Checking pipelined distributed global properties for post-silicon debug

Citation for published version (APA):

Document status and date:
Published: 01/01/2010

Publisher Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Checking Pipelined Distributed 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—
While multi-processor system-on-chips (MPSOCs) with network-on-chip (NOC) interconnect are becoming increasingly common to meet the constant performance demand, it is due to communication delays in the NOC extremely complicated to ensure that software executes correctly. In this paper, we extend our architecture that non-intrusively observes global properties at run time using distributed monitors such that not only single tokens but also pipelined tokens can be monitored. We detail the solution for a given race and compare the alternatives of having one large monitor versus multiple small monitors.

Index Terms—Validation, races, monitors, distributed property checking

I. INTRODUCTION

The increasing demand for performance is addressed using multi-processor system-on-chips (MPSOCs), which consist of a number of processors and supporting peripherals, with network-on-chip (NOC) interconnects, combined in a single integrated circuit (IC). While NOC-based MPSOCs meet the performance demand, it is hard to ensure that an MPSOC meets its specification due to its hardware and software complexity.

Pre-silicon verification of software and hardware does not imply that the complete (final) system meets its 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 debug of an MPSOC is a challenge because an MPSOC typically contains a number of unsynchronized 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 functional performance and/or end product cost and is therefore not desirable. Adding extensive additional infrastructure for post-silicon debug is also costly. Third, there is no common time reference in the system due to the use of multiple clock domains, which complicates creating a globally consistent view on the system [1]. 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.

We have previously developed an architecture to monitor for communication delays in NOC-based MPSOCs [2]. In particular, we showed the ability to check for races where one token consisting of a number of words is produced and consumed correctly. However, in general, multiple tokens can be pipelined. A producer can generate an arbitrary number of tokens and it can be very difficult to detect which specific word(s) cause(s) the problem. The fundamental problem is to distinguish and associate reads and writes in the memory with tokens. Hence, there is a need to check pipelined tokens.

Based on our work [2], we detail the two alternatives: 1 N-token monitor and N 1-token monitors. We made a case study where we detail the overhead of each approach.

The 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

Observing the state of the system is very difficult due to the limited access to the state of the internal nodes (e.g. flip-flops). A straight-forward approach to observe the system’s state is to reuse the scan-chains, which are present to enable manufacturing test. Scan-chains, flip-flops with additional multiplexers, are commonly used to apply manufacturing tests. The scan-chains allow to capture the state of the system at a given time [3]. While it is cost-effective to reuse the scan-chains, as they are already present for manufacturing test, their use is intrusive, as the system must be stopped while the content of the flip-flops is serially shifted out. It also only allows a single snapshot to be taken of the system at a time after the MPSOC has been stopped. Stopping the clocks for a globally consistent snapshot is difficult due to the multiple clock domains [4], [5]. After taking the snapshot, the system execution has to be resumed or restarted. Resuming the execution of the system to make additional snapshots is also difficult, as the precise clock relations among the internal clock domains, and the synchronization with the external environment at the moment of stopping may not be restorable.
A non-intrusive debug approach is to make use of trace buffers, which is common in today’s processors [6]. However, the constant increase of complexity and circuit speeds enforce larger trace buffers, and techniques to compress data have been developed [7]. In MOPSOCs, there is even a need for multiple trace buffers, monitors to trigger on events, and communication between monitors. An MOPSOC debug architecture with a separate interconnect for debug is proposed in [8]. In a similar set-up, re-use of the functional interconnect is proposed to send debug data [9], or synchronization tokens [10]. 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 but our previous work [2] details races that cannot be envisioned at the software level, or demonstrates how to detect these races. Different from our previous work [2] where a single token of a number of words is considered, we address in this paper multiple pipelined tokens.

### III. HIGH-LEVEL OVERVIEW

Figure 1 shows a task graph that consists of nodes and directed edges where a node represents a computation task and an edge between nodes represents communication between tasks. Figure 2 shows the task graph in Figure 1 mapped on an example MOPSOC. Task \( T_0 \) receives and distributes inputs to task \( T_1 \), executing on \( CPU_0 \), and task \( T_3 \), executing on \( CPU_0 \). Tasks \( T_2 \) is executed on \( CPU_0 \) together with Task \( T_1 \), while \( T_4 \) is executed on \( CPU_2 \) together with Task \( T_3 \). Given the results of task \( T_2 \) and \( T_4 \), task \( T_5 \) produces the outputs of the MOPSOC.

The communication between tasks, represented by edges between the tasks, can be implemented by First-In First-Out (FIFO) queues. A FIFO queue can be hardware-based, i.e. implemented with dedicated hardware, or software-based, i.e. assigning a part of the shared memory for the FIFO. In this paper, we assume that the FIFO queues are implemented as parts of the shared memory between \( CPU_0 \) and \( CPU_1 \).

A software-based FIFO is often implemented as a circular buffer. The advantage with this implementation is that only pointers and not elements have to be updated when operating on the FIFO. Figure 3 details a circular FIFO on which two tasks (a producer and a consumer) operate and the four associated pointers, \( FIFO_{top} \), \( FIFO_{bottom} \), \( RD_{ptr} \), and \( WR_{ptr} \). The pointers \( FIFO_{top} \) and \( FIFO_{bottom} \) define the size of the FIFO and the pointer \( RD_{ptr} \) defines where to read from the FIFO at a given point in time and \( WR_{ptr} \) defines where to write to the FIFO at a given point in time. The valid data of the FIFO is always between \( RD_{ptr} \) and \( WR_{ptr} \); however, as the pointers are constantly updated based on FIFO reads and writes, the valid area changes. The pointers \( RD_{ptr} \) and \( WR_{ptr} \) always have to be between \( FIFO_{top} \) and \( FIFO_{bottom} \). However, even then two alternatives exist for the valid area; alternative one - the case where \( RD_{ptr} \) is larger than \( WR_{ptr} \), and alternative two - the case where \( WR_{ptr} \) has passed \( FIFO_{top} \) and restarted at \( FIFO_{bottom} \), and is smaller than \( RD_{ptr} \). As soon as \( RD_{ptr} \) also passes \( FIFO_{top} \) and restarts at \( FIFO_{bottom} \), alternative one is valid again.

In order to minimize the memory latency and traffic in the NOC, it is common practice that pointers are kept locally at the producer and consumer. The \( FIFO_{top} \) and \( FIFO_{bottom} \) pointers are static and are not changed during the application and therefore copies can be kept locally at the consumer and the producer to ensure pointer consistency. However, the \( RD_{ptr} \) and \( WR_{ptr} \) are constantly updated during application. The general scheme is that the producer keeps \( WR_{ptr} \) and a copy or \( RD_{ptr} \) (named \( RD'_{ptr} \)) while the consumer keeps \( RD_{ptr} \) and a copy of \( WR_{ptr} \) (named \( WR'_{ptr} \)).

Figure 4 shows two communicating tasks, \( T_1 \) and \( T_2 \), where the producer task \( T_1 \) generates elements that are used by the consumer task \( T_2 \). In Figure 4, Task \( T_1 \) is mapped to \( CPU_1 \) while Task \( T_2 \) is mapped on \( CPU_2 \).

Before writing to or reading from the FIFO the producer and consumer poll the read and write pointers. Figure 4 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.

The producer repeatedly enters new tokens into the FIFO by the transactions, as detailed in Figure 5, while the consumer repeatedly requests tokens from the FIFO by the transactions, as detailed in Figure 6. Figure 4 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 4 over time are shown in Figure 7, using time lines of [11]. 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 and check the pointers, followed by the successful transfer of one token.

\[(p.1)\text{while (true)} \quad (c.1)\text{while (true)}\]
\[(p.2)\text{read RDptr} \quad (c.2)\text{read RDptr}\]
\[(p.3)\text{read WRptr} \quad (c.3)\text{read WRptr}\]
\[(p.4)\text{if ok_to_write} \quad (c.4)\text{if ok_to_read}\]
\[(p.5)\text{write data} \quad (c.5)\text{read data}\]
\[(p.6)\text{write WRptr} \quad (c.6)\text{write RDptr}\]

Fig. 5. Producer side  
Fig. 6. Consumer side

IV. 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 transactions are delivered and executed. As a result, read and write transactions issued in a 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 [12].

Figure 8 shows an execution trace of the transactions for the example of Figure 4 that although issued in a valid order may lead to an incorrect execution. The problem is that the update of WRptr (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 WRptr (c.3) at o, but subsequently still reads old data (read data c.5) at r. To detect this race, which is a global property, 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. A race similar to WRptr can occur for RDptr.

Figure 9 details reads and writes in terms of tokens where a token consists of a number of words. In our previous work (\cite{2}) we solved the problem when writing and reading single tokens at a time. However, pipelined tokens remained unsolved. Before addressing how to detect possible races for pipelined tokens, we shortly revisit the supportive debug architecture.

V. DISTRIBUTED DEBUG ARCHITECTURE

Figure 10 shows a high-level overview of an MPSOC, including the NOC and IP blocks like processors, memory, etc., that are connected to a local bus using its arbiter (A) and to a network protocol shell (S). This shell translates a specific bus protocol to a stream of data words. These data words are then transported by the NOC from network interface (NI) to NI, using intermediate routers [12].

Figure 10 also shows the Event Distribution Interconnection (EDI) \cite{4} (shaded), which is routed parallel to, but is 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) [13], or IP blocks, who can be programmed to either use, e.g. to stop communication and/or computation, or ignore them.

The monitor is non-intrusive as it only observes the bus to which it is attached and the events from other monitors that arrive over the EDI. Likewise, the outputs from the monitor are sent to other monitors using the EDI. A key advantage of the EDI is that all communication (and delays) are deterministic and fixed in time.

The monitor, shown in Figure 11, consists of a bus reader, three data matchers (DMs), and a state machine (SM). The bus reader is specific to the bus protocol, and forms the interface between the bus and the monitor. The bus reader takes the inputs from the bus and extracts address (adr), data (write_data and read_data), along with valid signals for each (adr_valid, write_valid, read_valid), and command information (cmd). The cmd from the bus reader is pipelined and turned into a signal c indicating a read or write operation. 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 (refer to Figure 12). 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 signal (c) are fed to a programmable state machine.

The state machine (Figure 13) is RAM-based to allow full programability. The RAM input (A - address) is given by events, EDI_n and the current state (CS)/ next state (NS). The RAM output (D - data) consists of EDI_out and the current state (CS)/ next state (NS).
VI. MONITORING PIPELINED TOKENS

We compare below the alternative of having one $N$-token monitor, capable of handling $N$ tokens simultaneously, against $N$ 1-token monitors, each capable of handling one token, but communication among each other via the EDI to also be able to handle $N$ tokens simultaneously. We assume below a token size of 8 words. However, the approach applies to any token size.

Each monitor consists of three data matchers and a state machine implemented in RAM. The $N$-token monitor requires a larger state machine (RAM) to capture all states but reuse its data matchers and less additional EDI signaling between monitors. We compare RAM sizes, number of EDI signals, and number of data matchers.

- $N$ 1-token monitors

In the case of $N$ 1-token monitors, naturally $N$ monitors are needed. The total size of the RAM ($R$) in bits (refer to Figure 13) depends on the number of address bits ($a$) and the data word size ($d$), and is given by $R = d \cdot 2^a$. The number of state bits ($s$) in a 1-token monitor is 6 (as derived in [2]). In addition, each of the $N$ 1-token monitors uses a single EDI input to receive events from monitor $M_2$, located at the on-chip control memory, three event inputs from the local data matchers (refer to Figure 11), an additional input to distinguish a read operation from a write operation, and three EDI inputs (and outputs) for inter monitor signaling. This brings the total number of address bits $a$ to $s + 1 + 3 + 1 + 3 = 6 + 8 = 14$. On the RAM output side, two error signals and three inter monitor EDI signals result in $d = s + 2 + 3 = 6 + 5 = 11$. The total memory size is therefore given by $R_N = N \cdot 11 \cdot 2^{14}$ for the
use case with $N$ 1-token monitors. For the corresponding EDI architecture, there is one EDI layer to receive events from monitor $M_2$, and per monitor two layers for error signaling and three layers for inter monitor signaling, hence the total number of EDI layers equals $5 \cdot N + 1$. In addition, $N$ 1-token monitors require three data matchers in each monitor, so $3 \cdot N$ in total.

- **$1$ $N$-token monitor**

A $N$-token monitor capable of handling $N$ pipelined tokens needs 8 states to record the writing of the eight words of the first token. For subsequent pipelined tokens, reads of the previous token(s) and writes of the current token can arrive interchanged, hence, each pipeline stage plus the final stage connecting with the initial state requires 8-8 states. The total number of states is therefore given by $8 + 64 \cdot N$ for $N \geq 2$. The monitor has to be able to record up to $N$ EDI events from monitor $M_2$ for $N$ tokens (including zero). Hence, $\log_2[N + 1]$ bits are needed for EDI administration. The total number of state bits is given by: $s = \lceil \log_2(N + 1) \cdot (8 + 64 \cdot N) \rceil$. The monitor needs one EDI input to receive events from monitor $M_2$, two outputs for error signaling and two inputs and outputs for inter monitor signaling, which leads to $d = s + 5)$. The number of inputs $a = s + 2 \cdot N$ where $2 \cdot N$ is needed to distinguish a protocol violation from a network delay problem for each token. The total number of memory bits is: $(s+2\cdot N)2^{(s+5)}$. For the EDI architecture, there is one EDI from monitor $M_2$ and two per monitor for error signaling, hence: $1 + 2 \cdot N$. Using a single monitor leads to three required data matchers.

In Figure 14, the resulting silicon area estimates of one $N$-token monitor and $N$ 1-token monitors are reported for $N$ ranging from 1 to 8. For $1 < N < 4$ the $N$-token monitor is more cost effective; however, at higher $N$, the $N$ 1-token monitors become more effective mainly because the higher number of required states increases the RAM rapidly for the $N$-token monitor.

Overall, we prefer the implementation of $N$ individual monitors over one single monitor, because the individual monitors will be more adaptable to other debug use cases as well.

**VII. Conclusions**

Software running on multi-processor system-on-chips with an advanced interconnect, such as a network-on-chip, may suffer from races that are difficult to detect. In this paper we extended our architecture to non-intrusively monitor for races, such that not only single tokens of a given number of words, but also pipelined token violations can be detected. The violations can be classified as timing errors or FIFO protocol violations. We have compared two implementation alternatives; 1 N-token monitor and $N$ 1-token monitors.

**REFERENCES**