An Automated Infrastructure for Real-Time Monitoring of Multi-Core Systems-on-Chip

George Kornaros*
Electronics & Computer Engineering Department
Technical University of Crete
Kounoupidiana, Chania, Greece
Email: kornaros@mhl.tuc.gr

Ioannis Christoforakis, Maria Astrinaki
Applied Informatics & Multimedia Department
Technological Educational Institute
Estavromenos, Heraklion, Greece

Abstract—**Requirements for rapid turnaround development of complex multi-core Systems-on-Chip nowadays have advanced to the level at which a number of different in principle validation techniques have to be performed in short time. Quite common are hybrids of passive debugging of Systems-on-Chip and event-driven active verification. On top of these, we present a novel highly flexible verification infrastructure, in which parameters of monitoring can be accessible in real-time while the measurement itself is being performed. Instead of simply observing components under development, the proposed infrastructure enables the designer to interact, monitor and adjust in real-time system parameters or application software. This paper explores different microarchitecture alternatives to efficiently support flexible real-time monitoring via hardware configurable monitors which can provide abstractions of the information. A quantitative evaluation of the proposed methodology on a system-on-FPGA provides results that can serve as guidelines for system-level designers, proving the need for flexible and at the same time efficient filters for real-time monitors inside complex multi-core SoCs.**

I. Introduction

In recognition of the complexity of *debugging* (in this article we use the term *debugging* to refer to both debugging for correctness as well as to discovering the critical paths for performance tuning and power optimization) multi-core systems-on-chip in conjunction with parallel applications, run-time mechanisms are becoming increasingly important to identify multiprocessing problems and evaluate partitioning and mapping of tasks to cores or their run-time behavior with increased level of confidence and accuracy. Therefore, it is important to develop tools and methodologies which provide advanced multi-core development and debugging capabilities while supporting both heterogeneous operating systems and/or heterogeneous processor architectures. Efforts are in the direction of system-aware profiling, run-time debugging tools, while it is widely known that it is extremely hard to find multiprocessing problems like race-conditions, time-consuming tasks, deadlocks or live-locks.

The paper addresses the problem of reducing the time spent on prototyping an SoC design, to improve time-to-market. In the prototyping process, the SoC design is implemented on an FPGA and trace programs are run on the SoC while measuring signals that are internal to the designed systems, because these signals need to be observed to validate the operation of hardware and software components. The purpose of this phase of prototyping is to find implementation errors and performance bottlenecks. To observe the system-internal signals, the proposed approach includes non-intrusive monitoring circuitry in the HDL of the SoC design. The monitoring circuitry can be automatically generated based on a prototyping-engineers specification to observe the behavior pre-determined signals. The monitoring hardware can contain comparison units, counters, timers, and memory, implemented either as dedicated hardware structures or using software running on monitoring-specific CPUs. Particular focus is on filtering the large data volume from FPGA-based prototyping experiments, to monitor only certain signals in conjunction to given events, analogous to setting breakpoints in a software debug tool. The proposed monitoring circuitry contain such filters and these filters can be reconfigured to match new events during the operation of the FPGA experiment.

The proposed infrastructure is conceived as an amalgamation of ideas from hardware verification methodologies for SoCs, hardware emulators and monitoring and debugging for distributed, real-time or parallel systems. Increasingly reduced time-to-market requirements force designers to devise automated ways to verify and isolate a root cause of the failures, or of the time-critical path inside complex multi-core SoCs. Key enabling concepts in electronic system level (ESL) verification methodologies are using transaction-based communication and synchronization combined with the controlled time concept [1], or hardware-abstraction layers [2], to deliver high verification throughput.

This work presents *an automated methodology* using advanced script-based tools, yet easy and fast to deploy; these tools feature a ***non-invasive nature***, with the capacity to capture the timing, performance or energy deficiencies inside complex SoCs when the real-time failing conditions do actually occur. While this hybrid infrastructure utilizes both *hardware monitors* and *software run-time management* for increased flexibility, it is straightforward and easy to employ. The hardware monitors need to be attached to the suspicious signals to verify, while the monitors' software interface offers a simple API to the developer, and does not incur any other programmer visible effects. Moreover, the proposed scheme can be incorporated

Also with the Technological Educational Institute of Crete, Applied Informatics & Multimedia Department, Heraklion, Greece

978-1-4673-1188-5/12/$31.00 ©2012 IEEE

to assist in monitoring and efficient handling of multiple processes or parallel applications in embedded systems. Debugging and control of parallel embedded applications can be facilitated with a real-time tool capable of capturing both hardware and software deficiencies.

The remainder of this paper is organized as follows. Section II addresses previous work on debugging and environments for monitoring SoCs. In section III the essential monitoring methodology features are described. Section IV analyzes the internal architecture of the monitors and provides an assessment of the capacity of the system when scaling to multi-cores, along with the interaction and the capabilities of the associated monitoring software. Finally, section V concludes the paper.

II. Background and Related Work

In providing observability for bus-based systems research and industry has already demonstrated valuable results, such as ARMs Coresight technology [3]. Usually, the presence of extra logic and special debugging modules is intrusive and sometimes degrades system performance. On top, integrating suitable debug and trace options on the system often requires the specialist knowledge of the tool manufacturer.

In the research domain, Cota et al., describe in [4] one approach to provide a Test Access Mechanism (TAM) for testing the nodes of a SoC; this technique re-uses the network resources to minimize the cost and improve the speed of testing probes. The key observation is that due to its role, network-on-chip is a central piece of the SoC. TAM interfaces with test wrappers, built around the cores, to apply test vectors to the cores under test, and also collects and delivers the possible responses. However, this type of operation is intrusive and useful only for off-line testing.

Different Design-for-Testability (DfT) approaches have also been proposed, to provide the means for testing a network-on-chip [5]. In [6] a debug framework for reconfigurable hardware is presented to address the problem of co-debugging in hybrid systems. A custom tool named JHDL with a new special design language has been developed to emulate the system based on abstractions of the structural view of hardware. However, real-time features of this approach and of the work presented in [7], combined with handling of the complexity of debugging multi-core SoCs are not present.

Hybrid techniques have been proposed which are based on combining a number of packet selection approaches. For instance, Scholler proposes to add packet sampling into the packet and create a scheme combining certain advantages of filtering and sampling [9].

In [10] Ciordas et al. present, in detail, how monitoring services can be taken into account at design time and how designers can integrate the monitoring functionality and placement in a network-on-chip through the system design flow. In addition, the cost of the complete monitoring solution is quantified; this cost includes the monitors, the extra network interfaces (NIs), NI ports or enlarged topology needed to support monitoring in addition to the original communication infrastructure. Results show an area-efficient solution for integrating monitoring in NoC designs. However, the filtering solutions are not detailed.

One of the main problems in NoC monitoring is that processing the entire contents of every packet imposes high demands on packet monitors and their hardware resources. One way for reducing the volume of the data is by utilizing certain techniques for filtering, aggregation, and sampling just as it is done in the case of telecommunication network monitoring [8]. In [11] the debug environment of the AEthereal NoC architecture is presented so as to visualize a SoC's state at the logical communication level. Complementary to this work the following sections introduce a more practical infrastructure which is quantitatively evaluated with experiments, so as to explore its effectiveness for multi-core SoCs.

III. Debugging Framework Based on Real-Time Monitoring

The main objectives of the proposed methodology are briefly summarized in:

- providing an easy, fast and efficient tool to jointly monitor hardware designs and software code,
- integrating monitors with the SoC design in a non-invasive way,
- leaving the software application without any instrumentation code run as it would on the actual embedded system.

The design flow is outlined in Fig. 1. Usually prototyping on a reconfigurable-logic-based platform assumes a SoC design that is re-synthesized and verified many times before an error free system is produced. In this context, but not limited to pre-production validation, the infrastructure shown in Fig. 1 uses a design described in an HDL (and potentially the embedded application code) as input. The designer mainly specifies the *trace* and the *trigger* signals before running the tool. Each of these signals can be either a single bit or a bit vector, or part of this bit vector. These can also be generics, or just the name of the signal without specifying bit width; the tool parses the design to discover it. The signal name can also be part of the actual signal name, which is very convenient when the design that is used as input to the flow has already been synthesized, and thus, the suffix of the signal name may be modified (due to vector naming conventions, or duplicate logic to reduce fanout).

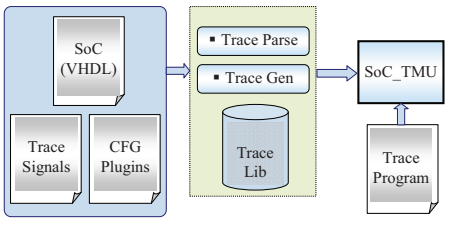

Fig. 1. Automated infrastructure to integrate Trace Monitor Units (TMUs) with a System-on-Chip design.

The debugging framework employs configurable plug-ins that can be specified optionally depending on the designer's requirements and resource constraints. After the inputs are

scanned and processed the configuration plug-ins are considered to produce the final top level synthesizable SoC. The developed plug-ins include:

- *Free-running timer*. Timestamps can be enabled to identify the exact timing of captured events. It is useful to obtain events from different clock domains associated with a local timestamp when insight is needed at a block level. However, currently we opt for a centralized counter/clock to avoid distributed time consensus issues. A tuple {timestamp, event} forms an event-time structure.
- *Statistic counters*. Currently a single 32-bit counter is used to accumulate the captured events and is controlled by the software of the monitor CPU.
- *Compression unit*. While filtering the trace data reduces the amount of information, if the designer desires to increase the window of observation then, the data must be either compressed or use a high-throughput link to transfer them to the real-time on-chip monitor CPU, or to the off-chip monitor host. Even though hardware compression units are efficient, their energy and silicon cost may be prohibitive for comparatively small designs.
- *CAM-based Event Classification*. A Content-Addressable-Memory (CAM) helps to categorize the events in classes specified as rules by the designer. A single filter is a first aid to select the events of interest out of a vast amount of sampled data. Further on-the-fly processing though, necessitates the use of a CAM. Due to the silicon cost of a hardware CAM the testbed evaluation discussed in section IV is using a single hardware filter combined with a CAM in software.
- *External Memory Interface*. Depending on the combination of processing of traced events, of link throughput that is used to transfer these data and of the level of filtering more on-chip memory, or off-chip memory can be selected.

The organization and internal architecture of these components is further analyzed in the following section.

A. Architecture

The minimal configuration of the monitoring unit may include a single interface (either UART or parallel interface) in order to forward the captured traces to a monitoring host. Similar to Chipscope Pro from Xilinx, a on-chip real-time observer tool for FPGAs, the designer defines which signals can be observed; these are captured through a monitor controller to an outgoing interface. Thus, users can define trigger conditions and capture data at or close to the system clock rate.

Unless the traced data can be forwarded to the monitoring host at least with the system's operating frequency, buffering must be enabled. The default option is to use on-chip memory, utilizing multiple memory blocks of size 64×64 if desired. The number of these memory blocks is automatically computed based on the bit width of the specified traced signals, so as to form a wide memory. Fig. 2 outlines an example organization of the monitoring module connected to an on-chip CPU. Alternatively, an off-chip memory can be specified with a potential risk to affect the system's throughput if the memory or the interface cannot sustain the aggregate bandwidth of the SoC and of the monitors.

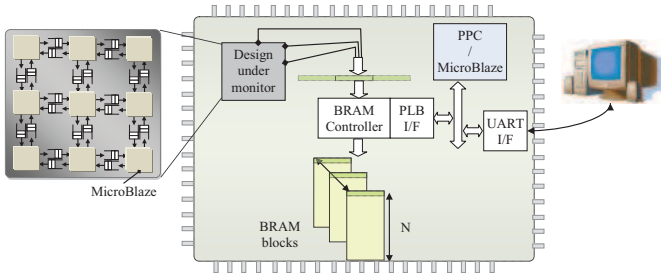

Fig. 2. On-chip memory organization for real-time event tracing

The functionality of the Trace Monitor Unit (TMU) encompasses a programmable filter to reduce the amount of captured data and the required time to process these data in software. In addition, the proposed real-time monitoring and debugging system features on the fly update of the filter of the monitor, which gives a clear advantage over all other on-chip verification tools, such as Chipscope, or ARM's CoreSight. This is particularly useful when a condition is met that triggers subsequent events to monitor, or more importantly when the execution of the embedded application is benefited from these events. Moreover, a series of checkpoints is a usual debugging strategy towards providing guarantees for a working algorithm. In the current implementation the monitoring processor communicates with the TMU using a register based interface. Thus, the filter can be programmed at run-time in the following way:

[T-1] Mask the traced vector using conditioning circuitry to collect only the samples of interest.
[T-2] Perform exact pattern match if strong conditioning is desired.
[T-3] Use a maskable trigger event to enable breakpointing, or versatile run-time control of trigger function.

Actually, the sampled data must satisfy a group of conditions set by the trigger fields and the masks so as to be captured. However, default masks allow an easy and straightforward usage by the designer. Figure 3 shows the internals of a single TMU attached to a on-chip CPU, which for the evaluation testbed is a MicroBlaze soft-CPU core from Xilinx. A few words can be transferred via point-to-point links (Fast-Simplex-Links named FSL, [12]) utilizing FIFOs for synchronization decoupling.

In fact, the TMU is a modular design allowing decoupling of its internal architecture with the type of memory storage used to capture the traces. It is created based on the user's configuration of the *TVdata* and *TVtrigger* signals, while the user can optionally program the mask with software at run-time.

There is the issue of targeting the host platform with these tools. At this point, generic solutions don't exist because of the range of multi-core hardware. Tools primarily target

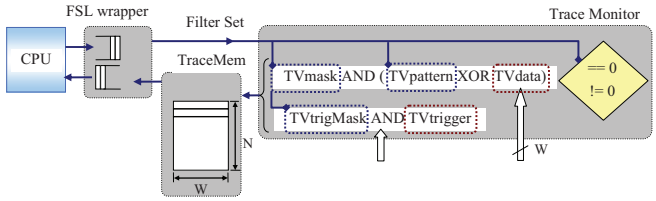

Fig. 3. Architecture of a single programmable trace monitor. The designer specifies the Trace Vector (TV) signals: TVdata and TVtrigger at configuration time, while TVmask and TVpattern can be programmed and modified at run-time.

only one class of hardware or even one vendor's hardware. In the developed framework, besides including a monitor processor or sending the captured information directly out of the chip to a monitoring host, the TMU component is modular and independent from the FSL wrapper. The FSL wrapper is used to exchange information with the MicroBlaze CPU, receiving the filtered samples and setting the appropriate masks according to the designer's code. However, the TMU module is quite generic and can be used easily in other platforms, besides Xilinx's.

Additionally, multiple TMUs can be integrated with a single MicroBlaze processor through the processor's configurable direct FSL extensions. The limitation of attaching up to 16 FSL custom blocks to the MicroBlaze is easily overcome, since a TMU wrapper-manager can form a hierarchical TMU structure with a larger number of TMUs, restricted only by the available on-chip resources. Finally, a potential issue that a designer must tackle is the processing capacity of a centralized monitoring CPU, as the monitoring software has to manage a large number of TMUs.

One must notice that the monitoring infrastructure employs easy to implement language constructs, so as to facilitate real-time decisions. These constructs are primitive to provide ease of use and fast deployment, and very efficient as well:

- change monitor signals, with masks by software at run-time, or with multiplexing in hardware (configuration / generation time)
- provide causal capture (if "event e1 happened" then look for event e2); currently, only four conditions are allowed to respect a reasonable area budget, while at the same time a designer will hardly ever use more conditions.

For instance, a user specification in the *Trace Signals* file of the form:

```
event1 > trigsignal1,... ,trigsignalN1
event2 > trigsignal2,... ,trigsignalN2
```

is translated in a HDL description as follows:

```
case TVdata:
  when event1=> TVtrigger <= {trigsignal1,... ,trigsignalN1};
                TVdata    <= {signal1,... ,signalN1};
  when event2=> TVtrigger <= {trigsignal2,... ,trigsignalN2};
                TVdata    <= {signal2,... ,signalN2};
  when event3=> TVtrigger <= {trigsignal3,... ,trigsignalN3};
                TVdata    <= {signal3,... ,signalN3};
  when event4=> TVtrigger <= {trigsignal4,... ,trigsignalN4};
                TVdata    <= {signal4,... ,signalN4};
end case;
```

This allows zero overhead for the monitor CPU to change trigger conditions, but requires re-synthesis of the system in case of errors, and thus more design time.

IV. Evaluation and Implementation Results

The evaluation testbed utilizes Xilinx's Virtex4-FX20 device as a substrate for experiments, along with simple kernel tasks to investigate rapid deployment, potential hardware overheads and programming model transparency. The testbenches are structured in two parts with increasing degree of complexity.

A. TMU response under software control

The non-invasive monitoring of embedded SoC applications for debugging, reliability, or performance, can be considered as the clear value addition by using this methodology. To this direction an embedded System-on-FPGA is developed, which uses the MicroBlaze soft-processors (see fig. 2 to perform an example application, namely an edge detection filter. Even though this SoC was hosted on a ML405 platform from Xilinx with external SRAM and DRAM memories, the embedded application is located in the local BRAM of the MicroBlaze. A neighboring MicroBlaze core serves as the monitor CPU which communicates with a hardware timer on a PLB bus, and with a UART interface to an external host. The ELF image to apply the filter is also pre-loaded in the local data BRAM. We attached a TMU to the address bus of the data BRAM of the supervised MicroBlaze. This worker MicroBlaze maintains a message buffer in its local Data BRAM while the monitor MicroBlaze observes when this buffer is accessed via its TMU. The physical address of the buffer in the local data BRAM is known after dis-assembly of the ELF executable; , or after profiling; then, the filter masks are programmed. In the current infrastructure a post-compilation script searches the executable code to provide the address of the buffer.

Figure 4 shows the measurement results as reported from the monitoring MicroBlaze. Initially, the TMU is disabled; the monitored MicroBlaze runs an edge detection kernel (based on Sobel filter) which processes an image of 7×180 pixels in 7M clock cycles as measured with the on-chip hardware timer. The buffer is accessed (read or write is not important) in every loop iteration for each row of the image.

Next, the TMU is enabled and the code in the monitoring MicroBlaze (after manual optimizations and compilation with -O3 option) continuously reads the FSL wrapper to extract the captured traces from the TMU memory. In the current implementation one access of the MicroBlaze results in reading of four 32-bit words from the TMU memory. A software timer is incremented with every access of the monitoring MicroBlaze to the TMU. Alternatively, this can be considered as a sampling rate indicator. Multiple experiments with diverse conditions of the application in the worker MicroBlaze and with the monitoring software (communication with neighboring processors is disabled during the kernel processing), the worst case response of the TMU when inquired from the monitor CPU is depicted in Fig. 4 (a). When only the on-chip hardware counter is utilized, the monitor CPU reads the

sampled value from the TMU in 650K clock cycles in worst case; when only the software counter is used, the worst case is 9.2K times. The monitor has the capacity to sample the specified signals which a designer requests in every clock cycle; however the designer must evaluate the time required for software to process the captured data against the capacity of the trace memory.

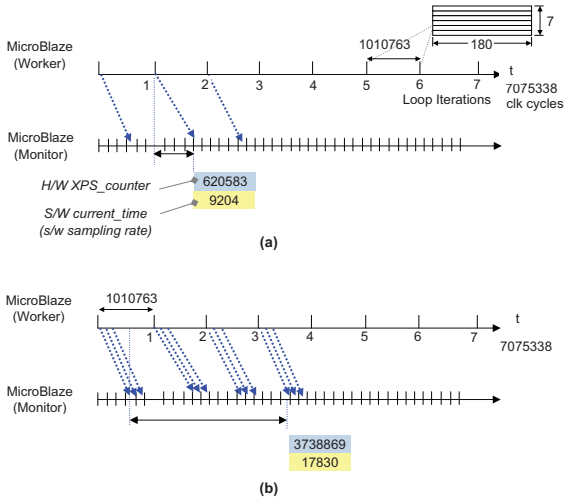

Fig. 4. A monitoring MicroBlaze captures the progress of image filtering running on a different MicroBlaze processor; the goal includes to identify response latency using software and hardware timers

In part (**b**) of Fig. 4 the monitoring CPU manages a 16 entry CAM in software. When the same trace is encountered more than ten times it is logged inside the CAM. Now, in the application code the local **b**uffer in the data BRAM is accessed three times per loop iteration. The result is that an entry is stored in the CAM in almost 3.74M clock cycles in worst case. This is due to the transfer rate of the captured traces from the TMU memory to the processor (4 words per access). The monitor provides indications *empty* and *full* to the monitor master, as regards the status of the TMU memory.

After the edge detection kernel on the pre-loaded image completes, the worker MicroBlaze calculates the gray-scale threshold using an availa**b**le range of 255 values. The addresses of the threshold ta**b**le are extracted from the code and the masks in the monitor are set appropriately. Thus, the monitoring CPU computes and reports the results shown in figure 5 after the application completes. Positive and negative deviations from the average sampling time are also reported to identify potential instantaneous interferences. Notice that the filter allowed only addresses conforming to **b**its [15..7] of the mask 0x00006880 for a total of 16 **b**uffers in local memory.

Alternatively, the free-running timer per monitor can **b**e ena**b**led for attaching timestamps to each event, instead of presenting the average time **b**etween samples.

Ta**b**le I summarizes the implementation cost of indicative monitor configurations; Unless we integrate complex functions in hardware, such as compression or classification of events, the cost of integrating even multiple TMUs is affordable.

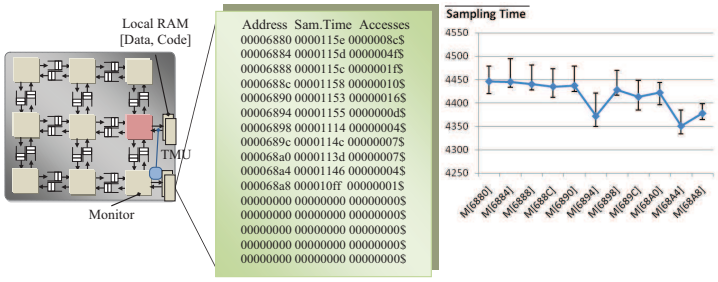

Fig. 5. Monitor report during execution of image filtering, threshold computations; in addition to correctness the goal is to offer real-time services towards monitoring tasks with hard real-time constraints

TABLE I
TRACE MONITOR UNIT IMPLEMENTATION RESULTS

Configuration	Slices	RAMB16s	Freq
TMU 72**b**it monitor with UART Tx i/f (ext. Host)	358	6	405.8
CAM plugin (16×192)	1364	18	211
TMU 32**b**it monitor with FSL i/f to MicroBlaze	432	3	418

B. Monitor aware multi-core SoC

If we assume that the monitoring system is not only a passive o**b**server **b**ut the worker CPU considers and reacts on the outcome of the monitor CPU, then, the designer must take into account the total *round-trip time* with respect to each individual source of delay. Hence, as refers to the delay **b**etween the occurrence of an event and the reaction of the worker processor, critical issues arise: (i) whether the monitor infrastructure faces **b**ounded **j**itter as regards the processing of the events, (ii) what is the most efficient scheme to notify the worker processors (interrupt triggered, message passing, etc.) considering the NoC organization, the memory hierarchy and the interrupt latency of a particular CPU.

If multiple monitors are attached to a monitor processor, then, the processor software in the centralized configuration, as in the implementation shown in section IV-A, must sweep all the monitors in a round-ro**b**in fashion, assuming all are of the same priority.

The **b**oundary condition for an effective scheme can **b**e formulated as:

T_{event2}-T_{event1} >= $T_{monproc}$ + $T_{reaction}$ + T_{TMU}

where,

$T_{monproc}$: the time for the monitor processor to process the captured event,

$T_{reaction}$: the time for a MicroBlaze node to understand the message of the monitor processor, and,

T_{TMU}: the time for the TMU to filter an event, store it in a local **b**uffer or memory and send it to the monitor processor.

To achieve this target each time-consuming factor must **b**e addressed. The proposed infrastructure in this work is efficient if applied at task level granularity, since, as the experiments show, even with optimized code in the monitor CPU we need carefully selected filters to remove gar**b**age traces. Shifting to hardware management techniques and distri**b**uted schemes will

consequently open the way to sub-microsecond monitoring of hundreds of cores.

In order to pause, reconfigure, or perform synchronization operations we investigated the latency of interrupt-based monitor-SoC communication, and thus we developed the evaluation scenario shown in Fig. 6. This includes just one set of the processing and the monitor MicroBlaze in interactive mode. Even though the monitor's role can be non-intrusive, it can potentially interfere to provide monitoring services for deterministic replay, or for more difficult cases like debugging shared variables, or interrupt handling. The key in these cases is to provide predictability.

In the depicted scenario the monitor probe collects the sampled traces and decides to communicate with one CPU via interrupt, so, the sequence of events shown in the figure takes place. The monitor CPU inquires the monitor probe, then sends a message to the worker and gives up polling the monitor interface; after, continuously reads a one-word (i.e., four bytes) message from the incoming port of the mailbox. The worker CPU is notified when a message arrives in the mailbox FIFO via interrupt. Each counter connected to the PLB adds a small overhead, but its usage only serves this measurement scenario. To avoid risking large penalties, integrated counters in place can be used.

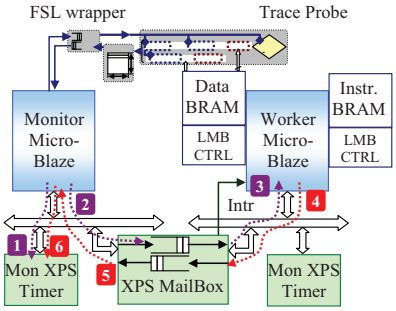

Fig. 6. Testbench setup of an integrated real-time monitor manager MicroBlaze with a monitor unit connected via FSL, measuring round-trip time while exchanging messages with the worker processor.

The average round-trip time to react for an identified event of interest costs 7070 clock cycles. Experiments with the process initiated from the worker MicroBlaze exhibit an average latency of 6920 clock cycles. In this scenario, the worker MicroBlaze starts its local XPS timer, then the sequence of events 4→5→2→3; the event 3 causes an interrupt and inside the interrupt service routine the value of the XPS timer is sampled. To further understand the performance difference from the application's perspective, the current application consumes 650K clock cycles between subsequent samples, which translates to almost 80 messages with response from the hardware processing units (i.e., the MicroBlaze).

Notice that if we integrate a hardware CAM to classify the captured events, with only 3 clock cycles per search operation, then, we can significantly accelerate $T_{monproc}$ of the monitor CPU. Nevertheless, the area cost of this plug-in is significant as shown in table I, making it affordable only for large SoCs. The method and dedicated logic or hardware components may have a significant impact on the ability to observe and detect specific execution behavior or system states at runtime. Predictable real-time system operation can be dependent on meeting tight timing constraints that may be perturbed during online observation or debugging. Different embedded application behavior, as well as distributed monitoring schemes will be explored in future work.

V. Conclusions

A novel infrastructure is introduced to address the real-time monitoring of multi-core SoCs for debugging in an non-invasive way, and for performing on-line adjustments, optimizations, and control in multi-process environments. This new methodology is applied in a reconfigurable environment using soft-cores. The generated centralized monitoring system allows a variety of configurations as regards the memory used for traces, the interfaces to the off-chip host and to the on-chip monitoring CPU. The internal organization of the template-based monitors allows a compact and flexible filtering easily configured off-line and programmed on-line.

The evaluation of the presented infrastructure shows that a well-balanced co-design of hardware debug blocks to provide comprehensive on-chip instrumentation and debug solutions, together with efficient software management of these hardware monitors opens remarkable possibilities for a system designer in the era of large multiprocessor SoCs.

References

[1] M. Kudlugi, S. Hassoun, C. Selvidge, and D. Pryor, "A transaction-based unified simulation/emulation architecture for functional verification," in *Proceedings of 38th Design Automation Conference DAC 01*, pp. 623–628, 2001.

[2] A. Raynaud and M. Mohtashemi, "Transaction-level co-emulation added to vmm methodology," *Synopsys Insight*, vol. 2, no. 4, 2009.

[3] A. Coresight, [Online] Available: http://www.arm.com/products/solutions/CoreSight.html.

[4] E. Cota, L. Carro, and M. Lubaszewski, "Reusing an on-chip network for the test of core-based systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 9, no. 4, pp. 471–499, 2004.

[5] A. Amory, E. Briao, E. Cota, M. Lubaszewski, and F. Moraes, "A scalable test strategy for network-on-chip routers," in *Proceedings of IEEE International Test Conference (ITC)*, Nov. 2005.

[6] B. Roesler and E. Nelson, "Debug methods for hybrid cpu/fpga systems," in *Proceedings of Field-Programmable Technology*, 2002, pp. 243–250.

[7] M. Aguirre, J. Tombs, V. Baena-Lecuyer, J. Mora, J. Carrasco, A. Torralba, and L. Franquelo, "Microprocessor and fpga interfaces for in-system co-debugging in field programmable hybrid systems," *Microproc. and Microsyst.*, vol. 29, no. 2-3, pp. 75–85, Apr. 2005.

[8] P. Amer and L. Cassel, "Management of sampled real-time network measurements," in *Proceedings of 14th Conference on Local Computer Networks*, October 1989, pp. 62–68.

[9] M. Scholler, T. Gamer, R. Bless, and M. Zitterbart, "An extension to packet filtering of programmable networks," in *Proceedings of 7th International Working Conference on Active Networking*, Nov. 2005.

[10] C. Ciordas, A. Hansson, K. Goossens, and T. Basten, "A monitoring-aware network-on-chip design flow," *J. Syst. Archit.*, vol. 54, no. 3-4, pp. 397–410, 2008.

[11] K. Goossens, B. Vermeulen, and B. Nejad, Ashkan, "A high-level debug environment for communication-centric debug," in *Proceedings of Design, Automation and Test in Europe*, Apr. 2009.

[12] Fast Simplex Link (FSL) Bus (v2.11a), available: www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pdf, Xilinx Inc., Data Sheet DS449, Jun. 2007.