System-Level Observation Framework for Non-Intrusive Runtime Monitoring of Embedded Systems

SYSTEM-LEVEL OBSERVATION FRAMEWORK FOR NON-INTRUSIVE RUNTIME MONITORING OF EMBEDDED SYSTEMS

by

Jong Chul Lee

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2014

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Jong Chul Lee
entitled System-Level Observation Framework for Non-Intrusive Runtime Monitoring of Embedded Systems
and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

_____ Date: 10/23/14
Dr. Roman Lysecky

_____ Date: 10/23/14
Dr. Ali Akoglu

_____ Date: 10/23/14
Dr. Meiling Wang

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

_____ Date: 10/23/14
Dissertation Director: Dr. Roman Lysecky

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

SIGNED: _____

Jong Chul Lee

ACKNOWLEDGEMENTS

I have many people to thank for this dissertation. First and foremost, I would like to give my tremendous thanks to my advisor, Professor Roman Lysecky, for all the advice, supports, encouragements, and trusts he has offered to me without any reservation. He guided me through the years and helped me to grow passion and become more and more mature in academic research. Because of his support and understanding, I had a pleasant and valuable experience during my Ph.D. study in the Electrical and Computer Engineering Department at the University of Arizona.

Special thanks go to Professor Ali Akoglu, Professor Meiling Wang, Professor Susan Lysecky and Professor Wei Hua Lin for the excellent courses they taught and for serving on my Ph.D. Written and Oral Comprehensive Exams as a committee member.

I would like to thank Tami Whelan for handling all the paper work necessary for the completion of my degree. I would like to express my appreciation to the supporting staff in the Department of Electrical and Computer Engineering for maintaining a friendly and stable learning environment.

I would also like to thank Doohwan Kim, a president of RTSync, and Chungman Seo, a senior research engineer at RTSync, for the excellent internship opportunity.

I would like to acknowledge my fellows and friends Hyun Jin Park, Sungjong Yoo, Junseok Kim, my labmates Karthik Shankar, Sachidanand Mahadevan, Vijay Gopinath, Jingqing Mu, Nathan Sandoval, Adrian Lizarraga and many other friends at University of Arizona, for their help and encouragement during my six years graduate student life here.

Last but not least, my family is to be thanked: Munam Lee, Booja Chung, Hoajin Lee, Bongjae Kim, Hyo Jin Lee, Chang Hwan Yoon, Jinchul Lee and Ko Eun Choi.

DEDICATION

To my parents, Munam Lee and Booja Chung, without whose love and encouragement throughout the years this thesis would not have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

As system complexity continues to increase, the integration of software and hardware subsystems within system-on-a-chip (SOC) presents significant challenges in post-silicon validation, testing, and in-situ debugging across hardware and software layers. The deep integration of software and hardware components within SOCs often prevents the use of traditional analysis methods to observe and monitor the internal state of these components. This situation is further exacerbated for in-situ debugging and testing in which physical access to traditional debug and trace interfaces is unavailable, infeasible, or cost prohibitive.

In this dissertation, we present a system-level observation framework (SOF) that provides minimally intrusive methods for dynamically monitoring and analyzing deeply integrated hardware and software components within embedded systems. The SOF monitors hardware and software events by inserting additional logic within hardware cores and by listening to processor trace ports. The SOF provides visibility for monitoring complex execution behavior of software applications without affecting the system execution.

The SOF utilizes a dedicated event-streaming interface that allows efficient observation and analysis of rapidly occurring events at runtime. The event-streaming interface supports three alternatives: (1) an in-order priority-based event stream controller, (2) a round-robin priority-based event stream controller, and (3) a priority-level based event stream controller. The in-order priority-based event stream controller, which uses

efficient pipelined hardware architecture, ensures that events are reported in-order based on the time of the event occurrence. While the in-order priority-based event stream controller provides high throughput for reporting events, significant area requirement can be incurred. The round-robin priority-based event stream controller is an area-efficient event stream ordering technique with acceptable tradeoffs in event stream throughput. To further reduce area requirement, the SOF supports a priority-level based event stream controller that provides an in-ordering method with smaller area requirements than the round-robin priority-based event stream controller.

Comprehensive experimental results using a complete prototype system implementation are presented to quantify the tradeoffs in area, throughput, and latency for the various event streaming interfaces considering several execution scenarios.

CHAPTER 1

INTRODUCTION

As system complexity continues to increase, the integration of software and hardware components within embedded systems presents key challenges in monitoring and analyzing complex hardware and software interactions. The deep integration of software and hardware components within embedded systems often prevents the use of traditional analysis methods to monitor and analyze the internal state of these components. This situation prevents the use of a logic analyzer to observe the interaction within embedded systems and may affect the system correctness during monitoring the erroneous behavior.

Existing debugging methods that require the system execution to be halted are intrusive, either requiring significant hardware resources or leading to system perturbations that can change the execution behavior, and pose considerable challenges for in-situ analysis. For example, JTAG [29] scan chains allow all registers within an SOC design to be monitored or controlled at runtime. However, in order to access those registers, the system execution must be halted. This perturbs the system execution such that observing the desired behavior may no longer be possible. Therefore, for in-situ analysis of monitored events, such intrusive methods are often infeasible, and when utilized may lead to system failure due to timing constraint violations—e.g. missed execution deadlines. Although many challenges exist for runtime in-situ system monitoring and testing, pre-silicon verification and testing methods provide a wealth of information that can potentially be utilized to efficiently monitor system execution at

runtime. For example, important pre-silicon verification requirements can be effectively reused within post-silicon validation and testing procedures [5][6].

Intrusive debugging methods pose considerable challenges in real-time systems, for which hard execution constraints are critical to system correctness. If a task within the system does not complete its execution within the required time, the task can be considered to have failed. Whereas failure to meet hard execution deadlines may result in complete system failure, failure to meet soft execution deadlines lead to undesired system behavior that can incur system failure. Soft deadlines must still be met to meet the desired systems goals. Hence, new debugging and verification methods are needed to provide in-situ analysis methods capable of monitoring software and hardware interactions without perturbing the system execution.

To overcome the challenges of traditional JTAG interfaces, numerous approaches have focused on trace-based methods for logging system events in both hardware and software components using dedicated trace and debug ports. For example, ARM's CoreSight [7] and Embedded Trace Microcell [8] can be synthesized within an SOC design to provide system-level trace capabilities using a dedicated trace port. However, these system-level trace methods are often limited in the amount of data that can be traced and stored in real-time or limited by the bandwidth of the trace port in reporting data to external test equipment.

In this dissertation, we present an event-driven system-level observation framework (SOF) providing low-overhead methods for observing and analyzing complex interactions across hardware and software boundaries at runtime. The SOF provides in-

situ support for controlling event probes within software and configuring hardware components using blocking, non-blocking, and cascading configurations. For serializing and reporting rapidly occurring event, the SOF provides three types of a priority-based event streaming interfaces. The contributions in this dissertation are: 1) a configurable, nonintrusive framework for monitoring designer-specified hardware and software events; 2) advanced observation methods for analyzing complex system events using blocking, non-blocking, and cascading event probe specifications; 3) a high-throughput pipelined, priority-based event streaming interface for serializing and analyzing monitored events at runtime; 4) area-efficient priority-based event streaming interfaces for efficiently reporting monitored events at runtime; and 5) a software sorting algorithm for efficiently sorting the event stream to provide a time ordered stream of observe events.

In Chapter 3, we present an initial framework for minimally intrusive hardware observability that provides designers with the ability to monitor complex application-specific hardware execution behavior at runtime with zero or minimal impact on system execution. In Chapter 4, we present an event-driven system-level observation framework that provides low-overhead methods for observing and analyzing designer specified hardware and software events at runtime. In Chapter 5, we present a system observation framework for monitoring and analyzing rapidly occurring software events. This system observation framework provides runtime support for defining and controlling software events with using blocking, non-blocking, and cascading event probes. In Chapter 6, we present an area-efficient event stream ordering technique that significantly reduces area requirements and two software sorting algorithms with acceptable tradeoffs in event

stream throughput. In Chapter 7, we present the priority-level based event streaming interface and a software sorting algorithm. In Chapter 8 and Chapter 9, we conclude and highlight future work.

CHAPTER 2

RELATED WORK AND BACKGROUND

In this section, we provide an extensive overview of related work on runtime trace and debug methods for hardware and software components. Table 1 provides a summary and classification of related work highlighting the collection method, target components, analysis method, storage, intrusiveness, and runtime configurability for each approach. The *collection method* defines how an approach collects data within the target system using trace based, scan based, or event driven alternatives. The *target* highlights the components within the SOC the approach seeks to monitor categorized as hardware or software. The *analysis method* indicates how and where the observed information is analyzed, including in-situ on-chip, in-situ off-chip, or offline. The *storage* defines where the collected information is stored within the system, including on-chip buffers, off-chip memory, none, or user-defined. The *intrusiveness* of an approach is defined as how the approach affects the execution of the system categorized as non-intrusive, minimally intrusive, and intrusive. A non-intrusive approach is one that in no way affects or perturbs the system execution. In contrast, an intrusive approach exhibits considerable impact on the system execution to the extent that it can affect both the correctness of the system execution and the validity of the information. A minimally intrusive approach is one that may impact the system, but the impact is either minor or can be controlled at runtime to minimize or eliminate the negative effects of the monitoring method. Lastly, the *runtime*

Table 1: Summary of related work in dynamic trace and debug methods for hardware and software components of embedded systems.

Reference	Collection Method[a]	Target[b]	Analysis Method[c]	Storage[d]	Intrusive-ness[e]	Runtime Config.[f]	Overheads and Note
JTAG [29]	S	H/S	OF	N	I	N	Requires systems to be halted to access internal signals.
[2][3]	T/E	H	ION/IOC	B	N	C	Reconfigurable hardware resources for implementing monitoring logic, programmable through JTAG
[7][8]	T/S	H/S	ION/IOC/OF	B	N/I	N	Limited by the number of signals that can be traced and bandwidth of trace port.
[11][27][28][79]	T	S	ION/IOC	N	N/M	C	Utilizes the hidICE emulator to observe multicore SOC designs. The hidICE provides real-time trace with minimal interference.
[18]	E	H	OF	N	I	N	Halts systems to report external debug events.
[21][22]	E	H/S	IOC	O	N/M	N	Requires host systems to analyze trace events.
[24][25][75][76]	T/S	H	IOC/OF	B	N/I	N	Limited by the number of signals and duration that can be traced due to on-chip memory and JTAG bandwidth limitations.
[26][77][78]	E	S	IOC	N	M	N	Lightweight software instrumentation utilized to monitor software events.
[31]-[40][60]	E	H	OF	B	N/M	C	Utilizes efficient methods for controlling when to trace and how to store data within available trace buffer. Limited by size of trace buffers and bandwidth of off-chip access.
[41][42][43][74]	T/S	H	OF	B	I	C	Option exists for implementing custom analysis logic in hardware to filter trace signals
[49]-[54]	E	H	OF	B	N/M	C	Method for custom creation of interconnection fabric for trace buffers.
[67]	E	H	IOC	N	M	N	Utilizes off-chip assertion checker implemented within an FPGA.
[70][71]	E	H	IOC	O	I	C	Profile of monitored events is transmitted over system bus.
[72]	S	S	ION	N	M	N	To limit intrusion, an extension of GDB enables a non-stop mode in which only a single task is stopped during debugging and all other tasks can execute normally.
[86]	S	H	OF	N	I	N	Uses custom scan-chain for monitoring a subset of signals within system.
SOF	***E***	***H/S***	***ION***	***U***	***N/M***	***C***	***Event-driven in-situ observation framework supporting configurable method for observing, configuring, analyzing, and reporting observations.***

a. T=Trace, S=Scan-chain, E=Event-driven
b. H=Hardware, S=Software
c. ION=In-situ on-chip, IOC=In-situ off-chip, OF=Offline
d. B=On-chip Buffer, O=Off-chip memory, N=None, U=User-define
e. N=Non-intrusive, M=Minimally intrusive, I=Intrusive
f. C=Configurable, N=Non-configurable

configurability indicates if an approach can be configured at runtime to select which signals or events to monitor.

A software debugger allows an engineer to debug a software design by halting the execution of software at a particular point and examine the state of the software by observing the state of the processor's internal registers and system memory. A software breakpoint works by inserting a special instruction in the software design to be debugged. When the instruction is called, it invokes the debugger's exception handler. Similar tools exist for hardware designs [18][86], but it is difficult to pragmatically match the utility of a software debugger given the inherently parallel execution of hardware cores for two reasons. First, software is fundamentally linear. While high-level programming languages may obscure the fact, at the machine interface, software is a linear sequence of instructions. Second, the regularity of the load-store computer architecture means that intermediate results almost usually return to the memory system.

Furthermore, a debugger has a high utility only when testing a subsystem in isolation. As the number of subsystems that the debugger does not control increases, the utility of the debugger decreases dramatically. Halting one subsystem is of low value if the rest of the system—e.g. sensors, actuators, physical processes—continues to operate. Debugging within real-time systems presents additional challenges as proper operation is dependent on meeting tight timing constraints that can be easily perturbed during debugging.

truss [59] is an example of an indiscriminate trace tool that allows users to capture data regarding every system call that a program makes. This execution trace can be a powerful analysis tool because it isolates a very specific identifiable behavior, e.g. system calls, and makes a history available to the designer. If, however, a program's suboptimal

behavior is caused by the timing or frequency of making specific system calls, indiscriminately tracing with *truss* may slow the program down enough to cause it to leave the set of states we want to observe.

Solaris DTrace [19][59] and Linux SystemTap [68] are examples of dynamic software observability that are differentiated by several key attributes. Solaris DTrace implements several optimizations of dynamic tracing that are important to consider, as DTrace partially served as the inspiration for supporting hardware and system-level observability. First, disabled probes have no insertion penalty. In other words, a disabled probe incurs no execution overhead. Second, insertion of a probe does not require a design-time decision. Third, the code inserted for a probe point must be inserted in a manner consistent with the security and reliability requirements of the system. Unlimited insertion of arbitrary code would be likely to permit a user to circumvent security restrictions or insert unstable software.

In the context of real-time systems, previous work has focused on reducing the overhead of traditional software debuggers. When debugging a single task within a multitasked system, stopping all tasks during debugging is extremely intrusive, and can lead to incorrect behavior and even system failure. To minimize this intrusion, an extension of GDB enables a non-stop mode in which only a single task is stopped during debugging and all other tasks can execute normally [72]. This allows a user to control tasks explicitly in ways that are not possible in all-stop mode, meaning that all tasks of execution stop during debugging.

Leatherman and Stollen [41][42][43][74] present a debug methodology that incorporates distributed on-chip instrumentation (OCI) components allowing designers to configure how to trace—e.g. by defining the trace width and depth—and when to trace—e.g. by defining triggers that start the trace process. The distributed OCI components can then be connected by a dedicated bus to an on-chip analyzer that can control and process the trace data before making that data accessible off-chip through a JTAG interface. The authors further propose a HyperJTAG interface that combines existing JTAG interfaces for processors within a multicore system with the distributed OCI components available through a single IP interface.

Similarly, Vermeulen and Goel [24][25][75][76] present a silicon debugging strategy for multiple clock domain systems using a JTAG port, in which an on-chip memory is utilized to trace specific internal signals that can later be accessed through the JTAG port. The proposed on-chip debug infrastructure and debugging software provide support for both real-time and time-intrusive monitoring. In order to support real-time, i.e. nonintrusive, monitoring, an on-chip memory is utilized to trace specific internal signals, which can then be accessed through the JTAG port. However, due to limitation in the availability on an on-chip memory, the duration and number of signals that can be monitored in real-time are limited.

Abramovici et al. [2][3] present a distributed reconfigurable fabric of multiplexers enabling designers to select a subset of signals to monitor. The selected signals are processed by a debug monitor that can directly forward the captured signals or perform basic processing on those signals—e.g. the debug monitor can directly process signals to

only report anomalous ones. Each configurable multiplexer and debug monitor operates within a single clock domain. The outputs from the distributed debug monitors are collected together by a trace component that records the signals into an on-chip memory that can then be accessed by a JTAG port. The proposed distributed debug offers the advantage of being able to limit the amount of data that is traced, supporting multiple clock domains, and eliminating the need to route all probes to a single trace component, which can often lead to a unroutable design during the synthesis process.

Ko et al. [31]-[40][60] propose a system-level debug architecture targeted for post-silicon validation that utilizes configurable event triggers, a network of trace buffers, and a configurable communication framework for efficiently storing data samples within the available trace buffers. The event triggers enable designers to specify conditions that will start tracing of specific signals. The trigger conditions can be configured by designers through a set of configurable comparators. This approach enables designers to change event triggers at runtime. The proposed debug architecture includes a network of trace buffer for handling the simultaneous tracing of multiple data signals. The trace buffer architecture controls how traced signals are stored among available trace buffers according to designer specified priorities. While this debug methodology provides support for runtime configuration, the proposed approach is not focused on enabled runtime in-situ analysis of tracing events and data.

Liu and Xu [49]-[54] propose a methodology for creating an area efficient trace interconnection fabric. Given the set of signals that need to be traced, a custom interconnection fabric is created in which multiplexers are utilized to trace mutually

exclusive signals—i.e. signals that are unlikely to occur simultaneously—and a custom crossbar network is utilized to trace concurrently accessible signals. Using either designer specified identification of the types of signals to trace or analysis of the circuit structure, a customized interconnection fabric can be generated.

Rather than incorporate a scan-chain directly within an RTL design, Yang et al. [86] propose an alternative automated method for selecting and extracting a subset of internal signals to be monitored. The selected signals are monitored by a separate FPGA-based test platform that provides scan-chain access to these signals for use within a co-simulation environment. The propose methodology has the advantage of being able to utilize the original testbench developed for the RTL design, while relying on automated tools to extract the desired signals for post-silicon co-simulation and verification.

Watterson and Heffernan [26][77][78] propose an online software monitoring method with the specific focus on developing a minimally intrusive method so as not to affect the execution of the application. Within the proposed method, events from the processor can be generated by minimally intrusive software instrumentations that report an event to a dedicated on-chip monitoring core. The monitoring core can then process the events and report the required data to the external environment.

The Owl framework [70][71] is a distributed approach that incorporates monitoring modules within the specific parts of the system to be monitored. The distributed monitoring modules communicate the profile data to a specific location in main memory or to a separate memory dedicated for profiling. However, because the monitors will need to transmit this data via the system bus, the proposed approach can be intrusive due to

bus contention. As the intended target is a system realized within an FPGA, the monitoring modules themselves are reconfigurable. This allows a designer to change how the monitoring process is implemented at runtime. For example, a designer can reconfigure the monitoring modules to alter the frequency at which the profile data is written to memory to reduce the profiling traffic on the system bus.

The MAMon monitoring system [21][22] proposes a methodology for monitoring hardware and software based events within SOC designs by incorporating dedicated logic within hardware components to detect occurrences of specific events being monitored. A probe unit is utilized to capture and log all occurrences of these events within an external memory. Events within the MAMon system are defined as conditional expressions that are evaluated during each clock cycle. However, a host workstation is required to view and analyze the event log with capabilities for filtering and searching the monitored events.

Hardware assertions are typically used during the validation and verification stages of hardware developments. Assertions can be specified to formally define design requirements often utilized within simulations to verify correct behavior. If the requirements are violated, an error will be asserted such that designers can identify and correct the incorrect behavior. A common method for specifying design requirements using assertions is the Property Specification Language (PSL) [4][30].

Research efforts have also resulted in methods and tools for automatically generating hardware assertion checkers in the form of VHDL or Verilog code from assertion specified using PSL [1][14]-[17]. For example, the FoCs tool [1] was developed

to automate the verification process by generating hardware assertion checkers from a PSL specification that can be directly utilized within standard simulation environments. The FoCs tool significantly reduces development time and costs by eliminating the need for designers to manually create HDL for the assertions and eliminating the need to utilize specialized tools. Although the FoCs tool was initially targeted at design time simulation and verification, several efforts have explored the benefit of online assertion checkers [12][13][55]-[58][62]-[66]. Online assertion checking directly integrates the hardware assertion checkers within the prototype—or even final—hardware implementation. Such an approach offers several advantages over simulation, including the ability to verify the system within a deployed environment. In contrast to the proposed observability framework, online assertion checker only logs the occurrence of an assertion, and does not provide any mechanism for analyzing or responding to these assertions at runtime.

In [67], a debugging and verification environment is presented that is capable of monitoring multiple internal signals—or hardware probes—within a hardware design and provides real-time trace for a subset of those signals via a dedicated debug port. Given a large number of signals a designer may want to monitor, the debug port provides a reconfigurable data filter that can be rapidly reconfigured to select the subset of signals that are traced. This provides a balance between the inputs and outputs necessary for the debug port and the number of internal probes that can be supported. The trace data is transmitted to an assertion checker implemented by using an FPGA that can be utilized to

verify correct execution of the device—typically by verifying properties defined within an assertion language, such as PSL.

Backasch et al. [11] presented a runtime verification approach to observe multicore SOC designs and verify designer-specified system properties. The approach utilizes the hidICE (hidden ICE) emulator [27][28][79] that transfers trace data to external analysis tools. The behavior and the instructions carried out by the target SOC design can be precisely reconstructed and emulated by the hidICE emulator. The hidICE emulator enables observability of multicore SOC activities (e.g., bus control events, bus reads, interrupts, processors power state changes) to capture real-time, and concurrent trace of processors and hardware cores in shared bus multicore SOC. This framework utilizes a combination of on-chip analysis to extract the synchronization event needed between the target SOC and emulator, and off-chip analysis within the emulator and host device for analysis the system execution.

CHAPTER 3

HARDWARE OBSERVABILITY FRAMEWORK FOR MINIMALLY INTRUSIVE ONLINE MONITORING OF EMBEDDED SYSTEMS

3.1 Overview

As the complexity of digital systems rapidly increases, designers are presented with significant challenges in monitoring, analyzing, and debugging the complex interactions of various software and hardware components. Existing hardware tests and debugging methods are often intrusive, either requiring significant hardware resources or requiring the execution of the system to be halted thus leading to system perturbations that can change the execution behavior to an extent that the erroneous behavior can no longer be observed—or lead to system failure due to missed execution deadlines.

In this chapter, we present an initial framework [44] for hardware observability that extends such dynamic observation capabilities to hardware and mixed hardware/software environments, thereby providing a runtime environment permitting system-wide observability. The hardware observability framework provides a flexible mechanism that can detect arbitrary designer-specified hardware events and allows users to observe those events through user-defined observation software.

3.2 Hardware Observability Framework

Figure 1 provides an overview of the proposed hardware observability framework for a system incorporating a Xilinx MicroBlaze processor [61][82] and several hardware cores.

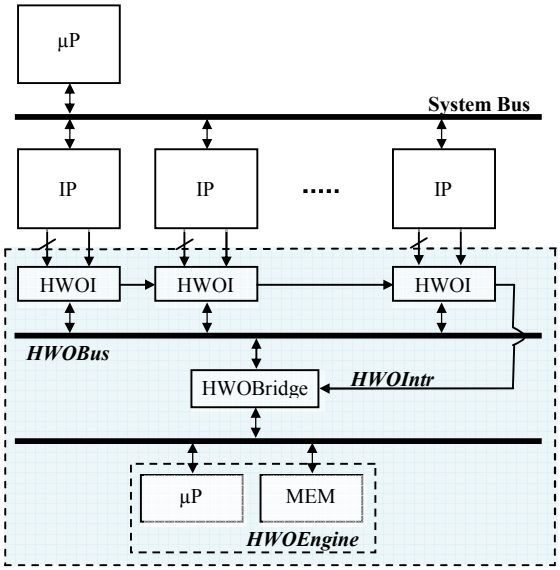

Figure 1: Nonintrusive hardware observability framework for a MicroBlaze processor system consisting of several hardware observability interfaces (*HWOI*), a dedicated hardware observability bus (*HWOBus*), a hardware observability bridge (*HWOBridge*), and a hardware observability engine (*HWOEngine*) to execute the observation software.

We utilize a MicroBlaze processor-based system as an illustrative example throughout this article to closely match our experimental results in which the hardware observability framework was implemented and evaluated using a Xilinx FPGA. In contrast to existing methods for debugging, testing, and tracing of hardware designs, the hardware observability provides a nonintrusive framework for monitoring complex designer-specified hardware events. In response to those hardware events, designers can create customized observation software to analyze and monitor the hardware events for the specific testing, debugging, or monitoring tasks at hand.

The nonintrusive hardware observability framework for a MicroBlaze processor systems consists of one hardware observability interface (*HWOI*) for each hardware core,

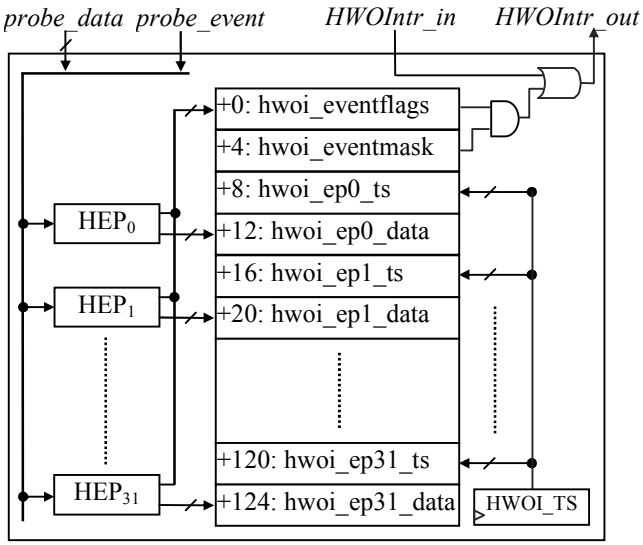

Figure 2: Hardware observability interface consisting of up to 32 hardware event probes (HEPs), timestamp register (HWOI_TS) providing relative cycle-level execution counts, and memory-mapped interface for both controlling the hardware observability monitoring and accessing runtime information for each HEP.

a dedicated hardware observability bus (*HWOBus*), a hardware observability bridge (*HWOBridge*), and a small secondary microprocessor for the hardware observability engine (*HWOEngine*) to execute the observation software.

3.2.1 Hardware Observability Interface (HWOI)

The hardware observability interface (HWOI), presented in Figure 2, provides the fundamental framework for dynamically monitoring designer-specified elements within hardware circuits without intrusion or interruption to the circuit operation. The HWOI consists of one or more hardware event probes (HEPs), a timestamp register (HWOI_TS), and a memory-mapped interface for controlling individual event probes and accessing the observation data for each probe.

Within the hardware observability approach, the basis element that can be observed is a hardware event. For each hardware IP core, the hardware events that need to be monitored at runtime area dependent on each specific hardware design. Hence, a designer must specify both the set of events to be observed and the set of probe signals from the IP cores that are needed to make these observations. While this approach requires additional effort on behalf of the designer, the core designer is the best source of knowledge for determining which event probes are needed to provide the highest level of observability for that hardware circuit. We note that although manual effort is currently required to incorporate additional hardware probes, our immediate future work focuses on developing automated tools for specifying hardware probes and automatically inserting the required logic within the hardware core. This event-driven observation approach provides significant flexibility in that complex execution behavior can be effectively observed at runtime and rapidly integrated within the observation framework.

The probes exported from the hardware cores can either consist of direct connections to internal signals or dedicated logics needed to detect an internal event. For example, consider a hardware event probe intended to monitor when access is granted to a shared bus for a burst transaction. To monitor this event, only the bus acknowledge signal needs to be exported to the HWOI. On the other hand, in order to observe a complex event such a specific execution order of states within a finite state machine, additional logic may be needed to track the sequence of events needed to trigger the event probe. Alternatively, the state register for the FSM can be exported to the HWOI, in which the logic required to detect the execution sequence can be incorporated.

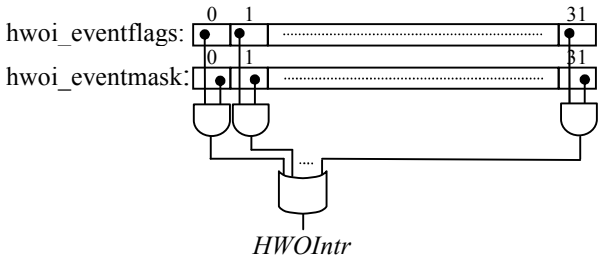

Figure 3: HWOI interrupt (*HWOIntr*) generation indicating the one or more *unmasked* hardware events have been observed.

The HWOI includes a timestamp counter (HWOI_TS) that provides a simple mechanism for analyzing the relative timing behavior between individual events. The HWOI_TS is a 32-bit free running counter that is incremented each hardware observability cycle. Considering a 100 MHz clock, the HWOI_TS counter provides the ability to measure the timing of event occurrences up to 40 seconds apart, beyond which a user would be required to develop additional timing capabilities within the observation software. For systems with higher clock frequencies the size of the HWOI_TS can be increased appropriately.

Each hardware event probe (HEP) defined by the designers will be associated with a one-bit event flag, a one bit event mask, a 32-bit timestamp register, a 32-bit data register, and HEP controller for the controlling the observation processes.

The events flags and event masks for all hardware event probes are accessible through two memory-mapped registers, *hwoi_eventflags* and *hwoi_eventmask*. Reading the *hwoi_eventflags* register will return to the current state for all HEPs within HWOI, where a logical one indicates the hardware event has been observed, and a logical zero

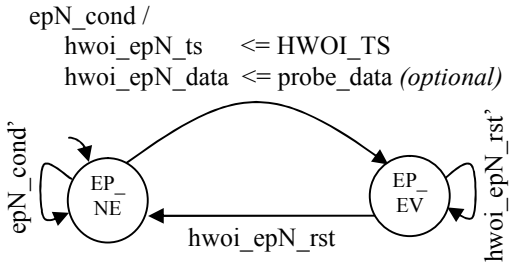

Figure 4: Hardware event probe (HEP) controller.

otherwise. Individual HEPs can be masked—or disabled—by configuring the *hwoi_eventmask* register.

If any unmasked hardware event has been observed, the HWOI will assert an interrupt, *HWOIntr*, as shown in Figure 3. The *HWOIntr* is daisy-chained through all HWOIs within the system, providing a single interrupt to the hardware observability engine (HWOEngine) to indicate that one or more hardware events have been observed.

Each HEP contains an HEP controller implemented as a simple state machine consisting of two states, as shown in Figure 4. Initially, the HEP controller waits in the *no event* state, EP_NE, until the desired event is observed, defined by the logical expression *epN_cond*. When the hardware event is observed, the HEP controller will capture the current HWOI_TS value and probe data into the HEP's timestamp, *hwoi_epN_ts*, and data, *hwoi_epN_data*, registers—both of which are optional and can be defined by the designer. The HEP controller will then transition to the *event* state, EP_EV. In the EP_EV state, the HEP controller waits until a reset signal, *hwoi_epN_rst*, is asserted for the probe. Writing a zero into the corresponding bit of the hwoi_eventflags register generates the reset signal for an individual probe.

Minimally, to define an HEP, as designer must specify a Boolean condition corresponding to the target event and optionally define which data value should be captured on occurrence of this event. This specification can be directly incorporated within the state machine for the HEP controller. Alternatively, for monitoring complex sequential events, a designer can incorporate additional states within HEP controller to detect complex sequences for the target event.

We have currently designed the HWOI to support up to 32 HEPs, requiring a total of the 66 memory-mapped registers within the HWOI. The memory-mapped addresses for each register are illustrated in Figure 2. If a timestamp or event probe data is not needed for an HEP, the associated registers will be replaced with logical zeros in order to maintain a consistent memory mapped address space. Although currently supporting a maximum of 32 HEPs, the HWOI can be efficiently optimized if fewer HEPs are needed. Finally, we note that the proposed HWOI does not support logging multiple occurrences of a single HEP. Although this currently limits the frequency at which individual events can be monitored, future work specially focuses on providing low overhead methods to capture such occurrences for specific designer-specified event probes.

3.2.2 Hardware Observability Bus and Bridge

To avoid any contention over the system bus, a separate hardware observability bus (HWOBus) is utilized for communication with the HWOIs for the hardware IP cores. The HWOBus is a simple synchronous bus with unidirectional data input and output signals optimized for the small address space needed to support the HWOIs. A hardware observability bridge (HWOBridge) component is utilized to interface the HWOBus to the

processor local bus of the hardware observability engine. The HWOBridge implements the necessary address mapping and synchronization to interface between the two buses.

3.2.3 Hardware Observability Engine

The hardware observability engine (HWOEngine) is implemented as a set of user-defined software functions executing on a lightweight processor. The HWOEngine provides a set of APIs that can be utilized to hide the software interfacing details required to access the memory-mapped registers within each HWOI.

In response to the *HWOIntr* signal generated whenever any hardware event is observed, the HWOEngine first reads the *hwoi_eventflags* register within all HWOIs. For each hardware event that has been observed, a user can associate an arbitrary software function into which the user provides the required observation capabilities. Once the observability software has completed, the HWOEngine must reset the HEP by writing a logical zero to the *hwoi_eventflags* register within the HWOI. Importantly, care must be taken to only reset those HEPs that have been processed by the HWOEngine, as additional hardware events may have been observed within the HWOI during the execution of the hardware observation software. Furthermore, users are provided the option to either automatically reset the HEPs through the provided APIs or leave this control to the user-defined observation software.

This method for system observability offers several advantages. First, designers and end-users are provided great flexibility in controlling how and when to observe the various hardware components. Secondly, while the HEPs are fixed at design time, the user can update the observation software if observation needs change. Thirdly, the

hardware observability framework can be readily integrated with existing methods for software observability, such as DTrace, thereby providing a single framework for system observability.

Given the goal of providing nonintrusive framework for hardware observability, the HWOEngine is implemented as a separate lightweight microprocessor system, as shown in Figure 1. The HWOEngine includes a dedicated processor (MicroBlaze), small memory (BRAM) for the observation software, interrupt controller, and communication device (UART) for controlling the observation software and providing minimal external reporting and tracing capabilities. Thus, the observation process is completely isolated from the main system, thereby eliminating any potential performance impacts on the main system execution. Furthermore, this implementation provides a secure method for ensuring that the user-defined observation software cannot corrupt and/or harm the system execution.

3.3 Experimental Results

To evaluate the hardware observability framework, we consider a base system consisting of a 100 MHz MicroBlaze processor and three hardware IP cores connected to the PLB. The primary hardware core for this system is a 13-tap FIR filter that periodically processes data provided by the software application executing on the MicroBlaze microprocessor.

In addition, two configurable bus transaction cores, *bustran1* and *bustran2*, were developed. These cores are designed to model varying bus transfer patterns. In previous

efforts, we developed a complex streaming application utilizing the MicroBlaze processor and multiple dedicated hardware cores. During initial development efforts for this system, aperiodic delays in bus accesses and decreased throughput resulted in noticeable performance degradations. In determining the source of the execution delays, significant effort was required to directly incorporate additional logic within all hardware cores to monitor bus transactions, including bus requests, bus acknowledgments, and bus throughput. To create an analogous execution scenario, we utilize the configurable bus transaction hardware cores to generate similar—but controllable—bus access patterns.

We incorporated our hardware observability framework using the nonintrusive architecture presented in Figure 1. Three HEPs were defined for each hardware IP core. The FIR core implementation utilizes a mix of floating point inputs/outputs and fixed-point internal computations. Thus, we incorporated three HEPs within the FIR core to detect and monitor various overflow conditions. Within each of the bus transaction cores, HEPs were defined to monitor the start of a bus request, bus access granted to the hardware core, and the bus transaction completion. The corresponding observability software executed on the HWOEngine reports any overflow conditions within the FIR core and computes various bus access statistics including average bus transaction wait time and data throughput.

The two bus transaction cores were configured to periodically transfer 4000 bytes of data using a burst transaction. *bustran1* periodically performs this transfer once every 1000 clock cycles. For the *bustran2* core, we varied the transfer period from once every 2000 cycles to once every 1000 cycles. Figure 5 presents the resulting average bus

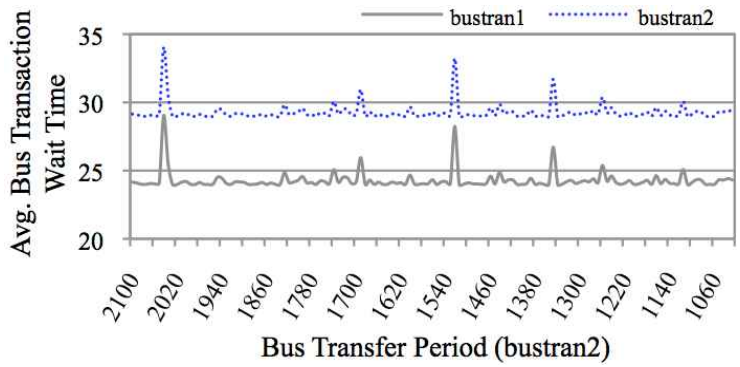

Figure 5: Average bus transaction wait time for the configurable bus transaction cores, *bustran1* and *bustran2*, measured using the hardware observability framework.

transaction wait time for both *bustran1* and *bustran2*. The analysis demonstrates that for periodic bus transactions with differing transfer rates, overlapping periods of bus transaction can result in noticeable transfer delays of up to 13%. Furthermore, the bus priority assignment for these cores can have a significant effect on performance. Even with identical transfer periods, the *bustran1* components exhibits a 17% longer average wait time compared to *bustran2*.

We note that the proposed hardware observability framework is designed to be integrated within the final system implementation, such that system designers and end-users can utilize the hardware observability framework for testing, debugging, and monitoring the execution of a system both during testing and implementation phases as well as after system deployment.

The base system was synthesized using Xilinx Platform Studio (XPS) 11.5 [80] onto a Virtex-5 (XC5VLX110T) FPGA development board. Table 2 reports the area required by the base system in lookup tables (LUTs) and flips-flops (FFs). The base

Table 2: System and hardware observability area requirements reported in lookup tables (LUTs) and flip-flops (FFs) for a Xilinx Virtex-5 FPGA (XC5VLX110T) for various components.

	LUTs	*FFs*	*LUT+FF*	*%*
Base System	10383	8751	19134	100
System with HWO	13527	11676	25203	132
HWO Eng. & Bridge	2637	2214	4851	25
HWO Bridge	354	320	674	4
HEPs (1)	89	68	157	0.8
HEPs (2)	136	104	240	1.3
HEPs (4)	205	176	381	2
HEPs (8)	273	320	593	3
HEPs (16)	478	608	1086	6
HEPs (32)	878	1184	2062	11

system executes at 100 MHz and requires 10,383 LUTs and 8,751 FFs. Table I further presents the area requirements for the complete system with hardware observability framework along with the area requirements for various components, including the HWOEngine, HWOBridge, and increasing number of HEPs ranging from 1 to 32. The area is reported both in LUTs, FFs, and percentage of combined LUTs and FFs compared to the base system. Overall, the complete system with hardware observability framework requires 13,527 LUTs and 11,676 FFs, which is roughly a 32% increase in area compared to the base system. In addition, the HWOBridge requires only 4% additional resources compared to the base system. The primary area overhead incurred by our proposed approach is due to the need for implementing a second processor for the HWOEngine, requiring over two thirds of the total area overhead along with 8.1 KB of memory to implement the observation software.

We further analyzed the power consumption overhead incurred by the proposed observability framework compared to the base system implementation. Overall, the

hardware observability framework requires 14% additional power, of which the HWOEngine requires 92%.

However, we note that our base system is fairly simple compared to most SOCs—only consisting of one microprocessor and three IP cores. As such, the area overhead of the proposed approach is expected to be significantly smaller for larger systems more represented of modern SOC designs.

Given increasing size and complexity of SOC designs, the scalability of incorporating additional hardware event probes is important. As the number of HEPs increases from 1 to 32, the area requirements increase from 157 total LUTs and FFs to 2062 total LUTs and FFs. The area for each HEP is primarily attributed to the HEP's timestamp register and corresponding memory-mapped interface. As the logic required to implement the memory-mapped interface does not increase linearly, the area required for each HEP scales well. For a HWOI with 32 HEPs, only 64 LUTs and FFs are required for each HEP. If an HEP is configured to capture data in the HEP's data registers, an additional 48 LUTs and FFs are needed on average.

CHAPTER 4

EVENT-DRIVEN FRAMEWORK FOR CONFIGURABLE RUNTIME SYSTEM OBSERVABILITY FOR SOC DESIGNS

4.1 Overview

The deep integration of software and hardware components within complex system-on-chip (SOC) designs prevents the use of traditional analysis and debug methods to observe the internal state of these components. This situation is further exacerbated for in-situ debugging, verification, and certification efforts in which physical access to traditional debug and trace interfaces is unavailable, infeasible, or cost prohibitive.

We previously presented an initial framework for monitoring hardware events by inserting additional hardware for detecting specific events within individual hardware cores. However, rather than report the event occurrences to a trace port or log them within a trace buffer, an isolated processor is incorporated on-chip to observe, process, and analyze the occurrences of events in real-time. In this chapter, we present an event-driven system-level observation framework (SOF) [45] providing robust methods for observing complex interactions across hardware and software boundaries at runtime. The SOF provides a configurable framework for monitoring and analyzing designer-specified system events with minimal—or no—effect on the system execution. In contrast with existing debug and trace capabilities, the observation framework provides designers with always on runtime monitoring of deeply embedded system events without requiring additional test equipment or offline analysis. Furthermore, designers can utilize the

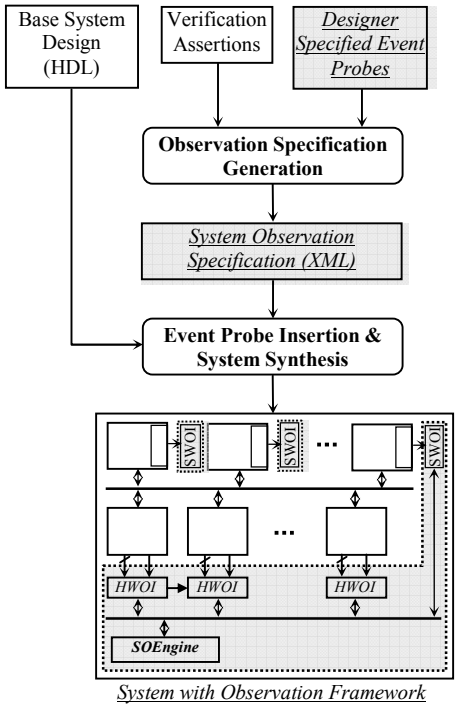

Figure 6: Overview of system observability integration methodology utilizing pre-silicon verification specifications to automatically create a post-silicon, in-situ observation framework.

observation framework for testing, verification, debugging, and system performance analysis tasks with minimal effort.

4.2 System-level Observation Framework

Figure 6 presents an overview of our proposed system observability integration methodology utilizing pre-silicon verification specifications to automatically create a runtime in-situ observation framework. Starting with the base SOC design and verification/testing requirements utilized by designers and test engineers—e.g. system properties utilized within assertion-based testing—a system observation specification can be determined that defines a set of *events* that should be monitored to achieve the

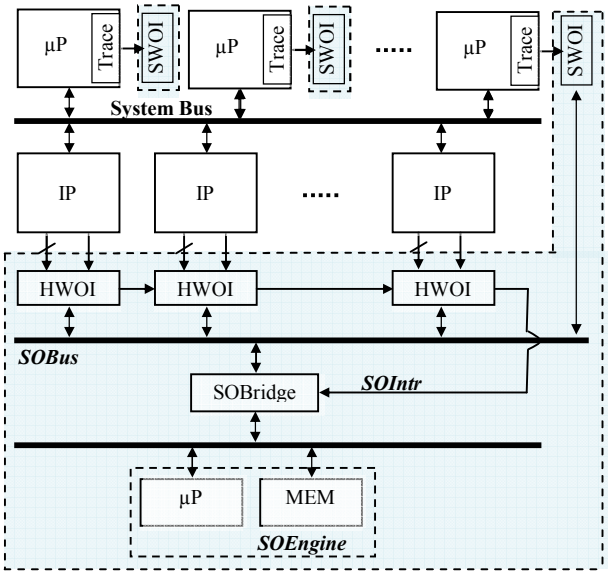

Figure 7: Nonintrusive system-level observation framework consisting of several software observation interfaces (SWOIs), hardware observation interfaces (HWOIs), a dedicated system-level observation bus (SOBus), a system-level observation bridge (SOBridge), and a system-level observation engine (SOEngine) to execute the observation software.

required level of in-situ observability within the final system. In addition to utilization verification requirements to determine these events, a designer can also define a set of events for individual hardware and software components within the SOC that should be monitored. This system observation specification can then be utilized to automatically integrate the circuitry required for each event and connect the resulting event probes within the SOF.

Figure 7 presents an overview of the proposed system-level observation framework integrated within a multiprocessor SOC design. We utilize a MicroBlaze processor based system as an illustrative example throughout this paper to closely match our experimental results in which a Xilinx FPGA is used to prototype our target system

design with the SOF. The SOF consists of a software observation interface (SWOI) connected to the trace port of each processor and a hardware observation interface (HWOI) connected to each hardware IP core to be observed. To avoid perturbing or affecting the execution of the main system, the SOF utilizes a dedicated system observation bus (SOBus) and an auxiliary lightweight processor for the system observation engine (SOEngine) that executes the runtime observation software. Depending on the processor implementing the SOEngine, a bus bridge may be needed to interface between the SOBus and the SOEngine's bus.

In contrast to existing methods for debugging, testing, and tracing of hardware designs, the SOF provides a nonintrusive framework for monitoring complex designer-specified hardware and software events. The software and hardware observation interface—described in the following subsections—provide a generic structure for monitoring system events that will be integrated and synthesized within the final system design. Although the system observation framework provides methods for controlling how to monitor the system at runtime, a designer must specify which events need to be monitored within the various processors and hardware cores of the SOC design. In response to system events, designers can create customized observation software to analyze and monitor the events at runtime for the specific testing, debugging, or monitoring tasks.

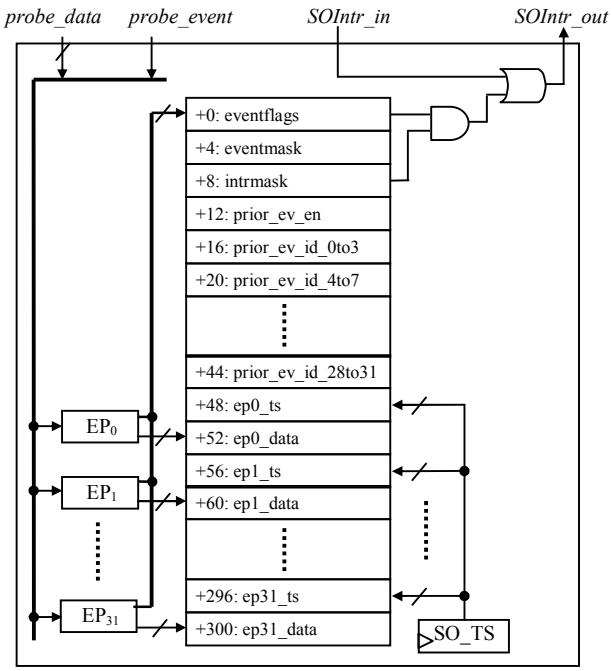

Figure 8: Generalized structure of HWOI and SWOI consisting of up to 32 event probes (EPs), timestamp counter (SO_TS), and memory-mapped interface for managing control registers and accessing the observation data in each EP.

4.3 Hardware Observability

Figure 8 describes the HWOI that provides a generic framework for dynamically monitoring designer-specified events within hardware IP cores without affecting the hardware core's execution. The HWOI consists of one or more event probes (EPs), a timestamp counter (SO_TS), and a memory-mapped interface for managing control registers and accessing the observation data in individual event probes. The EP is the basic element for monitoring events within hardware cores. As a generic framework, the HWOI supports up to 32 EPs, where each EP is associated with a 1-bit event flag, a 1-bit event mask, a 1-bit interrupt mask, a 32-bit timestamp register, and a 32-bit data register.

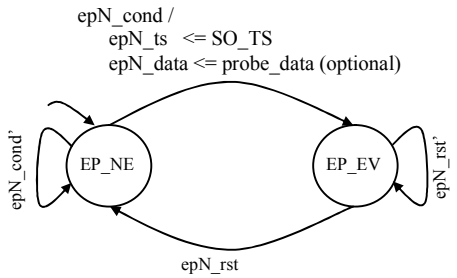

Figure 9: Event probe (EP) controller.

For each hardware core, the hardware events that need to be monitored at runtime are dependent on each specific hardware design. Hence, a designer must specify both the set of events to be observed and the set of probe signals from the IP cores that are needed to make these observations. While this approach requires additional effort on behalf of the designer, the designer is the best source of knowledge for determining which event probes are needed to provide the requisite level of observability for that hardware circuit.

Each EP contains an EP controller implemented as a simple state machine consisting of two states, *EP_NE* and *EP_EV*, as shown in Figure 9. Initially, the EP controller waits in the *no event* state, *EP_NE*, until the desired event is observed, defined by the logical expression *epN_cond*. When the hardware event is observed, the EP controller will capture the current SO_TS value and probe data into the EP's timestamp, *epN_ts*, and data, *epN_data*, registers, respectively. The controller will then transition to the event state, *EP_EV*. In the *EP_EV* state, the EP controller waits until a reset signal, *epN_rst*, is asserted for the event probe. The SOEngine can reset the event probe by writing a logical zero into the corresponding bit of the *eventflags* register, thereby returning the EP controller to the *EP_NE* state. This behavior can be described as a

blocking event probe, such that once an event is observed, the EP will *block* until the probe is reset.

Minimally, to define an EP, a designer must specify a Boolean condition corresponding to the target event and optionally define which data value should be captured on occurrence of this event. This specification can be directly incorporated within the state machine for the EP controller.

The events flags, event masks, and interrupt masks for all hardware event probes are accessible through three memory-mapped registers, *eventflags*, *eventmask*, and *intrmask*. Reading the *eventflags* register returns the current state for all EPs, where a logical one indicates the hardware event has been observed, and a logical zero otherwise. Individual EPs can be masked—or disabled—by configuring the *eventmask* register. When disabled, the EP controller will remain within the *no event* state, *EP_NE*.

The HWOI reports an occurrence of any unmasked event by asserting a system observation interrupt signal, *SOIntr*. The *SOIntr* is daisy-chained through all HWOIs and SWOIs within the system, providing a single interrupt to the SOEngine indicating that one or more events have been observed and need to be analyzed by the observation software. Within the HWOI, the interrupt for each EP can be individually disabled by setting the appropriate bit within the *intrmask* register. Whereas the *eventmask* register provides control over which events should be monitored, the *intrmask* register provides control which events generate an observation interrupt.

The HWOI includes a timestamp counter (SO_TS) that provides a simple mechanism for analyzing the timing behavior between individual events. The SO_TS is a

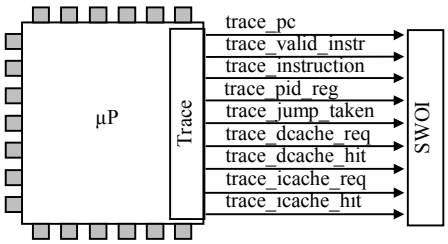

Figure 10: Processor trace interface signals used in SWOI.

32-bit counter that is incremented each observation cycle. For example, considering a 100 MHz clock, the SO_TS counter provides the ability to measure the timing of event occurrences up to 40 seconds apart, beyond which a user would be required to develop additional timing capabilities within the observation software. For systems with significantly higher clock frequencies or longer observation periods, the size of the SO_TS can be increased appropriately.

4.4 Software Observability

To observe the behavior of software executing on microprocessors within an SOC, the SOF leverages information provided by the processor's trace ports. Figure 10 demonstrates the interface between the trace signals of a processor and the SWOI. Importantly, the possible events that can be observed within the SWOI are limited to the information that is provided by the processor's trace port. Table 3 provides a description of the trace interface signals [82] currently utilized by the SWOI interfacing to a MicroBlaze processor. For MicroBlaze processors, *trace_pc* and *trace_instruction* are 32-bit signals, *trace_pid_reg* is an 8-bit signal, and *trace_valid_instr*, *trace_jump_taken*,

Table 3: Description of processor trace interface signals.

Trace Interface	Signal Description
trace_pc	Program counter of current instruction
trace_valid_instr	Indicates a valid instruction is being executed
trace_instruction	Instruction being executed
trace_pid_reg	Current task's process identifier (PID)
trace_jump_taken	Branch or jump taken
trace_dcache_req	Data cache request
trace_dcache_hit	Data cache hit
trace_icache_req	Instruction cache request
trace_icache_hit	Instruction cache hit

trace_dcache_req, *trace_dcache_hit*, *trace_icache_req*, and *trace_icache_hit* are 1-bit signals.

The SWOI utilizes a structure, semantics, and memory-mapped interface similar to the HWOI. As the signals provided by the trace interface are determined by the processor manufacture, the designer does not need to define the events to be monitored from scratch. Instead, a set of predefined configurable software events is provided. The system designer can select which software event probes and how many instances of those software event probes to incorporate within the SWOI. The following provides an overview of the software event probes.

Program Counter (PC) Event Probe: a software EP that detects the occurrence of a configurable program counter value. For each program counter EP, an additional memory-mapped register is included to store the value of the PC being monitored. This allows the PC value of the EP to be configured by the system observation software. Again, as it may be necessary to monitor more than one PC value, a designer can specify how many PC EPs to incorporate within the SWOI.

Instruction Opcode Event Probe: a software EP that detects the occurrence of a configurable instruction opcode. For each instruction opcode EP, an additional memory-

mapped register is included to store the value of the opcode being monitored. This software EP provides the means by which the system observation software could perform detailed profiling of the execution behavior of specific instruction types. As with the PC EP, a designer can specify how many instruction opcode EPs to incorporate within the SWOI.

Branch Taken Event Probe: a software EP that detects the occurrence of a jump or branch instruction being taken. In other words, this EP will be triggered when a branch or jump instruction occurs and the next PC value is not the next sequential program counter value. For the MicroBlaze processor, this event is a direct connection to the trace interface's *trace_jump_taken* signal. As this EP is independent of any specific instruction, one instance of this probe is required within the SWOI.

Context Switch Event Probe: a software EP that detects the occurrence of a context switch. To observe the occurrence of a context switch. To observe the occurrence of a context switch, the SWOI internally stores and monitors the PID of the current tasks being executed. When utilizing Xilinx's xilkernel [69][83], the current PID is provided by the trace interface signal *trace_pid_reg*. If the current PID differs from the previously stored PID, the context switch EP will be triggered.

Instruction/Data Cache Hit Probe: a software EP that detects instruction or data cache hits. For the MicroBlaze processor, this event is a direct connection to the trace interface's *trace_icache_hit* or *trace_dcache_hit* signal.

Instruction/Data Cache Miss Probe: a software EP that detects instruction or data cache misses. To detect the occurrence of a cache miss, the SWOI monitors both the

cache request trace signal—e.g. *trace_icache_req*—and cache hit trace signal—e.g. *trace_icache_hit*.

When a software event is observed, the EP controller will capture the current SO_TS value into the EP's timestamp register, *epN_ts*. In addition, a configurable data value can also be stored within the EP's probe data register, *epN_data*. The EP's probe data register can be configured at runtime to store one of the following values: current program counter, current timestamp, current instruction word, current PID value, number of instruction/data cache hits, or number of instruction/data cache misses.

4.5 System Observation Engine

The system observation engine (SOEngine) is implemented as user-defined software executing on a lightweight auxiliary processor. The SOEngine provides a set of APIs that can be utilized to hide the software interfacing details required to access the memory-mapped registers within the HWOIs and SWOIs. The SOEngine provides several advantages compared to existing trace and debug methods. Designers are provided great flexibility in controlling how and when to observe the various system observation events. Additionally, while the software and hardware EPs are fixed at design time, a designer can configure both how those EPs are monitored and what observation behavior is executed in response to those events.

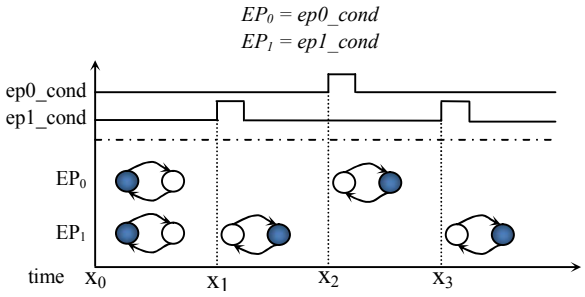

Figure 11: Example system observation event behavior for two events EP_0 and EP_1 without event cascading enabled.

4.6 Cascading Event Probe

Using the SOF, a designer can implement many different analysis methods in response to the observed system events. A common analysis for many designs is measuring the latency between two events. To monitor the latency between two events, two events probes can be created to observe the occurrence of each event. When these events occur, the system observation software can read the timestamp for both events, compute the latency, and reset the EPs to observe the next occurrence. However, this behavior may lead to incorrect latency calculation as the time at which the EPs are reset can influence the correctness of the calculation. Figure 11 presents an example behavior for two event probes, *EP0* and *EP1*. At time x_0, a reset signal is asserted for *EP0* and *EP1*, causing the EPs to return to the *EV_NE* state. When the condition *ep1_cond* for the *EP1* is observe at the time x_1, *EP1* will transition to the *EP_EV* state, assert the corresponding event flag, and capture the current timestamp. When the condition *ep0_cond* for the *EP0* is at time x_2, *EP0* will transition to the *EP_EV* state, assert the corresponding event flag, and capture the current timestamp. Calculating the latency between the occurrence of *EP0* and *EP1* is impossible, as the captured occurrences do not represent the correct temporal relation.

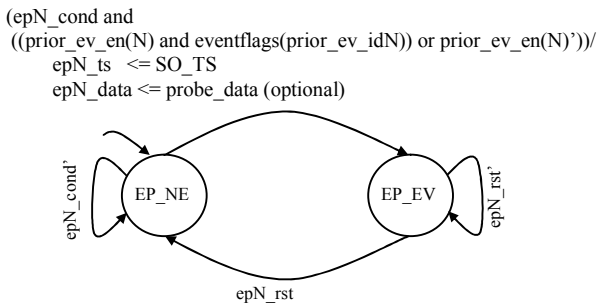

Figure 12: Cascading event probe (CEP) controller.

To support correct and predictable latency measurements within the SOF—and any observation requiring a cause and effect relationship—we further support a cascading event probe (CEP) presented in Figure 12. The CEP allows an EP to be both dependent on the event's condition and the occurrence of a prior event. For each CEP, a *prior_ev_en* signal is utilized to configure the EP that is dependent on a prior event. If enabled, a *prior_ev_id* signal is utilized to specify the prior event. Note that the prior events are currently constrained to the same HWOI or SWOI.

Figure 13 presents the behavior for two event probes, EP_0 and EP_1, in which EP_1 is configured as a CEP with an EP_0 as the prior event, using the same reset timing of Figure 6. Using the CEP, at time x_1, although *ep1_cond* is asserted, EP_1 will remain in the *EP_NE* as the prior event EP_0 has not yet occurred. At time x_2, the condition *ep0_cond* for the EP_0 is observed. EP_0 will transition to the *EP_EV* state, assert the corresponding event flag, and capture the current timestamp. Subsequently at time x_3, the condition *ep1_cond* and the prior occurrence of EP_0 will be observed, and EP_1 will transition to the *EP_EV* state, assert the corresponding event flag, and capture the current timestamp. For

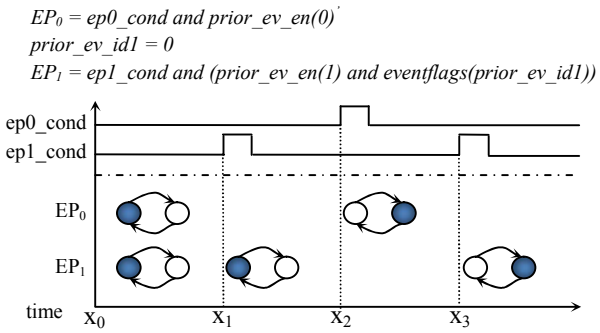

Figure 13: Example system observation event behavior for cascading events EP_0 and EP_1 in which EP_1 is dependent on EP_0 having previously occurred.

all occurrences of the sequence of events EP_0 and EP_1, the correct latency can be calculated within the system observation software.

4.7 Experimental Results

To evaluate the system-level observation framework, we consider an FPGA-based prototype of a SOC design consisting of a 100 MHz MicroBlaze processor and several hardware IP cores, presented in Figure 14. In addition to hardware cores implementing basic system functionality—e.g. timers, interrupt controllers, memory controllers, UARTS—the system design includes three additional cores accelerating specific operations: 1) a 13-tap FIR filter [20]; 2) a sobel edge detection (SED) [23] processing 640x480 grayscale images; and 3) a TFT controller [84] for displaying the resulting images using a DVI display output. Finally, the system software executing on the system processor is a multitasked software application implemented using Xilinx's xilkernel 4.00.a. The system was synthesized using Xilinx Platform Studio (XPS) 11.5 targeting a Virtex-5 FPGA (XC5VLX110T).

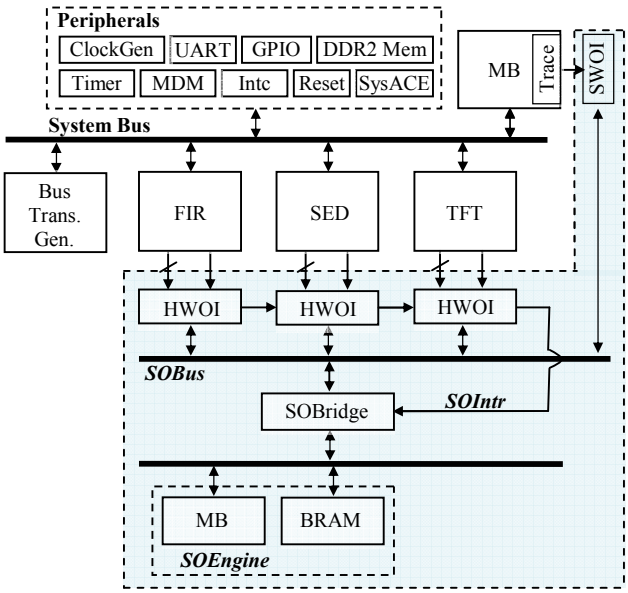

Figure 14: Overview of complete system design including three HWOIs and one SWOI.

We incorporated the SOF within our system design to verify the correct system execution, considering several system latency requirements summarized in Table 4. From these requirements, we determine the event probes that must be monitored within the hardware cores and software executing on the MicroBlaze processor. In total, 21 event probes were required, of which 12 were configured as cascading event probes. Finally, we implemented the required system observation software to configure and analyze

Table 4: System latency requirements for target system.

Req #	Description	Requirement
R1	FIR total latency	12.93 us
R2	FIR HW execution latency	1.79 us
R3	FIR HW initialization latency	7.32 us
R4	FIR synchronization latency	4.90 us
R5	SED total latency	86096 us
R6	SED HW execution latency	85153 us
R7	SED HW initialization latency	0.62 us
R8	SED synchronization latency	978 us
R9	SED SW execution latency (640x480)	6227 ms
R10	TFT frame rate (640x480)	16667 us

Table 5: Area requirements for base system and SOF reported in lookup tables (LUTs), flip-flops (FFs), and BRAMs.

	LUTs	FFs	BRAMs
Base System	11814	10963	19
System with SOF	17361	16258	23
HWOI for FIR	247	266	--
HWOI for SED	152	205	--
HWOI for TFT Controller	510	595	--
SWOI for MicroBlaze	1804	1126	--

events to detect and report runtime violations of the system requirements.

Table 5 reports the area required for the various HWOI, SWOI, and complete system with the SOF in lookup tables (LUTs), flip-flops (FFs), and block rams (BRAMs). The base system requires 11,814 LUTs, 10,963 FFs, and 19 BRAMs. Table IV further presents the area requirements for the complete system with the SOF, the HWOIs for three additional cores, and the SWOI for the main MicroBlaze processor. Overall, the complete system requires 17,361 LUTs, 16,258 FFs, and 23 BRAMs, which is roughly a 47% increase in area compared to the base system. We note that over two thirds of the area overhead is required by the SOEngine that is implemented as a secondary MicroBlaze processor with 17.7 KB of memory to implement the observation software. The area required for the HWOI and SWOI components is only 2,713 LUTs and 2,192 FFs, which corresponds to a 22% increase in area compared to the base system. Additionally, the SWOI requires significantly more area than the HWOIs. Whereas the HWOI is customized for the designer specified events, the SWOI supports all possible software event probe types. The area required for the SWOI can be substantially reduced by customizing the set of supported event probes.

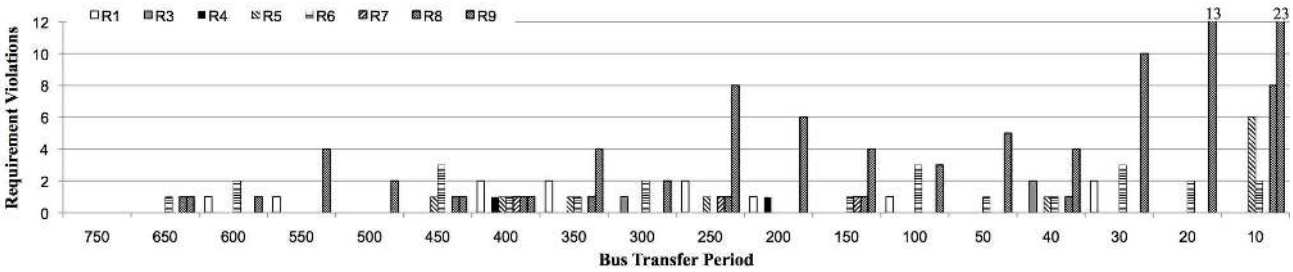

Figure 15: Number of requirement violations within 30 minutes of system execution for decreasing idle periods between bus transactions.

For the base system implementation, all requirements were dynamically verified using the SOF. To further evaluate the SOF, a configurable bus transaction generation core was incorporated within the system to allow for the generation of periodic data transfers on the system bus. We configured the bus transaction core by varying the idle period between transactions from once every 1000 clock cycles to once every 10 clock cycles. For each configuration, we executed the system for 30 minutes and logged the recorded number of requirement violations, reported in Figure 15. Requirements R2 and R10 are met for all considered idle periods, and for idle periods greater than 750 clock cycles all requirements are met. Although a gradual increase in total violations can be observed for decreasing idle periods, the effect on specific requirements is inconsistent. For example, while the SED's software execution latency requirement (R9) is violated most frequently and across all idle period shorter than 750, the number of violations within a 30 minute period does not monotonically increase. In addition, for several of those experiments, while the SW execution latency is violated, the total SED execution latency requirement (R8) is met. The SOF framework provides a robust method to

evaluate and analyze this behavior, which would be difficult or time consuming to evaluate using other testing and debugging methods.

CHAPTER 5

SYSTEM OBSERVATION OF BLOCKING, NON-BLOCKING, AND CASCADING EVENTS FOR RUNTIME MONITORING OF REAL-TIME SYSTEMS

5.1 Overview

The complexity of multitasked applications in real-time embedded systems presents key challenges in the reliability of task execution. Interactions between periodic and aperiodic tasks can incur unpredictable deviations from ideal execution times. Runtime observations can provide visibility for analyzing real-time execution behavior of vulnerable tasks and detect when such deviations may lead to system failure—potentially allowing correction or failsafe mechanisms to be utilized. However, such runtime observation must be non-intrusive, as even small perturbations to the system execution can significantly impact the system.

We previously presented a framework for monitoring hardware events by inserting additional logic for detecting designer-specified events within hardware cores to observe complex interaction across hardware and software boundaries at runtime. This observation framework utilized a memory-mapped interface to store and configure individual events using a dedicated bus. The memory-mapped interface imposed constraints on how rapidly events could be observed and analyzed within the proposed system. Specifically, all events adhered to a *blocking* semantic in which once an event was detected the event would wait until the designer-provided observation software reads

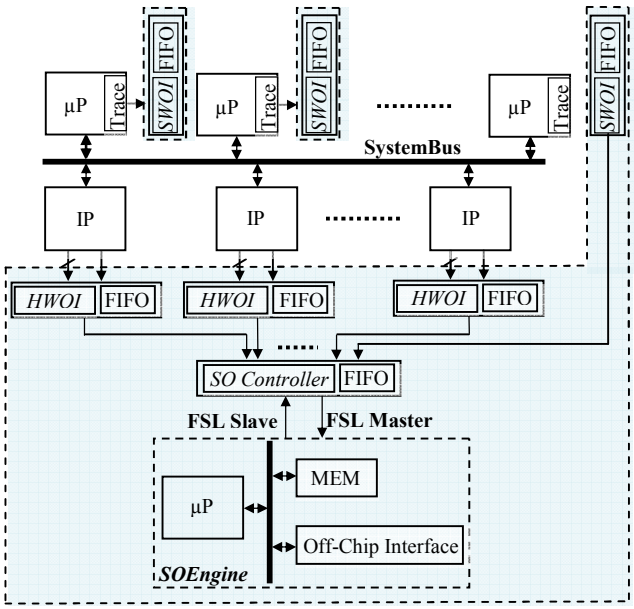

Figure 16: System observation framework (SOF) consisting of several software observation interfaces (SWOIs), hardware observation interfaces (HWOIs), a system observation controller (SOController), and a system observation engine (SOEngine) executing the observation software.

and resets the event before the same event could be detected again. This limits how rapidly events could be observed and creates overhead within the observation software.

In this chapter, we present a system observation framework [46] for monitoring and reporting rapidly occurring events using a pipelined, priority-based event streaming interface. The SOF provides in-situ support for configuring and controlling event probes within software and hardware components using *blocking*, *non-blocking*, and *cascading* event probes. We demonstrate the use of the SOF for nonintrusive analysis of real-time software systems by analyzing the task completion time and scheduling jitter for an example system.

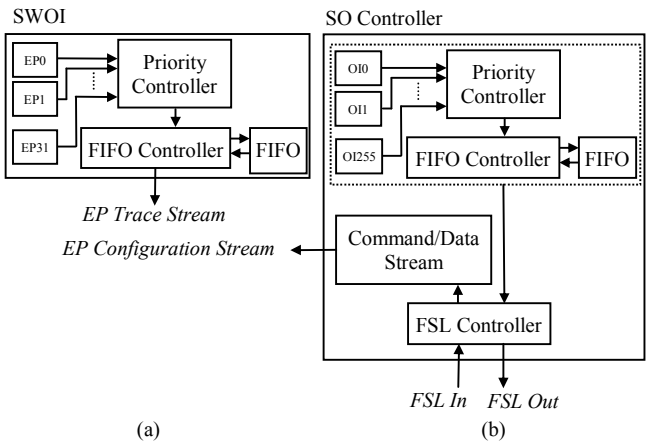

Figure 17: Overview of priority-based event streaming hierarchy and configuration stream interfaces for SOF.

5.2 System-level Observation Framework

Figure 16 presents an overview of the SOF integrated within a multiprocessor system-on-a-chip (SOC) design. We utilize a MicroBlaze-based system throughout this article to closely match our experimental results that use a Xilinx FPGA to prototype our target system design and SOF. The SOF consists of a software observation interface (SWOI) connected to the trace port of each processor core and hardware observation interfaces (HWOI) connected to each hardware IP core to be observed. The SOF utilizes a pipelined, priority-based event streaming interface to report monitored event data occurring monitored events, as shown in Figure 17(a). Within each SWOI and HWOI, a pipelined, priority-based event stream controller (PESC) serializes and stores observed events within a small FIFO. The system observation controller (SOController) serializes and stores monitored events across multiple HWOIs and SWOIs using the same pipelined, priority-based control mechanism. The observed events are finally reported to the system observation engine (SOEngine) using a Fast Simplex Link (FSL) [81][85]. The SOEngine

is an auxiliary processor that executes observation software for analyzing the event probe stream. Importantly, the software executing within the SOEngine is defined by the user to implement the analysis, testing, verification, or debugging needs of the system.

5.3 HWOI and SWOI Interface

Each HWOI or SWOI consists of one or more events probes (EPs), a timestamp counter, a configuration register of each EP, a PESC, a small FIFO, and a FIFO controller for streaming events to the SOController. Both HWOI and SWOI are synthesized on the SOC design. The EP is the basic element for monitoring events within software executions. The HWOI and SWOI interfaces support up to 32 EPs, where each EP is associated with a 10-bit configuration register that contains bits for configuring the cascading, blocking, data capture, timestamp capture, and event mask along a 5-bit prior event ID for configuring the cascading event. The timestamp counter provides a simple mechanism for analyzing the relative timing behavior among observed events. The timestamp counter is a 32-bit counter that is incremented each observation cycle.

5.4 Blocking, Non-blocking, and Cascading Event Probes

Each EP contains an EP controller implemented as a state machine consisting of three states, *EP_NE*, *EP_EV*, and *EP_BL*, as shown in Figure 18. Initially, the EP controller waits in the *no event* state, *EP_NE*, until the desired event is observed, defined by the logical expression *epN_cond*. When the desired event is observed, the EP controller will capture the current timestamp value and probe data, and then generate an event probe

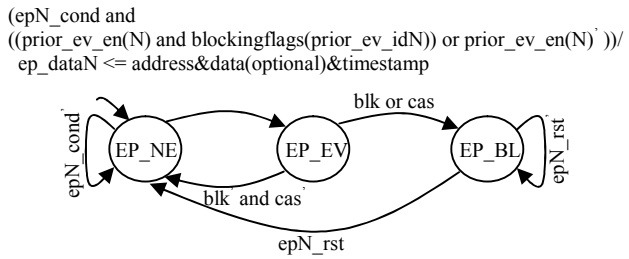

Figure 18: Event probe (EP) controller for blocking, non-blocking, and cascading event probe.

data, *ep_dataN*, consisting of an event probe address, data, and timestamp. The EP controller will then transition to the *event* state, *EP_EV*. The subsequent behavior of the EP depends on the configuration of the EP. The EP can be configured as a blocking event probe, a non-blocking event probe, or a cascading event probe.

A blocking probe will not detect another occurrence of an event until the observation software resets the probe. Thus, the EP controller will transition to the *blocking* state, *EP_BL*, and wait until a reset signal, *epN_rst*, is asserted for the event probe. In contrast, a non-blocking probe will continue to detect and report all occurrences of the event without requiring a reset from the observation software. For non-blocking—and non-cascading—probes, the EP controller will immediately transition back to the *EP_NE*.

Cascading event probes allow an EP to be dependent on both the event's condition and the occurrence of a prior event. When configured as a cascading probe, a *prior_ev_en* signal is utilized to configure the EP that is dependent on a prior event and a *prior_ev_id* signal is utilized to specify the prior event. Note that the prior event can be configured as a cascading probe itself, thereby supporting a sequence of cascading probes to detect and

analyze sequences of multiple events. A common analysis for many design is measuring the latency between two events. To monitor the latency between two events, two events probes can be created to observe the occurrence of each event. When these events occur, the system observation can read the timestamp for both events, compute the latency, and reset the EPs to observe the next occurrence.

Cascading probes can be configured as blocking and non-blocking. A cascading, non-blocking EP will continue to detect and report occurrences of the cascading event without requiring an explicit reset from the observation software. As a cascading probe is dependent on one or more prior events, all prior events should be reset in order to detect the same sequence of events. Thus, the EP controller will first transition to the *EP_BL* state, where it waits for a reset signal. Within the HWOI and SWOI interface, a cascading probe reset logic component detects the occurrence of the last non-blocking probe within the cascading event chain, and simultaneously reset all EPs—returning all EPs within the chain to the *EP_NE* state.

A cascading, blocking probe will remain in the *EP_BL* state until explicitly reset by the observation software. As with the non-blocking probe, the cascading probe reset logic will reset all probes within the cascading chain when this reset is received.

5.5 Software Event Probes

To monitor events within the execution of software, the SWOI interfaces with the trace ports of the processor core. A set of predefined configurable software events is supported

within each SWOI. A system designer can select which software event probes and how many instances of those software event probes to incorporate within the SWOI at design time. These probes can then be configured and controlled at runtime using the SOF.

Program Counter (PC) Event Probe: a software EP that detects the occurrence of a configurable program counter value. For each program counter EP, an additional register is included to store the value of the PC being monitored. This allows the PC value of the EP to be configured by the system observation software.

Instruction Opcode Event Probe: a software EP that detects the occurrence of a configurable instruction opcode. For each instruction opcode EP, an additional register is included to store the value of the opcode being monitored. This software EP provides the means by which the system observation software could perform detailed profiling of the execution behavior of specific instruction types.

Branch Taken Event Probe: a software EP that detects the occurrence of a jump or branch instruction being taken. In other words, this EP will be triggered when a branch or jump instruction occurs and the next PC value is not the next sequential program counter value.

Context Switch Event Probe: a software EP that detects the occurrence of a context switch. To observe the occurrence of a context switch, the SWOI internally stores and monitors the PID of the current tasks being executed.

Instruction/Data Cache Hit Probe: a software EP that detects instruction or data cache hits.

Instruction/Data Cache Miss Probe: a software EP that detects instruction or data cache misses.

5.6 Event Probe Configuration Stream

All EPs can be configured at runtime using software APIs implemented within the SOEngine. These APIs allow a user a user to configure and control the EPs. All configuration and control commands are transmitted through the FSL link of the SOEngine, as shown in Figure 17(b). All configuration and control commands consist of an initial configuration word:

<*R*, *CPC*, *CAS*, *BL*, *EM*, *DM*, *TM*, *PRID*, *EPID*, *OID*>

where *R* is a one-bit reset flag, CPC is a one-bit custom probe configuration flag, *CAS* is a one-bit cascading event probe configuration flag, *BL* is a one-bit blocking probe configuration flag, *EM* is a one-bit event mask, *DM* is a one-bit data mask, *TM* is a one-bit timestamp mask, *PRID* is a 5-bit prior event ID, *EPID* is a 5-bit event probe ID, and *OID* is an 8-bit observation interface ID. Each SWOI and HWOI are assigned an 8-bit observation interface ID to uniquely identify each interface, and within those observation interfaces, each EP is assigned a 5-bit event probe ID. This allows all EPs to be uniquely identified within the observation software using 13-bits.

If the reset flag *R* is set, the blocking event probe specified by the *EPID* and *OID* will be reset. For cascading probes, all EPs within a sequence of cascading event probes will be reset simultaneously. This ensures correct observance of the cascading sequence

of events being detected. For instance, consider cascading event probes using EP_0, EP_1, and EP_2. If EP_0, EP_1, and EP_2 are reset sequentially in order across multiple cycles EP_0 could be observed again before EP_1 or EP_2 is reset. This could then result in the incorrect observation—e.g. latency measurement—of the cascading probe sequence.

The custom probe configuration, *CPC*, is utilized to configure probe specific configuration data. For example, a program counter event probe requires configuration data to specify the address of the instruction being monitored. If the *CPC* bit is set within the configuration command, an additional configuration word containing the probe specific configuration data will be transmitted. For example, to configure a non-blocking program counter event probe EP_3 within software observation interface OI_2 to detect the occurrence of the address 0x44001264, the following command would be utilized:

<0, 1, 0, 0, 0, 0, 0, 0, 3, 2>

<0x44001264>

The event mask, *EM*, is used to enable and disable individual event probes. When the event mask bit is set, the detection of the corresponding probe will be disabled. The data mask, *DM*, and timestamp mask, *TM*, are used to control the capture of probe specific data and timestamps when reporting the event occurrences.

The blocking, *BL*, and cascading, *CAS*, configuration bits control the type of event probe. When the *CAS* bit is set, the prior event ID, *PRID*, is used to specify the source of the prior event that must be detected for the current event probe to be triggered. For

instance, to configure event probe EP_1 within the interface OI_2 as a blocking, cascading event probe with a prior event probe EP_3, the following command would be utilized:

<0, 0, 1, 1, 0, 0, 0, 3, 1, 2>

5.7 Pipelined, priority-based event stream controller

A pipelined, priority-based event stream controller (PESC) is incorporated within the SWOIs, HWOIs, and SOController to choose the order in which observed events are reported to the SOEngine. A pipelined binary tree structure of PESC components with $\log_2 N$ states—where N is the number of probes within the SWOI or HWOI—is utilized to select forward observed events from the EPs to the SOEngine.

Within the SWOIs and HWOIs, each individual PESC component compares two input events and selects the event with the highest priority to forward to the next stage during each clock cycle. When that forwarded event is read by the following pipeline stage, the PESC will again compare the two current event inputs to determine the next event to forward. When an observed event reaches the final stage of the pipelined binary tree within a specific OI, the observed event is written to a FIFO. The output of the FIFOs from individual SWOIs and HWOIs is then connected to the SOController that uses the same PESC components to control the order in which events from different observability interfaces are reported to the SOEngine. This pipelined binary tree structure of PESC components achieves an overall throughput of one observed event per clock cycle.

We note that many different priority schemes exist for reporting events. A simple scheme would utilize the event ID to control how events are ordered, in which the lower the event ID the higher the priority. This could further be extended to the SOController for selecting events between observability interfaces. While this simple scheme would require the least amount of area to implement, it is not appropriate for all systems. Such a scheme could lead to starvation in which a lower priority event is always delayed due to one or more rapidly occurring events with higher priority. A round robin priority control scheme could be utilized to overcome this challenge.

5.8 In-order priority controller

In the current SOF implementation, an in-order PESC is used that reports events in-order based on the events' timestamps. A lower timestamp indicates the event was observed earlier, and thus needs to be reported first. In the case that two events have the same timestamp, the event probe with a lower ID is given priority.

Figure 19 demonstrates the cycle-by-cycle execution behavior of the pipelined in-order priority controller for an example system consisting of four non-blocking, non-cascading EPs. The in-order PESC requires a two stage pipelined tree structure. For illustrative purposes, the EPs are setup such that they will continually trigger once reset. Thus, all four EPs are initially triggered simultaneously and will have the same timestamp at clock cycle x_0.

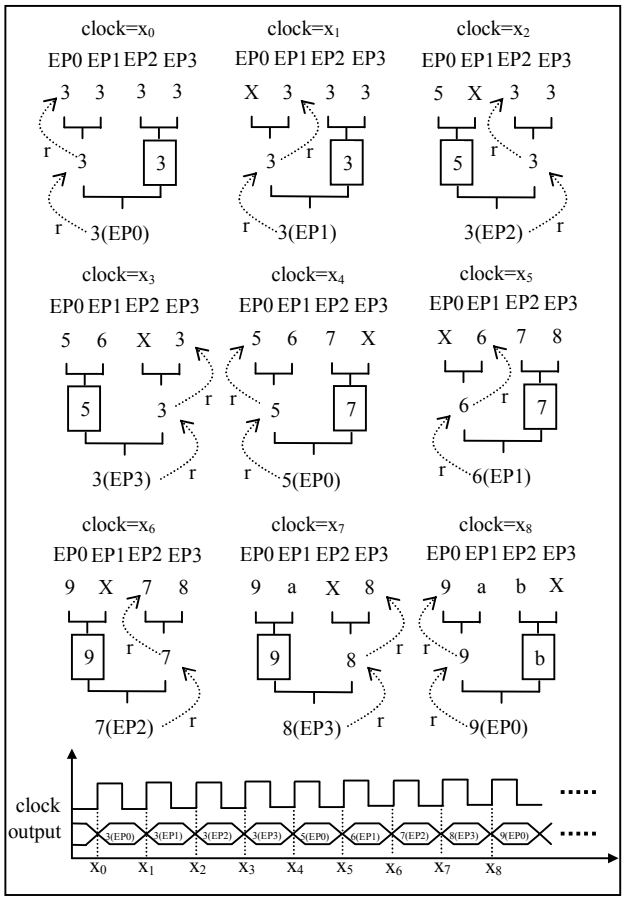

Figure 19: Operation of in-order pipelined, priority-based event stream controller highlighting the cycle by cycle operation for a system in which all EPs are triggered simultaneously.

In the first stage, the in-order PESC compares timestamps for EP_0 and EP_1 and compares timestamps for EP_2 and EP_3. As all events currently have the same timestamp, the events with the lowest IDs—EP_0 and EP_2 respectively—in each comparison will be forwarded to the next stage. In the second stage, the in-order PESC compares the events EP_0 and EP_2, outputting EP_0 as the first observed event. Whenever an EP is initially read from the EP controller or the reported event is forwarded to the next pipeline stage, a reset/read signal is asserted that allows the EP to detect another event or allows the

previous stage of the pipeline to select the next event. Note that not all pipeline stages may have a valid event at all times—indicates an X in the figure.

5.9 Experimental Results

To evaluate and demonstrate the capabilities of the SOF, we present a case study in which the SOF is utilized to nonintrusively analyze the task completion time and scheduling jitter within a real-time software system. We consider a system consisting of a 125 MHz MicroBlaze processor and basic system peripherals—e.g. timers, interrupt controllers, memory controllers, and UART. The multitasked application within this system consists of five periodically executing tasks from the SNU benchmark suite [73], specifically binary search (*bs*), FFT using Cooley-Tukey algorithm (*fft1*), integer-based forward discrete cosine transform from JPEG image encoding standard (*jfdctint*), matrix multiplication (*matmul*), and matrix inversion (*minver*). The Xilinx xilkernel 4.00a was utilized as the operating system and configured for priority-based scheduling. The scheduling priorities for individual tasks were assigned based upon their execution periods, summarized in Table 6.

To observe the completion time and scheduling jitter for all application tasks, ten configurable software event probes were implemented within the SWOI for the MicroBlaze processor—five of which were configured as cascading probes. We highlight the configuration for the cascading event probes utilized to observe the completion time of the *jfdctint* task. To measure the completion time, two EPs are needed: one EP—e.g. EP_4—to observe when the task is scheduled to execute and a second EP—e.g. EP_5—to

Table 6: Summary of periodic applications tasks based on applications within the SNU real-time benchmark suite.

Benchmarks	Periods (ms)	Priority
bs	200	4
fft1	310	5
jfdctint	140	2
matmul	190	3
minver	120	1

observe when the task has completed its execution. Both EPs will be implemented as a PC event probe within the SWOI. For the *jfdctint* task, a PC address of 0x50002A70 corresponds to the scheduling of the task's execution, and a PC address of 0x50002D3C corresponds to the end of the task's execution. Thus, EP_4 and EP_5 will be configured to observe the PC addresses 0x50002A70 and 0x50002A70, respectively.

As EP_4 monitors the scheduling of the task's execution, it is configured as a blocking and non-cascading probe. EP_5 is then configured as a cascading probe with EP_4. Notably, by configuring EP_4 as a blocking probe, and EP_5 as a cascading probe within EP_4 as the prior event, when EP_5 is reset by the observation software, EP_4 and EP_5 will be reset simultaneously. Both probes are located within observation interface OI_3. To configure these probes, the following configuration commands can be utilized:

<0, 1, 0, 1, 0, 0, 0, 0, 4, 3>

<0x50002A70>

<0, 1, 1, 1, 0, 0, 0, 4, 5, 3>

<0x50002D3C>

The system was synthesized using Xilinx Platform Studio (XPS) 11.5 targeting a Virtex-5 FPGA (XC5VLX110T). For all experiments, the system was executed for two minutes.

5.9.1 Monitoring Task Completion Time

We analyzed the task completion times for all software tasks. The completion time for a task is the time between the firing of the periodic task and its completion for that invocation. For real-time systems, the task completion time must be shorter than the task's deadline—i.e. task execution period for the tasks considered. The SOF enables a nonintrusive approach for observing the task completion time that can be used to analyze the system execution behavior, verify that tasks are executing correctly, or even predict when a task will not complete in time and take corrective action.

Figure 20 presents the task completion for all executions of the tasks, *bs*, *fft1*, *jfdctint*, *matmul*, and *minver*. The results demonstrate that significant variations in task completion times exist for several of the application tasks. However, these variations can be primarily attributed to the interference caused by higher priority task executions. Importantly, the results demonstrate that the completion for a task will vary based upon which task—or tasks—preempt the lower priority task. The various preemptions can be readily observed in Figure 20. For example, consider the *matmul* task, which can be preempted by *minver* and *jfdctint*. The observed completion times can be approximately categorized into four different levels: 1) no preemption occurs; 2) preempted by *minver* only; 3) preempted by *jfdctint* only; and 4) preempted by *minver* and *jfdctint*.

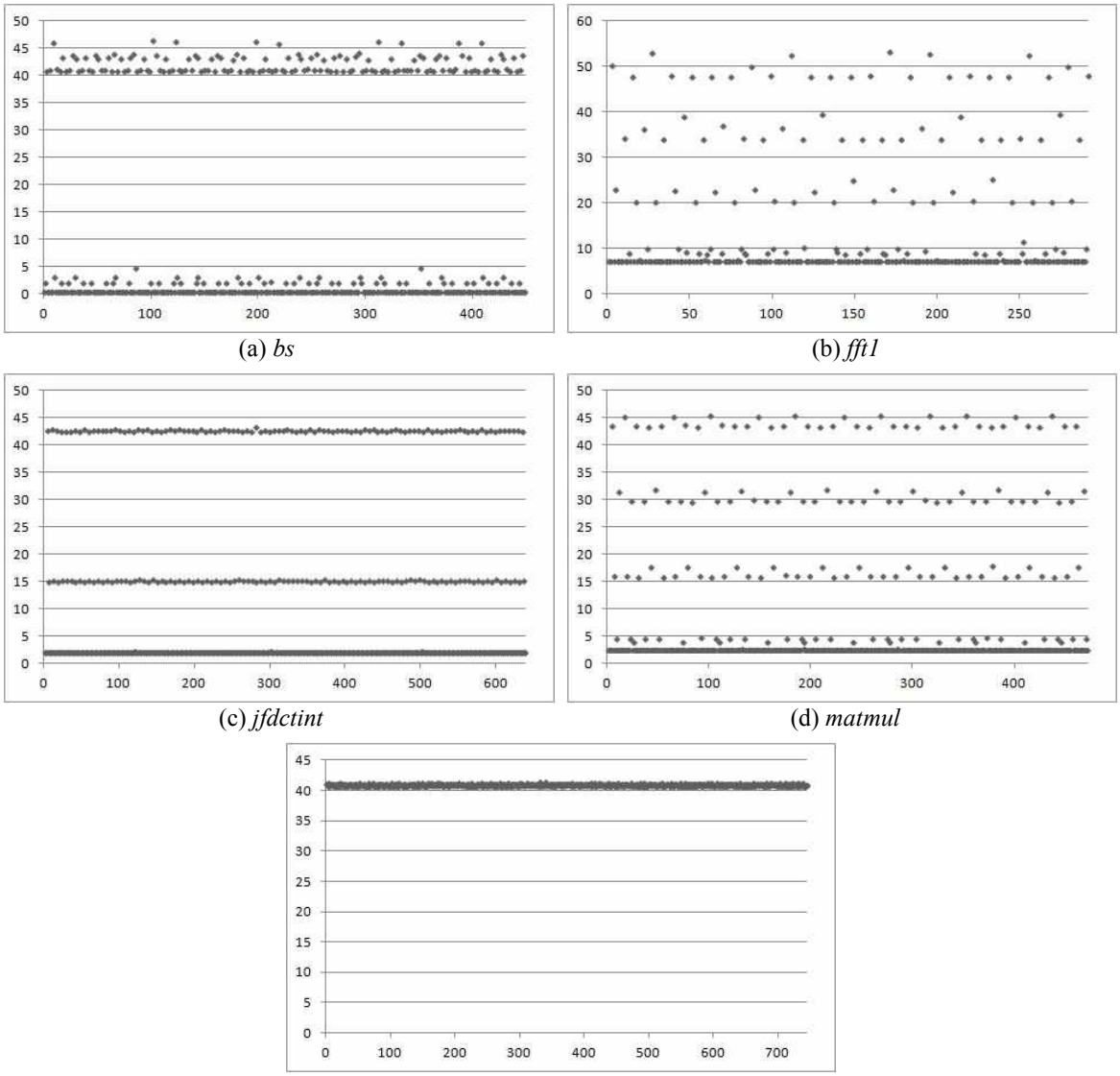

Figure 20: Task completion time of application tasks (a) bs, (b) fft1, (c) jfdctint, (d) matmul, and (e) minver. The x-axis is the number of observed events and the y-axis is time (ms) for 2 minute execution.

The resulting average completion times and deviations from those averages can be summarized as follows. The average completion time for the *bs* task is 0.74 ms. For 33.2% of those task executions, the average completion time is 42.1 ms. The average completion

time of the *fft1* task is 7.46 ms. For 8.3% of those task executions, the average completion time is 21.4 ms, for 8.3% of the task executions, the average completion time is 35.3 ms, and for 8.6% of the task executions, the average completion time is 49.0 ms. The average completion time of the *jfdctint* task is 2.04 ms. For 16.6% of those task executions, the average completion time is 15.14 ms, and for 16.6% of the task executions, the average completion time is 42.7 ms. The average completion time of the *matmul* task is 2.83 ms. For 8.3% of those task executions, the average completion time is 16.54 ms, for 8.3% of the task executions, the average completion time is 30.4 ms, and for 8.3% of the task executions, the average completion time is 44.1 ms. The completion time of the *minver* task that has the highest priority is average 41.0 ms without being affected by interference from other tasks.

5.9.2 Monitoring Task Scheduling Jitter

To further analyze the variations in task executions, we utilized the SOF to analyze the scheduling jitter of each task. The scheduling jitter is difference between the desired starting time of a task—i.e. the task's period—and the actual starting time of the task. Scheduling jitter can be measured by determining the difference between the starting times of two subsequent task executions. To measure this, two program counter EPs were configured to both detect the execution of the starting address for the task. One of these probes was configured as a cascading probe with the other probe as the priori event. This allows the SOF to precisely measure the time between that start of two subsequent task executions.

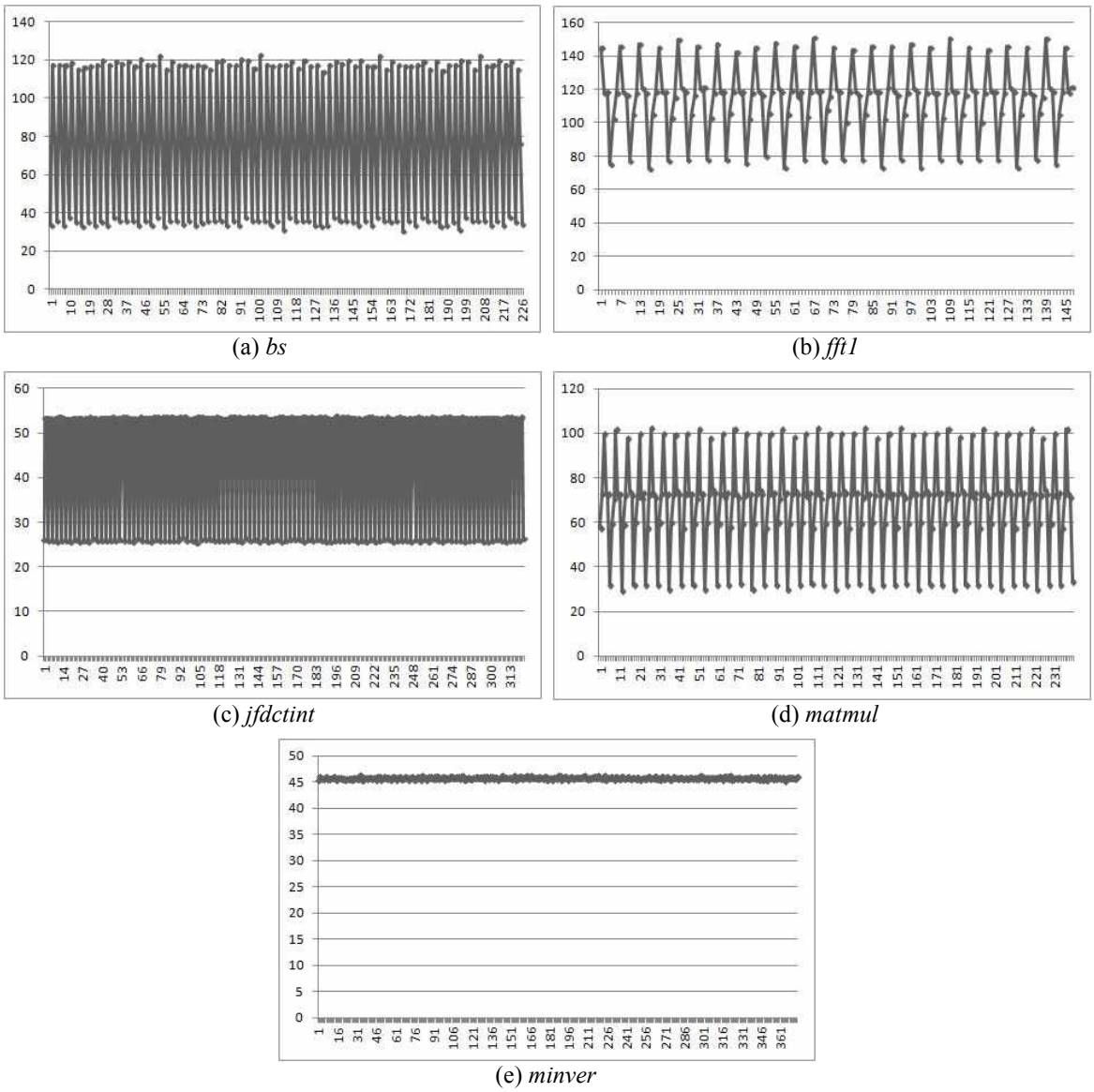

Figure 21: Scheduling jitter for application tasks (a) bs, (b) fft1, (c) jfdctint, (d) matmul, and (e) minver. The x-axis is the number of observed events and the y-axis is time (ms) for 2 minute execution.

Figure 21 presents the scheduling jitter for all executions of the tasks, *bs*, *fft1*, *jfdctint*, *matmul*, and *minver*. The variations in scheduling jitter demonstrate that when a task is scheduled to be executed, the task's actual execution will be delayed while higher priority tasks are executed. One may notice that the scheduling jitters are always greater

than 0, implying that the tasks' executions do not execute at the defined periods, but rather execute on average with a period longer than desired. These results help to identify the amount of scheduling overhead within the system implementation using Xilinx xilkernel.

The resulting average scheduling jitters and deviations from those averages can be summarized as follows. The average scheduling jitters for *bs*, *fft1*, *jfdctint*, *matmul*, and *minver* are 35.0 ms, 76.4 ms, 25.9 ms, 31.3 ms, and 45.8 ms, respectively. This corresponds to an average deviation from the tasks' period of 16% to 38%. For many systems, such overheads may not be tolerated, and knowledge of them helps to isolate potential sources of error.

5.9.3 Area, Throughput, and Latency Results

While the PESC utilized within the SWOI and SOController provides throughout of one event per cycle, the interface between the SOController and SOEngine using the FSL limits the event observation throughput of the SOF. To measure the effective maximum event observation throughput, a sample system was constructed that continually generates events from the SWOIs and HWOIs. The total number of events that could be processed by the SOEngine within a fixed time interval was measured to determine that the current SOF implementation is capable of observing and analyzing 346,136 events per second. While the PESC achieves a throughput of one event per cycle, the FSL interface and latency of the observation software is the bottleneck that limits the number of the events that can be observed and analyzed.

Table 7: Area requirements for SOF components reported in lookup tables (LUTs), flip-flops (FFs), and BRAMs.

	LUTs	FFs	BRAMs
SWOI+EPFIFO(77x128)	5271+35	1791+42	0+2
SO Controller+SOFIFO(77x256)	2253+35	1032+123	0+2
EP(2)+Priority Controller(1)	1282	767	--
EP(4)+Priority Controller(3)	2017	953	--
EP(8)+Priority Controller(7)	3578	1457	--
EP(16)+Priority Controller(15)	5099	1710	--
EP(32)+Priority Controller(31)	6674	1749	--

The latency from the occurrence of an event captured within an EP to the reporting of that event to the SOController is directly proportional to the total number of EPs enabled within the system. The worst case latency occurs when all EPs are simultaneously observed within the same clock cycle—resulting in a latency of MxN clock cycles for the EP with the lowest priority.

Table 7 reports the area required in lookup tables (LUTs), flip-flops (FFs), and block RAMs (BRAMs) for the EPs, SWOI, SOController, and FIFOs utilized within the respective components. Overall, the complete system requires 13,453 LUTs, 10,803 FFs, and 52 BRAMs. The SOEngine requires 2,000 LUTs and 1,924 FFs along with 15 KB of memory to implement the observation software. Table II also reports the area requirements for increasing the number of EPs ranging from 2 to 32 with the in-order PESC. As the number of EPs and the number of the PESC combined with EPs increase from 2 to 32 and from 1 to 31 respectively, the area requirements increase from 2,049 total LUTs and FFs to 8,423 total LUTs and FFs. The area for each SWOI and HWOI is primarily attributed to the EP's timestamp register and the in-order PESC. The area required by the EPs and pipelined PESC structures increases linearly in relation to the

number of event probes. Notably, the pipelined structure avoids the use of a dedicated bus connecting all SWOIs and HWOIs that can often create significant area overheads.

CHAPTER 6

AREA-EFFICIENT EVENT STREAM ORDERING FOR RUNTIME OBSERVABILITY OF EMBEDDED SYSTEMS

6.1 Overview

Previous system-level observation methods utilized the in-order pipelined, priority-based event stream controller to ensure events are reported in-order based on the event occurrence. The IO-PESC has a pipelined binary tree structure to directly sort events as they are reported. The pipelined binary tree structure provides an in-ordered throughput of one event per clock cycle. While providing high throughput for reporting events, this approach requires significant area resources. The area overhead is primarily attributed to the EP's timestamp register and the pipelined binary tree structure that increases linearly in relation to the number of EPs.

In this chapter, we present an overview of system observation framework [47] utilizing an area-efficient round-robin priority event stream controller (RR-PESC) for reporting observed events. While reducing area overhead according to the number of EPs, the RR-PESC cannot guarantee that observed events are output in order of their occurrence. However, an upper bound on order in which events are reported can be leveraged to efficiently sort the event stream in software. We present and analyze two software re-ordering algorithms, including an immediate sort and output algorithm and a delayed sort and output algorithm.

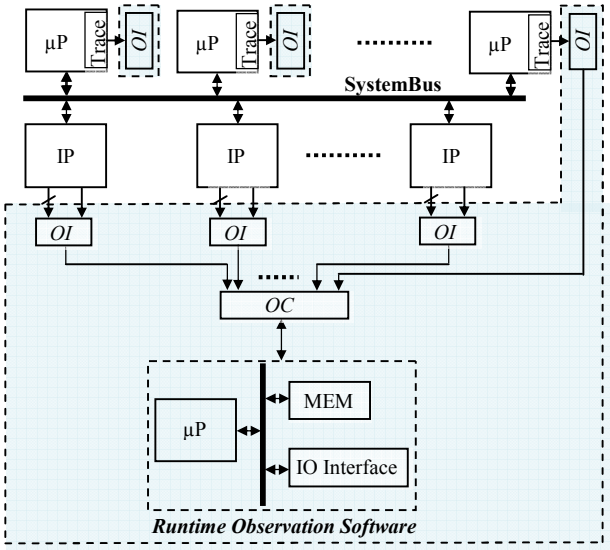

Figure 22: System observation methodology consisting of several observation interface (OIs) and in-situ observation software analyzing the event stream.

6.2 Overview of System-Level Observation

System-level observation methods provide the capabilities for monitoring and analyzing rapidly occurring events and in-situ support for configuring and controlling event probes within software and hardware components. Figure 22 presents an overview of a system-level observation framework integrated within a multiprocessor system-on-a-chip (SOC) design. The framework consists of a software observation interface connected to the trace port of each processor core and hardware observation interface connected to each hardware IP core to be observed. Each observation interface (OI) consists of one or more event probes (EPs), a timestamp register, a configuration register of each EP, a priority event stream controller, and a small FIFO for buffering events within the event stream. An EP is the basic element for monitoring events within software and hardware executions. The OI supports blocking, non-blocking, and cascading EPs to provide

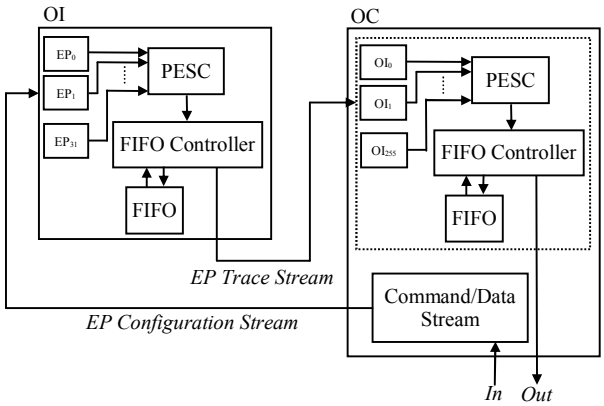

Figure 23: Overview of priority-based event streaming hierarchy and configuration stream interfaces for the system-level observation methodology.

runtime support for defining and controlling the EPs. Blocking and non-blocking EPs have different semantic behavior that affects how EPs are reported when the event stream buffers are full. A blocking EP will not detect another occurrence of an event until the observation software has processed the EP. In contrast, a non-blocking EP will continue to detect and report all occurrences of the event without requiring a reset from the observation software. If the buffers for events are full, the events detected by a non-blocking EP may be missed.

The framework utilizes a priority-based event stream controller (PESC) to serialize and report multiple observed events, as shown in Figure 23. Within each OI, the PESC serializes and stores observed events within a small FIFO. The system-level observation controller (OC) serializes and stores monitored events across multiple OIs using the same priority-based event stream control mechanism. The observed events are finally reported to the runtime observation software using a dedicated interface to an isolated processor executing the observation software to analyze the event stream in-situ. All EPs can be

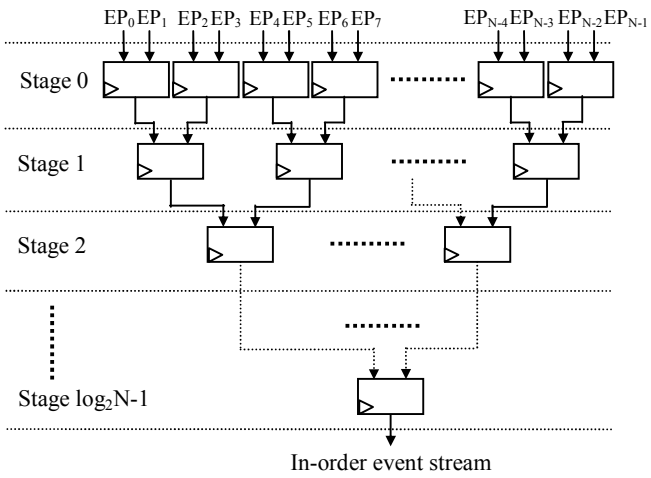

Figure 24: Overview of pipelined event ordering hardware.

configured using software APIs implemented within the runtime observation software. These APIs allow the observation software to configure and control the EPs in-situ.

We previously utilized a similar architecture for observing system events, in which a pipelined binary tree structure of event ordering components was utilized to directly sort the events as those events are reported. The pipelined event ordering hardware uses $\log_2 N$ stages where N is the number of events being monitored within the OIs. Figure 24 presents an overview of the pipelined sorting architecture for a system that consists of N EPs. Each sort component compares two input events and selects the event with the higher priority to forward to the next stage during each clock cycle, reporting events based on the events' timestamps. A lower timestamp indicates the event was observed earlier and thus needs to be reported first. In the case that two events have the same timestamp, the EP with a lower ID is given priority.

When the forwarded event is read by the following pipeline stage, the sorting components will again compare the two current input events to determine the next event

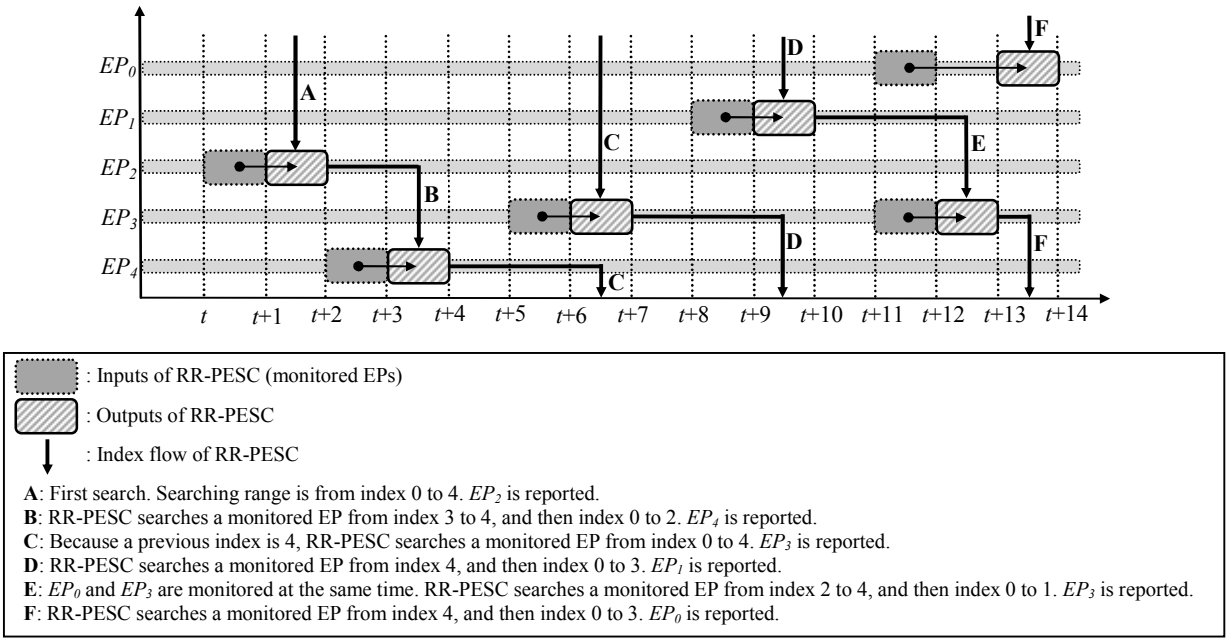

Figure 25: Example operation of round-robin priority controller. The x-axis displays time (in clock cycles), and the y-axis displays EPs.

to forward. When an observed event reaches the final stage of the pipeline, the observed event can be stored within a FIFO to buffer the event stream for the final output or analysis. This pipelined binary tree structure achieves an overall throughput of one observed event per clock cycle. However, the binary tree structure requires significant area overhead. To monitor N different events, N-1 PESC components are required within the $\log_2 N$ pipeline stages.

6.3 Round-Robin Priority-Based Event Stream Controller

Instead of utilizing a pipelined binary tree structure to directly sort observed events as they are reported to the OC, we present a round-robin priority-based event stream

controller (RR-PESC) that is an area-efficient event stream ordering technique that significantly reduces area requirements.

The RR-PESC is incorporated within the OIs and the OC to serialize and report observed events to the runtime observation software. Within the OIs, the RR-PESC compares all input events and selects the event to report according to a round-robin priority control scheme. The selected event is written to the output FIFO. The outputs of the FIFOs in each individual OIs ae then connected to the OC, which uses the same RR-PESC mechanism to control the order in which events from different OIs are reported to the runtime observation software.

In this article, we consider a set of N EPs in an OI $EP_i = \{EP_0, EP_1, ..., EP_{N-1}\}$. The EPs are assumed independent, i.e., no correlation exists between the EPs. The RR-PESC checks whether any EP detects a desired event each clock cycle. When observed events exist, the RR-PESC selects one observed event among multiple observed events in a cyclic fashion. The RR-PESC starts a search from index j. The index j initially starts at index 0. The EPs within the OI will be searched starting from index j to find EP_i, such that EP_i is observed and i is the smallest index greater than or equal to j with an observed event. After outputting EP_i, the RR-PESC will update j to i+1. If there are no EPs with index greater than j, the RR-PESC will continue to search starting with a search index of 0.

Figure 25 presents an example of the RR-PESC behavior for a set of five EPs. The RR-PESC will initially use a search index of j=0. At time t, EP_2 is first observed. The RR-PESC will output EP_2 and update the search index j to 3. At time t+2, EP_4 is

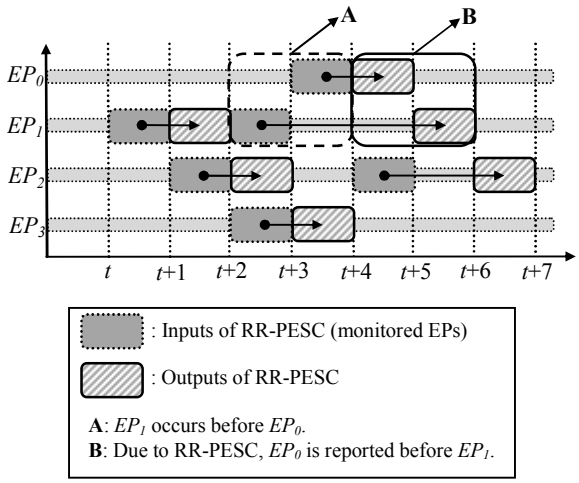

Figure 26: An EP observed later can be output before an EP observed earlier. The x-axis displays time (in clock cycles), and the y-axis displays EPs.

observed. The RR-PESC will output EP_4 and update the search index j to 0. Later at time t+8, EP_1 is observed, and the RR-PESC will output EP_1, updating the search index j to 2. Then, both EP_0 and the EP_3 are observed at the same time at time t+11. Because the search index is currently 2, the RR-PESC will first output EP_3 followed by EP_0.

Using the RR-PESC, events may be output in an order that is not sorted by the event timestamp, which is useful for system monitoring and analysis. Figure 26 shows an example scenario in which an EP observed later is output before an EP observed earlier for a set of four EPs. At time t+2, both EP_1 and EP_3 are observed at the same time. As the search index is currently 3, the RR-PESC will output EP_3 and update the search index j to 0. At time t+3, EP_0 is observed, and because the search index is currently 0, EP_0 will be output. It is not until time t+5, that EP_1 is output. As highlighted in the dashed box, EP_1 was observed before EP_0. However, EP_0 was output before EP_1, as highlighted in the solid box.

Thus, using a round-robin priority, the RR-PESC cannot guarantee that events are output according to their observation time. However, an upper bound on the difference between the event observance and the final output can be determined. To find this worst-case event output time, assume all EPs observe their respective events at the same time. Further, assume that the search index j is 0. In this case, the RR-PESC will output an observed event in the following order: $EP_0 \rightarrow EP_1 \cdots \rightarrow EP_{N-2} \rightarrow EP_{N-1}$. Event EP_{N-1} will wait for N event outputs before being output. Thus, the worst-case event output time for N EPs is N event outputs. Note that while events can be output at a maximum rate of one event per clock cycle, the observation software controls the effective event output rate. Hence, the worst-case event output time is defined in terms of event outputs and not clock cycles. This worst-case event output time implies that in a sequence of N event outputs, the timestamp for event EP_{N-1} must be greater than or equal to EP_0.

6.4 Online Event Stream Processing

While the output of the RR-PESC is not sorted by timestamp, the reported events are nearly sorted and the upper bound on the difference between the event observance and event reporting time can be leveraged to implement efficient software based sorting algorithms to reorder the events according to their timestamps. For in-situ analysis of system behavior, we present two software re-ordering algorithms. Both algorithms utilize a buffer to store incoming events that need to be sorted using a buffer size equal to twice the number of enabled EPs (*numEPs*). Because the maximum difference between an event observation and event reporting is N event outputs, after sorting the buffer, the first

```
Algorithm 1 Immediate Sort/Output
Input: EP Stream(addr, data, ts)
 1: buffer.InsertSorted(addr, data, ts);
 2: while (buffer[0].ts ≤ buffer[buffer.numElements-1].ts – numEPs)
 3:   OutputEvent(buffer[0]);
 4:   buffer.Remove(0);
 5: end while
 6: if (buffer.numElements == buffer.maxSize) then
 7:   for (k ← 0 to numEPs) do
 8:     OutputEvent(buffer[0]);
 9:     buffer.Remove(0);
10:   end for
11: end if
```

Figure 27: Immediate Sort/Output Algorithm.

half of the buffer can be output with a guarantee that any incoming event will not have a timestamp less than the output events.

6.4.1 Immediate Sort/Output

Figure 27 shows the pseudocode for the immediate sort/output algorithm. The immediate sort/output algorithm's goal is to sort events as the events are read from the event stream and written to the buffer. The procedure starts with reading an event from the OC. The event is immediately inserted within the buffer in sorted order using an insertion sort. The time complexity of this operation is O(n) where n is the number of events in a buffer. After inserting the new event, the algorithm will determine which events can be immediately output. If the difference in timestamp between any two events in the buffer is greater than *numEPs*, the event is output from the buffer. As the buffer is sorted, this process compares the first and last event in the buffer and outputs events as long as this condition holds. Furthermore, if the buffer reaches its maximum capacity of 2**numEPs*, the first half of the buffer is immediately output.

```
Algorithm 2 Delayed Sort/Output
Input:  EP Stream(addr, data, ts)
 1: if (buffer.numElements == buffer.maxSize) then
 2:   buffer.Sort();
 3:   while (k < numEPs)
 4:     OutputEvent(buffer[0]);
 5:     buffer.Remove(0);
 6:   end while
 7:   while (buffer[0].ts ≤ buffer[buffer.maxSize-1].ts – numEPs)
 8:     OutputEvent(buffer[0]);
 9:     buffer.Remove(0);
10:   end while
11: end if
```

Figure 28: Delayed Sort/Output Algorithm.

6.4.2 Delayed Sort/Output

Figure 28 shows the pseudocode of a delayed sort/output algorithm. The goal of the delayed sort/output algorithm is to only sort events when the buffer is full, which provides the opportunity to utilize an efficient sorting algorithm. When the buffer reaches its maximum capacity of 2**numEPs*, the algorithm executes an insertion sort to sort the events. While the worst-case time complexity for the insertion sort is $O(n^2)$, insertion sort performs well when the input is nearly sorted, where the performance approaches the best-case runtime of $O(n)$. Thus, we utilize insertion sort rather than a sort algorithm with $O(n \log n)$ complexity. We further experimentally verified that in practice insertion sort performs faster than both mergesort and quicksort for event stream reordering. After sorting the buffer, the first half of the buffer is output. The algorithm then checks to determine if any additional events can be output based on the difference in timestamps.

6.5 Experimental Results

We implemented the system observation framework using VHDL and utilized an FPGA-based prototype to evaluate the performance of this architecture. While the proposed approach is suitable for any SOC design, including ASIC and full-custom implementations, we utilized an FPGA-based prototype system to evaluate the capabilities of the round-robin priority-based event stream controller and software reordering algorithms. While evaluating the performance, area, and power of the proposed architecture for ASIC based designs remains future work, we expect the reported performance and area to scale accordingly. We created an example system consisting of a 125 MHz MicroBlaze processor with basic system peripherals—e.g., timers, interrupt controllers, memory controllers, UARTs. We implemented a real-time application consisting of five periodically executing tasks from the SNU benchmark suite, namely binary search (*bs*), FFT using Cooley-Tukey algorithm (*fft1*), integer-based forward discrete cosine transform from JPEG image encoding standard (*jfdctint*), matrix multiplication (*matmul*), and matrix inversion (*minver*). The execution periods for individual tasks range from 120 ms to 310 ms. The Xilinx xilkernel 4.00a was utilized as the operating system and configured for priority-based scheduling. Finally, we incorporated the system-level observation framework and implemented both our RR-PESC and the pipelined event ordering hardware. The system was synthesized using Xilinx Platform Studio (XPS) 11.5 targeting a Virtex-5 FPGA (XC5VLX110T).

Table 8: Area requirements for the pipelined event ordering hardware and round-robin priority-based event stream controller reported in lookup tables (LUTs) and flip-flops (FFs).

		32 EPs		64 EPs	
		LUT+FF	%	LUT+FF	%
IO-PESC	EPs	8704	100	17408	100
	PESC	10664	100	21672	100
	Total	19368	100	39080	100
RR-PESC	EPs	8704	100	17408	100
	PESC	2348	22	5127	24
	Total	11052	57.1	22535	57.7

6.5.1 Area Results

Table 8 reports the area required in lookup tables (LUTs) and flip-flops (FFs) for the pipelined event ordering hardware and the RR-PESC as a function of the number of event probes utilized within the system. To support 32 EPs, the pipelined event ordering hardware requires 8,215 LUTs and 2,449 FFs, while the RR-PESC only requires 2,204 LUTs and 144 FFs. The RR-PESC requires 78% less area than the pipelined event ordering hardware. The area for the pipelined event ordering hardware is primarily attributed to the binary tree structure of N-1 event ordering components. This requires 31 pipelined event ordering hardware components within 5 stages. As the number of EPs increases, the size for number of event ordering components increases linearly. For a system supporting 64 EPs, the pipelined event ordering hardware requires 16,695 LUTs and 4,977 FFs. In contrast, the RR-PESC requires 4,918 LUTs and 209 FFs, a 76% area reduction compared to the pipelined event ordering hardware.

6.5.2 Event Stream Latency Analysis

For the pipelined event ordering hardware, the latency between an event observation and the final output of that event within a timestamp-sorted event stream is proportional to the

Table 9: Latency (ms) for the pipelined event ordering hardware, the immediate sort/output and the delayed sort/output algorithms.

		CEP	FFEP	RTCS
IO-PESC	Max	0.0176	0.00149	0.00149
	Min	0.00146	0.00147	0.00147
	Avg	0.01755	0.00148	0.00148
Immediate Sort/Output	Max	0.0374	0.2468	121.04
	Min	0.0052	0.0416	0.0304
	Avg	0.0371	0.0426	22.97
Delayed Sort/Output	Max	0.0444	0.2496	536.32
	Min	0.0058	0.0424	0.1841
	Avg	0.044	0.0435	232.42

total number of EPs enabled within the system. However, the latency for the RR-PESC is dependent on both the number of enabled EPs and the latency of the software reordering algorithm.

Table 9 reported the latency of the pipelined event ordering hardware and the RR-PESC with two software re-ordering algorithms. We consider three monitoring scenarios: 1) a constant event probe (CEP) scenario; 2) a fixed frequency event probe (FFEP) scenario; and 3) a real-time system case study (RTCS) scenario. The CEP scenario is an artificial observation scenario to evaluate the maximum event stream throughput consisting of two event probes that are constantly observed every clock cycle. The FFEP and RTCS scenarios are more realistic observation scenarios, where the FFEP scenario is utilized to measure the latency of a single event probe with a fixed frequency of occurrence of 39.36 μs, and the RTCS scenario is designed to observe the start time and end time of each periodic task execution for the five software tasks considered.

For the CEP scenario, the average latencies of the pipelined event ordering hardware, immediate sort/output, and the delayed sort/output algorithms are 17.6 μs, 37.1 μs, and 44 μs, respectively. The difference in latency between the pipelined event

ordering hardware and the RR-PESC is 19.5 μs for the immediate sort/output algorithm and 26.4 μs for the delayed sort/output algorithm. Because the CEP scenario consists of constantly observed events, this difference represents the latency that can be attributed directly to the latency of the sorting algorithms.

For the FFEP scenario, the average latencies of the immediate sort/output and the delayed sort/output algorithms are 42.6 μs and 43.5 μs, respectively. When only one event probe is enabled, the two software re-ordering algorithms have similar latency. We note that latency in this scenario is dependent on the period of the event observations, as both algorithms require a second event to be input into the buffer before an event can be output. In contrast, the pipelined event ordering hardware achieves an average latency of only 1.48 μs, which is due to the binary tree structure that is not affected by the number of enabled EPs or the frequency of the event observation.

For the RTCS scenario, the average latencies of the immediate sort/output and the delayed sort/output algorithms are 22.97 ms and 232.42 ms, respectively. Again, the immediate sort/output algorithm achieves a lower latency, which is as much as 10X faster than the delayed sort/output. The RTCS scenario presents higher latency than others, which can be attributed to the period execution rates of tasks. In both immediate sort/output and the delayed sort/output algorithms, at least two events must be received before the algorithms can determine if the event can be output. As the RTCS scenario monitors the tasks' start and end times, the periodic rate of the fastest executing task will affect the overall latency. In the RTCS scenario, the period of the highest priority task is 120 ms. For the immediate sort/output algorithm, the maximum latency should be equal

Table 10: Throughput (events/second) for the pipelined event ordering hardware, the immediate sort/output and the delayed sort/output algorithms.

	CEP	FFEP	RTCS
IO-PESC	400791.0	25198.0	43.57
Immediate Sort/Output	219646.6	24847.3	43.54
Delayed Sort/Output	228360.8	24846.6	43.26

to this period, which is evidenced by the maximum measured latency of 121 ms. For the delayed sort/output algorithm, because the event stream buffer must be full before sorting the events, the maximum latency is affected by both the buffer size and the execution rates of the monitored tasks. The delayed sort/output algorithm has a maximum latency of 536 ms for the RTCS scenario.

For all three scenarios, the immediate sort/output has smaller latency than the delayed sort/output, which is due to the fact that the delayed sort/output must wait until the buffer is full before outputting any events.

6.5.3 Event Stream Throughput Analysis

Table 10 reports throughput for all three scenarios. To measure the effective maximum event throughput of the presented methods, we measure the total number of events that can be processed by the runtime observation software within a fixed time interval. While the pipelined event ordering hardware provides a maximum throughput of one event per clock cycle, the maximum effective throughput for the pipelined event ordering hardware, including the delay for the runtime software to read the events from the event stream, is 400,791 events per second. For the RR-PESC, the immediate sort/output and the delayed sort/output are capable of sorting and reporting 219,647 events per second and 228,361 events per second, respectively. Overall, the immediate sort/output algorithm achieves

both greater throughput and lower latency compared to the delayed sort/output algorithm. However, the throughput of the delayed sort/output is greater than the immediate sort/output in the CEP scenario because the immediate sort/output algorithm requires more comparisons to output events that are constantly observed every clock cycle.

CHAPTER 7

PRIORITY-LEVEL BASED EVENT STREAM TECHNIQUE FOR NON-INTRUSIVE RUNTIME MONITORING OF EMBEDDED SYSTEMS

7.1 Overview

We previously proposed a pipelined hardware architecture to ensure events are reported in-order based on the event occurrence with area overhead and a round-robin event stream ordering technique that significantly reduces area requirements with tradeoff in event stream throughput. The round-robin event ordering technique sequentially reports observed events based on the EP's ID, where the EP with the next larger ID will be output next. As such, the round-robin event stream ordering technique cannot guarantee observed events are output according to their occurrence time and requires a software sorting algorithm. While our proposed software sorting algorithm is effective, the round-robin priority scheme can affect the event stream throughput in the worst case [47].

In this chapter, we present a priority-level event stream ordering technique [48] that further reduces area requirement compared to the round-robin event stream ordering technique and supports different performance for different priority levels. Additionally, the priority-level event stream ordering technique allows designers specify priorities of different system components or events.

7.2 Priority-Level Based Event Stream Controller

To further reduce area requirement compared to the RR-PESC, the SOF supports a priority-level event stream controller (PL-PESC) that report events based on a priority

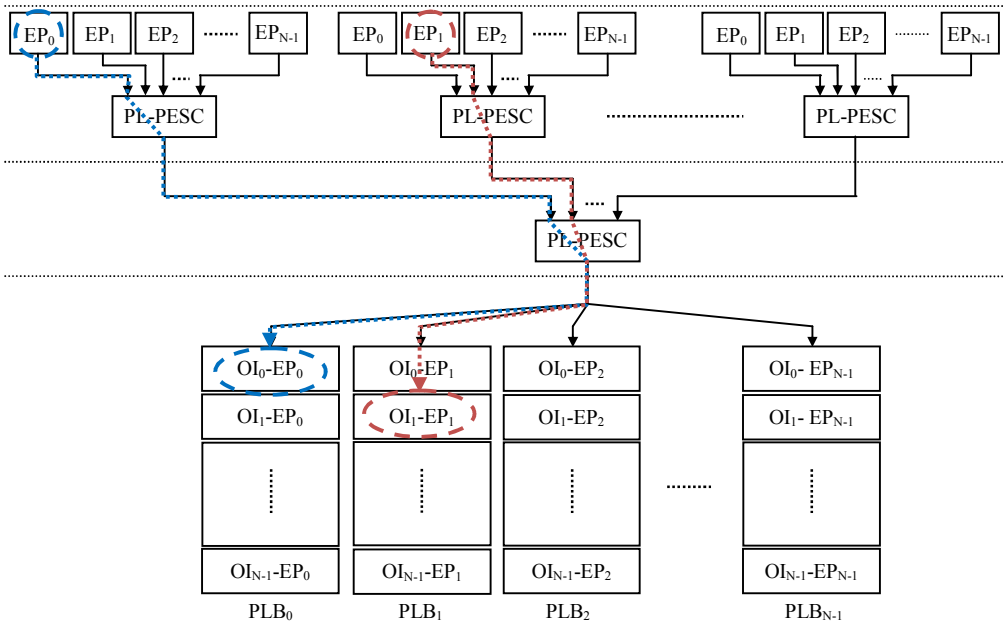

Figure 29: Example operation of priority-level event stream controller using an EP → PL priority assignment.

level assigned to EPs and OIs. The PL-PESC allows designers to specify priorities of different system components or events. To specify each priority, the PL-PESC utilizes a concept similar to fixed-priority preemptive scheduling that executes the highest priority task that is currently ready to execute. When observed events exist, the PL-PESC will select the observed event with the highest priority. The priorities of observed events can be configured in two different ways:

- EP → PL: the ID of the EP is utilized to determine the EP's priority level
- OI → PL: the ID of the OI is utilized to determine the EP's priority level

Figure 29 shows an operation example of the PL-PESC using an EP → PL priority assignment. Observed events from enabled EPs are stored to PLBs having the same ID of EPs, respectively. For example, using an EP → PL priority assignment, all

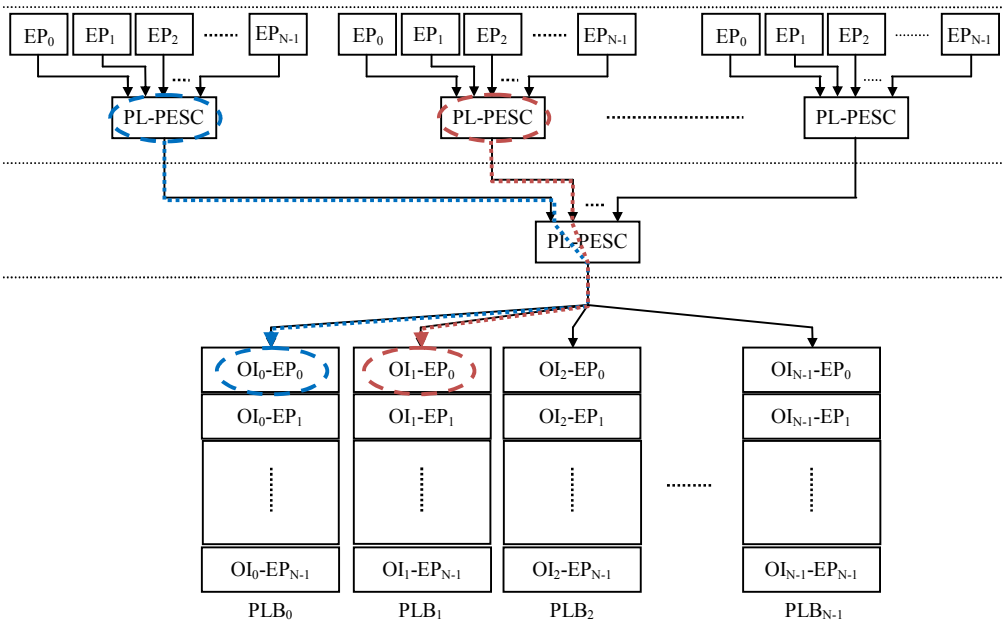

Figure 30: Example operation of priority-level event stream controller using an OI → PL priority assignment.

EPs with an ID of 0 will be mapped to the same PLB, namely PLB_0. This is further extended to the *SOCntrl* for selecting events between OIs. Figure 30 shows an operation example of the PL-PESC using an OI → PL priority assignment. Observed events from enabled EPs are stored to PLBs having the same ID of OIs, respectively. Using an OI → PL priority assignment, all EPs within the same OI will be mapped to the same PLB. For example, all EPs within OI_0 will be mapped to PLB_0.

The priority-level buffer (PLB) is a buffer to temporally store observed events that need to be sorted. The size of each PLB is based on the number of events assigned to a PL, the rate of event occurrence, and the number of higher priority events. We utilize an extension of the response time analysis [9][10]

to determine the individual PLB size requirements and the total event stream buffer size (*TESBS*).

The *TESBS* is the total buffers size required across all PLBs. The *TESBS* is calculated as:

$$TESBS = \sum_{l=0}^{N-1} PLBS_l$$

Where *l* is a priority level (PL) and *N* is the total number of the priority levels.

The priority-level buffer size (*PLBS*) for each PLB is a total event buffer requirement for a specific priority level *l*. The PLBS for priority level *l* ($PLBS_l$) is calculated as:

$$PLBS_l = \sum_{j \in PLB_l} EBR_j$$

where *j* represents all EPs mapped to PLB_l given the current priority assignment, and EBR is the event buffer requirement for a specific EP given by:

$$EBR_j = \left\lceil \frac{\max_{k \in PLB_l} (WCRT_k)}{MFR_j} \right\rceil$$

where *j* is an EP mapped to PLB_l, *k* represents all EPs assigned to PLB_l, $WCRT_k$ represents the worst case report time for all EPs *k*, and MFR_j represents the maximum firing rate for EP *j*.

In order to ensure a PLB can be efficiently sorted, the PLB must store events for a sufficient duration to ensure an occurrence of the lowest priority event mapped to the PLB can be observed and inserted into the sorted position according to the EP's timestamp. The worst case report time (WCRT) for an EP is the maximum time between

the event observation and the event being reported to the SOEngine. Hence, the maximum WCRT for all events mapped to a PLB defines an upper bound on the period of time that events within the PLB may need to be sorted to ensure the events within the PLB can be sorted. Given this upper bound, the EBR for each EP can be determined by considering the EP's maximum firing rate (MFR). The MFR for an EP is the maximum frequency at which the associated event is expected to be observed. For example, consider an event defined as the execution of a periodic task within a multitasked application. The MFR for that event is equal to the period of the task. Within the SOF, we assume a designer can specify the MFR for all EPs.

Finally, the WCRT of an event probe EP_i is calculated as:

$$WCRT_i = RT_i + \sum_{j \in hp_i} \left\lceil \frac{WCRT_i}{MFR_j} \right\rceil RTI_j$$

where RT_i is the report time (RT) in cycles from the event observation to the reporting of the event to the SOEngine without interference from higher priority EPs and RTI_j is the interference in cycles for each instance a higher priority EPs is reported. The WCRT equation can be solved iteratively starting with $WCRT_i = RT_i$.

For our SOF implementation, the RT_i is 2 and the RTI_j is 1 for all events. Thus, a simplified equation for $WCRT_i$ is

$$WCRT_i = 2 + \sum_{j \in hp_i} \left\lceil \frac{WCRT_i}{MFR_j} \right\rceil.$$

While the output of the PL-PESC is not sorted by timestamp, the reported events in each PLB are nearly sorted. After sorting each PLB, events of the PLB can be output with

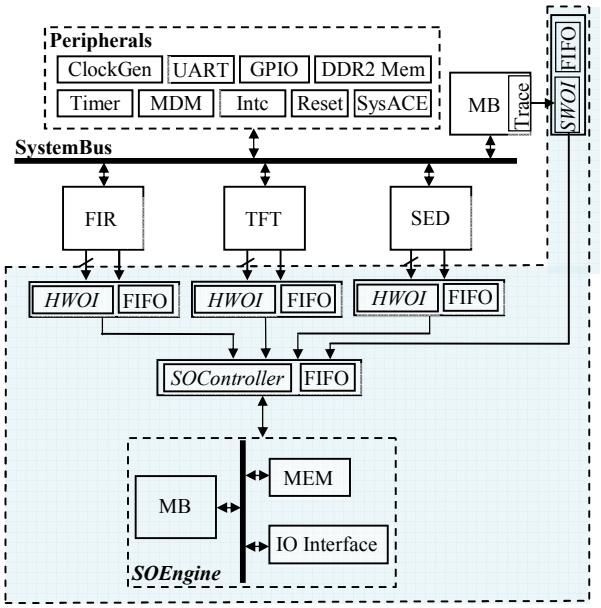

Figure 31: Overview of complete system design including three HWOIs and one SWOI.

a guarantee that any incoming event will not have a timestamp less than the output events. To sort each PLB, the immediate sort/output algorithm is utilized.

7.3 Experimental Results

To evaluate and demonstrate the system-level observation framework, we consider an FPGA-based prototype of a SOC design consisting of a 125 MHz MicroBlaze processor and several hardware IP cores, presented in Figure 31. In addition to hardware cores implementing basic system functionality—e.g. timers, interrupt controllers, memory controllers, UARTs—the system design includes three additional cores accelerating specific operations: 1) a 13-tap FIR filter; 2) a sobel edge detection (SED) processing 640x480 grayscale images; and 3) a TFT controller for displaying the resulting images using a DVI display output. We implemented a real-time application consisting of five

periodically executing tasks from the SNU benchmark suite, namely binary search (*bs*), FFT using Cooley-Tukey algorithm (*fft1*), integer-based forward discrete cosine transform from JPEG image encoding standard (*jfdctint*), matrix multiplication (*matmul*), and matrix inversion (*minver*). The execution periods for individual tasks range from 120 ms to 310 ms. Xilinx xilkernel 4.00a was utilized as the operating system and configured for priority-based scheduling. To observe the start time and the end time for all application tasks, ten configurable software event probes which were configured as non-blocking event probes were implemented within the SWOI for the MicroBlaze processor. Similarly, six configurable hardware event probes which were configured as non-blocking event probes were implemented within the HWOIs for the FIR, the SED and TFT cores. The system was synthesized using Xilinx Platform Studio (XPS) 11.5 targeting a Virtex-5 FPGA (XC5VLX110T).

We consider four monitoring scenarios: 1) a constant event probe (CEP) scenario; 2) a fixed frequency event probe (FFEP) scenario; 3) a real-time system case study (RTCS) scenario; and 4) a system-level case study (SLCS) scenario. The CEP scenario is an artificial observation scenario to evaluate the maximum event stream throughput consisting of two event probes that are constantly observed every clock cycle. The FFEP, RTCS and SLCS scenarios are more realistic observation scenarios, where the FFEP scenarios is utilized to measure the latency of a single event probe with a fixed frequency of occurrence of 39.36 μs, the RTCS scenario is designed to observe the start time and end time of each periodic task execution for the five software tasks considered, and the

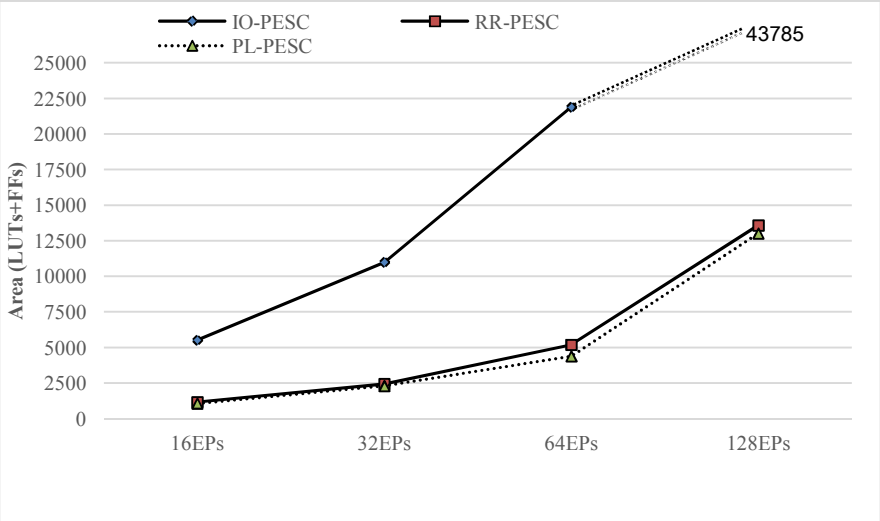

Figure 32: Area requirements for the IO-PESC, RR-PESC and PL-PESC reported in lookup tables (LUTs) and flip-flops (FFs).

SLCS scenarios is designed to observe the start time and end time of five software tasks for the RTCS scenario and three additional cores across multiple HWOIs and SWOIs.

7.3.1 Area Results

Figure 32 presents the area required in lookup tables (LUTs) and flip-flops (FFs) for the IO-PESC, the RR-PESC and the PL-PESC as a function of the number of event probes ranging from 16 to 128. The area requirements increase from 5,517 total LUTs and FFs to 43,785 total LUTs and FFs, from 1,164 total LUTs and FFs to 13,581 total LUTs and FFs, and from 1,068 total LUTs and FFs to 13,019 total LUTs and FFs, respectively. For 16 EPs, the RR-PESC requires 78.9% less area than the IO-PESC and for 128 EPs, the RR-PESC requires 68.98% less area than the IO-PESC. The area for the IO-PESC is primarily attributed to the binary tree structure of *N*-1 event ordering components. This ranges from 15 IO-PESC components within 4 stages to 127 IO-PESC components

Table 11: Latency (ms) for the IO-PESC, RR-PESC with the immediate sort/output algorithm, and PL-PESC with the immediate sort/output algorithm.

		CEP	FFEP	RTCS	SLCS
IO-IO	Max	2.853504	0.001488	0.001488	0.002328
	Min	0.001472	0.001472	0.001472	0.000416
	Avg	1.427552	0.001479	0.00148	0.000446
RR-RR	Max	4.915648	0.248696	121.0423	15.32952
	Min	0.008792	0.041584	0.030616	0.006776
	Avg	2.462265	0.042604	22.96781	6.975841
PL-PL	Max	4.579464	0.002112	0.004128	16.80478
	Min	0.002096	0.002096	0.002096	0.002328
	Avg	2.291998	0.002104	0.003247	14.13581

within 7 stages. As the number of EPs increases, the size for number of event ordering components increases linearly. In contrast, the PL-PESC has simpler structure than the RR-PESC because the PL-PESC does not require a previous location. For 16 EPs, the PL-PESC requires 80.64% less area than the IO-PESC and 8.25% less area than the RR-PESC and for 128 EPs, the PL-PESC requires 70.27% less area than the IO-PESC and 4.14% less area than the RR-PESC.

7.3.2 Latency Analysis

Table 11 reports the latency of the IO-PESC, the RR-PESC with an immediate sort/output algorithm and the PL-PESC with an immediate sort/output algorithm. For the CEP scenario, the average latencies of the IO-PESC, the RR-PESC with an immediate sort/output, and the PL-PESC with an immediate sort/output are 1.43 ms, 2.46 ms, and 2.29 ms, respectively. The difference in latency between the IO-PESC and the RR-PESC with an immediate sort/output is 1.03 ms and the difference in latency between the IO-PESC and the PL-PESC with an immediate sort/output is 0.86 ms. Because the CEP scenario consists of constantly observed events, this difference represents the latency that

can be attributed directly to the latency of the sorting algorithms. Additionally, the difference in latency between the RR-PESC and the PL-PESC represents the latency between the RR-PESC that depends on the previous search index and the PL-PESC that depends on the number of priority levels.

For the FFEP scenario, the average latencies of the RR-PESC with an immediate sort/output is 42.6 μs. When only one event probe is enabled, the latency of the RR-PESC is dependent on the period of the event observations, as the immediate sort/output algorithm requires a second event to be input into the buffer before an event can be output. In contrast, the IO-PESC achieves an average latency of only 1.48 μs, which is due to the binary tree structure that is not affected by the number of enabled EPs or the frequency of the event observations. While utilizing the same immediate sort/output algorithms, the buffer size for the PL-PESC is determined by the priority level. For the FFEP scenario, the buffer size of the immediate sort/output for the RR-PESC is two and one for the PL-PESC. Whenever a new event is updated, the buffer for the Pl-PESC is always full. Therefore, the PL-PESC with the immediate sort/output algorithm reports the event promptly. As a result, the average latency of the PL-PESC with an immediate sort/output is 2.1 μs.

For the RTCS scenario, the average latencies of the RR-PESC with an immediate sort/output and the PL-PESC with an immediate sort/output are 22.97 ms and 3.25 μs, respectively. Again, the buffer size of the RR-PESC is twice of the number of enabled EPs and the immediate sort/output requires a second event. For the RR-PESC, at least two events must be received before the algorithm can determine if the event can be

output. The RTCS scenario presents higher latency than the CEP and FFEP scenarios, which can be attributed to the period execution rates of tasks. As the RTCS scenario monitors the tasks' start and end times, the periodic rate of the fastest executing task will affect the overall latency. In the RTCS scenario, the period of the highest priority task is 120 ms. For the immediate sort/output algorithm of the RR-PESC, the maximum latency should be equal to this period, which is evidenced by the maximum measured latency of 121.04 ms. For the PL-PESC, because the buffer size that is determined by the priority level is one, the maximum latency that is not affected by both the buffer size and the execution rates of the monitored tasks is 4.13 μs.

The SLCS scenario observes five periodic tasks of the RTCS scenario and the start time and end time of three hardware IP cores. For the SLCS scenario, the average latencies of the RR-PESC with an immediate sort/output and the PL-PESC with an immediate sort/output are 6.98 ms and 14.14 ms, respectively. In the SLCS scenario, while the buffer size of PLB_2 to PLB_9 is one, the buffer size of PLB_0 to PLB_1 is four. The immediate sort/output algorithm of the PL-PESC is affected by frequency of event occurrence like the RR-PESC. Additionally, the RR-PESC has smaller average latency compared to the PL-PESC because the Pl-PESC is affected by SWOIs/HWOIs having higher priorities across multiple SWOIs and HWOIs unlike the RR-PESC following the round-robin priority control scheme.

For the IO-PESC, the latency between an event observation and the final output of that event within a timestamp-sorted event stream is proportional to the total number of EPs enabled within the system. However, the latency of the RR-PESC is dependent on

Table 12: Throughput (events/second) for the IO-PESC, RR-PESC with the immediate sort/output algorithm, and PL-PESC with the immediate sort/output algorithm.

	CEP	FFEP	RTCS	SLCS
IO-IO	400789.78	25197.56	43.574	143.30
RR-RR	228267.77	24847.25	43.544	143.25
PL-PL	243389.68	25000.44	43.571	141.27

both the number of enabled EPs and the latency of the software sorting algorithm. Similarly, the latency of the Pl-PESC is dependent on both the total number of priority levels and the latency of the software sorting algorithm.

7.3.3 Throughput Analysis

Table 12 reports throughput for all four scenarios. To measure the effective maximum event throughput of the presented methods, we measure the total number of events that can be processed by the runtime observation software within a fixed time interval. While the IO-PESC provides a maximum throughput of one event per clock cycle, the maximum effective throughput for the IO-PESC, including the delay for the runtime software to read the events from the event stream, is 400,790 events per second. The RR-PESC with the immediate sort/output and the PL-PESC with the immediate sort/output are capable of sorting and reporting 228,268 events per second and 243,390 events per second, respectively. The IO-PESC that is not affected by the number of enabled EPs and frequency of the event occurrence achieves greater throughput compared to the RR-PESC and the PL-PESC. Overall, the PL-PESC achieves greater throughput compared to the RR-PESC. However, the throughput of the RR-PESC is greater than the Pl-PESC in the SLCS scenario because SWOIs/HWOIs having lower priorities must wait the output of SWOIs/HWOIs having higher priorities across multiple SWOIs and HWOIs.

Table 13: Event stream buffer size for the RR-PESC with the immediate sort/output algorithm and PL-PESC with the immediate sort/output algorithm.

	CEP	FFEP	RTCS	SLCS
RR-RR	4	2	20	32
PL-PL	3	1	10	16

For the SLCS scenario, the Pl-PESC has lower throughput than the RR-PESC, which is due to the fact that the PL-PESC uses separate buffers for each PL while the RR-PESC uses a single buffer. Using the immediate sort/output algorithm, the overhead of sorting the individual buffers can be affected by the frequency of specific events. Additionally, the observation software incurs a slight overhead for determining which PLB to insert each incoming event.

7.3.4 Event Stream Buffer Size Analysis

Table 13 reports the event stream buffer size (ESBS) of the RR-PESC with the immediate sort/output and the Pl-PESC with immediate sort/output for all four scenarios. Because the ESBR of the RR-PESC with the immediate sort/output is always twice the number of enabled EPs, ESBSs for four scenarios are 4, 2, 20 and 32, respectively.

For the PL-PESC with the immediate sort/output, the ESBS is calculated based on the WCRTs and MFRs for each scenario. The CEP scenario consists of two EPs, namely EP_0 and EP_1, with WCRTs of 2 cycles and 4 cycles, respectively, and a MFR of 2 for both EPs. The total buffer requirements, or ESBR, for this scenario is 3. The FFEP scenario consists of only a single EP, requiring only a single buffer. The RTCS scenario consists of ten EPs, in which the WCRT for the EPs ranges from 2 cycles to 11 cycles and the MFRs range from $15*10^6$ cycles (once every 120 ms) to $38.75*10^6$ cycles (once

Table 14: Summary of OI IDs, EP IDs, MFR, PL mapping (using an EP → PL priority assignment) for all EPs, and EBRs in the SLCS scenario.

Task/Core	OI ID	EP ID	MFR (cycles)	PL Mapping	EBR
bs	0	{0, 1}	$25*10^6$	{ PL_0, PL_1}	{1, 1}
fft1	0	{2, 3}	$38.75*10^6$	{ PL_2, PL_3}	{1, 1}
jfdctint	0	{4, 5}	$17.5*10^6$	{ PL_4, PL_5}	{1, 1}
matmul	0	{6, 7}	$23.75*10^6$	{ PL_6, PL_7}	{1, 1}
minver	0	{8, 9}	$15*10^6$	{ PL_8, PL_9}	{1, 1}
FIR	1	{0, 1}	$51.25*10^6$	{ PL_0, PL_1}	{1, 1}
TFT	2	{0, 1}	$2.1*10^6$	{ PL_0, PL_1}	{1, 1}
SED	3	{0, 1}	$58.75*10^6$	{ PL_0, PL_1}	{1, 1}

every 310 ms). Due to the short WCRTs and long MFRs, the buffer requirement for each EP is only 1, resulting in an ESBR of 10. Table 14 summarizes the OI IDs, EP IDs, MFRs, priority level mapping for all EPs, and EBRs for all EPs within the SLCS scenario. WCRTs range from 2 cycles to 17 cycles with MFRs from $2.1*10^6$ cycles (once every 16.8 ms) to $58.75*10^6$ cycles (once every 470 ms). The resulting ESBR for the SLCS scenario is 16. Overall, the PL-PESC requires 50% smaller buffer compared to the RR-PESC.

CHAPTER 8

CONCLUSIONS

The current generation of tools for system observability fails to provide the required visibility into increasingly complex systems. First, they fail to provide a beyond-the-laboratory capability. Second, they fail to provide a unified approach to system observability that permits a system designer to simultaneously monitor both hardware and software consideration in-situ. The proposed system-level observation framework provides low-overhead methods for observing and analyzing designer-specified hardware and software events at runtime.

We presented a hardware observability framework system demonstrating a nonintrusive method by which complex events and states in hardware cores can be dynamically observed at runtime. Beyond the area required for the main hardware observability components, the proposed framework scales extremely well as additional HEPs are incorporated.

We presented a system-level observation framework capable of dynamically observing and analyzing designer-specified hardware and software events at runtime. The SOF integrates efficient methods for monitoring both hardware and software elements without affecting the execution of the system. Using a prototype SOC design, we demonstrated the capabilities of this approach. The SOF provides considerable flexibility in defining event probes, configuring event probes at runtime, and analyzing events

within the system observation software that exceed the capabilities of alternative test and debug methods.

We presented a system-level observation framework capable of dynamically observing and analyzing rapidly occurring software events at runtime. Using a prototype SOC design, we demonstrated the capabilities of this approach in nonintrusively analyzing the completion time and scheduling jitter for real-time application tasks. The SOF provides visibility for monitoring runtime execution behavior of software applications without affecting the system execution.

We presented area-efficient priority-based event stream controllers and a software sorting algorithm capable of dynamically ordering and reporting rapidly occurring system events at runtime. Using a prototype SOC design, we demonstrated the capabilities of these approaches in area requirement, latency and throughput. The RR-PESC with immediate sort/output algorithm requires 68.98% less area compared to the IO-PESC and the PL-PESC with immediate sort/output algorithm requires 70.27% less area compared to the IO-PESC. While the average throughput of the RR-PESC with immediate sort/output and PL-PESC with immediate sort/output decreased by 43% and 39% in the CEP scenario, for common observation situations, such as the FFEP, RTCS and SLCS scenarios, the average throughput decreased by less than 1.5% and as little as 0.007%. This decrease of throughput is primarily attributed to the operations for sorting and outputting events in the buffer.

CHAPTER 9

FUTURE WORK

Our future work will include accurately observing events across clock domains, automation tools for synthesizing event probes directly from system requirements, and providing self-configuration of the designer-specified event probes according to the system execution.

In most embedded systems, while each CPU runs in its own clock domain, other peripherals may run in another slower clock domain. Often designers create several different clock domains to optimize system performance and energy consumption. Future work will investigate how to extend the SOF to accurately measuring timestamps across different clock domains.

Future work includes designing automation tools that automatically generate event probes from system requirements, such as these specified using a specification language such as PSL. Automation tools can significantly reduce development time and costs by eliminating the need for designers to manually create HDL codes for the event probes and eliminating the need to utilize specialized tools.

The current SOF observes and analyzes events from the designer-specified event probes. The designer-specified event probe can be configured at runtime, but manually. The SOF will provide self-configuration of the designer-specified event probes according to the system execution. The designer can design observation scenarios of the event

probes, and the event probes can change observation conditions according to the system execution.

REFERENCES

[1] Abardanel, Y., I. Beer, L. Gluhovsky, S. Keidar, Y. Wolfsthal. FoCs: Automatic Generation of Simulation Checkers from Formal Specifications. Conference on Computer-Aided Verification (CAV), pp. 538-542, 2000.

[2] Abramovici, M., P. Bradley, K. Dwarakanath, G. Memmi, D. Miller. A Reconfigurable Design-for-Debug Infrastructure for SoCs. Design Automation Conference (DAC), pp. 7-12, 2006.

[3] Abramovici, M. In-System Silicon Validation and Debug. IEEE Design and Test of Computers, pp. 216-223, 2008.

[4] Accellera. Property Specification Language Reference Manual, Version 1.1, 2004.

[5] Adir, A., S. Copty, S. Landa, A. Nahir, G. Shurek, C, A. Ziv, C. Meissner, J. Schumann. A Unified Methodology for Pre-Silicon Verification and Post-Silicon Validation. Design Automation and Test in Europe Conference (DATE), pp. 1-6, 2011.

[6] Adir, A., A. Nahir, G. Shurek, C. Meissner, J. Schumann. Leveraging Pre-Silicon Verification Resources for the Post-Silicon Validation of the IBM POWER7 Processor. Design Automation Conference (DAC), pp. 569-574, 2011.

[7] ARM, Corp. CoreSight Components Technical Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0314h/index.html, 2009.

[8] ARM, Corp. Embedded Trace Macrocell Architecture Specification. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0014q/index.html, 2009.

[9] Audsley, N., A. Burns, M. Richardson, A. J. Wellings. Hard Real-Time Scheduling: The Deadline-Monotonic Approach. IEEE Workshop on Real-Time Operating Systems and Software, 1991.

[10] Audsley, N., A. Burns, M. Richardson, K. Tindell, A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. Software Engineering Journal, Volume 8, Issue 5, pp. 284-292, 1993.

[11] Backasch, R., C. Hochberger, A. Weiss, M. Leucker, R. Lasslop. Runtime Verification for Multicore SoC with High-Quality Trace Data. ACM Transactions on Design Automation of Electronic Systems (TODAES), Volume 18, Issue 2, Article 18, pp. 1-26, 2013.

[12] Borrione, D., M. Liu, K. Morin-Allory, P. Ostier, L. Fesquet. Online assertion-based verification with proven correct monitors. ITI 3rd International Conference on Information and Communications Technology (ICICT), pp. 125-143, 2005.

[13] Borrione, D., M. Liu, P. Ostier, L. Fesquet. PSL-based online monitoring of digital systems. Forum on Specification and Design Languages (FDL), pp. 5-22, 2005.

[14] Boulé, M., J. Chenard, Z. Zilic. Adding debug enhancements to assertion checkers for hardware emulation and silicon debug. IEEE International Conference on Computer Design (ICCD), pp. 294-299, 2006.

[15] Boulé, M., Z. Zilic. Efficient automata-based assertion-checker synthesis of PSL properties. IEEE International High Level Design Validation and Test Workshop (HLDVT), pp. 69-76, 2006.

[16] Boulé, M., Z. Zilic. Efficient automata-based assertion-checker synthesis of SEREs for hardware emulation. Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 324-329, 2007.

[17] Boulé, M., Z. Zilic. Automata-based Assertion-Checker Synthesis of PSL Properties. ACM Transactions on Design Automation of Electronic Systems (TODAES), Volume 13, Issue 1, Article 4, pp. 1-21, 2008.

[18] Camera, K., H. So, R. Brodersen. An Integrated Debugging Environment for Reprogrammable Hardware Systems. International Symposium on Automated Analysis-Driven Debugging (AADEBUG), pp. 111-116, 2005.

[19] Cantrill, B., M. Shapiro, A. Leventhal. Dynamic Instrumentation of Production Systems. USENIX Annual Technical Conference, pp. 15-28, 2004.

[20] Cetin, A. E., O. N. Gerek, Y. Yardimci, Equiripple FIR Filter Design by the FFT Algorithm, IEEE Signal Processing Magazine, Volume 14, Issue 2, pp. 60-64, 1997.

[21] El Shobaki, M., L. Lindh. A Hardware and Software Monitor for High-Level System-on-Chip Verification. Proceedings of the IEEE International Symposium on Quality Electronic Design (ISQED), pp. 56-61, 2001.

[22] El Shobaki, M. On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems. International Conference on Real-Time Computing Systems and Applications, 2002.

[23] Gao, W., X. Zhang, L. Yang, H. Liu. An Improved Sobel Edge Detection, IEEE International Conference on Computer Science and Information Technology (ICCSIT), pp. 67-71, 2010.

[24] Goel, S. K., B. Vermeulen. Data Invalidation Analysis for Scan-Based Debug on Multiple-Clock System Chips. IEEE European Test Workshop (ETW), pp. 61-66, 2002.

[25] Goel, S. K., B. Vermeulen. Hierarchical Data Invalidation Analysis for Scan-Based Debug on Multiple-Clock System Chips. IEEE International Test Conference (ITC), pp.1103-1110, 2002.

[26] Heffernan, D., S. Shaheen, C. Watterson. Monitoring embedded software timing properties with an SoC-resident monitor. IET Software, Volume 3, Issue 2, pp.140-153, 2009.

[27] Hochberger, C., A. Weiss. Acquiring an Exhaustive, Continuous and Real-Time Trace from SOCs. In Proceedings of the IEEE International Conference on Computer Design (ICCD), pp. 356-362, 2008.

[28] Hochberger, C., A. Weiss. A new methodology for debugging and validation of soft cores. In Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), pp. 551-554, 2008.

[29] IEEE Standard 1149.1-1993, IEEE Standard Test Access Port and Boundary Scan Architecture, IEEE Standards Board, October 1993.

[30] IEEE Standard 1850-2010 (Revision of IEEE Standard 1850-2005), IEEE Standard for Property Specification Language (PSL), IEEE Standards, pp. 1-188, 2010.

[31] Ko, H. F., A. B. Kinsman, N. Nicolici. Distributed Embedded Logic Analysis for Post-Silicon Validation of SOCs. IEEE International Test Conference (ITC), pp.1-10, 2008.

[32] Ko, H. F., N. Nicolici. On Automated Trigger Event Generation in Post-Silicon Validation. Design Automation and Test in Europe Conference (DATE), pp. 256-259, 2008.

[33] Ko, H. F., N. Nicolici. Automated Trace Signals Identification and State Restoration for Improving Observability in Post-Silicon Validation. Design Automation and Test in Europe Conference (DATE), pp. 1298-1303, 2008.

[34] Ko, H. F., N. Nicolici. Automated Scan Chain Division for Reducing Shift and Capture Power during Broadside At-Speed Test. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), pp. 2092-2097, 2008.

[35] Ko, H. F., N. Nicolici. A Novel Automated Scan Chain Division Method for Shift and Capture Power Reduction in Broadside At-Speed Test. International Symposium on Quality Electronic Design (ISQED), pp. 649-654, 2008.

[36] Ko, H. F., N. Nicolici. Resource-Efficient Programmable Trigger Units for Post-Silicon Validation. IEEE European Test Symposium (ETS), pp. 17-22, 2009.

[37] Ko, H. F., N. Nicolici. Algorithms for State Restoration and Trace-Signal Selection for Data Acquisition in silicon Debug. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), pp. 285-297, 2009.

[38] Ko, H. F., N. Nicolici. Combining Scan and Trace Buffers for Enhancing Real-time Observability in Post-Silicon Debugging. IEEE European Test Symposium (ETS), pp. 62-67, 2010.

[39] Ko, H. F., A. B. Kinsman, N. Nicolici. Design-for-Debug Architecture for Distributed Embedded Logic Analysis. IEEE Transactions on Very Large Scale Integration (VLSI) Systems (TVLSI), Volume 19, Issue 8, pp. 1380-1393, 2010.

[40] Ko, H. F., N. Nicolici. Mapping Trigger Conditions onto Trigger Units during Post-silicon Validation and Debugging. IEEE Transactions on Computers (TC), Volume 61, Issue 11, pp. 1563-1575, 2012.

[41] Leatherman, R., B. Ableidinger, N. Stollon. Processor and System Bus On Chip Instrumentation. Embedded Systems Conference, 2003.

[42] Leatherman, R., N. Stollon. Integrating On Chip Debug Instrumentation and EDA Verification Tools. DesignCon East, 2005.

[43] Leatherman, R., N. Stollon. An Embedded Debugging Architecture for SOCs. IEEE Potentials, Volume 24, Issue 1, pp. 12-16, 2005.

[44] Lee, J. C., A. S. Gardner, R. Lysecky. Hardware Observability Framework for Minimally Intrusive Online Monitoring of Embedded Systems. IEEE International Conference on Engineering of Computer-Based Systems (ECBS), pp. 52-60, 2011.

[45] Lee, J. C., F. Kouteib, R. Lysecky. Event-Driven Framework for Configurable Runtime System Observability for SOC Designs. IEEE International Test Conference (ITC), pp. 1-10, 2012.

[46] Lee, J. C., R. Lysecky. System Observation of Blocking, Non-Blocking, and Cascading Events for Runtime Monitoring of Real-Time systems. IEEE International Conference on Engineering of Computer-Based Systems (ECBS), pp. 49-58, 2013.

[47] Lee, J. C., R. Lysecky. Area-Efficient Event Stream Ordering for Runtime Observability of Embedded Systems. Design Automation Conference (DAC), Article 130, pp. 1-6, 2014.

[48] Lee, J. C., R. Lysecky. System-Level Observation Framework for Non-Intrusive Runtime Monitoring of Embedded Systems. Submitted to ACM Transactions on Design Automation of Electronic Systems (TODAES).

[49] Liu, X., Q. Xu. Interconnection Fabric Design for Tracing Signals in Post-Silicon Validation. Design Automation Conference (DAC), pp. 352-357, 2007.

[50] Liu, X., Q. Xu. Trace signal selection for visibility enhancement in post-silicon validation. Design Automation and Test in Europe Conference (DATE), pp. 1338-1343, 2009.

[51] Liu, X., Q. Xu. On multiplexed signal tracing for post-silicon debug. Design Automation and Test in Europe Conference (DATE), pp. 1-6, 2011.

[52] Liu, X., Q. Xu. On Signal Selection for Visibility Enhancement in Trace-Based Post-Silicon Validation. IEEE Transactions on Computer-Aided Design of

Integrated Circuits and Systems (TCAD) , Volume 31, Issue 8, pp. 1263-1274, 2012.

[53] Liu, X., Q. Xu. On efficient silicon debug with flexible trace interconnection fabric. IEEE International Test Conference (ITC), pp. 1-9, 2012.

[54] Liu, X., Q. Xu. On Multiplexed Signal Tracing for Post-Silicon Validation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) , Volume 32, Issue 5, pp. 748-759, 2013.

[55] Morin-Allory, K., D. Borrione. A proof of correctness for the construction of property monitors. IEEE International High-Level Design Validation and Test Workshop (HLDVT), pp. 237-244, 2005.

[56] Morin-Allory, K., D. Borrione. Online monitoring of properties built on regular expression sequences. Forum on Specification Design Languages (FDL), 2006.

[57] Morin-Allory, K., D. Borrione. Proven correct monitors from PSL specifications. Design Automation and Test in Europe Conference (DATE), pp. 1246-1251, 2006.

[58] Morin-Allory, K., L. Fesquet, B. Roustan, D. Borrione. Asynchronous On-Line Monitoring of Logical and Temporal Assertions. Lecture Notes in Electrical Engineering (LNEE), pp. 243-253, 2008.

[59] McDougall, R., J. Mauro, B. Gregg. Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (Solaris Series). Prentice Hall PTR, 2006.

[60] N. Nicolici, H. F. Ko. Design-for-debug for post-silicon validation: Can high-level descriptions help?. IEEE International High Level Design validation and Test Workshop (HLDVT), pp. 172-175, 2009.

[61] Obeidat, F., R. Klenke. Introducing MicroBlaze as an Infrastructure for Performance Modeling. IEEE International Conference on Microelectronic Systems Education (MSE), pp. 90-93, 2011.

[62] Oddos, Y., K. Morin-Allory, D. Borrione. On-Line Test Vector Generation from Temporal Regular Expressions. International Workshop on System-on-Chip for Real-Time Applications (IWSOC), pp. 135-140, 2006.

[63] Oddos, Y., K. Morin-Allory, D. Borrione. On-Line Test Vector Generation from Temporal Constraints Written in PSL. IFIP International Conference on Very Large Scale Integration (VLSISOC), pp. 397-402, 2006.

[64] Oddos, Y., K. Morin-Allory, D. Borrione. Prototyping Generators for on-line test vector generation based on PSL properties. IEEE Design and Diagnostics of Electronic Circuits and Systems (DDECS), pp. 1-6, 2007.

[65] Oddos, Y., K. Morin-Allory, D. Borrione. Assertion-Based Design with Horus. ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE), pp. 75-76, 2008.

[66] Oddos, Y., K. Morin-Allory, D. Borrione. Assertion-Based Verification and On-line Testing in Horus. Design and Test Workshop (IDT), pp. 249-254, 2008.

[67] Peterson, K., Y. Savaria. Assertion-based On-line Verification and Debug Environment for Complex Hardware Systems. International Symposium on Circuits and Systems (ISCAS), pp. 685-688, 2004.

[68] Prasad, V., W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, B. Chen. Locating System Problems using Dynamic Instrumentation. http://www.sourceware.org/systemtap/systemtap-ols.pdf, 2005.

[69] Saha, S., A. Chakrabarti, R. Ghosh. Exploration of Multi-thread Processing on XILKERNEL for FPGA Based Embedded Systems. International Conference on Control Systems and Computer Science (CSCS), pp. 58-65, 2013.

[70] Shultz, M., J. Tao, J. Jeitner, W. Karl. A Proposal for a New Hardware Cache Monitoring Architecture. Workshop on Memory Systems Performance (MSP), 2002.

[71] Shultz, M., B. White, S. McKee, H.-H. Lee, J. Jeitner. OWL: Next Generation System Monitoring. Conference on Computing Frontiers (CF), pp. 116-124, 2005.

[72] Sidwell, N., V. Prus, P. Alves, S. Loosemore, J. Blandy. Non-stop Multi-Threaded Debugging in GDB. Proceedings of the GCC Developers' Summit, pp. 117-128, 2008.

[73] SNU Real-Time Benchmark Suite. http://www.cprover.org/goto-cc/examples/snu.html

[74] Stollon, N., R. Leatherman, B. Ableidinger, E. Edgar. Multi-Core Embedded Debug for Structured ASIC Systems. DesignCon, 2004.

[75] Vermeulen, B., S. K. Goel. Design for Debug: Catching Design Errors in Digital Chips. IEEE Design & Test of Computers, Volume 19, Issue 3, pp. 37-45, 2002.

[76] Vermeulen, B. Functional Debug Techniques for Embedded Systems. IEEE Design & Test of Computers, Volume 25, Issue 3, pp. 208-215, 2008.

[77] Watterson, C., D. Heffernan. Runtime Verification and Monitoring of Embedded Systems. IET Software, Volume 1, Issue 5, pp.172-179, 2007.

[78] Watterson, C., D. Heffernan. A Runtime Verification Monitoring Approach for Embedded Industrial Controllers. IEEE International symposium on Industrial Electronics (ISIE), pp.2016-2021, 2008.

[79] Weiss, A., C. Hochberger. A New Methodology for the Test of SoCs and for Analyzing Elusive Failures. International Workshop on Microprocessor Test and Verification (MTV), pp. 18-23, 2008.

[80] Xilinx, Corp. EDK Concepts, Tools, and Techniques, http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/edk_ctt.pdf, 2009.

[81] Xilinx, Corp. Fast Simplex Link (FSL) Bus (v2.11b), http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pdf, 2009.

[82] Xilinx, Corp. MicroBlaze Processor Reference Guide, http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/mb_ref_guide.pdf, 2009.

[83] Xilinx, Corp. OS and Libraries Document Collection, http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/oslib_rm.pdf, 2009.

[84] Xilinx, Corp. Thin Film Transistor (TFT) Controller (v2.00a), http://www.xilinx.com/support/documentation/ip_documentation/xps_tft.pdf, 2009.

[85] Xu, S., H. Pollitt-Smith. A Multi-MicroBlaze Based SOC System: From SystemC Modeling to FPGA Prototyping. IEEE/IFIP International Symposium on Rapid System Prototyping (RSP), pp. 121-127, 2008.

[86] Yang, S., H. Shim, W. Yang, C.-M. Kyung. A new RTL Debugging Methodology in FPGA-based Verification Platform. IEEE Asia-Pacific Conference on Advanced System Integrated Circuits (APASIC), pp. 180-183, 2004.