI, to enable or disable them, and to define the patterns.

B. NI shells

Message-based debug relies on stopping the handshakes of requests between the (1) master and the master NI port (MNIP),

and (2) between slave NI port (SNIP) and slave; similarly for responses between (3) slave and SNIP, and (4) between MNIP and master. As shown in Figure 6 by the two AND gates, this is accomplished by masking the request accept signals at the MNIP (1) and the request valid signal at the SNIP (2). Omitted from the figure is the similar masking of the response accept signal at the SNIP (3), and the reponse valid signal at the MNIP (4). The signals are masked when an event has been received on the EDI (more on this below), and when the finite state machine of the NI port in the NI indicates that the message is complete. The latter information ("end of message" or EOM in the figure) is already present in NI for transaction-safe connection reconfiguration [26], [30]. It is read out non-intrusively by the stop module from the NI shell, as indicated by the dashed arrow [Y]. The event distribution interconnect determines when events arrive at the NIs, and hence when the message handshakes are masked.

For transaction-level debug only the message handshake between the master and the MNIP is masked, using the same infrastructure, as soon as the current message has finished.

C. Event distribution interconnect

The event distribution interconnect (EDI) delivers events from the monitors and other event generators, such as the TAP controller, to the relevant debug components, such as the NIs and TAP controller. Ideally, when an event is generated anywhere in the SOC, all ongoing handshakes of messages must be finished, and no new handshakes must be initiated. Essentially, this requires global single-cycle event distribution, which is not scalable and difficult to implement in lay-out. Our EDI is the best alternative: events are broadcast synchronously at the (high) NOC functional frequency by pipelined stop modules. A stop module sends incoming events to all its neighbours: to ensure the broadcast dies out it does not respond to incoming events in the next cycle. The EDI is a scalable solution, with minimal latency (1 cycle per stop module), to minimise the number of new message handshakes starting after the event occurred. Note that an asynchronous EDI implementation is quite possible, and "cycle" is then replaced by "handshake."

The EDI uses the same topology as the NOC, and can be placed and routed alongside the routers and NIs, to avoid changes to the top-level lay out, and to ensure that it runs at the NOC functional frequency. This is suggested in Figure 6 by positioning the related blocks vertically above one another.

With a flit size of three, the EDI propagates events three times faster than the data that caused the event; this is quick enough to distribute an event to all stop modules before the message on which the monitor triggered can leave the NOC (finish its handshake). The critical timing for this occurs when a monitor is attached to the link between a router and the destination NI. If the monitor triggers on the final word (in a flit) of a message, then that message must be the last one to complete its handshake on that NI port. This is achieved because the next flit, part of the next message, has latency of at least a two cycles before being offered to the IP block [26]. The event arrives at the NI shell also two cycles after it has been generated, and is in time to mask the appropriate valid or accept signal.

Stop modules tell the NI shells to finish ongoing messages and then stop, as described previously. However, if an ongoing message does not finish, e.g. due to a non-responsive / stopped IP block, infinitely long message, or message-level interdependencies, a second stop signal can be used to forcefully stops all ongoing messages. The first stop signal can be generated by both monitors or the TAP controller, but the second stop signal can be given only by the TAP controller. The stop monitors contain a small FSM, which implements this stop sequencing, and ensures that the broadcast dies out and that multiple concurrent events are handled correctly. For example, two monitors can trigger simultaneously. Or a monitor can trigger after some or all of the NIs are already masking the message handshakes because it may take some time to flush the NOC, i.e. allow all data to arrive at the slave NI ports.

D. Debug data distribution interconnect

The debug data interconnect (DDI) can be a dedicated interconnect, such as a bus. Alternatively, existing interconnects can be re-used, such as the functional NOC, or manufacturing-test scan chains accessible via an IEEE1149.1 *test access port* (TAP) [31].

The functional NOC can be re-used as DDI, but the process of flushing the NOC of functional data before it can be used to transport debug data is non-trivial. If it has to be possible to resume operation after stopping and debug, the functional data that was flushed out of the NOC must be restored, which is also difficult.

It is standard industry practice to use scan chains [32] to test for manufacturing defects. The entire NOC (and all IP blocks) will therefore typically contain scan chains. For this reason, we use scan chains to implement the DDI, as proposed by [33], [29], [13]. We use NXP's standard design flow with gate-level synthesis and scan-chain insertion for the NOC, with an exception for the optimised hardware FIFOs used in the routers [34] and NIs [26], which contain a dedicated test infrastructure [35].

IEEE1149.1-compliant scan-based manufacturing-test and debug infrastructure is accessed using a TAP. Using the TAP data can be sent in and out of the chip over four or five dedicated chip pins. The NOC and every IP block have a *test wrapper and test control block* (TCB) to isolate and control the block during manufacturing test, respectively. They also have an *access-control test point register* (AC-TPR) to select which internal scan chains to route to the TAP using the test access mechanism. In this way, the infrastructure that is used for manufacturing test is largely re-used for debug.

SOCs often have sophisticated programmable clock generation that allows modifying the clock signal per IP block at run time. The clock controller switches each IP block's clock between off, one or more functional frequencies, one or more test frequencies, and a debug frequency. The test and debug infrastructures are independent from the functional

interconnect in terms of wiring. However, the test and debug infrastructure cannot operate at the same time as the functional interconnect because they access the same state (registers and FIFOs) and because they operate at different frequencies (the functional NOC frequency is much higher).

The TAP controller orchestrates the interactions between the TCBs and clock controller 1) to manage the transition from functional state and clock to the test or debug state and clock for each block, 2) to obtain test or debug access to a block, 3) to use the AC-TPRs to select the chain chains of the block, and 4) to use the TAP to transport the scan data into or out of the chip.

The advantage of re-using the test and debug infrastructure as DDI is that it comes at virtually no extra cost because all IP blocks already contain the required hardware. The disadvantage is the relatively low speed at which they run (10 MHz), compared to a functional interconnect. Moreover, access to the state is sequential per scan chain, where a dedicated functional interconnect [36] could provide faster memory-mapped access, although at a higher area cost. Reading and modifying the IP and NOC state can be offered by both alternatives.

E. Debug control interconnect

In this section we describe how the monitors, EDI, and DDI are programmed and together offer message-based communication-centric debug.

The debug infrastructure (monitors, the DDI, clock controller) is programmable at run time. To be precise, it contains a number of programmable registers, to specify information such as: the data pattern to be matched by the monitor [11], [12], AC-TPR scan chain selection, the run / stop / second forced stop state of the EDI, the run / stop state of each NI shell and IP block, and the clock control registers.

The task of the DCI is to provide access to these registers using the TAP. The DCI can be implemented with a dedicated interconnect, such as a (low-speed) control bus (e.g. APB). Alternatively, existing interconnects can be re-used, such as the scan chains (extending [29], [37]) or the functional NOC (proposed for monitors in [11]). In our experiments we use the manufacturing-test scan chains for the DCI, like we did for the DDI. All debug control registers are accessible via control scan chains that are separate and independent from the (data) scan chains containing the state of NOC and IP cores. The control registers are readable and programmable also when the SOC is in functional mode. The control scan chains are accessed via the TAP at the debug clock frequency.

To program the monitors, and to read out NOC registers after a monitor event caused the NOC to stop, the following steps are performed:

1) On *reset* the TAP controller, TCBs, EDI and monitors are disabled.
2) Using the TAP (Fig. 6[A]), from outside the chip at the debug clock frequency, *program* the monitors with the desired pattern to be matched.
3) An *event* generated by a monitor is indicated in a monitor register, and is distributed by the EDI to all NIs, and all relevant valid and accept signals are masked (Fig. 6[B]).
4) At the same time, using the TAP and DCI *poll* from off-chip if the event has already occurred by reading out the stop module registers (Fig. 6[C]). Note that these registers are in the (fast) functional clock domain, and that they are polled from the (slow) debug clock domain through the control scan chains. Polling takes place independently from the NOC that continues to operate at its functional frequency. This is not a problem because once stopped, the debug state is stable.
5) When an event has been detected, wait until all ongoing messages have terminated and the NOC is in a *quiescent state*. This can be checked by polling the "stopped" debug status registers in the NIs, like the stop modules debug status registers.
   Alternatively, a second stop can be sent via the TAP controller (Fig. 6[A&C]) to *forcefully* stop all ongoing message handshakes, whether they have finished or not. Note that because the debug clock frequency is much lower than the EDI frequency, the rising edge of this TAP stop signal is detected and used to accomplish this safely.
6) Then, the *clock controller* is programmed via the TAP (Fig. 6[D]) to switch the NOC and/o IP blocks from the functional to debug clock.
7) At this point, the TAP controller and the NOC and IP blocks all operate at the debug clock frequency, and the state of the NOC and IP blocks (registers and hardware FIFOs) can be read out and/or modified (Fig. 6[F]) via the TAP (Fig. 6[G]) after programming the AC-TPR registers, which select the scan chains for scan out (Fig. 6[E]).

These steps are performed by off-chip debug software. All steps except 5 are required for traditional computation-centric debug. The following section illustrates some of these steps with simulation results.

F. Design flow

To apply the debug concepts and architecture we use the standard NXP design flow. The NOC is synthesised with gate-level synthesis tools, resulting in a netlist. Scan chains are then inserted in the netlist. The clock controller, reset controller, and TAP controller are inserted, before the design lay-out can be generated. In our example, we used register-based FIFOs instead of the optimised hardware FIFOs, to simplify the design flow.

VI. Results

We applied the debug architecture described in the previous section to a simple network containing two routers connecting two masters and two slaves, each with a dedicated NI. We focus on one master-slave pair, the other pair just generates traffic on the shared link between the two routers. In general, the number of monitors depends on the desired coverage of NOC components and IP blocks [38]. Here, each router has a monitor, which can observe all of its incoming links.

A. Performance and cost

Our proposed architecture is *modular*, and the monitors, event and data and control interconnects can be dedicated or re-used, and be different or the same. We have chosen for a dedicated event distribution interconnect (EDI) to ensure a quick reaction to event, to minimise missing data when the NOC is stopped. The EDI makes use of the NOC clock and lay-out, and therefore scales well in size. The data and control distribution interconnects (DDI and DCI) re-use the existing scan chains.

In terms of *area*, the cost of the monitors depends entirely on the desired functionality. Our monitor occupies $0.006mm^2$ in 0.13-micron CMOS, most of which is due to its programmable registers. The EDI consists of as many stop modules as there are routers, two in our example. The area of a stop module is negligible ($0.0002mm^2$). The DDI, DCI, and TAP controller do not add any area because they are present in all designs. The total area of the simple NOC was $2.26mm^2$, and the debug infrastructure added 4.5%. If the NOC area is around 4% of that of a SOC [39], the cost of SOC-level debug is less than 0.2%.

Speed. The EDI operates at the NOC functional speed, or 500 MHz in 0.13 micron CMOS. It can run faster, but we chose to share the NOC clock, for ease of integration. The maximum time it takes the EDI to distribute an event is equal to the the NOC functional clock period times the longest path in the NOC, assuming monitors are attached to NIs and/or routers. The entire test and debug infrastructure, i.e. DDI, DCI, TAP, and TAP controller, runs at 10 MHz. To scan out the 43050 registers in our small NOC only takes 4.3 milliseconds. In practice, the scan-out speed is much lower, around one state dump per second, due to restrictions in memory size of debugger hardware, and the low efficiency of board to PC communication. This is sufficiently fast because the human debugging the system usually requires more time.

B. Simulations

Figure 7 contains three sets of traces that show the debug architecture in action. The master communicates with a slave on DTL ports, via intermediate masterside_NI_shell, masterside_NI_kernel, two routers, slaveside_NI_kernel, and slaveside_NI_shell. The traces show the request and response signals at the masterside_NI_shell port (MNIP) and at the slaveside_NI_shell NI port (SNIP). The former is a target port, the latter an initiator port.

Normal operation: The top trace in Figure 7 shows normal operation, where the MNIP accepts four commands from the master, shown by the four marked pulses on the dtl_cmd_accept (labelled "write" and "read") immediately below the clock signal. The first and third command are writes (dtl_read_cmd is low), the others are read commands. The last of the eight write data words ("wdata") of the first write command is signalled by a '1' on the dtl_write_last signal. The write command and the write data are transported from the MNIP to the SNIP and offered to the slave (dtl_cmd_valid is high), as illustrated by the solid arrows. Note that the write data that was offered contiguously to the MNIP arrives spread over time at the slave, because the master and slave using a GS connection that happens to use non-contiguous TDMA slots.

Similarly, as illustrated with the dashed arrows in Figure 7, the read command is accepted by the MNIP, transported to the SNIP and then offered to the slave (dtl_cmd_valid is high). The slave responds with read data ("rdata"), which is transported and offered to the master (dtl_rd_valid). The master accepts the read data before offering another write and read command. Note that the writes are posted, and that the write and read commands are pipelined.

Transaction-level debug: In this scenario (Figure 7, middle set of traces), the monitor at the router connected to the slave NI triggers an event and generates a stop signal for all NIs. The event is raised immediately after the first read command. For transaction-level debug, only the master NI reacts to the stop signal (stop_in, in dashed circle). Thus, as during normal operation, the write and read commands are offered to the slave that reacts as before. The master also accepts the read data, and thus finishes all outstanding transactions. As explained in Section V-B, the NI shell tracks the completion of messages. It does not accept any new commands after the stop event, even when the master offers a new write (dtl_cmd_valid is high). This is illustrated by the absence of the second write and read commands.

The NI shell also tracks the number of outstanding transactions ("livetrns"), which goes low after the first read has completed. This, together with the asserted stop signal, blocks the shell ("blocked" goes high, see dashed circle). The TAP controller polls the blocked signal of the master NIs to observe when there is no longer any activity in the NOC. At this point it can lower the NOC clock from functional frequency to the test frequency, and scan out all NOC state. This is not shown in this and the next signal trace for lack of space.

Message-level debug: Message-level debug (Figure 7, bottom set of traces) is similar to the transaction-level debug. The same event is raised but now results in a stop signal to both master and slave NIs. The stop signal originates from the router that is closer to the slave NI, and therefore reaches the slave NI earlier than the master NI. In fact, it reaches the slave NI before the write command and data do. As a result, the message handshake is not initiated, and the NI shell keeps the write command and data in its FIFOs and does not offer them to the slave. This is evident from the absence of a pulse on the dtl_cmd_valid. Note that unlike for transaction-level debug, the master NI shell does not enter a blocked state because the write request remains pending at the slave NI.

VII. Related work

In the domain of multi-processors and their programming [8], [40] describe a system to reproducibly replay parallel program executions by saving traces between different computation threads. [41], [42] aim for the same by focussing on interactions using shared variables. These works focus on post-hoc replay and analysis of parallel programs, whereas we focus on actively controlling the concurrency.

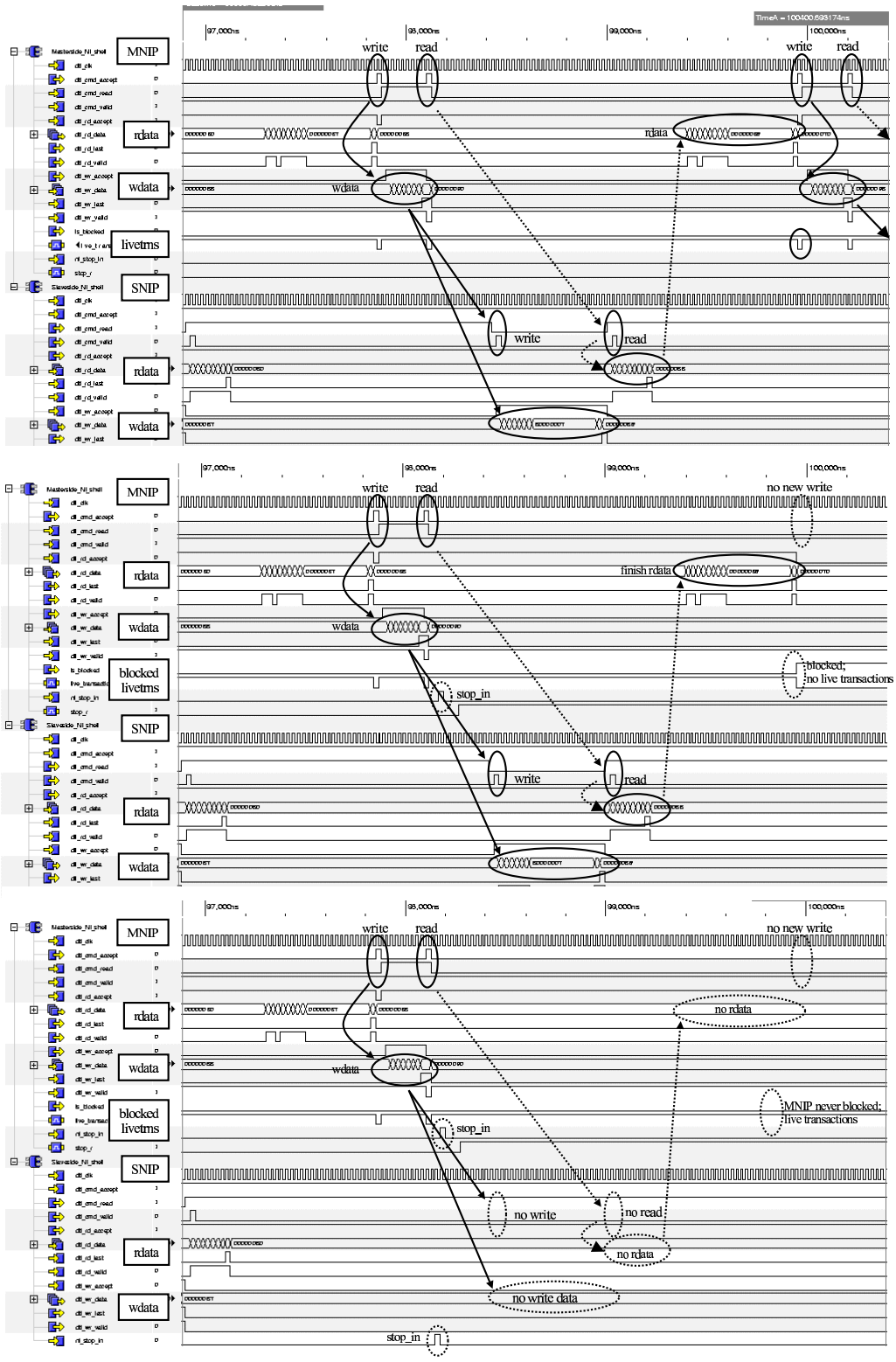

Fig. 7. Gate-level signal traces.

A good overview of SOC debug can be found in [5]. Our work is based on TAPs, like [33], [29], [37], [13]. Their approach is computation-centric whereas our approach is communication-centric instead.

[36] proposes TAP-based debug. In contrast to our DDI, their TAP controller uses the functional interconnect to access the state of processors. This assumes that the functional interconnect is stateless, which is not the case for NOCs. The NOC would have to be flushed of data before use by the TAP controller, and even more difficult, the state would have to be reinstated before single stepping or resuming operation.

[43] tackles the problem of debugging multiple processors using simulation with instruction-set simulators. The interconnect, or silicon debug are not considered.

[44] proposes synchro-tokens that are similar to our flit handshake for test and debug of GALS SOCs. However, they do not use their method for a communication-centric debug as proposed here.

VIII. Conclusions

In this paper we addressed the debugging of complex SOCs. This is hard because they contain multiple processors that interact through concurrent interconnects, such as NOCs. We classify the scope and temporal granularity of debug. We show that the scope of debug extends beyond the traditional debugging of processors, to include debugging of the interconnect, and system-level debug that concentrates on the interactions between IP blocks. Therefore, our debug methodology is *communication-centric.* Furthermore, processor and interconnect monitoring and debug coincide at a few points: the instruction/flit level, which is the smallest grain of control; the *message level* where interactions between master and slave IP blocks are visible, and the *transaction level*, where master behaviour alone is traced.

Based on these insights we define and implement a modular debug architecture, based on a NOC, monitors, and a dedicated high-speed event-distribution broadcast interconnect. The data distribution interconnect (DDI) and debug control interconnect (DCI) re-use the manufacturing-test scan chains and IEEE1149.1 test access ports (TAP).

To apply our debug concepts of the functional SOC only the NI shells require changes. The additional area cost for debug is limited to the monitors and the event distribution interconnect, which are 4.5% of the NOC area, or less than 0.2% of the SOC area. The debug architecture runs at NOC functional speed and reacts very quickly to debug events to stop the SOC close in time to the condition that raised the event. The speed at which data is retrieved from the SOC after stopping is 10 MHz, which is sufficient.

We proved our concepts and architecture with a gate-level implementation of a small SOC, consisting of behavioural IP blocks, gate-level NOC with scan chains, the broadcast event distribution interconnect, and clock, reset, and TAP controllers. Gate-level signal traces illustrated debug at message and transaction levels.

References

[1] ARM, "Multi-layer AHB. overview." 2001.

[2] L. Benini and G. De Micheli, "Networks on chips: A new SoC paradigm," *IEEE Computer*, vol. 35, no. 1, pp. 70–80, 2002.

[3] A. Jantsch and H. Tenhunen, Eds., *Networks on Chip*. Kluwer, 2003.

[4] C. E. McDowell and D. P. Helmbold, "Debugging concurrent programs," *ACM Computing Surveys*, vol. 21, no. 4, pp. 593–622, 1989.

[5] A. Hopkins and K. McDonald-Maier, "Debug support for complex systems on-chip: A review," *IEE Proceedings Computers and Digital Techniques*, vol. 153, no. 4, pp. 197–207, July 2006.

[6] R. Leatherman and N. Stollon, "An embedded debugging architecture for SoCs," *IEEE Potentials*, vol. 24, no. 1, pp. 12–16, Feb-Mar 2005.

[7] *CoreSight: V1.0 Architecture Specification*, ARM.

[8] S. Grabner, D. Kranzmüller, and J. Volkert, "Debugging of concurrent processes," in *Proc. Int'l Parallel and Distributed Processing Symposium (IPDPS)*, Jan. 1995, pp. 547–554.

[9] S. K. Goel and B. Vermeulen, "Hierarchical data invalidation analysis for scan-based debug on multiple-clock system chips," in *Proceedings IEEE International Test Conference (ITC)*, Oct. 2002, pp. 1103–1110.

[10] K. Goossens, J. Dielissen, and A. Rădulescu, "The Æthereal network on chip: Concepts, architectures, and implementations," *IEEE Design and Test of Computers*, vol. 22, no. 5, pp. 414–421, Sept-Oct 2005.

[11] C. Ciordaş, T. Basten, A. Rădulescu, K. Goossens, and J. van Meerbergen, "An event-based monitoring service for networks on chip," *ACM Transactions on Design Automation of Electronic Systems*, vol. 10, no. 4, pp. 702–723, Oct. 2005.

[12] C. Ciordaş, K. Goossens, T. Basten, A. Rădulescu, and A. Boon, "Transaction monitoring in networks on chip: The on-chip run-time perspective," in *Proc. Symposium on Industrial Embedded Systems (IES)*, Oct. 2006.

[13] B. Vermeulen, T. Waayers, and S. Goel, "Core-based Scan Architecture for Silicon Debug," in *Proceedings IEEE International Test Conference (ITC)*, Baltimore, MD, USA, Oct. 2002, pp. 638–647.

[14] *Device Transaction Level (DTL) Protocol Specification. Version 2.2*, Philips Semiconductors, July 2002.

[15] *AMBA AXI Protocol Specification*, ARM, June 2003.

[16] OCP International Partnership, "Open core protocol specification," 2001.

[17] T. Bjerregaard, "The MANGO clockless network-on-chip: Concepts and implementation," Ph.D. dissertation, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2006.

[18] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, "Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip," in *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2004.

[19] E. Beigne, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin, "An asynchronous NOC architecture providing low latency service and its multi-level design framework," in *Proc. Int'l Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2005.

[20] P. Guerrier and A. Greiner, "A generic architecture for on-chip packet-switched interconnections," in *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2000, pp. 250–256.

[21] D. Bertozzi and L. Benini, "Xpipes: A network-on-chip architecture for gigascale systems-on-chip," *IEEE Circuits and Systems Magazine*, pp. 18–31, 2004.

[22] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "QNoC: QoS architecture and design process for network on chip," *Journal of Systems Architecture*, vol. 50, no. 2–3, pp. 105–128, Feb. 2004, special issue on Networks on Chip.

[23] ARM, "AMBA specification. rev. 2.0," 1999.

[24] A. Hansson, M. Coenen, and K. Goossens, "Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip," in *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Apr. 2007.

[25] A. Rădulescu and K. Goossens, "Communication services for networks on chip," in *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*, S. S. Bhattacharyya, E. F. Deprettere, and J. Teich, Eds. Marcel Dekker, 2004, pp. 193–213.

[26] A. Rădulescu, J. Dielissen, S. González Pestana, O. P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens, "An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming," *IEEE Transactions on*

CAD of Integrated Circuits and Systems, vol. 24, no. 1, pp. 4–17, Jan. 2005.
[27] J. D. Day and H. Zimmerman, "The OSI reference model," in *Proceedings of the IEEE*, vol. 71, 1983, pp. 1334–1340.
[28] Y. H. Song and T. M. Pinkston, "On message-dependent deadlocks in multiprocessor/multicomputer systems," in *Proc. HiPC*, 2000.
[29] G. Rootselaar and B. Vermeulen, "Silicon Debug: Scan Chains Alone Are Not Enough," in *Proceedings IEEE International Test Conference (ITC)*, Atlantic City, NJ, USA, Sept. 1999, pp. 892–902.
[30] A. Hansson and K. Goossens, "Trade-offs in the configuration of a network on chip for multiple use-cases," in *Proc. Int'l Symposium on Networks on Chip (NOCS)*, May 2007.
[31] IEEE Computer Society, *IEEE Standard Test Access Port and Boundary-Scan Architecture-IEEE Std 1149.1-2001*. IEEE Press, 2001.
[32] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.
[33] K. Holdbrook, S. Joshi, S. Mitra, J. Petolino, R. Raman, and M. Wong, "microSPARC: A case study of scan-based debug." in *Proceedings IEEE International Test Conference (ITC)*, 1994, pp. 70–75.
[34] E. Rijpkema, K. Goossens, A. Rădulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander, "Trade-offs in the design of a router with both guaranteed and best-effort services for networks on chip," *IEE Proceedings: Computers and Digital Techniques*, vol. 150, no. 5, pp. 294–302, Sept. 2003.
[35] P. Wielage, E. J. Marinissen, M. Altheimer, and C. Wouters, "Design and DFT of a high-speed area-efficient embedded asynchronous FIFO," in *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2007.
[36] K. D. Maier, "On-chip debug support for embedded systems-on-chip," in *Proc. Int'l Symposium on Circuits and Systems (ISCAS)*, vol. 5, May 2003, pp. V–565–V–568.
[37] B. Vermeulen and G. van Rootselaar, "Silicon debug of a co-processor array for video applications," in *Proc. Workshop on High-Level Design Validation and Test (HLDVT)*. Los Alamitos, CA, USA: IEEE Computer Society, 2000, pp. 47–52.
[38] C. Ciordaş, A. Hansson, K. Goossens, and T. Basten, "A monitoring-aware NoC design flow," in *Proc. Euromicro Symposium on Digital System Design*, Aug. 2006.
[39] F. Steenhof, H. Duque, B. Nilsson, K. Goossens, and R. Peset Llopis, "Networks on chips for high-end consumer-electronics TV system architectures," in *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, vol. 2. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, Mar. 2006, pp. 148–153.
[40] T. J. Leblanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Transactions on Computers*, vol. C-36, no. 4, pp. 471–482, Apr. 1987.
[41] R. Carver and K.-C. Tai, "Replay and testing for concurrent programs," *IEEE Software*, vol. 8, no. 2, pp. 66–74, Mar. 1991.
[42] A. J. Goldberg and J. L. Hennessy, "Mtool: An integrated system for performance debugging shared memory multiprocessor applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 28–40, Jan. 1993.
[43] A. Wieferink, T. Kogel, R. Leupers, H. eyr, A. Nohl, and A. Hoffmann, "A generic tool-set for SoC multiprocessor debugging and synchronization," in *Proc. Int'l Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, June 2003, pp. 161–171.
[44] M. W. Heath, W. P. Burleson, and I. G. Harris, "Synchro-tokens: A deterministic GALS methodology for chip-level debug and test," *IEEE Transactions on Computers*, vol. 54, no. 12, pp. 1532–1546, Dec. 2005.