Architecture and Design Flow for a Debug Event Distribution Interconnect

Arnaldo Azevedo
Department of Software and Computer Technology
Delft University of Technology
Delft, The Netherlands
a.p.pereiradeazevedofilho@tudelft.nl

Bart Vermeulen
NXP Semiconductors
Eindhoven, The Netherlands
bart.vermeulen@nxp.com

Kees Goossens
Electronic Systems Group
Faculty of Electrical Engineering
Eindhoven University of Technology
Eindhoven, The Netherlands
k.g.w.goossens@tue.nl

***Abstract*—In this paper, we describe and analyze the architecture of the proposed Debug Event Distribution Interconnect (EDI). The EDI transmits debug events, which are 1-bit signals, between debug entities in different areas of the Network-on-Chip based Multi-Processor System-on-Chip. The EDI replicates the NoC topology with an EDI node instantiated for each underlying NoC data module. Contention in the EDI node is handled by replicating the EDI in layers. The EDI generation is automatic, and uses as input the cross-triggering patterns that are not required to follow the communication patterns in the NoC. The generation and routing tool is also presented in this paper. The EDI is evaluated with four different implementations varying complexity and handling of contention. The area of a single EDI Layer is around 0.9% of the area occupied by the tested NoCs, using the lower area implementation. These results show that the proposed implementation of the EDI incurs low cost on the overall system.**

I. Introduction

Future multi- and many-core processors will require a scalable infrastructure for debugging purposes. With the increasing number of cores on chip, the processor interconnect will have to shift to a more scalable option, such as Network on Chip (NoC). This is the case of Intel Knight Corner accelerator that features more than 50 cores [10].

Transaction based debug techniques for debugging NoC based multi- and many-core processors and Multiprocessor Systems on Chip (MPSoC) that use distributed debug modules have gained increased attention from the academia and industry. Implementations of such approaches are described in [15], [12], [6], [8], [2]. In these approaches, data transfers between cores and/or memories are used as breakpoint entities. In other words, communication is used in the same way as source code lines in software debug. These approaches focus on the interaction between processing cores as, for instance, bugs caused by race conditions [11]. To implement these approaches, two kinds of module are inserted in the interconnect. The first module, the monitor, observes the traffic on a specific link between two modules. It generates a debug event when the observed traffic matches the programmed pattern. The second module, the Protocol Specific Instrument (PSI), stops transactions when receiving a debug event and acts as a breakpoint unit.

Debug events need to be transmitted from monitors to PSIs. This requires either the adaptation of the NoC to transfer this information in-band [13], or a separated infrastructure [15], [6], [8]. The infrastructure can be tied with the data interconnect or be dissociated with it, such as in ARM CoreSight [2].

In this paper, we propose a specialized infrastructure to transport debug events in NoC-based MPSoCs. The proposed debug Event Distribution Interconnect (EDI) is a circuit switched, multicast, application-specific NoC. The EDI is programmable and automatically generated following the topology of the data NoC. The debug engineer defines under which circumstances a debug event will be generated by a monitor and to which monitor(s) and/or PSI(s) the debug event will be propagated. In this work, we consider the debug event to be, without losing generality, a 1-bit signal. As a result data can be duplicated for multicast and broadcast. Multicast and broadcast are required to halt the communication in several links on the MPSoC while a unicast is required to implement, among others, distributed cross-triggering. Also, the notion of contention is different than data NoCs, as detailed later on this work. The EDI is programmed via a scan chain, which is a separate control interconnect. The EDI modules have lower latency than NoC modules. The routing tool can therefore guarantee that the debug event arrives earlier than the data of the communication it is designed to halt. The AElite [9] is used as the underlying NoC.

The main contributions of this paper are as follows. We introduce a hardware architecture of the EDI that meets the requirements for the debug event transportation without interference in the underlying data NoC, and with small area. The proposed EDI allows distributed cross-triggering of debug events and enables the halting of communication links that are not part of the communication that triggered the debug event. We also present a design flow that automatically dimensions and generates the hardware of the EDI and the configuration to be used at run-time. The EDI satisfies design time Debug Use Cases (DUCs). Post silicon DUCs can be automatically mapped if enough resources are available. Moreover, multiple implementations of the EDI with different notions of contention and implementation costs are benchmarked.

This work was partially funded by projects EU FP7 288008 T-CREST and 288248 Flextiles, Catrene CA104 Cobra, and NL STW 10346 NEST.

The remainder of this paper is organized as follows. Section II briefly presents related work that deal with MPSoC debugging and specialized interconnects for debugging purposes. The debug infrastructure supported by the proposed interconnect is presented in Section III with the proposed specialized interconnect for debug communication itself being introduced in Section IV. Three additional implementations of the EDI with different contention handling are presented in Section V. It is followed by the description of the supporting generation tool in Section VI. In Section VII, an evaluation of the design options is presented. Conclusions are drawn in Section VIII.

II. Related Work

In this section, we briefly describe related work on debugging of MPSoCs and also of multi-core processors in general. The main attention will be given to the interconnect of the analyzed solutions.

This work extends the communication centric debug infrastructure proposed by Goossens et al. [8]. The approach to stop communications on a link after a specific attribute is found is described as well as monitors and PSIs. No detailed description, however, is given on the interconnect. Our aim is to close this gap and evaluate different options for the EDI.

Wen et al. [15] present a debug infrastructure for multi-core processors for detecting race conditions. In their work a monitor observes the communication of a cluster of processors and are connected in a ring topology. Similarly, ARM CoreSight [2] connects its debug modules via a bus.

Fiorin et al. [6] proposes another monitoring architecture for MPSoCs. The main focus of their work is monitoring resources for performance tuning, but the authors claim it can be easily adapted for debug. The authors propose using the NoC to transmit the observed attributes and events, because the monitors can store several attributes of the observed communications. The data is collected after the application finishes. This is different from our proposal as our focus is to perform intervention on the system when a faulty characteristic is observed.

Tang and Xu [13] propose an in-band debug event transmission. In this approach, the debug event is embedded in the communication packet by the monitor. Its PSI analyzes the incoming communication for the debug event. The debug event also carries more information that is used to signal which operation the PSI should perform. While this approach does not incur overhead in terms of silicon area, it limits the debug possibilities. It does not allow the debug event to be transmitted to a PSI that is not located in the packet path. Our proposed solution, on the other hand, allows for total flexibility in defining DUCs. This flexibility is important, for instance, to stop communication to all modules of a distributed memory (such as data and control memories) when a wrong memory access is requested.

III. Debug Infrastructure

To support the distributed cross-triggering debug approach [8], our approach requires four main components. The first component, the monitor, is a module that observes the communication for pre-programmed attributes and generates the debug event. The debug event generated by the monitor is propagated via the Event Distribution Interconnect (EDI), our specialized NoC and second required component, to other monitors and/or to the PSI modules. PSIs are modules that halt the communication on a link on the arrival of a debug event. The last component of our approach is the DUC. A DUC is a configuration of the debug infrastructure that maps the debug capabilities designed by the debug engineer corresponding to the data NoC's communication requirements. This section will present more details of each of these components, with the exception of the EDI that is presented in Section IV.

A. Monitors and PSIs

A monitor observes a communication link [5], [4], looking for a specific (set of) attribute(s) in the communication on the link. When a pre-programmed attribute is found on the observed communication, the monitor generates a debug event. This attribute can be an address range, a data value, a protocol command, etc, depending on where the monitor is instantiated, and on the protocol of the link.

To allow distributed cross-triggering of events, along with observing the communication link, we assume that monitors can also be triggered by a combination of connection events and debug events that are generated by other monitors. This capability allows triggering debug events in the presence of different attributes on different links. This is important for detecting, for instance, bugs caused by race conditions [11].

In the presence of a debug event, PSIs [7] are responsible for halting (stopping, single-stepping) the communication on the connection. Halting of the communication is implemented by gating the handshaking signals of the link accordingly to the protocol being utilized. With the handshaking signals gated, request and/or response signals are not accepted or propagated, which prevents the data transmission to finish. To be able to do so, the PSI is usually instantiated between two modules, such as an IP block, a router, or a bus, thereby physically splitting the link.

Details of monitors and PSIs depend on the specific link they are placed on. Their actual functionality can also change accordingly with the actual methodology being implemented. The analysis of monitors and PSIs is out of the scope of this paper and thus their impact, for instance in floorplan area cost, are not taken into consideration.

B. Debug Use Cases

The debug engineer can define several DUCs to be supported by the system. The routing of the debug events through the EDI is performed automatically and will be described in Section IV.

A Debug Connection is one producer and one or more consumers of a debug event. A monitor is a producer of a debug event, while monitors and PSIs are consumers. Figure 1 depicts several examples of Debug Connections. In this figure, monitors are labeled *M#*, PSIs *S#*, and Debug Connections

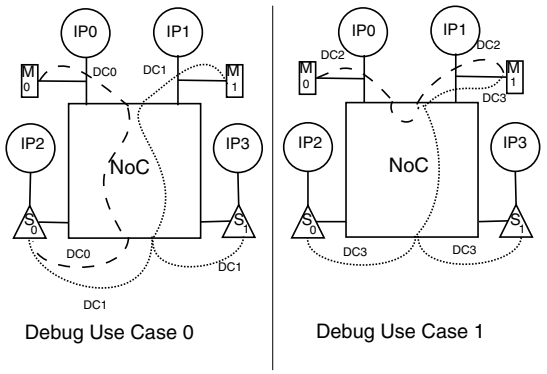

Fig. 1. Example of Debug Use Cases

DC#, where # is a number that identify a specific instance. *DC0* (dashed line) is a simple example of a Debug Connection composed of the monitor *M0* and the PSI *S0*. In this case, any data connection to the on-chip memory will be blocked when *M0* generates a debug event. A Debug Connection may or may not follow the path of the observed communication.

A debug event can also be used to block several data connections. This is depicted in Figure 1 by *DC1* (dotted line), where *S0* and *S1* block their respective data communications in the presence of a debug event from *M1*. This is useful to completely block part, or even all, of the system given a specified condition is observed by a monitor.

Debug Connections can be chained together in order to perform more complex event triggering through distributed cross-triggering. This is achieved with Debug Connections between monitors. As described in Section III, monitors can generate a debug event based on a combination of data attributes on the observed connection, and one or several debug events from other monitors. By chaining Debug Connections, and consequently debug event generation conditions, the Debug Infrastructure can handle distributed characteristics to stop the system in a coherent state [14]. In our example, *DC2* is chained to *DC3* via *M1*. *M1* is programmed to generate a debug event in the case when an attribute is observed in the communication and a debug event is received from *M0*.

In order to reduce the debug area overhead on the MPSoC, it is not required to have all desired Debug Connections in a single DUC. The same debug infrastructure can be re-utilized to implement other Debug Connections, if they are not required to be available simultaneously.

If all the Debug Connections described above need to be used simultaneously, it requires a more complex implementation of the monitors (as they are required to simultaneously observe more than one attribute), or the use of two monitors to observe a NoC link. These Debug Connections can be partitioned into separated DUCs, if the related events are not required to be observed simultaneously. For instance, the mentioned Debug Connections can be partitioned in two DUCs, the first containing *DC0* and *DC1* (*DUC0*) and the second *DC2* and *DC3* (*DUC1*), as depicted in Figure 1. In the depicted example, the monitors are used by both DUCs, however, with different functionality, reducing the overhead of the debug infrastructure.

IV. EDI Basics

Events generated by the monitors are transmitted to their target monitor(s) and/or PSI(s) via the EDI. The EDI is a specialized a circuit switched interconnect physically separated from the chip interconnect used to communicate data. The EDI follows the overall topology of the underlying NoC with an EDI node instantiated for each underlying NoC module, such as network interfaces, routers, and local buses. The EDI nodes are connected if there is a respective data link between the NoC modules. The EDI has several layers to accommodate the designer-specified DUCs and it is configured via a dedicated IEEE 1149.1 connection [1].

In this section, the EDI is described, including the reasons for a specialized NoC, the topology, the router architecture, the routing strategy and restrictions, the configuration system, and the support for multiple clock domains.

A. Topology

Although the EDI is physically separated from the underlying data NoC, it is based on the topology of the latter. An example of a NoC with its associated EDI is depicted in Figure 2. In the picture, processing components are labeled as *IP*, NoC network interface as *NI*, NoC routers as *R*, monitors as *M*, and PSIs as *S*. Solid lines depict a data link between modules and dotted lines depict EDI links between EDI nodes, which are depicted as gray boxes. As depicted in Figure 2, each NoC module is associated with an EDI node. This is the case for both logical and physical placement (chip floorplan). The reasons to have the EDI topology tied with the NoC topology are to guarantee that the EDI link will not become the critical path in the system and to benefit from the scalability properties of NoCs.

The links between the EDI nodes are determined by the existing data links between NoC modules. A pair of unidirectional 1-bit link is instantiated between EDI nodes where there is a respective NoC link. The EDI link, however, does not reflect all the properties of the NoC link. The link between EDI nodes is always bi-directional regardless of the NoC link (unidirectional or bidirectional). Also, a single bi-directional EDI link is instantiated even when there are multiple direct links between the NoC modules.

B. Broadcast EDI node

The EDI nodes are responsible for broadcasting and routing the debug events in the EDI. As previously mentioned, EDI nodes are placed alongside the modules of the data NoC, as depicted in Figure 2.

The EDI node works by propagating a debug event in any of its input ports to all output ports on the same EDI Layer. For routing purposes, the output ports of the EDI node can be gated. A route is configured in the EDI node by setting the content of a mask register associated with each output port. The architecture of a 4-port EDI node is depicted in

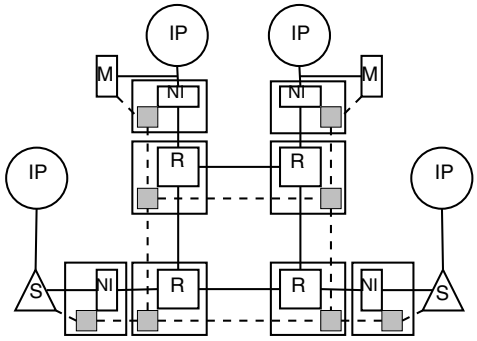

Fig. 2. Example of EDI Topology

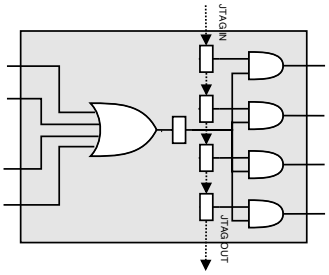

Fig. 3. Basic EDI node architecture

Figure 3. The mask registers are shift registers that implement the JTAG [1] network, and will be discussed on Section IV-D.

C. EDI Layers

In the EDI, several debug events can have a single source or destination, or even be routed through the same EDI node or Link, causing resource contention. Because the debug events do not carry their destination information, the adopted solution for contention is to replicate the EDI topology in different layers. Each EDI Layer is a complete replication of the EDI topology and is instantiated when more than one debug event has to be routed through an EDI node, or when more than one debug event is generated by or targeted at the same monitor or PSI.

The use of EDI Layers to handle contention aims to keep the required area for the debug infrastructure to its minimum. Routing more than one debug event on the same EDI Link would require packets or circuits, leading to an increase in complexity, and thus overhead, in the EDI. Another property of the EDI Layers is that they are not interconnected. This means that a debug event assigned to a specific layer cannot change layer, even if there are more available resources on a different layer. This is due to the increase in EDI node complexity and the relatively low associated reduction in EDI Layers, as will be demonstrated in Section VII.

D. EDI Configuration

The EDI is configured using a separated JTAG [1] that connects all the debug modules. All registers in monitors, PSIs and EDI nodes are part of this scan chain. Upon reset the values of the registers are set such that all components are inactive. To activate the modules, a bitstream file is loaded via a Test Access Port. The bitstream carries the activation values and the DUC to be configured in the EDI. The modules do not perform while a configuration is being loaded.

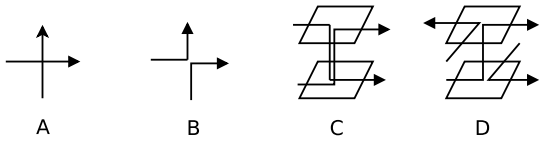

Fig. 4. Examples of contention types

The loading of a new configuration is also the way that run/stop debug is implemented. After a communication is blocked, a new bitstream needs to be loaded in order to deactivate the PSI, allowing the handshaking signal to be propagated.

V. Advanced EDI Nodes

In this section we present advanced EDI node designs. These design extend the capabilities of the Broadcast node described in Section IV-B while keeping the circuit switched interconnect characteristics. The configuration of the EDI Nodes is performed as described in IV-B.

The *Routing node* routes the event arriving from an input port only to the selected output ports. This allows routing several events on the same EDI Layer, as long as there is no conflicts on the input or output ports of the events being routed. Because in this architecture each output port has to select from which input port it accepts events, the cost of this alternative grows quadratically with the number of ports.

If two debug events have to be routed through the same input or output port in an EDI node, a debug event identification mechanism, such as destination addressing, would be required. This would lead to extra complexity of the EDI nodes and additional wires on each EDI Link to carry the extra information. Since the debug event is a single bit signal, any extra information would incur a linear link overhead.

BroadCross node is the Broadcast node extended with the capability of routing events across EDI Layers. In other words, an event arriving in an EDI Layer can be routed to an output port belonging to a different EDI Layer. This alternative architecture can reduce the number of layers by allowing a more efficient usage of the links between EDI nodes. This option incurs, however, a quadratic increase with the number of layers, related to the Broadcast node. The *RouteCross node* has same capability described above but applied to the original Routing node.

Each of the architectures described above can handle a specific combination of types of contention. Figure 4 depicts examples of contention. Contentions labeled *A* and *B* depict contentions caused by two or more debug events being routed through one EDI node. Contention labeled *C* depicts one signal per layer being accepted and routed to another layer, while type *D* depicts multiple debug events being routed per layer and across layers. Table I presents which types of contention can be handled by each described architecture and its cost.

TABLE I
CONTENTION HANDLING BY EACH EDI NODE ARCHITECTURE

	Type of Contention				
Architecture	A	B	C	D	EDI node cost
Broadcast	×	×	×	×	$ports$
Routing	✓	✓	×	×	$ports^2$
BroadCross	×	×	✓	×	$ports \times layers$
RouteCross	✓	✓	✓	✓	$ports^2 \times layers$

VI. EDI DESIGN FLOW

To generate the EDI and instantiate monitors and PSIs we developed the the EDIdim tool. The EDIdim tool receives the description of the NoC, the location of monitors and PSIs, and the description of DUCs as inputs. The tool is responsible for generating and dimensioning the EDI, for inserting the scan chain, for routing the DUCs, and for generating the configuration bitstreams.

The first step of the EDIdim tool is to create the EDI by replicating the underlying NoC topology. This is performed by instantiating an EDI node to each NoC module and followed by the instantiation of EDI links between the EDI nodes. To each pair of NoC modules that have a data link between them, the respective EDI nodes are connected by a pair of unidirectional EDI links, as described in Section IV.

To route a Debug Connection, first the Dijkstra algorithm [3] is used to find the shortest path between the monitor that generates the event, the initiator, and the first target module, another monitor or a PSI. The Dijkstra algorithm is applied over a graph representing the EDI topology. The distance value between two nodes is used as a contention value initialized as *1*. The list of nodes between the initiator and target, inclusive, that carries an debug event, following its visiting order is called a Debug Route.

For the case of multiple targets in a debug connection, first a Debug Route between an initiator and a target is routed. For each additional target, each node in the route becomes an origin node for the Dijkstra algorithm, and it will be added to the Debug Route. I.e., the routing algorithm will find the shortest route between the target and any node already used in the Debug Route and update it with the result. The objective with this strategy is to reduce the number of nodes in a Debug Route. Although this approach does not generate an optimal solution, it delivers satisfying results with low complexity.

In order to efficiently block an ongoing data communication, the debug event needs to reach the target PSI before the communication. This can be implemented by the edidim tool, by associating a data communication to a Debug Connection. The tool then can check the latency between the signals and reroute the Debug Connection if required. In the utilized NoC for our experiments, a flit has a latency of 3 NoC cycles per hop. The EDI incurs only one NoC cycle delay per hop.

To ensure minimum contention on the resulting EDI, once a debug connection is routed, the nodes in the path have its distance value increased by the number of nodes in the EDI. This is given the fact that for each debug event crossing a EDI node, a new EDI Layer needs to be instantiated. With this strategy the routes will avoid the nodes with high distance values, crossing them only when strictly necessary and for the shortest number of times. This reduction in node contention is expected to translate to a reduced number of layers.

After all Debug Routes are calculated, they are assigned to EDI layers. For each DUC, first an empty layer is created and the first calculated route is assigned to it. Each subsequent route tests for contention on each available layer, starting from layer zero. If there is no contention free layer, a new layer is created and the route is allocated to it. For each DUC a new set of layers is created. After all routes of all DUCs are allocated to layers, the number of layers of the DUC with the highest number of layers is the required total number of EDI layers.

For each DUC, a separated bitstream is generated. For the generation of each bitstream, each node contains a mask register which width is given by the number of ports times the number of EDI layers. First, the content of all mask registers are set to *1*, "do not propagate event". Each route is then visited and, for each module, the specific initiator port is set to *0*. This configures the nodes to propagate the debug events following the given route.

VII. EXPERIMENTAL EVALUATION

In this section, we evaluate the presented EDI against more complex structures in terms of number of layers and resulting logic gate area cost. An evaluation of the total area cost incurred by the debug infrastructure, with highlight to the presented EDI, is also given. To evaluate the proposed EDI in number of layers and area, first alternative architectures of the EDI node are introduced. This is followed by the description of the evaluation method used in the experiment. Finally, the results are presented and analyzed.

To evaluate the benefit of each variation of the EDI node, five implementations of the AElite NoC [9] of different sizes were chosen. The NoCs consist of a 2×2, three 4×4, and one 8×8 router meshes. The three 4×4 topologies variations allowed for 1, 2, and 4 IPs per router. A monitor and a PSI were assigned to each IP. Each NoC was evaluated under light, medium, and heavy Debug Connection (DC) loads. The light load consists of 1 DC per IP, while the medium and heavy loads consist of 2 and 4 DCs per IP, respectively. Twenty DUC were randomly generated to each DC load. Each randomly generated DC in a DUC has a 50% chance of being a 1-to-1 DC, 25% of being a 1-to-2 DC, 12.5% of being a 1-to-3 DC, and so on.

Figure 5 depicts the average number of EDI Layers for each NoC in the three DC loads, where the X in the "$\_nX$" suffix stands for the number of IPs per router. As presented in the figure, the proposed Broadcast EDI node incurs the worst number of layers from the evaluated options. It reaches up to more than four times the required number of layers than the Routing Crossing EDI node for the $4x4\_n4$ NoC, with an average of 74 EDI Layers for 256 DC per DUC where the latter required only 17. The $4x4\_n4$ NoC have a higher demand for EDI Layers because it has the highest ratio of DC

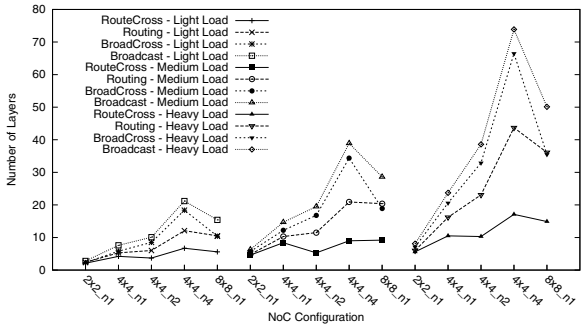

Fig. 5. Number of layers

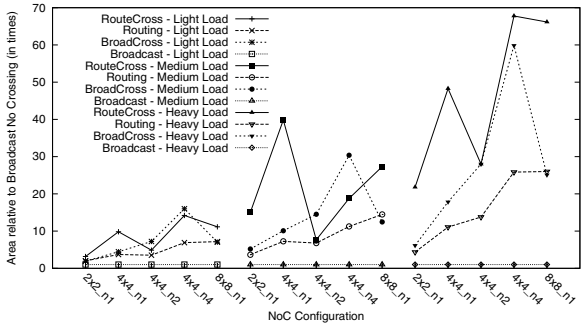

Fig. 6. Area of layers relative to the Broadcast no Crossing

per router. It is also depicted that broadcasting across layers has only a marginal improvement with 20% reduction in the number of layers.

While for the number of layers the Broadcast EDI node has the worst result, it does not translate in the total gate area cost. Figure 6 depicts the gate area of EDI Layers with Routing (with and without crossing) and the BroadCross EDI nodes compared with the area of the EDI Layer using the Broadcast node, accordingly with the EDI node cost presented in Table I.

Because of the simplicity of the Broadcast EDI node, the other implementations yield from 10 to 25 times the area of the EDI using the Broadcast EDI node. The main reasons are the fast growth in area of the alternative implementations and the average number of Debug Connections per layer. The EDI nodes with routing capability (Routing and RouteCross) require a number of mask registers that grows exponentially with the number of ports. While the NI has a low number of ports, the same is not usually the case for routers. Also, the capability of crossing layers, specially, results in large area overhead without a comparable reduction in the number of layers. It reduces the number of layers by half for RouteCross but the cost of each layer increases linearly with the total number of layers. From the number of layers depicted in Figure 5, it is possible to infer that the average number of Debug Connections allocated per layer by the routing tool is around 4 when routing to Broadcast and around 10 for RouteCross. This is only a 2.5 times increase in Debug Connection density.

The area of a single EDI Layer was measured to be around 0.9% of the area occupied by the NoC, where 0.3% is dedicated to the JTAG control in the EDI node and the remaining 0.6% is occupied by the mask registers and logic gates. For a system with 10 EDI Layers, a reasonable quantity as it can support 40 Debug Connections simultaneously; the area cost is around 6.5% of the NoC area. These results show that the EDI incurs low cost on the overall system.

VIII. Conclusions

In this paper we introduced a debug event distribution interconnect that carries the debug events between the debug modules monitor and PSI. The EDI mirrors the topology of the underlying NoC, providing a scalable solution for MPSoCs. The interconnect is composed of low cost nodes that implements routing by gating the output ports. The EDI allows distributed cross-triggering of events that are used to detect race conditions and violation of distributed assertions. The debug events in the EDI are propagated by the EDI nodes that broadcast debug events received in any of its ports to all output ports. Four architectures of the EDI node were evaluated, but while reducing the number of EDI Layers the overall area cost increased. The described routing tool was able to allocate four Debug Connections per layer in average, using the randomly generated test patterns. The area cost of the EDI per layer was measured to be smaller than 1% of the underlying NoC area. A ten-layer EDI was measured to be around 6.5% of the NoC area while allowing for approximately 40 active Debug Connections simultaneously. These results indicate that the proposed solution incurs a very small cost.

References

[1] IEEE Standard Test Access Port and Boundary-Scan Architecture. *IEEE Std 1149.1-2001*, 2001.

[2] ARM Ltd. Coresight architecture specification, 2004.

[3] M. Barbehenn. A note on the complexity of Dijkstra's algorithm for graphs with weighted vertices. *IEEE Trans. on Computers*, 47(2):263, 1998.

[4] C. Ciordas, K. Goossens, T. Basten, A. Radulescu, and A. Boon. Transaction monitoring in networks on chip: The on-chip run-time perspective. In *IES*, pages 1 –10, 2006.

[5] C. Ciordas, K. Goossens, A. Radulescu, and T. Basten. Noc monitoring: impact on the design flow. In *ISCAS*, 2006.

[6] L. Fiorin, G. Palermo, and C. Silvano. A monitoring system for NoCs. In *NoCArc*, pages 25–30, 2010.

[7] K. Goossens, B. Vermeulen, and A. B. Nejad. A high-level debug environment for communication- centric debug. In *DATE*, 2009.

[8] K. Goossens, B. Vermeulen, R. van Steeden, and M. T. Bennebroek. Transaction-based communication- centric debug. In *NOCS*, 2007.

[9] A. Hansson and K. Goossens. *On-chip Interconnect with Aelite: Composable and Predictable Systems*. Springer Verlag, 2010.

[10] Intel Co. Intel many integrated core architecture, 2011. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

[11] E. Larsson, B. Vermeulen, and K. Goossens. Distributed architecture for checking global properties during post silicon debug. In *ETS*, 2010.

[12] S. Tang and Q. Xu. A multi-core debug platform for NoC-based systems. In *DATE*, pages 870–875, 2007.

[13] S. Tang and Q. Xu. In-band cross-trigger event transmission for transaction-based debug. In *DATE*, pages 414–419, 2008.

[14] B. Vermeulen and K. Goossens. Interactive debug of socs with multiple clocks. *IEEE Design Test and of Computers*, 28(3):44 –51, 2011.

[15] C. Wen, S. Chou, C. Chen, and T. Chen. NUDA: A non-uniform debugging architecture and non-intrusive race detection for many-core systems. *IEEE Trans. on Computers*, PP(99):1, 2010.