RTSS'08, IEEE, pp. 481-491. 2008

Hardware Runtime Monitoring for Dependable COTS-based Real-Time Embedded Systems

Rodolfo Pellizzoni, Patrick Meredith, Marco Caccamo, Grigore Roşu
Department of Computer Science, University of Illinois at Urbana-Champaign
{*rpelliz2, pmeredit, mcaccamo, grosu*}*@cs.uiuc.edu*

Abstract

COTS peripherals are heavily used in the embedded market, but their unpredictability is a threat for high-criticality real-time systems: it is hard or impossible to formally verify COTS components. Instead, we propose to monitor the runtime behavior of COTS peripherals against their assumed specifications. If violations are detected, then an appropriate recovery measure can be taken. Our monitoring solution is decentralized: a monitoring device is plugged in on a peripheral bus and monitors the peripheral behavior by examining read and write transactions on the bus. Provably correct (w.r.t. given specifications) hardware monitors are synthesized from high level specifications, and executed on FPGAs, resulting in zero runtime overhead on the system CPU. The proposed technique, called BusMOP, has been implemented as an instance of a generic runtime verification framework, called MOP, which until now has only been used for software monitoring. We experimented with our technique using a COTS data acquisition board.

1. Introduction

The real-time embedded system industry is progressively moving towards the use of Commercial-Off-The-Shelf (COTS) components in an attempt to reduce costs and time-to-market, even for highly critical systems like those deployed by the avionic industry. While specialized hardware and software solutions are sometimes available for such markets, their average performance and ease of integration is lagging behind the development of COTS components. For example, a commercial plane like the Boeing 777 uses the SAFEbus backplane [10], which, while specially designed to meet the hard real-time constraints of an avionic system, is only capable of transferring data up to 60 Mbps. On the other side, a modern COTS peripheral bus such as PCI Express 2.0 [16] can reach transfer speeds of 16 Gbyte/s, over three orders of magnitude greater than SAFEbus.

Unfortunately, when trying to use COTS for building high-integrity, real-time embedded systems, current engineering practices face significant challenges. While one can capture relevant assumptions about COTS as formal specifications, they are hard or impossible to formally verify: this is both because manufacturers are unwilling to disclose details of their implementation, for fear of losing competitive edge, and because the increase in performance is often matched by a similar increase in design complexity (out-of-order execution and branch prediction are examples of this trend in CPU design). Modern COTS peripherals running in master mode are particularly challenging. A master peripheral can directly communicate with all other elements in the system, including main memory and other peripherals, thus reducing the load on the CPU. On the other side, providing fault-containment becomes extremely hard: a misbehaving, low-criticality master peripheral could very well disrupt the entire system.

Based on the above discussion, our proposal for the safe integration of COTS peripherals in critical embedded systems is to use *runtime monitoring*: the peripheral requirement specifications are checked at runtime against its current observable behavior. If any violation is detected, then a suitable recovery action can be taken to restore the system to a safe state. The validity of the runtime monitoring approach has been proved in the field of software engineering by a large number of developed tools and techniques (see Section 7). However, applying runtime monitoring to our scenario poses some new challenges. First of all, the behavior of a COTS peripheral is controlled both by the hardware of the peripheral itself and by its software driver, hence we must check the correctness of their interactions. Second, master peripherals can directly interact with the rest of the system without requiring any action by the CPU. Based on these two considerations, our monitoring solution must be able to detect and check all communication between the peripheral and the rest of the system. Finally, runtime monitoring typically comes with an unforgivable price: runtime overhead. We can split such overhead in two components: 1) overhead due to the observation and generation of relevant events 2) overhead due to running a monitor at each event to check if any property of the specification is violated. Both types of overhead tend to be unpredictable and thus unsuitable for real-time computation.

To combat these problems, we propose a distributed monitoring technique based on the development of a *mon-*

```
logic = ERE

declarations : {
  signal cntrlCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal cntrlOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
}

event countDisable : memory write address = base1 + X"220"
                     dbyte value in "---------------0"
event cntrlMod : memory write address in base1 + X"220"
                 {
                   cntrlOld <= cntrlCurrent;
                   cntrlCurrent <= value(15 downto 0);
                 }
event countEnable : memory write address = base1 + X"220"
                    dbyte value in "---------------1"

pattern: ((countEnable countDisable) + cntrlMod + countDisable)*

violation handler : {
  mem_reg <= '1';
  address_reg <= base1 + X"220";
  -- roll back to the previous cntr_cntrl2 value
  value_reg(15 downto 0) <= cntrlOld;
  cntrlCurrent <= cntrlOld;
  enable_reg <= "0011";
}
```

Figure 1: Example Property: SafeCounterModify

itoring device. The idea is to introduce an additional hardware component into the system that can check all peripheral communication and perform recovery actions, when necessary. Assuming "sniffing" data transfers does not add delay to the system, our solution prevents the first type of overhead. The second type of overhead is removed by running all monitors directly on the device, adding no runtime overhead to the CPU. Additionally, the system can run completely undisturbed as long as no recovery action is needed.

The speed of modern COTS communication architectures rules out the possibility of a software implementation for the device; instead, all logic is implemented on a reconfigurable FPGA. Finally, to show that a monitored system is safe, we need to prove that the monitoring logic monitors, indeed, the right properties. In our system, this is ensured by automatically synthesizing the monitoring logic from formal requirements specification, so that it is "correct by construction". In particular, we leverage on the Monitor Oriented Programming (MOP)[5] framework (see Section 2), which is highly extensible and supports multiple formalisms, by creating a new MOP instance: BusMOP.

Illustrative Example. An example of BusMOP can be seen in Figure 1. This example is a property used in the case study of Section 6 and related to the behavior of Counter 2, a counter on the PCI703A board we used in our experiments. This property, called SafeCounterModify, requires that any modification to cntr_cntrl2, the control register for Counter 2, happens only while the counter is not in use. This modification is captured by the cntrlMod event, because cntr_cntrl2 is at address X"220". The counter can be enabled/disabled by modifying bit 0 of cntr_cntrl2 (captured by the countEnable/countDisable events; "-" is the VHDL 'don't care").

logic = ERE tells BusMOP that the property will be expressed using an extended regular expression pattern. The declarations section declares two monitor-local registers, cntrlCurrent and cntrlOld, and initializes them to 0. These registers will hold the current and previous values of the cntr_cntrl2 register. This allows us to repair the register when/if the property is *violated* by writing the old value to the register on the peripheral itself (the value_reg assignment), and forcing the current value the monitor stores to be the previous value, as can be seen in the violation handler section of the specification.

The pattern itself, in the pattern section, matches any trace that consists of a cntr_cntrl2 modification, a disable of the counter, or an enable followed by a disable. The pattern is followed by *, allowing it to match repeatedly. The only way to violate this pattern, then, is to see a modification after an enable that is *not* followed by a disable first.

The implementation of the events, declarations, and the actions available to handlers is explained in Section 5.2. The formula/pattern implementation, and the use of handlers is explained in Section 5.3. The actual generated code is available in our Technical Report [18], or by running the online trial on our website [4].

Key contributions. We provide three main contributions. First, in Section 4 we describe the design of a monitoring device for the PCI/PCI-X bus (a brief overview of PCI is presented in Section 3). The monitoring device can be plugged in on a PCI bus segment, and monitor all peripherals attached to the segment. Whenever peripheral activity fails to conform to the specification, the device can perform a *corrective* action: either bring the peripheral back to a safe state if the error is recoverable, or otherwise disconnect it from the system. While certain implementation decisions are necessarily specific to our choice of PCI, we believe that the general design principles and lessons learned can be applied to most other communication architectures. Second, in Section 5 we provide a new instantiation of the MOP framework, called BusMOP, that is able to generate hardware modules; the generated monitoring logic is then integrated with the rest of the monitoring device design and synthesized on the FPGA. Third, in Section 6 we show the feasibility of the overall approach by applying our technique together with the developed monitoring device to check a COTS data acquisition board. Our experimental results reveal that the monitoring device is able to detect, and recover from, errors caused by faults in the driver that we discovered after manually inspecting it. We conclude by discussing related work in Section 7, and providing final remarks and future work in Section 8.

2. The MOP Framework

Monitoring-Oriented Programming (MOP) ([5] and citations there) is a formal framework for system development and analysis, in which the developer specifies desired properties using *definable* specification formalisms, along with code to execute when properties are violated or validated; it is important to note that a failure to confirm to the specification can be expressed as either the validation or violation of a property, see Section 6 for examples. Monitoring code is then automatically generated from the specified properties and integrated together with the user-provided code into

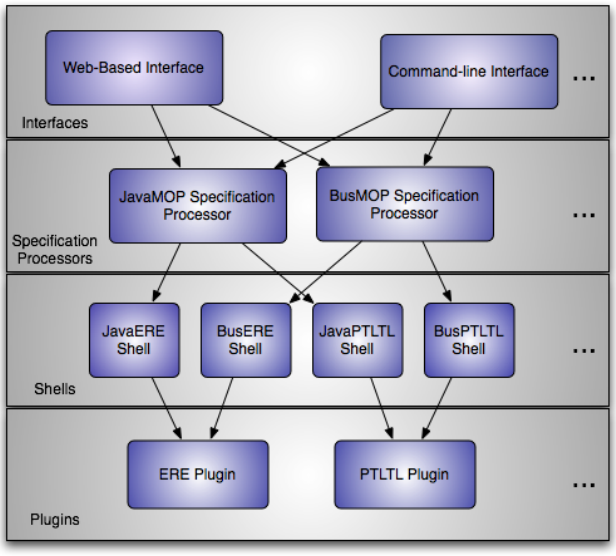


Figure 2: MOP Architecture.

the original system. MOP is a highly extensible and configurable runtime verification framework; currently there are two MOP instances: JavaMOP and BusMOP (the instance described in this paper).

Property specifications consist of event definitions, which are instance dependent (e.g., pointcuts in JavaMOP and bus transactions or interrupts in BusMOP), and logical formulae or patterns, which are not. The user is allowed to extend the MOP framework with his/her own logics via *logic plugins* which encapsulate the monitor synthesis algorithms. This extensibility of MOP is supported by a layered architecture which separates monitor generation and monitor integration. By standardizing the protocols between layers, modules can be added and reused easily and independently. By providing language specific shells, logic plugins can be reused between several different MOP instances. A graphical representation of the architecture can be seen in Figure 2.

The formula or pattern designates which "traces" (observed series of events) are valid or invalid. Because extended regular expression (ERE) and past-time linear temporal logics (PTLTL) are the two plugins used in this paper, we will describe which traces are valid or invalid for ERE patterns and PTLTL formulae. For EREs, valid traces are those which are strings in the language represented by the ERE, with events treated as the letters in the alphabet of the language. Neutral traces (which trigger no handlers) are prefixes of strings in the language, while violations are invalid strings. For PTLTL formulae, valid traces are any traces for which the formula evaluates to true, invalid traces are those for which the formula evaluates to false; there are no neutral traces. For more information on regular languages and temporal logic see [14] and [7], respectively.

3. PCI Bus Overview

The Peripheral Component Interconnect (PCI) is the current standard family of communication architectures for motherboard/peripheral interconnection in the personal computer market; it is also widely popular in the embedded domain [16]. The standard can be divided in two parts: a *logical* specification, which details how the CPU configures and accesses peripherals through the system controller, and a *physical* specification, which details how peripherals are connected to and communicate with the motherboard. While the logical specification has remained largely unaltered since the introduction of the original PCI 1.0 standard in 1992, several different physical specifications have emerged since then.

One of the main features of the logical layer is plug-and-play (automatic configuration) functionality. On start-up, the OS executes a PCI base driver which reads information from special configuration registers implemented by each PCI-compliant peripheral and uses them to configure the system. Of peculiar importance is a set of up to 6 Base Access Registers (BARs). Each BAR represents a request by the peripheral for a block of addresses in either the I/O or memory space; the PCI base driver is responsible for accepting such requests, allocating address blocks and communicating back the chosen addresses to the peripheral, by writing in the BARs. To communicate with the peripheral, the CPU can, then, issue write and read commands, called *transactions*, to either I/O or memory space; each peripheral is required to implement *bus slave* logic, which decodes and responds to transactions targeting all address spaces allocated to the peripheral. Typically, address spaces are used to implement either registers, which control and determine the logical status of the peripheral, or data buffers. Peripherals *can* also implement *bus master* logic: they can autonomously initiate read and write transactions to either main memory or the address space of another peripheral. Master mode is typically used by high-performance peripherals to perform a DMA transfer, i.e., transfer data from the peripheral to a buffer in main memory. The peripheral's driver can then read the data directly from memory, which is much faster than issuing a read transaction on the bus. Finally, each peripheral is provided with an interrupt line that can be used to send interrupts to the CPU.

There are two main flavors of physical architecture: PCI/PCI-X is parallel, while PCI-E is serial but runs at much higher frequency (2.5Ghz against up to 133Mhz for PCI-X). We have focused on PCI/PCI-X[1], which implements a shared bus architecture. The logical PCI tree is physically divided into bus segments, and most bus wires are shared among all peripherals connected to a single segment. We refer to [16] for detailed bus specifications. Each transaction seen on the bus consists of an address phase, which provides the initial address in either memory or I/O space, followed by one or multiples data phases, each of which carries up to 32 or 64 bits of data for PCI/PCI-X, respectively (individual bytes can be masked using *byte en-*

1 We also plan to extend our design to PCI-E; see Section 8.

ables). Since each bus segment is shared, arbitration is required to determine which master peripheral is allowed to transmit at any one time. Arbitration uses two active-low, point-to-point wires between the peripheral and the bus segment arbiter, REQ# and GNT#. A standard request-grant handshake is used, where the peripheral first lowers REQ# to request access to the bus, and the arbiter grants permission to start a new transaction by lowering GNT#.

4. Monitoring Device

We designed a prototype monitoring device based on a Xilinx ML455 board [19] using a mixed VHDL/Verilog register transfer level (RTL) description. The board is outfitted with a Virtex-4 FPGA and is can be plugged into a standard 3.3Volts PCI/PCI-X socket. The FPGA implements both a slave and a master peripheral module, together with the monitoring modules. Events for the system are specified in terms of read/write data transfers on the bus and interrupt requests; the device continuously "sniffs" all ongoing activities on the bus, and it is therefore able to monitor communication for all other peripherals located on the same bus segment. Whenever a failure to meet the specification is detected, the device can execute a recovery action using strategies based on the detected error.

For a vast category of errors that involves incorrect interaction between the peripheral and its software driver, it is often possible to recover from the failure by forcing the peripheral into a consistent state. The monitoring device implements a master module, and can therefore initiate transactions on the bus. For example, consider a common type of error, where the driver fails to validate some input from the user and as a result writes an invalid value to a register in the peripheral. We can recover by rewriting the register with a valid value. However, if the error is caused by a fault in the peripheral hardware, interacting with registers may not be enough to bring the peripheral to a consistent and safe state.

We propose a mechanism that lets the monitoring device disconnect the faulty peripheral from the bus. We developed a simple hardware device, the *peripheral gate* [17], that is able to force the REQ# signal from the peripheral to the bus arbiter to be high; hence, the peripheral never receives the grant and it is prohibited from initiating any further transaction on the bus[2]. The peripheral gate is implemented based on a PCI extender card, i.e., a debug card that is interposed between the peripheral card and the bus and provides easy access to all signals. A clarifying picture for monitoring of a single peripheral is provided in Figure 3(a). The monitoring device can output a *stop* signal, which closes the gate when active high. Finally, sometimes the monitoring device cannot perform a suitable recovery action by itself, but there is a higher level actor, such as the OS or the system user, that can provide better recovery; examples include complex software operations such as restarting the driver or the whole PCI stack, and physically interacting with the peripheral. In this case, the best strategy is to communicate the failure to the chosen actor. The study of OS-level reliability techniques is outside the scope of this paper; instead, for our prototype design we implemented a RS-232 controller that can be used to send information to the user over a serial connection.

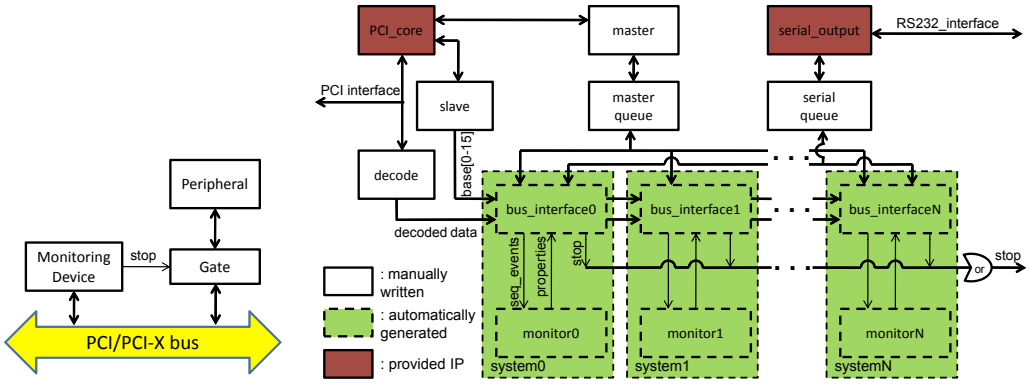
The reader should notice that the nature of our implementation is such that if a trace is seen, which does not conform to a specification, as a consequence of a bus transaction, that specific bus transaction can not be prevented from propagating to the rest of the system. For example, if a faulty peripheral performs a write transaction to an area in main memory which is not supposed to modify, we can detect the error, disconnect the peripheral and report the failure to the OS/user. However, the information in the overwritten area will be lost. As part of our future work, we are working to implement an interposed monitoring device: by sitting between the bus and a peripheral, it will be able to buffer all transactions that target that specific peripheral or are initiated by it. If a property is validated/violated, it is then possible to take *preventive* measures (i.e., either discard or modify the transaction before propagating it). While this solution will provide a higher degree of reliability, there is a price to be paid in terms of increased communication delay due to buffering in the monitoring device.

A simplified block diagram for the monitoring device is shown in Figure 3(b). We distinguish three types of blocks: 1) blocks provided by Xilinx as proprietary intellectual properties (IPs); 2) manually coded RTL modules provided by BusMOP, which are independent of the peripheral specification; 3) automatically generated RTL modules, which are dependent on the specification (see Section 5). PCI transaction signals are routed to two different modules: the PCI_core and the decode module.

The PCI_core module is a hard IP that implements all logic required to handle basic PCI functionalities such as plug-and-play. Bus slave and bus master logic is implemented by the slave and master modules, respectively. In particular, slave implements a set of 16 registers, base0 through base15. Since the OS configures the BAR registers at system boot, a peripheral cannot directly determine the location of address blocks used by another peripheral. Hence, the OS must also write the locations of the address blocks allocated to monitored peripherals in the base registers. The decode module is used to simplify event generation. It translates all transactions on the bus (except for those initiated by the monitoring device itself) into a series of I/O or memory reads/writes, one for each data phase, as well as the occurence of an interrupt, and forwards the translated information to the monitoring logic.

The system0, ..., systemI, ..., systemN blocks implement the monitoring logic for each of N user specified properties. Each systemI block consists of two automatically generated modules: bus_interfaceI contains all logic that depends on the specific choice of communication interface (PCI bus), while monitorI contains all logic that depends on the formal language used to specify the prop-

2 While technically it is always possible for a faulty peripheral to disrupt the bus by altering the state of the signals, in practice the described approach is effective since access to the bus is mediated by three-state buffers enabled by GNT#.



(a) Gated Monitoring Device. (b) Block Diagram.

Figure 3: Monitoring Device.

erty. This separation provides good modularity and facilitates code reuse. bus_interface1 first receives as input the decoded bus signals and generates events, which are sequentialized by the events_sequentializer submodule (see Section 5.2), and then passed to monitor1 using the seq_events wires. monitor1 checks whenever the formula for the I-th property is validated/violated and passes the information back to bus_interface1, which can then execute three types of recovery: 1) disconnect a monitored peripheral from the bus using the stop signal; 2) send information to the user using the serial_output module, which implements a RS-232 transmitter; 3) start a write transaction on the bus using the master module. Finally, since it is possible for multiple system1 modules to initiate recovery at the same time, we provide queuing functionalities for serial_output and master in modules master_queue and serial_queue, respectively.

It is important to notice that in the current implementation the time elapsed from any event that triggers a validation/violation to executing the corresponding handler is at most 4 clock cycles. This time is short enough to execute a recovery action before a faulty peripheral is allowed to start a new transaction, as PCI arbitration overhead prevents a peripheral from transmitting immediately.

5. Property Specifications

Properties are specified using a domain specific event syntax, and formulae or patterns written in the logic of a particular plugin. Additional monitor state can also be declared using the declarations section. The violation handler and validation handler sections allow for arbitrary code to be executed on the occurrence of a violation or validation, respectively. An example of how they are used can be seen in Figure 1 in Section 1. Currently, we have support for the extended regular expression (ERE) and past-time temporal logic (PTLTL) MOP Plugins, and adding most of the others will require a minimal amount of work, as only the monitor component changes from one logical specification formalism to another. This means that properties may be specified, formally, using an ERE pattern or a PTLTL formula.

A property is implemented in two main modules, a bus_interface, which generates logical events from bus traffic and handles monitor recovery, and a monitor implementing a property specification in hardware. A more detailed description will be given below.

5.1. Events

A formal description of the event syntax (using Backus Naur Form (BNF) [12] extended with $[p]$ and $\{p\}$, denoting zero or one repetitions of p and zero or more repetitions of p, respectively) can be seen below:

$$
\begin{array}{rcl}
\langle \textit{Event} \rangle & ::= & \text{“}event\text{”}\ \langle \textit{ID} \rangle : \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{MemoryOrI/O} \rangle \langle \textit{ReadOrWrite} \rangle \text{“}address\text{”}\ \text{“}=\text{”} \\
 & & \langle \textit{ArithmeticExp} \rangle \text{“}value\text{”} [\text{“}not\text{”}] \text{“}in\text{”}\ \langle \textit{Range} \rangle \\
 & & [\langle \textit{Action} \rangle] \\
 & | & \langle \textit{MemoryOrI/O} \rangle \langle \textit{ReadOrWrite} \rangle \text{“}address\text{”}\ \text{“}in\text{”} \\
 & & \langle \textit{Range} \rangle [\text{“}\{\text{”}\ \langle \textit{Action} \rangle\ \text{“}\}\text{”}] \\
 & | & \text{“}interrupt\text{”} [\langle \textit{Action} \rangle] \\
\langle \textit{MemoryOrI/O} \rangle & ::= & \text{“}memory\text{”} \quad | \quad \text{“}io\text{”} \\
\langle \textit{ReadOrWrite} \rangle & ::= & \text{“}read\text{”} \quad | \quad \text{“}write\text{”} \\
\langle \textit{Action} \rangle & ::= & \text{“”}\ \langle \textit{Arbitrary VHDL code} \rangle\ \text{“”} \\
\langle \textit{Range} \rangle & ::= & \langle \textit{ArithmeticExp} \rangle [\text{“,”}\ \langle \textit{ArithmeticExp} \rangle] \\
\langle \textit{ArithmeticExp} \rangle & ::= & \langle \textit{Number} \rangle \quad | \quad \langle \textit{ID} \rangle \\
 & | & \langle \textit{ArithmeticExp} \rangle\ \text{“}+\text{”}\ \langle \textit{ArithmeticExp} \rangle \\
 & | & \langle \textit{ArithmeticExp} \rangle\ \text{“}-\text{”}\ \langle \textit{ArithmeticExp} \rangle \\
 & | & \langle \textit{ArithmeticExp} \rangle\ \text{“}\&\text{”}\ \langle \textit{ArithmeticExp} \rangle \\
\langle \textit{Number} \rangle & ::= & \langle \textit{VHDL number or bitstring} \rangle \\
\langle \textit{ID} \rangle & ::= & \langle \textit{Capital or lower case letter} \rangle \{\langle \textit{LetterOrDigit} \rangle\} \\
\langle \textit{LetterOrDigit} \rangle & ::= & \langle \textit{Capital or lower case letter} \rangle \quad | \quad \langle \textit{Digit} \rangle
\end{array}
$$

There are three basic types of events in BusMOP: I/O accesses, memory accesses, and interrupts. It is important to distinguish between I/O and memory events because they require different enable functionality and different read/write signals. We show uses of interrupt events in [18]. I/O and memory events must specify at least an address, which may be an arithmetic expression over identifiers, VHDL numbers, addition, subtraction, and concatenation, and whether the event is a read or a write. An I/O or memory event may also specify a value range, which is the value of the address read or written by the bus transac-

tion. Ranges can consist of a single arithmetic expression, or a pair of comma separated arithmetic expression denoting the minimum and maximum values that may trigger the event (thus, ranges are inclusive). Value ranges must also specify a size, byte, dbyte (16 bits), or qbyte (32 bits), so that the correct comparison code and byte enables can be generated (values smaller than a byte require masking the proper bits). Address ranges are used in events that *do not* have specified value ranges. The reason for this is that when a value range is specified, the code generator must generate the proper byte enables based on address alignment, and alignment does not make sense for ranges. Address ranges are useful for some properties, e.g. a property that monitors accesses to a certain buffer in memory.

5.2. The bus_interface Module

The code for the declarations, and handler sections is copied verbatim into the VHDL module defining the bus_interface. Because of this copying, the code must be written in VHDL. The events are expanded to combinatoric statements implementing the specified logic. The output of the combinatoric statements is assigned to an events wire vector, which is connected to the monitor module through an event_sequentializer submodule. Each index in the bus corresponds to the truth value of a specific event, numbered with the 0'th index as the first event, and the n'th index as the n'th event from top to bottom in the specification. This ordering is important, because it directs the event linearization performed by the event_sequentializer submodule.

The event_sequentializer is necessary because the logical formalisms expect linear, disjoint events. The event_sequentializer takes coincident events and sends them to the monitor in subsequent clock cycles, in ascending index order, using the seq_events wire vector. Therefore, if events(0) and events(3) occur in the same cycle, the monitor will see 0 followed by 3. To see why simultaneous events are possible, consider, again, Figure 1 from Section 1. The cntrlMod event is asserted whenever the cntr_cntrl2 register (base1 + X"220") is written. Because both the countEnable and countDisable events require writes to the same address as the cntrlMod event, any time countEnable or countDisable are triggered, a cntrlMod is also triggered. As the property tries to enforce the policy that all modifications happen when the counter is not enabled, we must serialize events such that cntrlMod happens *after* a countDisable and *before* a countEnable. The ordering of events in Figure 1, is consistent with this, because countDisable is listed before cntrlMod, which is listed before countDisable.

The violation handler is placed in the module such that it is only executed if the monitor module denotes that the property has been violated. The situation is similar for a validation handler, save that it is executed only when the formula or pattern is validated. As can be seen in the Figure 3(b), the monitor module reports the validation, violation, or neutral state of the monitored property, via the properties wire vector, to the bus_interface module. Several actions are available in validation and violation handlers. Aside from manipulating any local state of the monitor (such as the write to cntrlCurrent in Figure 1), the bus_interface module makes available several registers which can be used used to execute the recovery actions detailed in Section 4. The registers are summarized in the table below:

	Write Interface
io_reg	write request in I/O space
mem_reg	write request in memory space
address_reg	write address
value_reg	write value
enable_reg	byte enables
serial_reg	ASCII value to serial output
stop_reg	Peripheral gate control

As can be seen in Figure 1, we perform a memory write to the cntr_cntrl2 register of its previous value. The address_reg is used to denote the address of cntr_cntrl2 (base1 + X"202"), while the value_reg is set to the old value of cntr_cntrl2, the mem_reg is asserted to tell the PCI bus that the write performed is a memory write, and the byte enables are set to "0011" to denote that the lower two bytes must be written.

5.3. The monitor Module

The monitor module is responsible for monitoring the property given serialized events. It encompasses the logic of the formula, and it is the only portion of our system dependent on the logical formalism used.

Extended Regular Expressions. Extended regular expressions (EREs) are the normal regular expressions [14], extended with negation. The same plugin used for JavaMOP's [5] EREs is used to transform the provided ERE to a minimized deterministic finite automata (DFA) defined in generic code. We convert the generic code to Verilog. The current state of the DFA is kept in a register. On each clock cycle, the current state of the DFA and the event are consulted to see if the property is violated or validated, and what state to transition to. Violations of EREs are tricky, because, if used normally, a DFA, once it reaches a violation state, will report a violation every event (because there is no valid transition out of the violation state). We chose to reset the DFA to the initial state whenever a violation is encountered, to avoid this problem. ERE pattern is as follows:

$$\begin{array}{lcl} \langle Pattern \rangle & ::= & \text{“}epsilon\text{”} \mid \langle Event\ Name \rangle \\ & \mid & \text{“}\sim\text{”}\langle Pattern \rangle \mid \langle Pattern \rangle \text{“}*\text{”} \\ & \mid & \langle Pattern \rangle \text{“}+\text{”}\langle Pattern \rangle \mid \langle Pattern \rangle \langle Pattern \rangle \end{array}$$

“epsilon” is the empty string, “ $\sim$ ” is negation, “ $*$ ” is zero or more repetitions, “+” is logical *or*, and $\langle Pattern \rangle$ $\langle Pattern \rangle$ represents concatenation.

Past-time Linear Temporal Logic. Past-time Temporal Linear Logic (PTLTL) [7] extends normal propositional logic with *temporal* operators. We modified the PTLTL plugin used in JavaMOP to make it more suitable for implementation as a logic circuit. The original, generic code out-

put by the plugin used a number of sequential assignments to an array of truth values. We take this sequential code and, using back substitution, change the sequential code into a series of parallel assignments. The resulting assignments are *entirely* parallel, allowing the operation of the monitor to be contained within a single clock cycle. A more in depth explanation of this transformation is omitted, but will appear in an upcoming technical report on PTLTL. The syntax for PTLTL formulas is as follows:

$$
\begin{array}{rcl}
\langle Formula \rangle & ::= & \text{“}true\text{”} \mid \text{“}false\text{”} \mid \langle Event\ Name \rangle \\
& \mid & \text{“}not\text{”}\langle Formula \rangle \mid \langle Formula \rangle \text{“}and\text{”} \langle Formula \rangle \\
& \mid & \langle Formula \rangle \text{“}or\text{”} \langle Formula \rangle \\
& \mid & \langle Formula \rangle \text{“}implies\text{”} \langle Formula \rangle \\
& \mid & \text{“}[*]\text{”}\langle Formula \rangle \mid \text{“}\langle * \rangle\text{”}\langle Formula \rangle \\
& \mid & \text{“}(*)\text{”}\langle Formula \rangle \mid \langle Formula \rangle \text{“}S\text{”} \langle Formula \rangle
\end{array}
$$

“not”, *“and”*, *“or”*, and *“implies”* are the ordinary logic operators. “$[*]$”, “$\langle * \rangle$”, “$(*)$”, and *“S”* are temporal operators denoting always in the past, eventually in the past, previously, and since, respectively.

A design decision relating to both logics we have implemented, and all future logics, is that properties cannot be violated or validated before an event arrives. This helps eliminate some strange interactions (examples can be found in our Technical Report [18]). JavaMOP has the same functionality, but in JavaMOP it is due to the fact that a monitor does not *exist* before the first event, whereas in BusMOP, the monitor exists as soon as the FPGA is configured.

6. Case Study: The PCI703A ADC/DAC Board

In this section, we show how our runtime monitoring technique can be applied to a concrete case by providing specification and runtime experiments for a specific COTS peripheral, the PCI703A board [8]. PCI703A is a high performance Analog-to-Digital/Digital-to-Analog Conversion (ADC/DAC) peripheral for the PCI bus. In particular, it can perform high-speed, 14-bits precision ADC at a rate of up to 450,000 conversions/s, and transfer data to main memory in bus master mode. At the same time, the peripheral is simple enough that we were able to carefully check all provided hardware manuals and to manually inspect its Linux driver; specifying formal properties for a peripheral clearly requires a deep understanding of its inner working. In our proposed model, the peripheral's manufacturer is responsible for writing the runtime specification. In this sense, the formal specification can be thought of as a correctness certification provided by the manufacturer, as long as the user employs a monitoring device and recovery actions can be proved to restore the system to a safe state.

To better mimic what we think would be a typical process for a COTS manufacturer, we produced a requirement specification for the PCI703A in two steps. First, we prepared a detailed description of the communication behavior of the peripheral in plain English. Then, we converted this informal description into a formal set of events and formulae for BusMOP. Inspection of the driver revealed two

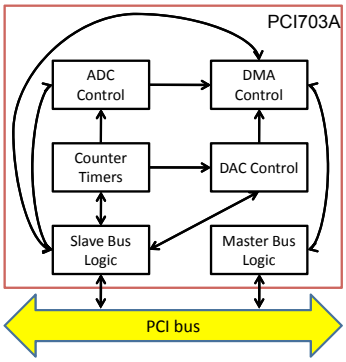


Figure 4: PCI703A Diagram.

software faults, both of which can cause errors that are detected and recovered by the monitoring device. While in this case we could have prevented errors by simply removing the faults, we argue that drivers for more complex peripherals can be thousands of lines long and neither code inspection nor testing is sufficient to remove all bugs. We further injected additional faults in the driver to test all written formal properties. It would have been nice to also show recovery for hardware faults, but we did not find any in the tested peripheral and injecting faults in the hardware is difficult. A list of both informal and formal properties can be found at [18]. In what follows, we first provide an overview of PCI703A and then we detail properties for an example subsystem, a counter used in the ADC process. The example is particularly instructive as we show how a small but representative set of properties is able to catch one of the aforementioned driver bug.

A block diagram for the PCI703A is shown in Figure 6. The bus slave logic implements two memory address blocks in BAR0 and BAR1, used for conversion data and control registers, respectively; the corresponding base addresses are written in base0 and base1 in the monitoring device. The ADC Control and DAC Control blocks control the ADC/DAC operations and write/read data into internal FIFOs. The DMA Control block can be programmed to move data between each FIFO and main memory using bus master functionality. Finally, the Counter Timers block implements four counters. Counter 0 and 1 are user programmable and can be used either for debugging purposes or to trigger a DA conversion. Counter 3 is also user programmable and produces an external output. Finally, Counter 2 is not meant to be user programmable; it is to be used exclusively to generate the clock for AD conversions. The C user library provided with the driver exports an ADConfig function used to configure ADC Control and the associated Counter 2. The library also provides a CTConfig function to be used to configure the user counters; unfortunately, under Linux the function can also be used to change the configuration of Counter 2. This is a problem, as any user in the system could erroneously or maliciously change Counter 2 while an ADC is in progress.

Three 16-bits control registers are relevant to our discussion: cntr_cntrl2 (at hexadecimal location 220 rela-

```
logic = ERE

declarations : {
  signal cntrlCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal cntrlOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
}

event cntrlMod : memory write address in base1 + X"220"
                 {
                   cntrlOld <= cntrlCurrent;
                   cntrlCurrent <= value(15 downto 0);
                 }
event setBit4 : memory write
                address = base1 + X"220"
                dbyte value in "----------1----"

pattern: setBit4

validation handler : {
  mem_reg <= '1';
  address_reg <= base1 + X"220";
  -- roll back to the previous cntr_cntrl2 value
  value_reg(15 downto 0) <= cntrlOld;
  cntrlCurrent <= cntrlOld;
  enable_reg <= "0011";
}
```

Figure 5: InterruptFix Specification

tive to BAR1), cntr_divr2 (228), and adc_cntrl (300). Bit 0 of cntr_cntrl2 determines whether Counter 2 is enabled, and bits 2-1 determine its clock source (either 20Mhz or 100Khz); when the counter is enabled, it first loads the content of cntr_divr2 and then starts counting down at the selected frequency. When it reaches zero, the value of cntr_divr2 is reloaded, a clock signal is sent to ADC Control, and finally if bit 4 of cntr_cntrl2 is set, an interrupt is generated. Register adc_cntrl controls the behavior of ADC Control; in particular, bit 0 enables/disables the ADC process and bits 2-1 determine the clock source, with a value of "00" indicating that Counter 2 is used.

We express three requirements:

Requirement 1 *Bit 4 of cntr_cntrl2 should never be set. While the functionality is relevant for Counters 0,1, in the case of Counter 2 setting bit 4 would cause the generation of spurious interrupts that increase load on the driver.*

Requirement 2 *If the ADC is using Counter 2, and the clock source for Counter 2 is set to 20 Mhz, then the value of cntr_divr2 must be at least 45 to avoid violating the maximum conversion speed of the peripheral.*

Requirement 3 *If the ADC is active and using Counter 2, then Counter 2 must also be active; furthermore, while Counter 2 is active no change to the counter configuration is allowed.*

Requirements 1-3 are able to catch the driver bug in the sense that an invalid counter configuration can not be set before starting the ADC, and furthermore while the ADC is active no counter modification is allowed. We wrote four (five including the example from Section 1) formal properties to capture the requirements:

InterruptFix. The InterruptFix specification is the formalization of Requirement 1, and can be seen in Figure 5.

```
logic = ERE

declarations : {
  signal clkSrc : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal src : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
}

event divrBad: memory write address = base1 + X"228"
               dbyte value in 0,44
event divrGood: memory write address = base1 + X"228"
                dbyte value in 45,65535
event clkSrcSet : memory write address in base1 + X"300"
                  { clkSrc <= value(15 downto 0); }
event srcSet :  memory write address in base1 + X"220"
                { src <= value(15 downto 0); }
event countEnable : memory write address = base1 + X"220"
                    dbyte value in "---------------1"

pattern : (divrBad (clkSrcSet + srcSet)* countEnable)*

validation handler : {
  if (clkSrc(2 downto 1) = "01") and (src(2 downto 1) = "00") then
    mem_reg <= '1';
    address_reg <= base1 + X"228";
    --set cntr_divr2 to 45
    value_reg(15 downto 0) <= X"2D";
    enable_reg <= "0011";
  end if;
}
```

Figure 6: SafeConversionSpeed Specification

Because we do not want the 4th bit set, we simply monitor the pattern setBit4, an event which corresponds to setting the 4th bit. We perform recovery when the pattern is validated by overwriting cntr_cntrl2 with the last valid value, similarly to SafeCounterModify in Figure 1.

SafeConversionSpeed. The SafeConversionSpeed specification is the formalization of Requirement 2, and can be seen in Figure 6. For this property we chose to show how event side effects can be used in handlers as part of checking that a property has been validated/violated. When the clkSrcSet or srcSet events are triggered, meaning that the cntr_cntrl2 or adc_cntrl registers have been modified, respectively, we store the value written to the register in monitor local registers (e.g., src <= value(15 downto 0)). The pattern specifies that the cntr_divr2 be set to a bad value (less than 45), followed by any number of updates to cntr_cntrl2 or adc_cntrl, followed by the enabling of the counter. If cntr_divr2 is set to a value larger than 44, the pattern will be violated, and the monitor will be reset. This means that the validation handler will be executed only when then value of cntr_divr2 is too low for safe conversion, but regardless of whether or not the board is actually using Counter 2. The handler then checks that it is, in fact using Counter 2, and that Counter 2 is using the 20Mhz source, before performing the recovery: setting cntr_divr2 to a valid value (45).

NoDisableWhileConverting. The NoDisableWhileConverting specification is the formalization of part of Requirement 3, and can be seen in Figure 7. This could have been written in a similar manner to SafeConversionSpeed, i.e., using event side effects to store current register values and checking them in the handler. We decided to use a fully formal specification, that defines events for setting the registers to good or bad values. The formula itself specifies that, if the ADC is enabled, and clkSrc2

```
logic = PTLTL

declarations : {
  signal cntrlCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal cntrlOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
}

event countEnable : memory write address = base1 + X"220"
                    dbyte value in "---------------1"
                    {
                      cntrlOld <= cntrlCurrent;
                      cntrlCurrent <= value(15 downto 0);
                    }
event countDisable : memory write address = base1 + X"220"
                     dbyte value in "---------------0"
                     {
                       cntrlOld <= cntrlCurrent;
                       cntrlCurrent <= value(15 downto 0);
                     }
event clkSrc2Good : memory write address = base1 + X"300"
                    dbyte value in "-------------01-"
event clkSrc2Bad : memory write address = base1 + X"300"
                   dbyte value not in "-------------01-"
event adcEnable : memory write address = base1 + X"300"
                  dbyte value in "---------------1"
event adcDisable : memory write address = base1 + X"300"
                   dbyte value in "---------------0"

formula : ( ((not adcDisable) S adcEnable) and
            ((not clkSrc2Bad) S clkSrc2Good) )
          implies
          ((not countDisable) S countEnable)

violation handler : {
  mem_reg <= '1';
  address_reg <= base1 + X"220";
  -- roll back to the previous cntr_cntrl2 value
  value_reg(15 downto 0) <= cntrlOld;
  cntrlCurrent <= cntrlOld;
  enable_reg <= "0011";
}
```

Figure 7: NoDisableWhileConverting Specification

is good, meaning that Counter 2 is being used to time the ADC, then Counter 2 must be enabled. The part of the formula before the implies keyword, states that the ADC is enabled and the ADC clock source is Counter 2, the second half of the formula is the requirement that Counter 2 not be disabled. The formula is true when correct behavior is exhibited, so we use a violation handler for the recovery action, which again is simply to set cntr_cntrl2 to the last valid value.

SafeDivrModify. The SafeDivrModify specification is the formalization of part of Requirement 3, and can be seen in Figure 8. In conjunction with NoDisableWhileConverting and SafeCounterModify (from Section 1), all of requirement 3 is covered. This specification ensures that cntr_divr2 is not modified while Counter 2 is enabled. This property is the same as SafeCounterModify from Figure 1, save that we are ensuring that cntr_divr2 is not modified, rather than cntr_cntrl2. We also used PTLTL rather than ERE, to show how two very similar properties look in both logics. These could be collapsed into one specification, but it would make recovery more complicated, because we only want to roll back the register that was actually modified (cntr_cntrl2 or cntr_divr2). The formula itself states that if cntr_divr2 has been modified and the counter has not been disabled since the last time it was enabled, than we must recover. Unlike

```
logic = PTLTL

declarations : {
  signal divrCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal divrOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
}

event countDisable : memory write address = base1 + X"220"
                     dbyte value in "---------------0"
event divrMod : memory write address in base1 + X"228"
                {
                  divrOld <= divrCurrent;
                  divrCurrent <= value(15 downto 0);
                }
event countEnable : memory write address = base1 + X"220"
                    dbyte value in "---------------1"

formula: (divrMod) and (*)((not countDisable) S countEnable)

validation handler : {
  mem_reg <= '1';
  address_reg <= base1 + X"228";
  -- roll back to the previous cntr_divr2 value
  value_reg(15 downto 0) <= divrOld;
  divrCurrent <= divrOld;
  enable_reg <= "0011";
}
```

Figure 8: SafeDivrModify Specification

SafeCounterModify we use a validation rather than a violation handler, because the formula was easier to express with recovery being on validation.

As a final consideration, note that the handlers of SafeCounterModify, InterruptFix and NoDisableWhileConverting can be invoked simulteously if an incorrect value is written to cntr_cntrl2, which results in the execution of multiple bus writes. However, this causes no problem since all handlers overwrite cntr_cntrl2 with the same valid value.

7. Related Work

There are two main run-time verification approaches: 1) offline, where a log, or trace is kept, which can then be used for purposes of debugging; and 2) online, where a property is checked while the program is running. As BusMOP is an online technique, we will only describe online approaches to runtime verification.

MaC [11], PathExplorer (PaX [9], and Eagle [3] use specific verification languages which cannot be changed, while BusMOP, as an extension of MO [5], will eventually support all the logics supported in JavaMOP. Temporal Rove [6] is a commercial runtime verification tool which uses future time metric temporal logic. It provides inline specification of monitors, where the monitors are written straight in the source file. Inline specification does not make sense for BusMOP, as there is no program being monitored per se. Program Query Language (PQL [15], is an approach somewhat similar to MOP, although it also only allows one specification language. PQL can support the full generality of context free languages. Tracematches [2] is very similar to JavaMOP. The biggest difference is that its choice of regular expressions for logical formalism is hardwired. It is an extension of the AB [1] AspectJ compiler. All of the above approaches are designed to monitor specific programs, and are

implemented in software. This has the effect of both adding runtime overhead, and performing a function different from that of BusMOP, which monitors COTS peripherals.

The PSL to Verilog compiler, P2 [13], is the sole attempt to perform formal runtime verification in hardware, of which we are aware. P2V is similar to BusMOP in that monitors are implemented in hardware rather than software, and that both approaches thus have no runtime overhead on the CPU. P2V, however, is more like the above approaches in that it is designed for monitoring actual programs rather than peripheral devices. Also it requires a dynamically extensible soft-core processor implemented on an FPGA, while our approach can potentially be applied to any COTS communication architecture. Further, P2V uses hardwired logic while BusMOP allows different formalisms.

8. Conclusions and Future Work

COTS peripherals are increasingly being adopted in the embedded market for performance reasons. However, COTS components introduce challenges in the development of critical systems, as they are unpredictable and often complete hardware specification is not publicly available. In this paper, we have proposed run-time monitoring of bus activities as a way to cope with such unpredictability. A monitoring device can be plugged on a PCI bus segment and check that all communication between peripherals and the rest of the system behaves according to specifications. Monitoring logic is automatically generated by the BusMOP framework and synthesized on FPGA, resulting in zero CPU runtime overhead. Finally, we showed the applicability of our monitoring infrastructure and recovery mechanisms on a real test case.

We plan to extend this work in two directions. From a system point of view, we plan to develop a interposing PCI/PCI-X/PCI-E monitoring device capable of executing preventive recovery actions as described in Section 4. From a formal specification point of view, we plan to extend BusMOP to support other MOP logic plugins. Most of them will require little work, with the exception of context free grammars (CFG)[3], which would require implementing, effectively, a hardware LR(1) parser. This extension is not trivial, since memory is required for the stack, and the monitor must be able to process each event in few clock cycles. The unbounded nature of an LR(1) parser's per event memory usage makes the CFG plugin an unlikely candidate for running in a release system.

References

[1] P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. ABC: an extensible AspectJ compiler. In *Proc. of the ACM Conf. on Aspect-oriented software development (ASOD'05)*, pages 87–98, 2005.

[2] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 589–608, 2007.

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, pages 277–306, 2004.

[4] BusMOP webpage. `http://fsl.cs.uiuc.edu/BusMOP`.

[5] F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 569–588, 2007.

[6] D. Drusinksky. Temporal rover, 1997-2007.

[7] E.A. Emerson. *Handbook of Theoretical Computer Science*. MIT Press, 1990. Chapter 16: Temporal and modal logic.

[8] Eagle Technology. *PCI 703 Series User's Manual*. `http://www.eagledaq.com/display_product_36.htm`.

[9] K. Havelund and G. Rosu. Monitoring Java programs with Java pathexplorer. In *Proc. First Workshop on Runtime Verification*, 2001.

[10] K. Hoyme and K. Driscoll. Safebus(tm). *IEEE Aerospace Electronics and Systems Magazine*, pages 34–39, Mar 1993.

[11] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[12] D. Knuth. Backus normal form vs. backus naur form. *Communications of the ACM*, 7(12):735–736, 1964.

[13] H. Lu and A. Forin. The design and implementation of p2v, an architecture for zero-overhead online verification of software programs. Technical Report MSR-TR-2007-99, Microsoft Research, 2007.

[14] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing, 1996. Chapter 1: Regular Languages.

[15] M. Martin, B. Livshits, and M. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 365–383, 2005.

[16] PCI SIG. *Conventional PCI 3.0, PCI-X 2.0 and PCI-E 2.0 Specifications*. `http://www.pcisig.com`.

[17] R. Pellizzoni, B. D. Buy, M. Caccamo, and L. Sha. Coscheduling of real-time tasks and PCI bus transactions. Technical report, University of Illinois at Urbana-Champaign, 2008. Available at `http://netfiles.uiuc.edu/rpelliz2/www/techreps/`.

[18] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Roşu. BusMOP: a runtime monitoring framework for PCI peripherals. Technical report, University of Illinois at Urbana-Champaign, 2008. Available at `http://netfiles.uiuc.edu/rpelliz2/www/techreps/`.

[19] Xilinx, Inc. *Virtex-4 ML455 PCI/PCI-X Development Kit User Guide*. `http://www.xilinx.com/support/documentation/boards_and_kits/ug084.pdf`.

3 More precisely, MOP supports DCFLs.