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

Erik Larsson, Bart Vermeulen and Kees Goossens

N.B.: When citing this work, cite the original article.
A Distributed Architecture to Check Global Properties for Post-silicon Debug

Erik Larsson  
Linköpings universitet  
Sweden  
Email: erila@ida.liu.se

Bart Vermeulen,  
NXP Semiconductors  
Netherlands  
Email: bart.vermeulen@nxp.com

Kees Goossens,  
Eindhoven University of Technology  
Netherlands  
Email: k.g.w.goossens@tue.nl

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.

Index Terms—Post silicon debug, validation, races, monitors, distributed property checking

I. 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-negligible 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 II and a high-level overview is in Section III. Races are described in Section IV, our distributed debug architecture in Section V. A case study where we find races at run time is given in Section VI. We conclude with Section VII.

II. RELATED WORK

The fundamental problem during silicon debug is how to observe the state of the system. One straight-forward 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 snap-shots. 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 setup, 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.

III. 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\(_{\text{top}}\), FIFO\(_{\text{bottom}}\), RD\(_{\text{ptr}}\), and WR\(_{\text{ptr}}\). The FIFO\(_{\text{top}}\) and FIFO\(_{\text{bottom}}\) pointers define the size of the FIFO and the RD\(_{\text{ptr}}\) and WR\(_{\text{ptr}}\) 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\(_{\text{ptr}}\) and RD\(_{\text{ptr}}\) 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.

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.

IV. RACES AND DISTRIBUTED CONDITIONS

Modern high-performance on-chip interconnects, such as multi-layer buses 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 transactions 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 WR_p (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 WR_p (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 WR_p can occur for RD_p. 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.

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.

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 $2^{4+n+q}$ words of $n + q$ bits each.

B. 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].
Fig. 7. One of three data matchers

Fig. 8. State machine

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.

Fig. 9. A NOC with monitors and EDI for debug

VI. CASE STUDY

In this section, we detail the properties that need to be monitored for the detection of the WR\textsubscript{ptr} race, the subsequent monitor implementation, and simulation results.

A. 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 \(M_3\).

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\textsubscript{ptr}, \(p.6\)) followed by the consumer reads the updated pointer (read WR\textsubscript{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\textsubscript{2}.

When the first property \((j \preceq o\) 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\textsubscript{top} and FIFO\textsubscript{bottom}). The first condition \((j \preceq o\) 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 \((r \preceq h)\), then the race has occurred. As a result, read and write accesses need only be checked in
a fixed range (or even a single location). In this 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.

B. Monitor Implementation

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

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 $WR_{ptr}$. Event $a$ is generated by pre-loading the address of $WR_{ptr}$ 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 I. 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 I 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 $EDI_{in}$ signals are ignored (marked as don’t care (-)).

![State diagram for monitor $M_2$](image)

Table I: RAM content for monitor $M_2$

<table>
<thead>
<tr>
<th>$EDI_{in}$</th>
<th>Current_state</th>
<th>event</th>
<th>$EDI_{out}$</th>
<th>next_state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>a w r c</td>
<td>0 1 2 3</td>
<td>a w r c</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0 0 0 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0 - - -</td>
<td>0 0 0 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1 - - 0</td>
<td>0 0 0 0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>- - - 0</td>
<td>0 0 0 0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0 - - -</td>
<td>0 0 0 0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1 - - -</td>
<td>0 0 1 0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

C. 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$_0$ 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, CPU₂ 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 (E₁, E₂, ..., E₈). 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 (E₈), this is signalled as a 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.

VII. 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