A Distributed Architecture to Check Global Properties for Post-silicon Debug

Erik Larsson[†], Bart Vermeulen[‡], Kees Goossens[*]
Linköping University[†] *NXP Semiconductors*[‡] *Eindhoven University of Technology*[*]
Sweden *Netherlands* *Netherlands*

Abstract

Post-silicon validation and debug, or ensuring that software executes correctly on the silicon of a multi-processor system-on-chip (MPSOC) is complicated, as it involves checking global properties that are distributed on the chip. In this paper we define an architecture to non-intrusively observe global properties at run time using distributed monitors. The architecture enables to perform actions when a property holds, such as stopping (part of) the system for inspection. We apply this architecture to the problem of software races that result in incorrect communication between concurrent tasks on different processors. In a case study, where we implemented monitors, event distribution, and instruments to stop communication between intellectual property (IP) blocks, we demonstrate that these races can be detected and classified as timing violations or as FIFO protocol violations.

1 Introduction

Multi-processor system-on-chips (MPSOCs) consist of a number of processors combined in a single integrated circuit (IC) with supporting peripherals. It is hard to ensure that MPSOCs meet their specification due to their hardware and software complexity. Post-silicon validation is required to check that the silicon meets its specification and software testing/validation/verification is needed to ensure that the software meets its specification. Pre-silicon verification of software and hardware individually does not imply that the combination of software and hardware, the complete system, meets the specification because execution models may not match, and fault models may not capture all failures. As a result post-silicon debug is often required to find out why the final physical system does not work as expected.

Post-silicon debugging of MPSOCs is challenging because they contain multiple unsynchronised clock domains. Global properties about the system therefore require communication between multiple distributed units in different domains (possibly far apart) with non-neglible communication delays. There are several problems to overcome. First, there is a need to be able to monitor local properties, preferably in a non-intrusive way such that the functional operation is not impacted. Second, results of local distributed monitors must be combined for global properties. Sending information over the functional interconnect may impact performance and/or cost and is therefore not desirable. Adding extensive additional infrastructure for post-silicon debug is costly. Third, there is no common time reference in the system due to the use of multiple clock domains, which complicates the notion of globally consistent view on the system. Fourth, the fact that local properties may become true millions of clock cycles apart from each other requires efficient handling and analysis of large data volumes.

In this paper we focus on software races, which are erroneous system executions, arising from a mismatch between the assumptions made about how (correct) software executes on a (correct) platform. For example, when a processor executes a store instruction to a remote memory, depending on the type of the cache, the write data may not be copied immediately from the local cache to the remote memory. Or, as shown in our case study, when writing data to different distributed memories, subtle incorrect (or unstated) assumptions about communication delays may result in incorrect system behaviour. In particular, we defined

1. possible races conditions,
2. properties to be monitored at distributed units,
3. flexible monitors to capture the properties,
4. a signalling infrastructure between monitors,
5. how to program monitors to capture races, and
6. made a case study that verifies that we can check for the occurrence of races at run-time and classify errors as timing or FIFO protocol related.

This paper is organized as follows. Related work is in Section 2 and a high-level overview is in Section 3. Races are described in Section 4, our distributed debug architecture in Section 5. A case study where we find races at run time is given in Section 6. We conclude with Section 7.

2 Related Work

The fundamental problem during silicon debug is how to observe the state of the system. One straightforward approach is to use the existing scan-chains to capture the state of the system at a given time [1]. While it is cost-effective, as scan-chains are present to enable manufacturing test, it is intrusive, as the system must be stopped. Hence it only allows a single snap-shot of the system to be made. Furthermore it is difficult to resume the system to make additional snapshots. Stopping the clocks for a globally consistent snap-shot is also difficult due to the multiple clock domains [2, 3].

A non-intrusive debug approach is to use trace buffers, which is common in processors of today [4]. The constant increase of complexity enforces larger trace buffers, and techniques to compress data have been developed [5]. With the advent of MPSOCs, there is a need for multiple trace buffers, monitors to trigger on events, and communication between monitors. An MPSOC debug architecture with a separate interconnect for debug is proposed in [6]. In a similar set-up, re-use of the functional interconnect is proposed to send debug data [7], or synchronization tokens [8]. The functional application is impacted by this debug activity, which is not desirable.

While a significant number of works have been proposed on silicon debug, no work details races that cannot be envisioned at the software level, nor demonstrates how to detect these races.

3 High-level Overview

Figure 1 shows two communicating tasks, t_1 and t_2, where producer task t_1 generates elements that are used by consumer task t_2. In the figure, task t_1 is mapped to CPU_1 while task t_2 is mapped on CPU_2.

The communication between tasks is handled using First-In First-Out blocks (FIFOs) mapped on memory. For efficiency, logical FIFOs are commonly implemented as circular buffers, as only pointers and not elements have to be updated when operating on the FIFO. Associated with each FIFO are four pointers: FIFO_{top}, $\text{FIFO}_{\text{bottom}}$, RD_{ptr}, and WR_{ptr}. The FIFO_{top} and $\text{FIFO}_{\text{bottom}}$ pointers define the size of the FIFO and the RD_{ptr} and WR_{ptr} pointers define where to read from, respectively, write to in the FIFO at a given point in time.

Before writing to or reading from the FIFO the producer and consumer poll the read and write pointers. To minimise the latency for this, it is common to keep pointers in a memory close to the producer and consumer. Figure 1 shows that the pointers WR_{ptr} and RD_{ptr} are kept in an on-chip memory which is accessible with a low latency, while the (usually much larger) FIFO data is kept in the large, slower off-chip memory.

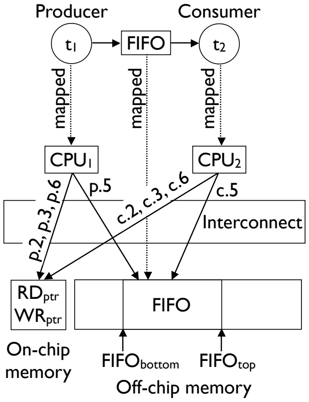

Figure 1: A task graph and an example mapping

The producer constantly enters new tokens into the FIFO by the transactions detailed in Figure 2 while the consumer constantly requests tokens from the FIFO by the transactions detailed in Figure 3. Figure 1 shows that the transactions $p.2$, $p.3$, $p.6$ operate on the on-chip memory while $p.5$ operates on the off-chip memory, and that the transactions $c.2$, $c.3$, $c.6$ operate on the on-chip memory while $c.5$ operates on the off-chip memory. The transactions by the producer and the consumer in Figure 1 over time are shown in Figure 4, using time lines of [9]. The producer, consumer, on-chip memory, and off-chip memory each have a time line indicating when transactions are issued and when they take effect. The example trace shows how both producer and consumer poll the points, followed by the successful transfer of one token.

```
(p.1) while (true)    (c.1) while (true)
(p.2)  read RDptr     (c.2)  read RDptr
(p.3)  read WRptr     (c.3)  read WRptr
(p.4)  if ok_to_write (c.4)  if ok_to_read
(p.5)   write data    (c.5)   read data
(p.6)   write WRptr   (c.6)   write RDptr
```

Figure 2: Producer side **Figure 3:** Consumer side

4 Races and distributed Conditions

Modern high-performance on-chip interconnects, such as multi-layer busses and networks on chip (NOC), are pipelined and concurrent, to serve many transactions at the same time. As a result, there is no single sequential system trace, as was the case for older sequential interconnects. Distributed memories, effects in the NOC such as different path lengths, congestion, differential Quality-of-Service guarantees, as well as slave arbitration and different slave speeds of execution, make it often hard to predict when transac-

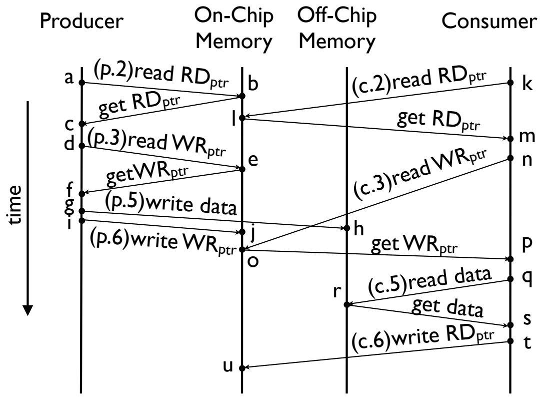

Figure 4: Time diagram for producer and consumer

tions are delivered and executed. As a result, read and write transactions issued in given order by a processor may execute in a different order at different slaves. Next, we illustrate how communication races may occur, using a NOC [10].

Figure 5 shows an execution trace of the transactions for the example of Figure 1 that although *issued in a valid order* may lead to an *incorrect execution*. The problem is that the update of $\mathrm{WR}_{\mathrm{ptr}}$ (p.6) issued at i is quickly transported to and written in the on-chip memory at j. Hence, it can overtake the slower *write data* (p.5), which is issued at g and written in the off-chip memory at h. The consumer reads the updated pointer $\mathrm{WR}_{\mathrm{ptr}}$ (c.3) at o, but the old data (*read data* c.5) at r. To detect this race, it is required to monitor properties at the on-chip memory, and off-chip memory, and then correlate these distributed local properties to detect the race, which is a global property.

A race similar to $\mathrm{WR}_{\mathrm{ptr}}$ can occur for $\mathrm{RD}_{\mathrm{ptr}}$. FIFOs with more than one token have a pipelined behaviour involving multiple tokens and concurrent producer and consumer transactions. For clarity, in the remainder, we focus on only one producer operation (sequence of transactions as in Figure 2) and one consumer operation (sequence of transactions as in Figure 3), although our approach is also valid for the more complex, pipelined cases.

5 Distributed Debug Architecture

We describe our proposed monitor and how it is integrated in the existing on-chip debug architecture, described in [2].

5.1 Monitor

The monitor, shown in Figure 6, consists of a *bus reader*, three *data matchers (DMs)*, and a *state machine (SM)*. The monitor is non-intrusive, and it only observes the bus to which it is attached, and events from other monitors that arrive over the Event Distribution Interconnection (EDI). The EDI is a separate dedicated debug interconnect, which is fast and low-cost [2]. Likewise, the outputs from the monitor are sent to other monitors using the EDI.

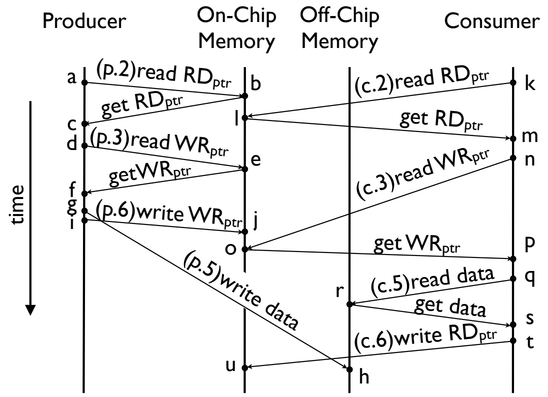

Figure 5: Time diagram of a race

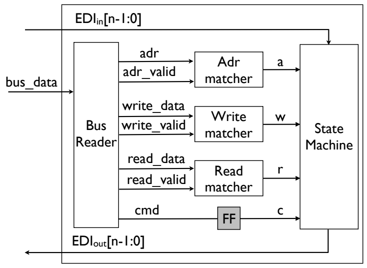

Figure 6: Overview of the monitor

The bus reader is specific to the bus protocol, and forms the interface between the bus and the monitor. It takes inputs from the bus and extracts address (*adr*), data (*write_data* and *read_data*), along with a valid signal for each (*adr_valid*, *write_valid*, *read_valid*), and command information (*cmd*).

The *cmd* from the bus reader is clocked and turned into c indicating read or write. Outputs from the bus reader is fed to three programmable data matchers, which produces three outputs, a, w, r, respectively. Each of these three data matchers consists of two symmetric parts, left and right (Figure 7). The low (high) register can be initialized to a pre-defined value or set during execution to the input data. The low and high registers can independently be updated to an input value or to an incremented value. The low and high register can be masked such that a set of bits are ignored. The masked outcome is compared against the input (data), which also can be masked. A data matcher can check:

1. if (part of) its input is (not) equal, less or greater than a given value or the previous input, or
2. if (part of) its input is in a static or moving range [min, max].

The outputs of the three programmable data matchers (a, w, r) and the read or write command (c) are feed to the state machine.

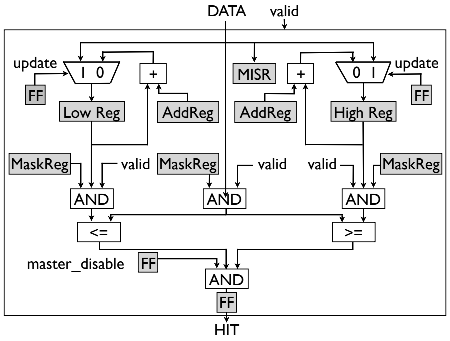

Figure 7: One of three data matchers

The state machine (Figure 8) is RAM-based to allow full programability. The RAM input of $4 + n + q$ bits is a concatenation of the output of the data matchers plus command (4 bits - (a, w, r, c)), the EDI output (n bits), and the state output of the state machine (q bits). The size of the RAM is $2^{(4+m+q)}$ words of $n + q$ bits each.

5.2 Interconnect

Our generic MPSOC, including the NOC, is shown in more detail in Figure 9. Each IP block, processor, memory, etc., is connected to a local bus with its arbiter (A) to a protocol shell (S) that translates a specific bus protocol to a stream of data words. These are then transported by the NOC from network interface (NI) to NI, using intermediate routers [10].

Figure 9 also shows the EDI [2] (shaded), which is routed parallel to, but independent from, the functional router network. Since the EDI broadcasts events, it is simpler, faster, and cheaper than the NOC. Extending our previous EDI implementations, it contains multiple planes, to allow for multiple events and identification of the event's originator. The EDI delivers events to monitors, protocol-specific instruments (PSI) [11], or IP blocks, who ignore or use them, e.g. to stop communication or computation.

6 Case Study

In this section, we detail the properties that need to be monitored for the detection of the WR_ptr race, the subsequent monitor implementation, and simulation results.

6.1 Distributed Conditions to Monitor

While the commands at the producer and the consumer all are issued in correct order, unexpected latencies in interconnects may result in the race detailed in Figure 5. For a generic MPSOC with debug facilities as in Figure 9, the monitors to detect the race must be located near the memories. Figure 10 shows the timing diagram including the monitors M_2 and M_3.

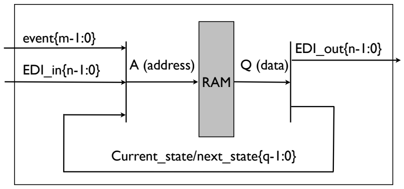

Figure 8: State machine

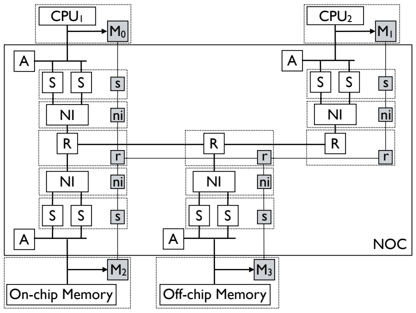

Figure 9: A NOC with monitors and EDI for debug

The race can be detected by, first, checking if the consumer has read an updated pointer in the on-chip memory and, second, when the condition is true, checking if the consumer reads in the wrong part of the FIFO.

We denote the first condition, producer updates write pointer (WR_ptr, p.6) followed by the consumer reads the updated pointer (read WR_ptr, c.3), by $j < o$, where $<$ denotes *occurs before*. When the property is true, monitor M_2 informs all other monitors by sending an event on its EDI_2.

When the first property ($j < o$ detected by monitor M_2) is true, the second property to check is if the consumer reads data (c.5) before the producer's data has arrived (p.5), i.e. monitor M_3 checks $r < h$. Checking updates or accesses of the read (or write) pointer only requires monitoring a single memory address. However, accessing FIFO data requires checking access to data in a range of addresses. If the producer and consumer software software operates correctly, i.e. access data at the right addresses, it is sufficient to check that the consumer reads and the producer writes within the FIFO's address range (defined by FIFO_top and $\text{FIFO}_\text{bottom}$). The first condition ($j < o$) indicates that there is a potential risk of a race. If the consumer then reads in the FIFO before the producer data arrives in the FIFO ($r < h$), then the race has occured. As a result, read and write accesses need only be checked in a fixed range (or even a single location). In this

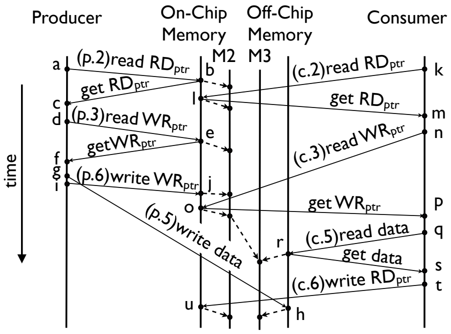

Figure 10: Inserted monitors

paper we only describe the detection of races for a single outstanding token (or producer/consumer synchronization). However, our approach is more general and allows pipelined monitor events, at the cost of a more advanced implementation.

6.2 Monitor Implementation

In this section we show how to program the monitors, to observe the race.

6.2.1 Monitor M_2

Monitor M_2 is to detect if $j < o$ and then signal this to monitor M_3 on EDI e_2. To detect condition j (p.6) and o (c.3), the data matcher must create an event a (in Figure 6) for transactions with address WR$_{\text{ptr}}$. Event a is generated by pre-loading the address of WR$_{\text{ptr}}$ in both high and low registers of the monitor (see Figure 7). If the input is true on both the left and the right hand side, the input is the pointer address. Read and write transactions generate events c and c' directly.

Given events a, c and c', the state machine is defined as in Figure 11 and Table 1. The state machine is in the initial state 0 until j, that is a & c' (p.6) is valid, and then the state machine moves to state 1. If in state 1 and a & c (c.3) becomes valid there is a potential race as the pointer is updated and the updated pointer has been read; hence, e_2 output is generated and the state machine returns to state 0.

Table 1 shows the implementation of the state machine. As a RAM is used, all states are to be defined (all possible combinations of don't care states (-)). We assume one EDI signal for each monitor, which for our example in Figure 9 gives four EDI_{in} signals. Monitor M_2 does not react to any other monitor; hence, all EDI$_{\text{in}}$ signals are ignored (marked as don't care (-)).

6.2.2 Monitor M_3

Monitor M_3 is to detect activity on e$_2$, which indicates that the property to be checked by monitor M_2 has triggered. If M_2 is activated, monitor M_3 is to detect $r < h$, and then signal on e_3 if a timing error occurs or on e_0 if a FIFO protocol error occurs, such that an event receiver, for example a PSI, can be notified. As discussed above, detecting where in the FIFO access is performed is not necessary as given a possible pointer violation (detected by monitor M_2), it is necessary to check that the consumer reads prior to the producer writes in the FIFO data area. To detect condition r and h, an a hit is needed on transactions in the FIFO. The event a is generated by pre-loading the FIFO$_{\text{top}}$ in the high and the FIFO$_{\text{bottom}}$ in the low registers of the monitor (see Figure 7).

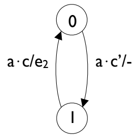

Figure 11: State diagram for monitor M_2 where a=address hit, c=read and c'=write

EDI_in	Current_state	event	EDI_out	next_state
0 1 2 3		a w r c	0 1 2 3	
- - - -	0	0 - - -	0 0 0 0	0
- - - -	0	- - - 1	0 0 0 0	0
- - - -	0	1 - - 0	0 0 0 0	1
- - - -	1	- - - 0	0 0 0 0	1
- - - -	1	0 - - -	0 0 0 0	1
- - - -	1	1 - - 1	0 0 1 0	0

Table 1: RAM content for monitor M_2

For the implementation of monitor M_3, we assume a token size of 8 words (8 individual reads/writes). We use six state bits (variable $state$) where bits 3 down to 0 keep track of the difference in number of writes ($\#w$) and reads ($\#r$), bit 4 indicates if a token has been completely written to the memory, and bit 5 defines if the FIFO protocol is followed. A signal on e_3 indicates a timing error due to the interconnect and a signal on e_0 indicates a FIFO protocol violation.

The state is included in the RAM, hence, the following if-statements are executed concurrently:

```
if $w(write)$ then
   increment $state$[3-0] // $\#w - \#r$
   if $state$[3-0] = 8 ($\#w - \#r$) then
      set $state$[4]=1 // token complete
   end if
end if
if $e_2$ then
   set $state$[5]=1 //FIFO protocol used
end if
if $r(read)$ then
   if $state$[4]=0 //token incomplete then
      if $state$[5]=1 //FIFO protocol used then
         output $e_3$; stop;
      else
         output $e_0$; stop;
      end if
      decrement $state$[3-0] // #w-#r
      if $\#w - \#r = 0$ then
         $state$[4]=0 //reset - token incomplete
         $state$[5]=0 //reset - FIFO protocol not used
      end if
   end if
end if
```

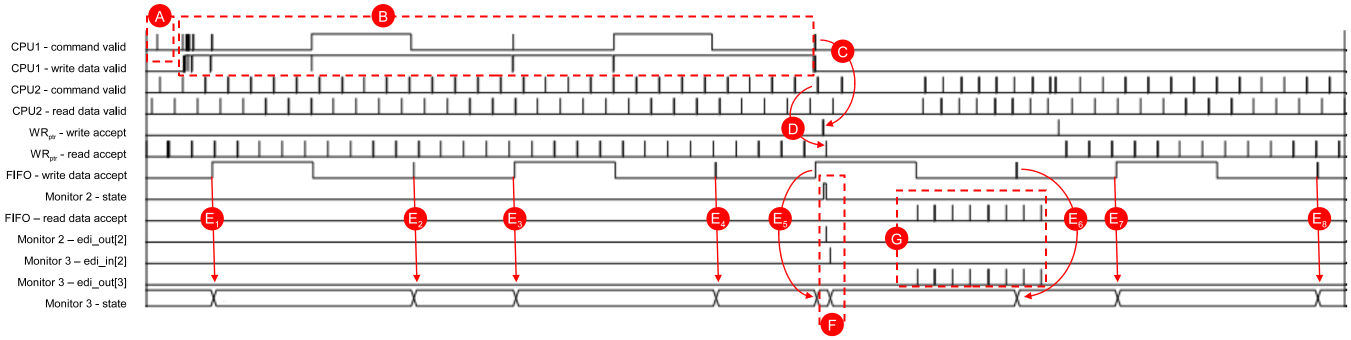

Figure 12: Simulation results of run-time race checking.

6.3 Experimental Results

We have implemented the MPSOC in Figure 1 and Figure 9. We simulate the producer-consumer interaction using a shared memory FIFO, see Figure 12. CPU_1 queries the value of the RD_{ptr} (A) to check whether there is room in the FIFO. It subsequently writes eight 32-bit data words as one token into the FIFO (B), and updates the write pointer (C). In parallel, CPU_2 continuously queries this write pointer to see if there is data in the FIFO. When it subsequently reads the write pointer after the producer has updated it (D), Monitor 2 changes state, and informs Monitor 3 via EDI 2 (F). All the while, Monitor 3 is keeping track of the data words writing by the producer in the FIFO (E_1, E_2, $\ldots$, E_8). When a read request from the consumer comes in before the final data word of the token is written by the producer into the FIFO memory (E_8), this is signalled as an race violation on EDI 3 (G). In our case, all eight data words in the token are read before the final data word is written. Of those eight, the last three data words are actually read before they are written, and therefore contain old, incorrect data.

7 Conclusions

Software running on multi-processor system-on-chips with an advanced interconnect, such as a network-on-chip, may cause races that are difficult to detect. We have in this paper detailed races, shown that distributed monitoring can detect the races, defined properties needed to detect a particular race, proposed a flexible and programmable monitor to check for properties and shown how to include and use the monitors in an existing debug interconnect. We have made a case study showing that a race can be detected and eventual errors can be classified as timing errors or FIFO protocol violations by programming distributed monitors.

References

[1] K. Holdbrook *et al.*, “Microsparc: a case-study of scan based debug,” in *International Test Conference*, 1994, pp. 70–75.

[2] B. Vermeulen *et al.*, “Debugging distributed-shared-memory communication at multiple granularities in networks on chip,” in *International Symposium on Networks-on-Chip*, 2008, pp. 3–12.

[3] B. Vermeulen and K. Goossens, “Debugging multi-core systems on chip,” in *Multi-Core Embedded Systems*, G. Kornaros, Ed. CRC Press/Taylor & Francis Group, Sep. 2010, ch. 5, pp. 153–198.

[4] ARM, “Embedded trace buffer,” ARM Ltd., Tech. Rep., http://www.arm.com/.

[5] E. Daoud and N. Nicolici, “Real-time lossless compression for silicon debug,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 9, pp. 1387–1400, Sept. 2009.

[6] R. Leatherman and N. Stollon, “An embedding debugging architecture for SOCs,” *Potentials, IEEE*, vol. 24, no. 1, pp. 12–16, Feb.-March 2005.

[7] S. Tang and Q. Xu, “In-band cross-trigger event transmission for transaction-based debug,” in *Design, automation and test in Europe*, 2008, pp. 414–419.

[8] C.-N. Wen *et al.*, “Nuda: a non-uniform debugging architecture and non-intrusive race detection for many-core,” in *Design Automation Conference*, 2009, pp. 148–153.

[9] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[10] A. Hansson and K. Goossens, “An on-chip interconnect and protocol stack for multiple communication paradigms and programming models,” in *Int’l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct. 2009.

[11] K. Goossens, B. Vermeulen, and A. Beyranvand Nejad, “A high-level debug environment for communication-centric debug,” in *Proceedings Design, Automation, and Test in Europe (DATE)*, Apr. 2009.