Spring 6-2-2015

Hardware/Software Interface Assurance with Conformance Checking

Li Lei
Portland State University

Hardware/Software Interface Assurance with Conformance Checking

by

Li Lei

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Fei Xie, Chair
Jingke Li
Suresh Singh
Fu Li

Portland State University
2015

ABSTRACT

Hardware/Software (HW/SW) interfaces are pervasive in modern computer systems. Most of HW/SW interfaces are implemented by devices and their device drivers. Unfortunately, HW/SW interfaces are unreliable and insecure due to their intrinsic complexity and error-prone nature. Moreover, assuring HW/SW interface reliability and security is challenging. First, at the post-silicon validation stage, HW/SW integration validation is largely an ad-hoc and time-consuming process. Second, at the system deployment stage, transient hardware failures and malicious attacks make HW/SW interfaces vulnerable even after intensive testing and validation.

In this dissertation, we present a comprehensive solution for HW/SW interface assurance over the system life cycle. This solution is composited of two major parts. First, our solution provides a systematic HW/SW co-validation framework which validates hardware and software together; Second, based on the co-validation framework, we design two schemes for assuring HW/SW interfaces over the system life cycle: (1) post-silicon HW/SW co-validation at the post-silicon validation stage; (2) HW/SW co-monitoring at the system deployment stage.

Our HW/SW co-validation framework employs a key technique, conformance checking which checks the interface conformance between the device and its reference model. Furthermore, property checking is carried out to verify system properties over the interactions between the reference model and the driver. Based on

the conformance between the reference model and the device, properties hold on the reference model/driver interface also hold on the device/driver interface. Conformance checking discovers inconsistencies between the device and its reference model thereby validating device interface implementations of both sides. Property checking detects both device and driver violations of HW/SW interface protocols. By detecting device and driver errors, our co-validation approach provides a systematic and efficient way to validate HW/SW interfaces.

We developed two software tools which implement the two assurance schemes: DCC (Device Conformance Checker), a co-validation framework for post-silicon HW/SW integration validation; and CoMon (HW/SW Co-monitoring), a runtime verification framework for detecting bugs and malicious attacks across HW/SW interfaces. The two software tools lead to discovery of 42 bugs from four industry hardware devices, the device drivers, and their reference models. The results have demonstrated the significance of our approach in HW/SW interface assurance of industry applications.

DEDICATION

To the memory of my grandma, Binying

To my parents, Ruiling and Xiaoming

To my wife, Qi

ACKNOWLEDGMENTS

First and foremost, I would like to express my deep gratitude to my advisor, Prof. Fei Xie for his support, guidance, and encouragement. Fei taught me how to conduct research systematically: how to identify research problems, how to seek for practical solutions, and how to realize the proposed solutions. When I encountered problems, he spent multiple hours discussing the problems with me and helping me approach to the feasible solutions. During my Ph.D. study, he kept challenging me to be a better student and a better researcher. Most importantly, Fei always reminders me to carry out practical research to impact the real world. His research philosophy will also benefit my future career and life.

I would like to thank my committee members, Prof. Fu Li, Prof. Jingke Li, and Prof. Suresh Singh for their contributions to my dissertation. I highly appreciate their perspectives on my research and precious feedbacks to my thesis.

Dr. Juncao Li has helped me a lot since he was a graduate student in Portland State University (PSU). His passionate attitude towards computer science often motivates me to eagerly explore what I have not seen. His vision from the industry guided me to develop practical solutions and tools. Dr. Kang Li hosted my internship at Virtual Device Technologies (VDTech). He provided me many valuable suggestions for my dissertation research and helped me compile my Ph.D. research into a commercial software of VDTech. I would thank Juncao and Kang's generous help and constructive advices.

My study at Department of Computer Science in PSU would not have been enjoyable without PSU professors and students. Prof. Bryant York enriched my knowledge with his broad perspective of computer science. He also offered me valuable feedbacks when I was writing my first academic paper. Prof. Cynthia Brown gave me a lot of help and suggestions on my study and TA work during my first year at PSU. I am deeply indebted to them. I would also like to thank my fellow graduate students, Dr. Kecheng Hao, Bo Chen, Kai Cong, Christopher Havlicek, Bin Lin, Disha Puri, and Zhenkun Yang. The discussions with them benefited me a lot and made my Ph.D. study more enjoyable.

My grandma, Binying Xi, passed away during my Ph.D. study. I didn't get a chance to see her one last time. This dissertation is dedicated to her. I would like to thank my parents, Ruiling Li and Xiaoming Lei, for their unconditional supports and sacrifices throughout these years. I could never have achieved this without their education and key values instilled in me. Last but not the least, I would thank my wife, Qi Tong for her love and care. Over the past five years, Qi shared every joyful moment with me and encouraged me to pass through every trouble and frustration. She deserved greater thanks than what I could give.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 MOTIVATION

Computer systems are pervasive ranging from smartphones to desktops and to servers. Our daily life heavily depends on computer systems. For example, nowadays we intensively use tablets or laptops to access on-line banking account or make purchase over the Internet. Moreover, we take the airplanes or cars, which all embed electronic systems. Due to all these dependencies, computer systems must be reliable and secure.

Hardware/Software (HW/SW) interfaces are pervasive in these computer systems. For example, about 70% Linux kernel implements device drivers [14] for operating hardware devices and in Windows XP, there are over 35,000 device drivers with over 100,000 versions of hardware devices [43]. However, HW/SW interfaces are unreliable and insecure. Most system failures are caused by incorrect interactions between the devices and their drivers [51]. In Windows XP, about 85% system failures are caused by driver errors [55] and there are seven times more driver failures than the errors caused by the rest part of Linux kernel [14]. To exacerbate these matters, many of these failures are transient. They disappear on a system reboot, often to resurface at a later time. What is even worse, the pervasive, deeply embedded, and strongly connected nature of these systems makes

them increasingly vulnerable to malfunctions, malicious attacks, and tampering. Regarding to the error-prone and vulnerable nature of HW/SW interfaces, effective HW/SW interface reliability and security assurances are highly desired.

1.2 PROBLEM STATEMENT

Assuring HW/SW interface reliability and security is difficult, not only due to the intrinsic complexity of HW/SW interfaces, but also because of various challenges posted at different stages of the system development life cycle. Generally, effective HW/SW interface assurance is highly needed in two major stages of the system life cycle: (1) system validation/testing stage; (2) system deployment stage. At the system validation/testing stage, hardware and software are integrated once the first version of the silicon hardware prototype is available. Such a stage where validations are conducted on the real silicon prototype is also denoted as the post-silicon validation stage. At the system deployment stage, HW/SW interfaces have been released and deployed with the system to the end-users. We discuss the difficulties and challenges at these two stages respectively.

1.2.1 Challenges at Post-silicon Validation Stage

New computer systems like smartphones and tablets, are entering the market place at an ever-accelerating pace. This brings enormous pressures on the product development teams to shorten the *time-to-market*. Moreover, according to recent industry reports [29], validation accounts for nearly 60% of the overall product cost. At the post-silicon validation stage, HW/SW integration validation, validating hardware and software together, is a major component of system validation. A recent study [4] indicates that the cost of HW/SW integration validation has experienced significant increases. Therefore, regarding to the time-to-market pressure,

effective HW/SW integration validation is required. However, currently HW/SW integration validation largely involves ad-hoc and manual work. There are several key challenges:

1. **Lack of HW/SW interface observation.** HW/SW integration validation often relies on testing the entire system with high-level application test scenarios. However, HW/SW interfaces are often not sufficiently observed and certain interface bugs escape detection. For example, unspecified bit-flipping in hardware interfaces often posts security threats and incurs unnecessary power consumptions. However, without observing these bits in a systematic and efficient manner, these bugs are not detected.

2. **Difficulty in attributing HW/SW interface bugs.** When a bug is discovered in HW/SW integration validation, it is often unclear if it is a hardware or software bug, due to the close involvement and interaction of both hardware and software. For example, an invalid software command to the hardware could trigger the hardware to hang. However, such a bug usually appears as a hardware bug rather than a software bug as the hardware stops responding to any new command.

3. **Difficulty in debugging HW/SW interfaces.** Hardware interacts with its control software frequently, producing a huge number of I/O events. To troubleshoot, the engineers usually have to sift through thousands of I/O events and analyze them manually. To exacerbate this situation, as hardware and software share a large range of I/O interface registers or memory, it is often difficult to pinpoint the location of the bug. For example, Intel PCI Ethernet adapter e1000 has 128KB I/O interface memory and produces more than ten thousands of I/O events just for bringing up the driver. When an

error occurs, engineers often manually analyze each driver requests, which incurs significant human effort.

1.2.2 Challenges at System Deployment Stage

At the system deployment stage, HW/SW interfaces are still vulnerable even after many iterations of validation and testing. There are two major reasons. First, transient hardware failures are common. According to several reports [3, 5, 48], a significant number of reported failures cannot be reliably reproduced under the same stimuli triggering the failures. Most of them are transient errors. Second, recently HW/SW interfaces have become a major target of malicious attacks, entailing serious security threats. For example, the infamous Stuxnet worm specifically targets the interactions between programmable logic controllers and their control software, ushering in a new era for virus and worm attacks [59]. Furthermore, globalization of computer system production generates major concerns about potential security backdoors planted in devices and drivers, which are increasingly produced by third-party suppliers who may not be fully trustworthy [17].

Summary. Given the ubiquity and seriousness of these challenges at both the post-silicon validation stage and the system deployment stage, it is highly desired to develop systematic methods to validate HW/SW interfaces and automatically detect and analyze interface bugs. Moreover, even after extensive validation, HW/SW interfaces still need to be continuously protected against hardware transient failures and malicious attacks.

1.3 OVERVIEW OF OUR APPROACH

In this dissertation, we present a comprehensive solution for assuring HW/SW interface reliability and security over the life cycle of computer systems. This

solution is composited of two major parts: (1) HW/SW co-validation framework; (2) Two HW/SW interface assurance schemes: *post-silicon HW/SW co-validation* and *HW/SW co-monitoring*. The HW/SW co-validation framework is central to our assurance solution, which validates HW/SW interactions. The two assurance schemes apply the co-validation framework in different fashions at the post-silicon validation stage and the system deployment stage respectively.

1.3.1 HW/SW Co-validation Framework

Our co-validation framework employs a key technique, conformance checking, which checks the conformance between a device and its reference model. The general work flow of conformance checking has three steps: (1) recording the driver requests issued to the device and the device interface state before each request. Essentially we record a sequence of driver requests with device interface states. We denote such a sequence as a *device trace*; (2) executing the reference model by taking the recorded driver request sequence; (3) checking if there are any inconsistencies in interface states between the device and the reference model. By discovering their inconsistencies, conformance checking validates device interface implementations of both the device and the reference model. Through conformance checking, the reference model shadows the device execution. Beyond conformance checking, we conduct property checking which verifies system properties over the reference model and driver interactions. As the reference model shadows the device execution, properties hold on the reference model and driver interface also hold on the device/driver interface. Property checking discovers both device and driver violations of HW/SW interface protocols. Through the two-tier checking infrastructure, our framework essentially validates the device/driver interactions.

Workflow. Figure 1.1 presents the workflow of our co-validation approach. We

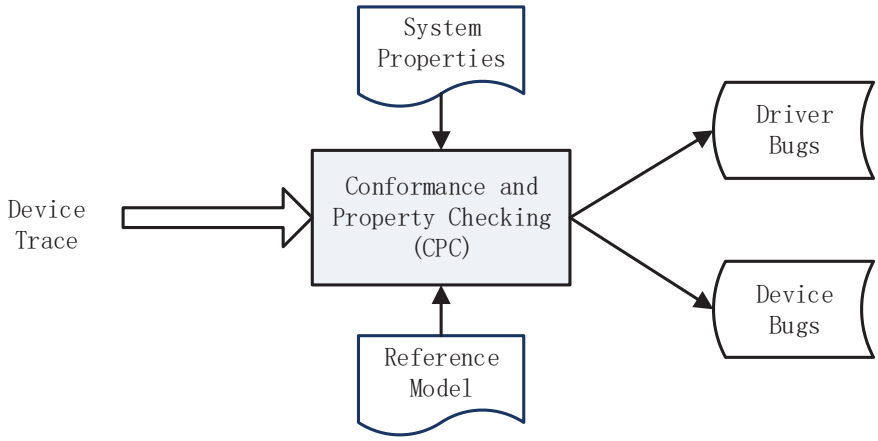


Figure 1.1: Workflow of conformance and property checking

implement conformance and property checking in the engine, conformance and property checker (CPC). It takes a device trace captured from the target device/-driver interface, the reference model, and the system properties as inputs. CPC carries out conformance and property checking over the device trace and outputs driver and device bug reports. CPC can be conducted in two manners: (1) *off-line checking:* the device trace is saved into trace files and CPC does conformance and property checking off-line; (2) *on-line monitoring:* the device trace is captured from device/driver interface at runtime while CPC does conformance and property checking over the captured device trace simultaneously. Based on the two manners, we develop two assurance schemes for different stages of the system life cycle respectively (see Section 1.3.2).

Our framework entails four major techniques which are the key technical contributions of this dissertation research. These techniques are described as follows.

1. **Conformance checking with reference models.** Conformance checking detects the inconsistencies between the device and the reference model by simulating the device behaviors over the reference model [38]. We use

a technique, symbolic execution [31], to simulate the device behaviors on the reference model. Symbolic execution is used to overcome the limited observability of hardware silicon. In HW/SW integration validation, the device internal registers are generally not observable. Moreover, the external environment inputs to the device are also hard to capture. Conformance checking models them using variables with symbolic values when replaying the recorded device trace on the device. In this way, symbolic execution covers all the possible values of the internal registers and the external environment inputs.

2. **Property checking.** The property checker verifies these system properties over the device indirectly through the reference model. Based on the conformance between the device and the reference model, the property verified on the reference model and the driver also holds on the device and the driver. Property checking helps detect both device and driver errors in device/driver interactions.

3. **Adaptive concretization.** Symbolic execution helps us overcome the limited observability challenge. However, it introduces significant overhead as it usually explores a large number of program paths, which makes conformance checking a time-consuming process. To address this challenge, we propose an optimization, adaptive concretization, to reduce the overhead of symbolic execution [36]. We exploit the fact that most of the virtual prototype states conforming to the device state are generated by an execution path accessing none of or only a few of symbolic values. Adaptive concretization eliminates unnecessary symbolic values to prune unnecessary paths explored by symbolic execution.

4. **Extended conformance checking of DMA interfaces.** The original conformance checking aims to detect inconsistencies on device interface registers. However, besides interface registers, Direct Memory Access (DMA) I/O interfaces are also an important type of HW/SW interfaces. To validate DMA interfaces, we extend the conformance checking by capturing DMA interface states and detecting DMA interface inconsistencies between the device and the reference model [37].

1.3.2 HW/SW Interface Assurance Schemes

Our solution aims to provide HW/SW interface assurance over the computer system life cycle. As Section 1.2 illustrated, HW/SW interface assurance is required in two major stages of the system life cycle: the post-silicon validation stage and the system deployment stage. Moreover, each stage has specific requirements for assurance since the system operates under different environments and faces to different kinds of users. We list these requirements as follows.

1. **Requirements of post-silicon validation stage.** At the post-silicon validation stage, both hardware and software evolve over many iterations. This stage requires the integration validation to fully check each version of HW/SW interfaces and detect as many bugs as possible from both sides. While the validation is conducted, a large amount of test cases are issued to manipulate the HW/SW interface in different ways. As a result, a significant number of traces are generated for analysis. Therefore, post-silicon validation is often heavyweight and is required to be effective in detecting errors.

2. **Requirements of system deployment stage.** At the deployment stage, the device and its driver have been released. The device and the driver are

embedded into the production system and utilized by the end users. It is often impossible to deploy the development-time validation tools with the device and the driver, and the validation overhead must be minimum. Therefore, HW/SW co-validation in this stage must be lightweight and detects faults under runtime performance constraints.

To meet the different requirements, we realize our HW/SW co-validation approach in two assurance schemes respectively: (1) post-silicon HW/SW co-validation for the development stage; (2) HW/SW co-monitoring for the deployment stage. Figure 1.2 illustrates how the two assurance schemes fit into the system life cycle. They utilize the same underlying techniques, conformance and property checking. However, there are three key differences between these two schemes.

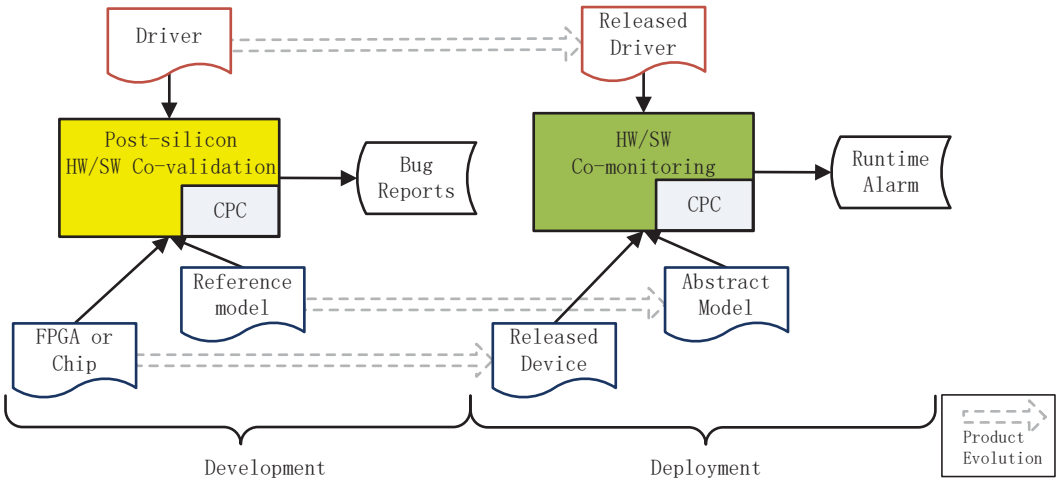


Figure 1.2: Two schemes of HW/SW interface assurance

1. **On-line and off-line.** As Section 1.3.1 illustrated, conformance and property checking can be carried out in two manners: off-line and on-line. Post-silicon HW/SW co-validation is conducted in an off-line fashion. Since at the

post-silicon validation stage, a significant number of I/O events is produced under intensive testing, the off-line approach does not introduce unnecessary runtime overhead and would not be disrupted by the system failures. Oppositely, HW/SW co-monitoring is conducted as an on-line approach, as the system failures should be caught immediately at runtime to protect the target system.

2. **Target device/driver interfaces.** Post-silicon HW/SW co-validation validates the preliminary versions of drivers and the device prototypes which are silicon chips. The co-validation mainly focuses on detecting design flaws in device/driver interfaces. HW/SW co-monitoring monitors the released devices and drivers which have passed intensive testing process. Therefore, HW/SW co-monitoring mainly focuses on detecting malicious exploits and transient errors across device/driver interfaces at runtime.

3. **Reference models.** Comparing to our post-silicon HW/SW co-validation, HW/SW co-monitoring uses a lightweight reference model which abstracts unnecessary implementation details. The abstract reference model helps reduce runtime overhead introduced by conformance and property checking.

Our two assurance schemes have been realized in our two software tools, DCC (Device Conformance Checker) and CoMon (HW/SW Co-monitoring) which implement our HW/SW co-validation approach. We applied our two software tools to four real industry designs. DCC and CoMon discovered 42 non-trivial bugs from devices, drivers, and the reference models. Moreover, CoMon has successfully detect all the malicious attacks across HW/SW interfaces.

1.4 RELATED WORK

This dissertation work is related to HW/SW interface assurance, driver testing and monitoring, device validation and testing, and symbolic execution with its related optimizations.

1.4.1 HW/SW Interface Assurance

There has been much research on HW/SW interface assurance in the pre-silicon verification stage. HW/SW co-verification and HW/SW co-simulation are two mainstream techniques. In HW/SW co-verification, model checking is widely used [33, 39, 60, 41]. It verifies properties by analyzing the interface implementation statically; however, it often encounters the state explosion problem. Our co-validation framework conducts validation over the execution trace captured at runtime which largely avoids the state explosion problem. Research on co-simulation [8, 20, 23, 25, 47, 50, 52] typically utilizes design models of the hardware and does not directly work with the implementation of the hardware/software interfaces. Moreover, neither co-verification nor co-simulation is conducted in the real environment, particularly on the real silicon devices. Therefore, how to eliminate the false negatives and reproduce the detected bugs is often a major challenge. Our approach is conducted on the real devices and drivers, all the detected bugs are real bugs which have already occurred in the runtime system and these bugs can be reproduced by replaying the I/O event sequence.

1.4.2 Driver Testing and Monitoring

Driver testing includes static analysis and dynamic testing. SDV [6] tests Windows device drivers through static analysis of driver programs. It discovers bugs

that are related to incorrect usage of Windows Kernel API. This approach requires Windows driver source code. Dynamic testing tools, such as Driver Verifier [42], verifies the driver over the driver concrete execution trace. DDT [34], and SymDrive [49], discover driver bugs by simulating different inputs to the target driver. Both DDT and SymDrive use symbolic execution, to explore driver paths during testing. Runtime monitoring is an alternative approach to driver reliability assurance. Nooks [54] uses a mechanism called "shadow driver" to monitor the runtime behaviors of the target Linux driver and replace the real driver to handle driver exceptions. A similar approach [30] monitors the driver at runtime and checks if the driver can survive when taking the faulty device inputs. Several other approaches [58, 19, 61, 24] monitor the driver and use software module isolation techniques to prevent driver errors from affecting operating systems. All such driver quality assurance research focuses on detecting memory access failures, invalid kernel API usages, and how to protect the kernel from driver failures. They seldom consider the device behaviors when testing the driver. Other approaches [30, 34] do analyze the hardware interface inputs at certain points of the driver execution. However, without monitoring the critical part of device interfaces and fully exploring the device states, device and driver violations of the HW/SW interface protocol are often missed.

1.4.3 Device Testing and Validation

Device testing and validation are usually carried out at the post-silicon stage. Post-silicon validation is performed on silicon prototypes and testing devices. A significant amount of research has been focused on detecting and localizing bugs in silicon chips. A major difficulty of post-silicon bug detection and localization is the limited observability of silicon hardware internals. Several approaches [1, 46] have

developed hardware on-chip monitors to collect hardware execution traces with internal signals. Assertion-based verification [11, 26, 44] and formal method [18] have been used to analyze and debug the execution traces from on-chip monitors. Our approach also works on detecting and troubleshooting post-silicon bugs. Instead of validating internal implementations of silicon hardware, we focus on HW/SW interfaces.

In summary, current device and driver testing/validation methods mainly focus on one side of the HW/SW interface, rather than validating their integration together. However, as devices and drivers are highly correlated by nature and their interactions often follow complicated protocols. Discrete testing and validation are not effective in detecting HW/SW interface errors.

1.4.4 Symbolic Execution

Symbolic execution [31] is widely used for software testing. SAGE [22], KLEE [12], and S2E[13] use symbolic execution to test software systems that intensively interact with environments. Other tools [57, 53, 7] also employ symbolic execution to generate test cases for testing software programs. Symbolic execution often suffers from the path explosion problem. There has been many research targeting reducing the overhead of symbolic execution. A major effort has been to avoid path explosions by pruning redundant paths. RWSet [10] and path subsumption [2] employ a similar heuristic whereas a path which is identical to the one previously explored can be safely pruned. In [35], a method is proposed for automatically merging states to reduce the number of paths explored in symbolic execution. Several other approaches [21, 53, 56] leverage the benefits of concolic execution to partially concretize the target programs thereby the number of explored paths is reduced. In our adaptive concretization, we reduce the number of explored paths

by concretizing some of the variables with symbolic values.

1.5 DISSERTATION OUTLINE

The reminder of this dissertation is organized as follows. Chapter 2 introduces the background of our research. Chapter 3 presents our post-silicon HW/SW co-validation including conformance and property checking. Chapter 4 elaborates on how we extend our post-silicon HW/SW co-validation framework to validate DMA interfaces. Chapter 5 presents our optimization algorithm of conformance checking. Chapter 6 presents the HW/SW co-monitoring for HW/SW interface assurance at the system deployment stage. In Chapter 7, we conclude and discuss the future directions.

Chapter 2

BACKGROUND

In this chapter, we introduce some relevant concepts: QEMU virtual devices which we adopt as our virtual prototypes, Formal Device Model (FDM) which is used as the reference model in HW/SW co-monitoring, and symbolic execution with which we replay driver requests on virtual devices and FDMs.

2.1 QEMU AND VIRTUAL DEVICES

QEMU [9] is a virtual machine that can emulate different processor architectures, such as x86, SPARC, and ARM. It also emulates virtual devices for different peripheral devices, e.g., network adapters and mass storage devices.

A QEMU virtual device is a software component integrated into QEMU. We illustrate the virtual device concept with the Intel e1000 network adapter, a PCI (Peripheral Component Interconnect) device. As Figure 2.1 shows, the e1000 virtual device has the following major components:

- PCI device state, as defined by $E1000State$, which keeps track of the PCI device state;
- Device configuration, as defined by $e1000\_info$, which stores the PCI configuration information for this device (multiple configurations may be provided);
- PCI device functions: (1) the entry functions such as $e1000\_mmio\_writel$

```
typedef struct E1000State_st {
    PCIDevice dev;
    NICState *nic;
    NICConf conf;
    uint32_t mac_reg[0x8000];
    uint16_t phy_reg[0x20];
    uint16_t eeprom_data[64];
    ......
} E1000State;
```

```
static PCIDeviceInfo e1000_info = {
    .qdev.name  = "e1000",
    .qdev.desc  = "Intel Gigabit Ethernet",
    .vendor_id  = PCI_VENDOR_ID_INTEL,
    .device_id  = E1000_DEVID,
    .revision   = 0x03,
    .class_id   = PCI_CLASS_NETWORK_ETHERNET,
    ......
};
```

```
static void e1000_mmio_writel(void *opaque, phys_addr_t addr, uint32_t val)
{
  switch(addr){
    case E1000_TDT:
          set_tdt(s, addr, val); break;
    ......
  }
}
static void set_tdt(E1000State *s, int index, uint32_t val)
{
  s->mac_reg[index] = val;
  start_xmit(s);
}
static void start_xmit(E1000State *s)
{ ...... }
static ssize_t
e1000_receive(VLANClientState *nc, const uint8_t *buf,  size_t size)
{ ...... }
```

Figure 2.1: Excerpts from the e1000 QEMU virtual device.

which are invoked by the QEMU VM when the driver issues I/O commands;(2) the interface functions such as *set_tdt* which are invoked through the entry functions; (3) the module functions, for example, *start_xmit* and *e*1000_*receive*, which model the device internal transactions such as packet transmission and reception.

2.2 FORMAL DEVICE MODEL

A FDM is an executable transaction-level model, specifying HW/SW interfaces and hardware functionalities [41]. It is derived from the device specification and written in a restricted subset of the C language with three key extensions: transaction, non-determinism, and relative atomicity.

Transaction. The FDM focuses on the design logic rather than the implementation details of HW/SW interfaces and hardware functionalities. For example, a data-transfer command is usually processed in multiple clock cycles; however, from the perspective of the software, it may only be necessary to describe this command as one hardware state transition. We define a hardware transaction to represent a hardware state transition in an arbitrarily long but finite sequence of clock cycles. Hardware transactions are atomic to software.

Non-determinism. A FDM utilizes non-determinism mainly in two ways: (1) updating the state variables, which contributes to the data flow of the specification; (2) deciding the conditions of branches or loops, which contributes to the control flow of the specification. For both ways, the use of non-determinism abstracts away unnecessary details. For example, one important utilization of non-determinism is how a FDM models the hardware concurrency.

- *Non-deterministic interleaving.* Hardware is concurrent in nature. For example, a network card processes driver commands and receives data concurrently. To specify such hardware concurrency, FDMs use a method, namely **non-deterministic interleaving**, which has three steps: (1) identify the concurrent modules (e.g., command unit, receive unit, etc.) of the target hardware device; (2) specify the modules using separate C functions which are defined as module functions; (3) non-deterministically invoke these module functions in a hardware transaction function. When the transaction function is executed multiple times, these module functions are executed in a non-deterministic sequence. From the view point of software, the effect of hardware concurrency is modeled by the set of hardware states after non-deterministic many executions of the hardware transaction function.

Relative atomicity. Relative atomicity has two key ideas: (1) hardware transactions are atomic from the viewpoint of software; and (2) Interrupt Service Routines (ISRs) are atomic to other lower-priority software routines. In device/driver interactions, when hardware fires an interrupt, the OS calls the ISRs that are registered in the interrupt vector table sequentially until an ISR acknowledges its ownership of the interrupt. During this process, only one ISR can run at a time and other hardware interrupts are suppressed. The interrupted thread can continue its execution only after the interrupting ISR terminates.

2.3 BÜCHI PUSHDOWN SYSTEM (BPDS)

2.3.1 Büchi Automaton (BA).

A *BA* $\mathcal{B}$ [32] is a non-deterministic finite state automaton accepting infinite input strings. Formally, $\mathcal{B} = (\Sigma, Q, \delta, q_0, F)$, where Σ is the input alphabet, Q is the

finite set of states, $\delta \subseteq (Q \times \Sigma \times Q)$ is the set of state transitions, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. $\mathcal{B}$ accepts an infinite input string iff it has a run over the string that visits at least one of the final states infinitely often. A run of $\mathcal{B}$ on an infinite string s is a sequence of states visited by $\mathcal{B}$ when taking s as the input. We use $q \xrightarrow{\sigma} q'$ to denote a transition from state q to q' with the input symbol σ. A FDM is modeled as a BA, capturing device behaviors.

2.3.2 Labeled Pushdown System (LPDS).

A *LPDS* $\mathcal{P}$ [40] is a tuple $(I, G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$ where I is the input alphabet, G is a finite set of global states, Γ is a finite stack alphabet, $\Delta \subseteq (G \times \Gamma) \times I \times (G \times \Gamma^*)$ is a finite set of transition rules, and $\langle g_0, \omega_0 \rangle$ is the initial configuration. LPDS can take inputs, which is different from PDS. A LPDS transition rule is denoted as $\langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g', w \rangle$, where $\tau \in I$ and $((g, \gamma), \tau, (g', w)) \in \Delta$. A configuration of $\mathcal{P}$ is a pair $\langle g, \omega \rangle$, where $g \in G$ is a global state and $w \in \Gamma^*$ is a stack content. The set of all configurations is denoted by $Conf(\mathcal{P})$. A device driver is modeled as a LPDS.

2.3.3 Büchi Pushdown System (BPDS).

A *BPDS* $\mathcal{BP}$ [40] is defined as the Cartesian product of a BA $\mathcal{B}$ and a LPDS $\mathcal{P}$: (1) the input alphabet of $\mathcal{B}$ is defined as the power set of the set of propositions that may hold on a configuration of $\mathcal{P}$ (i.e. a symbol in Σ is a set of propositions); (2) the input alphabet of $\mathcal{P}$ is defined as the power set of the set of propositions that may hold on a state of $\mathcal{B}$ (i.e. a symbol in I is a set of propositions); and (3) two labeling functions is defined as follows:

- $L_{\mathcal{P}2\mathcal{B}}: (G \times \Gamma) \rightarrow \Sigma$, associates the head of a LPDS configuration with the set of propositions that hold on it. Given a configuration $c \in Conf(\mathcal{P})$, we

```
    void foo(int a, int b)
    {
1:      if (a > 5)
2:          b = b + 1;
3:      else
4:          b = b - 1;
    }
```

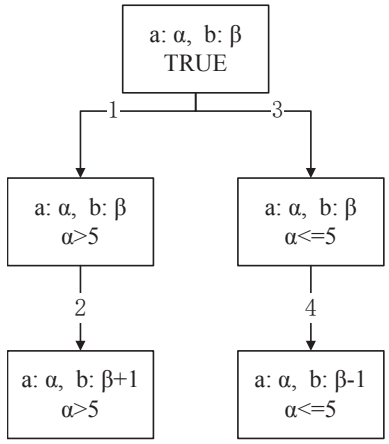


Figure 2.2: An example of symbolic execution.

write $L_{\mathcal{P}2\mathcal{B}}(c)$ instead of $L_{\mathcal{P}2\mathcal{B}}(head(c))$ for simplicity.

- $L_{\mathcal{B}2\mathcal{P}}: Q \rightarrow I$, associates a state of $\mathcal{B}$ with the set of propositions that hold on it.

$\mathcal{BP} = ((G \times Q), \Gamma, \Delta', \langle (g_0, q_0), \omega_0 \rangle, F')$ is constructed by taking the Cartesian product of $\mathcal{B}$ and $\mathcal{P}$. A BPDS rule $\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle \in \Delta'$ iff $q \xrightarrow{\sigma} q' \in \delta$, $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle g, \gamma \rangle)$ and $\langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g', w \rangle \in \Delta$, $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q)$. A configuration of $\mathcal{BP}$ is referred to as $\langle (g, q), \omega \rangle \in (G \times Q) \times \Gamma^*$. The set of all configurations is denoted as $Conf(\mathcal{BP})$. The labeling functions define how $\mathcal{B}$ and $\mathcal{P}$ synchronize with each other. $\langle (g_0, q_0), \omega_0 \rangle$ is the initial configuration. $\langle (g, q), \omega \rangle \in F'$ if $q \in F$. Basically, the combination of the device and the driver is modeled as a BPDS, where the driver is modeled as LPDS, the device is modeled as BA, and their interactions are captured by labeling functions $L_{\mathcal{P}2\mathcal{B}}$ and $L_{\mathcal{B}2\mathcal{P}}$. A path of $\mathcal{BP}$ is a sequence of BPDS configurations, $c_0 \Rightarrow_{\mathcal{BP}} c_1 \ldots \Rightarrow_{\mathcal{BP}} c_i \Rightarrow_{\mathcal{BP}} \ldots$, where $c_i \in Conf(\mathcal{BP}), i \geq 0$.

2.4 SYMBOLIC EXECUTION

Symbolic execution [31] executes a program with symbolic values as inputs instead of concrete ones and represents the values of program variables as symbolic expressions. Consequently, the outputs computed by the program are expressed as functions of input symbolic values. The symbolic state of a program includes the symbolic values of program variables, a path condition, and a program counter. The path condition is a Boolean expression over the symbolic inputs; it accumulates constraints which the inputs must satisfy for the symbolic execution to follow the particular associated path. The program counter points to the next statement to be executed. A symbolic execution tree captures the paths explored by the symbolic execution of a program: the nodes represent the symbolic program states and the arcs represent the state transitions.

We use the program in Figure 2.2 to illustrate how symbolic execution is conducted. At the entry, a and b have symbolic values α and β, respectively, the path condition is `TRUE`, and the program counter is 1. At the branching point, the path condition is updated with conditions on the inputs to select between the two alternative paths. At an assignment statement, the symbolic value of the relevant variable is updated.

Chapter 3

POST-SILICON HW/SW CO-VALIDATION

3.1 MOTIVATION AND OVERVIEW

Post-silicon validation is a critical stage in the system life cycle. In this stage, not only hardware silicon validation is conducted, but also HW/SW integration validation. A recent study [4] indicates that the cost of HW/SW integration validation has increased significantly. As the complexities of systems grow, there are several key challenges in the post-silicon integration validation:

1. **Lack of HW/SW interface observation.** In HW/SW integration validation, HW/SW are combined together and treated as a black box. To test hardware and software combination, some common test scenarios are created and issued from high-level applications. The testers can discover some bugs by observing if there are any system errors or crashes. Nevertheless, HW/SW interfaces are often not sufficiently observed and certain interface bugs will not be detected.

2. **Difficulty in attributing HW/SW interface bugs.** When a bug is discovered in HW/SW integration validation, it is often unclear if it is a hardware bug or a software bug due to the close involvement and interaction of both hardware and software.

3. **Difficulty in debugging HW/SW interfaces.** Hardware interacts with its control software frequently, producing a huge number of I/O events. To

troubleshoot, the engineers usually have to sift through thousands of I/O events and analyze them manually.

Regarding to these serious challenges of the HW/SW integration validation, systematic and effective validation approaches are highly desired to validate both hardware and software together. To conduct an effective co-validation over HW/SW interfaces, a major challenge is how to design a reference model which can effectively track the device behaviors. Recently, virtual prototyping has emerged as a promising technique to enable early software development. A virtual prototype is a system-level, executable software model of a hardware device with full observability. The device interface modeled by the virtual prototype is required to be functionally equivalent to that of the silicon device. Thus, virtual prototypes can be used as reference models and have a major potential in facilitating HW/SW co-validation.

In this chapter, we present a HW/SW co-validation framework for post-silicon HW/SW integration validation. We utilize the virtual prototype of the device as a reference model for validating HW/SW interfaces. As Figure 3.1 shows, there are two stages in our framework. We illustrate the two stages as follows.

1. **Runtime recording:** In the runtime recording stage, the device/driver interactions are recorded, including the driver requests issued to the device and the device interface state before each request. We denote the recorded sequence of driver requests associated with the corresponding device interface states as **device trace**. The framework saves the recorded device trace into a trace file and inputs it to the conformance and property checker (CPC).

2. **Off-line checking:** In the off-line checking stage, CPC takes the trace file, the virtual prototype of the device, and system properties governing the device/driver interactions as inputs. CPC performs conformance checking and property checking. As a result, it reports the discovered bugs on device/-driver interfaces.

CPC implements a two-tier checking infrastructure: conformance checking and property checking. Conformance checking checks the interface register conformance between the device and its virtual prototype, thereby validating the interface implementations of both sides. Through conformance checking, the virtual prototype shadows the device execution. Property checking leverages the virtual prototype to expose the device state transitions and verifies the properties over the virtual prototype/driver interactions. Based on the conformance between the virtual prototype and the device, the properties violated in the virtual prototype-/driver interface are also violated in the device/driver interface. By checking the properties, invalid driver inputs to the device and invalid device interface state are both detected.

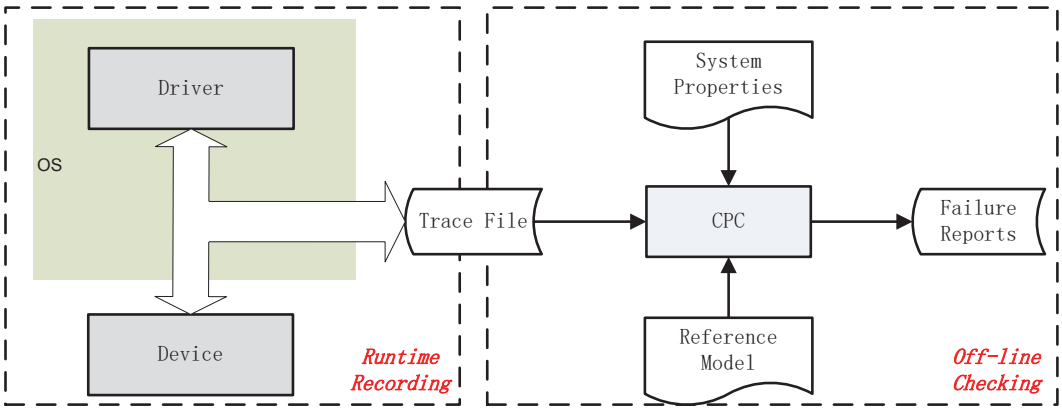


Figure 3.1: Architecture of post-silicon HW/SW co-validation

Outline. The reminder of this chapter is organized as follows. Section 3.2 presents our conformance checking between the device and its virtual prototypes. Section 3.3 illustrates the property checking based on conformance checking. In Section 3.4, we present some implementation details for realizing our HW/SW co-validation framework. Section 3.5 elaborates on experimental results.

3.2 CONFORMANCE CHECKING WITH VIRTUAL PROTOTYPES

We present the basic workflow of conformance checking [38]. As illustrated in Figure 3.2, the workflow has two major components: a trace recorder and a conformance checker.

The trace recorder records the driver request sequence to the device. The conformance checker replays the sequence on the virtual prototype and checks the conformance. The discovered inconsistencies are recorded. An inconsistency record contains the inconsistent registers, the driver request causing the inconsistency, and the virtual prototype execution trace under the driver request.

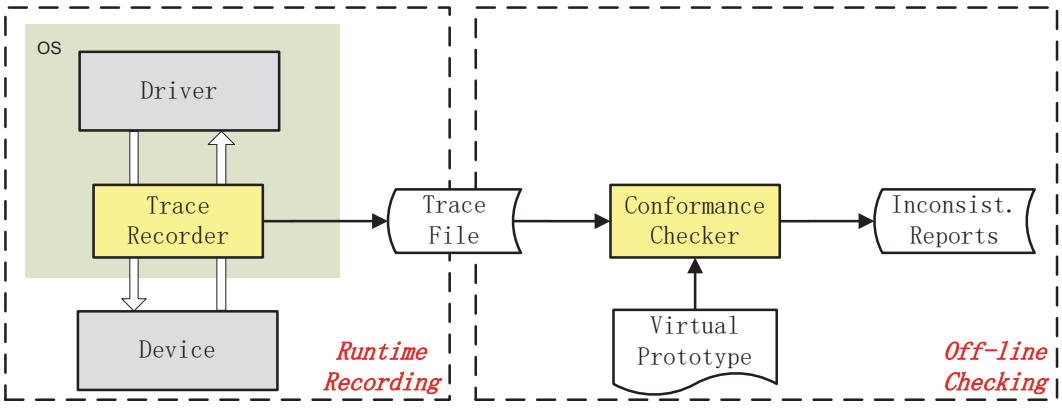


Figure 3.2: Workflow of conformance checking

3.2.1 Preliminaries

Before discussing the details of this workflow, we first introduce our notion of conformance, which is defined between the states of the device and its virtual prototype. The state of the device is determined by the values of its interface and internal registers. The interface registers of the device are observable while the internal registers are generally not observable and are sometimes even unknown. The virtual prototype is a model of the device. It models interface registers of the device with a set R_I of corresponding variables and defines a set R_N of variables to capture device internal behaviors. However, the variables in R_N often have no correspondence with the internal registers of the device. We define a virtual prototype state as follows.

Definition 3.1. *A* **virtual prototype state** *is denoted as* $V=\{V_I, V_N\}$ *where* V_I *is the device interface state, i.e., the assignments to variables in* R_I *and* V_N *is the device internal state, i.e., the assignments to variables in* R_N.

We represent the device state with the same sets of variables: R_I and R_N. The variables in R_I are assigned values observed from the corresponding interface registers of the device. The variables in R_N are assigned symbolic values with no constraints since the device internal is not observable.

Definition 3.2. *A* **device state** *is denoted as* $S=\{S_I, S_N\}$ *where* S_I *is the assignments to variables in* R_I *and* S_N *is the symbolic assignments to variables in* R_N.

A **concrete device state** is a device state whose state variable values are all concrete. A **symbolic device state** is a device state some of whose state variable values are symbolic and there can also be constraints on these symbolic values. A symbolic device state can be viewed as a set of concrete states. In our approach,

we treat both V and S as symbolic states, which can be viewed as two set of concrete device states, denoted as $set(V)$ and $set(S)$ respectively. Based on this generalization, Definition 3.3 defines the conformance between a device state and a virtual prototype state.

Definition 3.3 (state conformance)**.** *A device state S and a virtual prototype state V conform to each other if $set(S) \cap set(V) \neq \emptyset$.*

To compute $set(S) \cap set(V)$, we denote the device state variables as var_1, var_2, ..., var_n and the values of the state variables of S as $Val(var_1)_S$, $Val(var_2)_S$, ..., $Val(var_n)_S$. We construct the expression of S as $Expr(S)$: $(var_1 == Val(var_1)_S) \wedge (var_2 == Val(var_2)_S) \wedge ... \wedge (var_n == Val(var_n)_S)$. Similarly, assume the constraints of V as $Cont(V)$, the expression of V, $Expr(V)$, is $(var_1 == Val(var_1)_V) \wedge (var_2 == Val(var_2)_V) \wedge ... \wedge (var_n == Val(var_n)_V)) \wedge Cont(V)$. Given $Expr(S)$ and $Expr(V)$, $set(S) \cap set(V) \neq \emptyset$ if and only if $Expr(S) \wedge Expr(V)$ is satisfiable.

3.2.2 Trace Recorder

The trace recorder captures: (1) each driver request issued to the device; (2) the device interface state before each driver request is issued. A sequence of such state-request pairs captured on the device can be viewed as a **device trace**. We define such a device trace as $T = \langle S_{I_0}, D_0 \rangle, \langle S_{I_1}, D_1 \rangle, ..., \langle S_{I_n}, D_n \rangle$, where the pair $\langle S_{I_k}, D_k \rangle$ $(0 \leq k \leq n)$ represents a driver request D_k to the current device interface state S_{I_k}.

3.2.3 Conformance Checking Algorithm

The conformance checker replays T on the virtual prototype using symbolic execution. Algorithm 3.1 presents this workflow. It takes a device trace T and a virtual

prototype F as inputs. The conformance checking algorithm works as follows:

1. Initialize the virtual device state V_0 to be S_0 from T' and set $k = 0$.

2. Take the next driver request D_k of T' and symbolically execute the virtual device from V_k on D_k. Symbolic execution may produce a set G of virtual device states.

3. Check the conformance between G and S_{k+1} (see below for details). If not conforming, report an inconsistency; otherwise continue checking.

4. Set the virtual device state V_{k+1} to be the silicon device state S_{k+1}; Increment k and go to step 2.

5. The conformance checker terminates when it finishes the last driver request of T'.

Algorithm 3.1 conformance_checking(T, F)

```
1: T' ← convert_trace(T)
2: /* Take ⟨S_k, D_k⟩ from T'*/
3: for k: 0 → n do
4:    /*Set VP state V_k to be device state S_k*/
5:    V_k ← S_k
6:    /*Symbolically execute VP by taking D_k at V_k state*/
7:    G ← sym_exec(F, V_k, D_k)
8:    H ← conformance_check(G, S_{k+1})
9:    if H == false then
10:      report_incon()
11:   end if
12: end for
```

Major functions in Algorithm 6.1 are described below.

1. Given $T = \langle S_{I_0}, D_0\rangle$, $\langle S_{I_1}, D_1\rangle$, ..., $\langle S_{I_n}, D_n\rangle$, function *convert_trace* generates a new device trace $T' = \langle S_0, D_0\rangle$, $\langle S_1, D_1\rangle$, ..., $\langle S_n, D_n\rangle$, where $S_k (0 \leq k \leq n)$ is a device state derived from S_{I_k}. (cf. Definition 3.2).

2. Function *sym_exec* symbolically executes the virtual prototype and generates a set of virtual prototype states denoted as $G = \{g_i \mid 0 \leq i \leq m\}$.

3. Function *conformance_check* checks the conformance between G and the next device state under D_k, denoted as S_{k+1}. Definition 3.4 defines their conformance. If G and S_{k+1} conform to each other, function *conformance_check* returns *true*, otherwise, it returns $false$.

4. When function *conformance_check* returns $false$, there is an inconsistency and function *report_incon* reports the inconsistency.

Definition 3.4 (Device Conformance)**.** *Given* $G = \{g_i \mid 0 \leq i \leq m\}$ *and* S_{k+1}, *the virtual prototype and the device conform to each other at* D_k *if* $\exists g_i \in G$ *where* $0 \leq i \leq m$, $set(S_{k+1}) \cap set(g_i) \neq \emptyset$.

Discussions. Our conformance definition is essentially the conformance between the interface states of the device and the virtual prototype since the internal variables of S have unconstrained symbolic values. Therefore, our algorithm may not detect internal state non-conformance. Moreover, to reduce symbolic execution complexities, we synchronize the virtual prototype state to the device state after each drive request. This may miss inconsistencies that only surface after several driver requests. Under this conformance definition, our approach is sound theoretically as symbolic execution explores all possible interface states of the virtual prototype. Nevertheless in practice, for practicality and efficiency, our approach

may introduce false positives, i.e., false alarms, due to optimizations of symbolic execution (cf. Section 3.4.4). Furthermore, our approach might also miss some device bugs. For example, if the virtual prototype and the device have a same error, this error will not be discovered. In Section 3.3, we will show property checking can help us detect such errors.

3.3 PROPERTY CHECKING

This section overviews our property checking design [37]. Based on the conformance checking workflow described in Figure 3.2, we build a property checker over the conformance checker in the off-line checking stage. As Figure 3.3 illustrates, the property checker takes a trace file as its input, verifies the system properties over the device state transitions exposed by the conformance checker, and reports property failures.

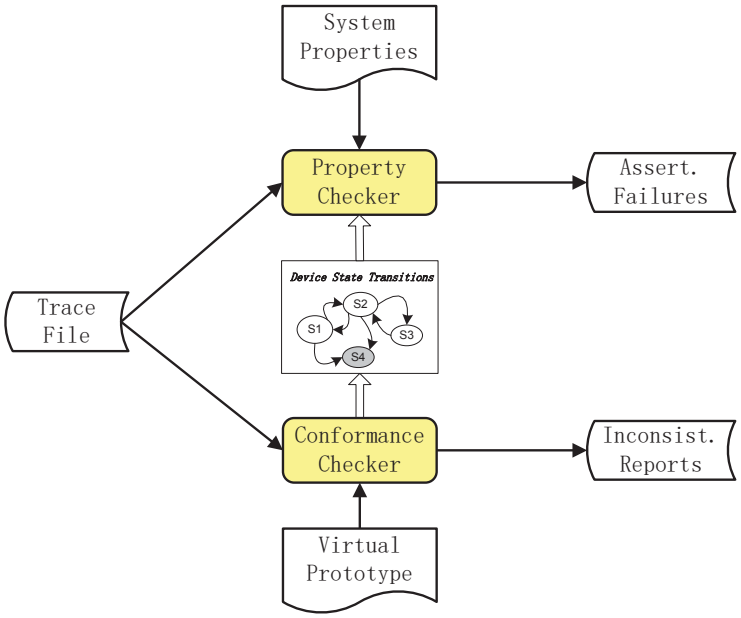


Figure 3.3: Workflow of property checking

Property checking verifies two types of properties: (1) stateless properties,

the properties without involving device states; (2) stateful properties, the properties related to device states. Moreover, each type also contains two categories: (1) device properties specifying how the device should behave in device/driver interactions; (2) driver properties specifying how the driver should behave in device/driver interactions. As examples, we present four properties specified in the eepro100 specification [27] as follows.

Property 1*: If some register bits are marked as "reserved", the driver cannot set these bits.*

Property 2*: If some register bits are marked as "reserved", the device cannot set these bits.*

Property 3*: If the device Command Unit (CU) is not in* `SUSPENDED` *status, the driver cannot send* `RESUME` *to the device.*

Property 4*: If the driver does not require the device to fire an interrupt after the device completes the driver request, the device should never fire such an interrupt.*

Properties 1 and 2 are stateless properties while properties 3 and 4 are stateful properties. Moreover, failures of properties 1 and 3 indicate driver violation of device/driver interface protocols while failures of properties 2 and 4 indicate device errors.

Remark. Conformance checking can detect device violations of device/driver interface protocols. However, as Section 3.2.3 mentioned, conformance checking may miss some device errors. By verifying the device properties, property checking essentially provides a way to detect some of the device errors missed in conformance checking.

3.3.1 Virtual Prototype Instrumentations

As we leverage the virtual prototype to infer the device state transitions, the virtual prototype can be directly used as a validation vehicle. For property checking, we instrument the virtual prototype with assertions generated from specified properties. In this way, while the conformance checker simulates the device behaviors on the virtual prototype, the property checker detects if any assertion fails during the simulation. Currently, we instrument the assertions manually. In future, we will develop a method that allows the users to specify assertions and automatically instruments the virtual prototype with the assertions. Figure 3.4 shows the four assertions corresponding to properties 1, 2, 3, and 4 respectively. A special API function *dcc_assert* is used to specify these assertions.

3.3.2 Detecting Assertion Failures

The property checker evaluates the assertions when the conformance checker executes the virtual prototype. Symbolic execution of the virtual prototype usually explores multiple program paths, we denote such a set of paths as $P = \{p_i \mid 0 \leq i \leq n\}$. Definition 3.5 defines the condition that the property checker detects a property violation.

Definition 3.5 (Property Violation)**.** *Given a property ψ, a set of paths $P = \{p_i \mid 0 \leq i \leq n\}$ explored under a driver request D, ψ is violated under D if $\forall p_i \in P$, the assertion failure of ψ is reachable on p_i.*

The set P represents all the possible device behaviors under the driver request D. Only if all of these possible behaviors lead to the violation of the property ψ, the property checker can ensure there is an property violation in the device/driver interface.

```
static void eepro100_cu_command(EEPRO100State * s, uint8_t val)
{
    // Assertion for property 1
    dcc_assert(!val & RESERVED_BITS);

    // Assertion for property 3
    if (s->cu_state != CU_STATE_SUSPENDED)
      dcc_assert(val != CU_CMD_RESUME);

    ......

   // Assertion for property 2
    dcc_assert(s->mac[CU_CMD] & RESERVED_BITS);
}

static void eepro100_write_mdi(EEPRO100State *s, uint32_t val)
{
   ......
   // Assertion for property 4
   if (!val & MDIC_INT)
   {
      dcc_assert(!s->mac[SCB_INT] & MDI_INT);
   }
}
```

Figure 3.4: Assertions instrumented in the eepro100 virtual device

Discussions. Our property checking has a major advantage in verifying stateful properties. In the state of the art driver implementations, to runtime verify a property related to the device states, the driver has to be instrumented to keep a partial device state machine where only property-related states and corresponding state transitions are modeled. This approach has three limitations: (1) modeling a state machine for every property incurs redundant human efforts; (2) ad-hoc state machine instrumentation is intrusive to the driver implementation; (3) the state transitions inferred by the driver are sometimes out of synchronous with the real device state transitions, as the driver hardly checks the real device states. Our approach leverages the virtual prototype to systematically model and maintain the complete device state machine while the normal workflow of the device/driver interface is not affected. Furthermore, through conformance checking, the virtual prototype is largely guaranteed to be synchronous with the device.

3.4 IMPLEMENTATION

This section presents the techniques for implementing our HW/SW co-validation approach.

3.4.1 Selective Capturing

The trace recorder captures values of the interface registers of the device. However, it is difficult to capture all interface registers since a device often has a large range of interface registers. For example, Intel e1000 network adapter, a PCI device, has 128KB of interface registers. Capturing all these registers incurs excessive memory transactions, which will heavily degrade the system performance. To address this problem, we propose a method, namely selective capturing, which captures a smaller set of important registers rather than the complete set.

To decide which registers to capture, we statically analyze the virtual prototype [16]: symbolically execute the virtual prototype by using symbolic inputs and record the registers accessed in execution. As the registers can be accessed by using symbolic addresses, which may lead to an unnecessarily large range of registers to record. Therefore, we only record the registers accessed by concrete addresses. This may miss certain registers. As a supplement, we allow the user to specify which registers they want to capture. Selective capturing does not affect the soundness of our approach although it may miss inconsistencies.

3.4.2 Incremental Trace Recording

The trace recorder captures the device trace at runtime, which is a sequence of driver requests associated with device interface states. In practice, testing a device usually produces thousands of driver requests in a short period of time. As each driver request corresponds to a set of captured registers, saving the complete device trace would occupy a significant amount of memory and disk space. Moreover, in post-silicon HW/SW co-validation, the trace recorder is usually running on a testing machine with the target device while the conformance checker can be running on any other machines. Therefore, transferring a trace file from the testing machine to the checking machine costs significant time if the file size is large.

We observe that in a device trace, between two consecutive driver requests, there is only a small number of interface registers whose values are changed. Based on this observation, we develop an incremental trace recording method which only records the interface registers whose values are changed instead of recording a complete set of selected registers. This method has three steps:

1. Before the first driver request D_0 is issued, the trace recorder captures the interface state S_{I_0} and saves it in the device trace T.

2. Before the driver requests D_k where the current device interface state is S_{I_k} and $0 < k \leq n$, the trace recorder computes $\Delta_{S_{I_k}}$, where $\Delta_{S_{I_k}} = \mathcal{D}(S_{I_k}, S_{I_{k-1}})$. The function $\mathcal{D}$ returns the different registers and their values between S_{I_k} and $S_{I_{k-1}}$. The trace recorder saves $\Delta_{S_{I_k}}$ in T.

3. Given a device trace $T_\Delta = \langle S_{I_0}, D_0 \rangle$, $\langle \Delta_{S_{I_1}}, D_1 \rangle$, ..., $\langle \Delta_{S_{I_n}}, D_n \rangle$, for $\Delta_{S_{I_k}} (0 < k \leq n)$, the conformance checker recovers S_{I_k} by using function R where $\mathcal{R}$: $\{S_{I_{k-1}}, \Delta_{S_{I_k}}\} \rightarrow S_{I_k}$. In this way, the conformance checker recovers T based on T_Δ and conduct the conformance checking in the native way.

3.4.3 Harness Generation for Virtual Prototypes

A virtual prototype is not a stand-alone program, which is executed as part of the virtual platform. Therefore, we need an execution harness for symbolically executing the virtual prototype. We generate an execution harness based on the concepts of non-deterministic interleaving and symbolic inputs.

- *Non-deterministic interleaving.* As Section 2.2 illustrates, to capture the hardware concurrency, it requires non-deterministic many executions of a loop where the module functions are invoked non-deterministically. We define such a loop as the **main loop** of the execution harness. The condition of the main loop is a non-deterministic choice and module functions are invoked non-deterministically in the main loop.

- *Symbolic inputs.* As outside environment inputs are not captured from the device, we assign symbolic values to these input variables so that symbolic execution can cover the possible inputs from the outside environment.

```
......
dcc_make_symbolic(buff, BUFF_SIZE, "buff");
dcc_make_symbolic(size, sizeof(uint32_t), "size");

//Non-deterministic many executions
while(choice()) {

    //Non-deterministic interleaving
    switch (choice()) {

      // Respond to write/read registers
     case 0:     Access_Register(); break;

      // Receive packets
      case 1:     e1000_receive(nc, buff, size); break;

      // Do nothing
      default:   break;
}
......
```

Figure 3.5: Excerpts of execution harness of e1000 virtual prototype

Example. We illustrate harness generation using the e1000 network adapter. Figure 3.5 shows an excerpt from the harness we generate for the e1000 virtual prototype. There are two module functions: (1) `Access_Register`; (2) `e1000_receive`. The function `Access_Register` models how the device responds to a driver request, e.g., writing to or reading from a register. The function `e1000_receive` models how the device receives packets from the network, which takes several input parameters. We call the function `dcc_make_symbolic` to assign symbolic values to the input variables. The function `choice()` implements a non-deterministic choice which returns a symbolic value. In the main loop, the two module functions are invoked non-deterministically.

3.4.4 Termination of Symbolic Execution

Symbolic execution might not terminate when it encounters a loop without a statically known number of iterations, e.g., the main loop in the execution harness. We refer to such a loop as an unbounded loop. To address this issue, we set constant bounds for all such loops in the virtual prototype. We leverage runtime behaviors of the virtual prototype in the QEMU virtual machine to decide the loop bound for each unbounded loop. The method contains three steps:

1. We statically analyze the virtual prototype through symbolically executing the virtual prototype using symbolic inputs, to identify the unbounded loops.

2. When the virtual prototype is running within the QEMU virtual machine, for each unbounded loop identified by static analysis, we record the largest number of iterations that the loop has been executed. If we encounter an unbounded loop while replaying the device trace, we use its recorded maximum number of iterations as its bound.

3. As a supplement, we allow the user to adjust the loop bound for a specific loop. For example, if using a large bound induces high time and memory costs or even path explosions, the user may lower the bound.

Remarks. Loop bounding may lead to false positives since it potentially reduces the virtual prototype behaviors. However, we argue that the false positive ratio is low due to two reasons. First, static analysis shows that for most unbounded loops, increasing the numbers of loop iterations does not affect the virtual prototype interface state. Therefore, the conformance checking result will not be affected most of the time. Second, the loop bounds cover most virtual prototype behaviors if the runtime test cases for identifying loop bounds have a high coverage of the virtual prototype (herein we use the code coverage metrics such as statement coverage). Moreover, a discovered false positive may be eliminated thereafter by the user incrementing the loop bounds. However, since setting the bounds too large may lead to high time and memory costs and even path explosions, sometimes false positives cannot be completely eliminated. Therefore, the user may need to search for a "sweet spot" to achieve minimum false positives with reasonable symbolic execution costs.

3.4.5 Implementation Details

Trace Recorder Implementation

We implement our post-silicon HW/SW co-validation on Linux. The trace recorder is implemented as a Linux kernel library. A standard Linux device driver always calls Linux kernel functions to access its device. For instance, a driver calls function `writel` to write a long integer to a device register. We hook these kernel functions. As a result, the trace recorder is invoked to record the driver requests when the driver calls these functions to issue requests.

Conformance and Property Checking

We construct our conformance and property checker using the symbolic execution engine KLEE [12]. We modify KLEE in three aspects. First, we set the loop bounds during symbolic execution. Second, we realize our own module for conformance and property checking. Third, KLEE is a testing tool rather than just a symbolic execution engine. It provides some functionalities which are unnecessary in our approach. We remove these functionalities from KLEE. For example, KLEE generates test cases for the explored paths, which is not essential for symbolic execution. We remove this functionality to avoid the I/O operations during symbolic execution.

3.5 EVALUATION

In this section, we present our evaluation results including two parts: (1) design flaws discovered in real industry designs; (2) performance of our post-silicon co-validation framework with optimizations of conformance checking.

3.5.1 Experiment Setup

All experiments were conducted on a workstation with a dual-core Intel Pentium D Processor at 3.20 GHz and 4GB of RAM, running Linux with kernel version 2.6.35. The devices evaluated are three types of widely used network adapters. We use their QEMU virtual devices as the virtual prototypes. Information of these devices and their virtual prototypes are summarized in Table 3.1. It also shows the size of the registers we selectively capture in each network adapter. The virtual device size is measured in Lines of Code (LOC). Intel e1000, Intel eepro100, and Realtek rtl8139 virtual devices are included in QEMU 0.15.1 source code. Broadcom bcm 5751 virtual device is newly created following the QEMU 0.15.1 requirements.

Table 3.1: Devices and virtual prototypes for HW/SW co-validation

Devices	Virtual Device Size (LOC)	Selective Captured Size (Bytes)
Intel e1000 Gigabit NIC	2099	1224
Broadcom bcm5751 Gigabit NIC	4519	412
Intel eepro100 10/100M NIC	2178	74

3.5.2 Bug Detection

Inconsistencies and Device Bugs

Conformance checking of our framework discovered 26 inconsistencies between the three network adapters and their virtual prototypes under test: 12 in e1000, 8 in bcm5751, and 6 in eepro100. By analyzing the inconsistency reports generated by the conformance checker, there are 22 bugs from the virtual devices, and 4 bugs from the devices. As the result shows, most of these inconsistencies are caused by the bugs of the virtual devices. This is because on one hand the devices are stable products which have gone through extensive testing and bug-fixing procedures; on the other hand, their virtual prototypes are not heavily tested through any rigorous testing procedures. However, these virtual prototype bugs are still possible to appear in silicon prototypes at the early stage of hardware development, since these bugs are common violations of hardware designs. We believe that if this approach is conducted at the post-silicon validation stage before devices are released, it can also discover many inconsistencies caused by the bugs of devices/prototypes.

Types of device bugs. We summarized the bugs which cause the inconsistencies. As shown in Table 3.2, there are 9 types of device bugs we discovered by analyzing the inconsistencies. Most of these bugs are very common violations of hardware

designs. For example, firing interrupts too many times and failing to fire interrupts are both common defects in hardware devices. We discuss the device bugs and the virtual prototype bugs respectively.

- *Device bugs.* The bugs of the first type are real device bugs. The device updates the register specified as reserved in the device specification. This bug can be serious since it may cause unnecessary device behaviors, expose additional device information, and consume extra power.

Table 3.2: Types of bugs in virtual prototypes and devices

No.	Bug Description	Num.	Device Type	Distribution
1	Reserved Bits/Registers are updated	4	Device	e100, e1000
2	Generate unnecessary interrupts	2	VP	eepro100, e1000
3	Fail to generate interrupts	1	VP	bcm5751
4	Fail to clear interrupts	1	VP	bcm5751
5	Fail to update registers	7	VP	eepro100, e1000, bcm5751
6	Update registers with wrong values	2	VP	e1000
7	Two or more registers are out of sync.	2	VP	bcm5751
8	Registers reset to incorrect values	3	VP	eepro100, e1000, bcm5751
9	Incorrect data types for modeling device states	1	VP	bcm5751
10	VP does not model device concurrency	3	VP	eepro100, e1000, bcm5751

```
static void
set_mdic(E1000State *s, int index, uint32_t val)
{
      ... ...
    s->mac_reg[MDIC] = val |E1000_MDIC_READY;
    set_ics(s, 0, E1000_ICR_MDAC);
}
```

Figure 3.6: Excerpt of e1000 virtual device

- *Virtual prototype bugs.* The bugs of second to fourth types are all related to interrupts. The bugs from the fifth type to ninth type can cause the driver to read incorrect values. These bugs often cause serious driver and system errors or even crashes, and similar device errors have been reported [30].

Consequences of inconsistencies. These inconsistencies can have serious consequences. Here we use an inconsistency found in Intel e1000 as an example. In this scenario, the device driver writes certain values to register `MDIC` to transfer data into the internal module of the device. After the data transfer finishes, according to the value of a specific bit in register `MDIC`, the device determines whether to fire an interrupt.

However, Figure 3.6 shows how the virtual prototype responds under such the scenario by invoking the function `set_MDIC`. In this function, no matter what is the value of register `MDIC`, the virtual prototype always generates an interrupt by invoking the interrupt function `set_ics`. Due to this feature, the driver developed on the virtual prototype may always expect an interrupt after the device finishes transferring data. However, the device does not always generate an interrupt to notify the driver when the data transfer is completed. Therefore, if the driver

is not well written, it will treat no interrupt as an incorrect data transfer in the device, and report an exception by mistake. The driver's normal work flow will be disrupted on the device. By detecting such an inconsistency, our tool helps users easily figure out why the driver does not work properly with the silicon device. This case illustrates how our approach can help post-silicon device/driver co-debugging.

Property Checking and Driver Bugs

By using property checking, we detect two driver bugs shown in Table 3.3. Property checking verified 31 properties in total, of which there are 10 stateless properties and 21 stateful properties. These driver violations are harmful to the system. Updating reserved and read-only registers are likely to incur unnecessary behaviors of the devices.

Table 3.3: Summary of driver bugs

Bug Description	**Num.**	**Bug Types**	**Distribution**
Update interface register reserved bits	1	Stateless Property Violation	eepro100
Update interfac register read-only bits	1	Stateless Property Violation	e1000

Summary. The results demonstrated that our framework addresses the first three key challenges of HW/SW integration validation presented in Section 3.1. First, our framework is effective to detect the design flaws in HW/SW interfaces. For example, the discovered bug of updating the reserved bits can be easily missed if the HW/SW interface is not observed. Second, our framework can easily identify

a HW/SW interface bug as a hardware bug or a software bug. For example, an invalid driver input often appears like a device bug as the device usually hangs under the invalid input. By detecting the invalid input through property checking, the framework clearly identifies this bug as a driver bug. Third, conformance checking and property checking are carried out automatically, which reduces significant human efforts.

3.5.3 Efficiency

We evaluate the efficiency of our approach, in terms of time usages, memory usages, and false positive ratios. We issue four kinds of test cases to the network adapters to collect device traces. These test cases are all common usages of network adapters as shown in Table 3.4. "NIC test-suite" contains a family of typical test cases on network interface controllers (NIC), which manipulate a NIC in different ways, e.g., sending UDP packets and setting MTU size.

Table 3.4: Test cases for evaluating HW/SW co-validation

Test Cases	**Description**
Reset Network Interface	Bring down and then bring up the network interface
Ping	Ping another network interface
Transfer files	Copy large files with total size 3.2 GB
NIC test-suite	A set of typical test cases on NIC

Time and memory usages

We evaluate the time and memory usages of conformance checking. Table 3.5 shows the results. The "Time Usage" column shows the average time usages for

the conformance checker processing each driver request of the device trace collected under the test cases. We also recorded the maximum values of memory usages. Consider that our approach is an offline checking approach, the time usage is acceptable and the memory usage is low.

Table 3.5: Time and memory usages and false positives

Devices	**Test Cases**	**Time Usage (sec)**	**Memory Usage (MB)**	**Inconsistency (Discovered /Verified)**
e1000	Reset NIC	0.24	212.60	8/8
	Ping	2.92	300.00	8/8
	Transfer files	3.11	308.14	**12/9**
	NIC test-suite	3.06	288.23	11/11
bcm5751	Reset NIC	0.19	166.51	9/9
	Ping	2.88	255.16	8/8
	Transfer files	2.87	251.02	**8/6**
	NIC test-suite	2.33	218.65	7/7
eepro100	Reset NIC	0.26	207.73	4/4
	Ping	2.10	220.15	2/2
	Transfer files	2.45	236.77	2/2
	NIC test-suite	2.31	226.84	4/4

False positive ratios

To assess the number of false positives introduced by our optimizations, we verified all the inconsistencies discovered. In the "Inconsistency" column of Table 3.5, we show the numbers of discovered inconsistencies and verified inconsistencies.

Most of the inconsistencies are verified. We encountered false positives in the traces of transferring files on e1000 and bcm5751 (marked as bold). Both virtual prototypes have only one unbounded loop whose number of iterations affects the virtual prototype interface state. The number of iterations of the loop depends on the total size of packets received by the device between two consecutive driver requests. In the virtual prototype, one iteration of the loop would receive a fixed number of packets. Therefore, one iteration of the loop captures the device behaviors when the network traffic is modest. Occasionally when the network traffic is heavy, it requires executing the loop more than once. Therefore, our setting the bound to one produces false positives. Nevertheless, as we adjust the bound by incrementing it to two, all previously encountered false positives are eliminated while the time and memory costs remain modest. This demonstrates that (1) our approach has a low false positive ratio; (2) The supplementary loop bounding method is effective in eliminating false positives.

3.6 SUMMARY

We have presented an approach to HW/SW co-validation at post-silicon stage. This approach entails two checking techniques: (1) conformance checking with virtual prototypes; (2) property checking, which help detect errors in HW/SW interface implementations between a device and its driver. Our co-validation framework can effectively detect the bugs from the devices, the virtual prototypes, and the drivers. Preliminary evaluation shows that our approach is useful and efficient. In three network adapters, we discover many bugs while incurring low memory and time usages. Furthermore, our validation framework has major potential in addressing the key challenges of HW/SW integration validation presented in Section 3.1. First, the framework records and validates the device interface state; thus

the errors in the interface registers are effectively detected. Second, the framework identifies both the device and driver errors over the device/driver interface thereby it can easily attribute device/driver interface bugs as device or driver bugs. Finally, our framework only requires minimum manual efforts, which significantly saves validation human efforts.

Chapter 4

HW/SW CO-VALIDATION FOR DMA INTERFACES

4.1 MOTIVATION AND OVERVIEW

Direct Memory Access (DMA) is a way by which peripheral devices can directly access the system memory without involving CPU. For most peripheral devices, I/O interfaces based on Direct Memory Access (DMA) are a critical part of their HW/SW interfaces. For example, in Intel EERPO100 Ethernet adapter specification [27], 25% of all pages describe the DMA interface implementations. Therefore, DMA interface validation is a critical task in HW/SW co-validation.

This chapter presents a HW/SW co-validation framework for validating DMA interfaces. In general, a device interface includes interface registers and the DMA interface. Chapter 3 presents an approach to conformance checking over device interface registers. Our framework for validating DMA interfaces essentially extends the conformance checking in Chapter 3 to check the conformance on not only interface registers but also the DMA interface. Thereby our extended approach can detect not only DMA interface bugs but also new bugs in interface registers whose values have dependencies on DMA interface state. Nevertheless, the straightforwardly extended conformance checking is not scalable to complicated device designs due to two limitations:

1. **Large overheads of recording DMA interface states.** A DMA interface is essentially a shared memory between the device and its driver. The

size of the DMA interface can be fairly large. For example, the Intel e1000 Ethernet adapter has 8 MB DMA memory. Therefore, recording the DMA interface state, i.e., DMA memory state, under each driver request may heavily degrade system performance.

2. **Missed bugs due to imprecise environmental input simulation.** A large part of DMA-based I/O involves handling the environmental inputs, e.g., receiving data in an Ethernet adapter. As conformance checking does not record environmental inputs, it cannot simulate the DMA operations under environmental inputs precisely on the virtual prototype. As a result, some DMA interface bugs are often missed (cf. Section 4.3).

To address the challenges above, we developed three key techniques: (1) **record-on-write policy** which records the DMA interface state only when it is updated; (2) **partial record** which records part of a FIFO ring-based DMA memory instead of the complete ring; (3) **environmental input prediction** which predicts when the device receives inputs from its external environment, thereby facilitating precise simulation of the device behaviors on the virtual prototype. The first two techniques reduce recording overheads. The last helps discover DMA interface bugs related to environmental inputs.

We have applied our framework to four Ethernet adapters and their drivers using their virtual prototypes from QEMU [9]. Our approach has discovered 12 bugs in DMA interface implementations of the devices, their virtual prototypes, and their drivers. Moreover, the techniques for reducing recording overheads make our framework applicable to two devices with complicated designs.

In summary, our co-validation of DMA interfaces framework makes following key contributions:

1. We present a HW/SW co-validation framework for DMA interface implementations of devices and their drivers using their virtual prototypes.

2. Besides validating the DMA interface implementations, our extended conformance checking further validates interface registers related to the DMA interface (see details in Section 4.2.3).

3. The three key optimizing techniques make our framework scalable and effective on real industry designs.

4.2 OUR APPROACH

4.2.1 Preliminaries

We first briefly review the work flow of DMA-based I/O. A DMA interface is a piece of shared memory between the device and its driver and can be accessed by both. The device and the driver exchange data and commands through the shared memory. A data structure called descriptor is typically used in the DMA work flow. The work flow of a device interacting with its driver through the DMA interface is as follows.

1. The driver builds a descriptor d which contains a command c. The driver puts d into the DMA interface and updates a special interface register Reg of the device to notify the device that there is a command in the DMA interface.

2. Once Reg is updated, the device reads the descriptor d from the DMA interface and executes the task specified by c.

3. When the device completes the task, it updates the status of d and writes d back to the DMA interface. It may also update some relevant interface registers.

From this work flow, it can be observed that two aspects of the DMA interface are validated: (1) the device implementation of the DMA interface that handles the DMA inputs; (2) the driver implementation that produces DMA inputs.

4.2.2 DMA Interface Validation Framework

As Figure 4.1 shows, our HW/SW co-validation framework is built on the conformance checking work flow. It takes the virtual prototype and a trace file generated from the trace recorder, and outputs an inconsistency report and a property failure report. It consists of two major components as follows.

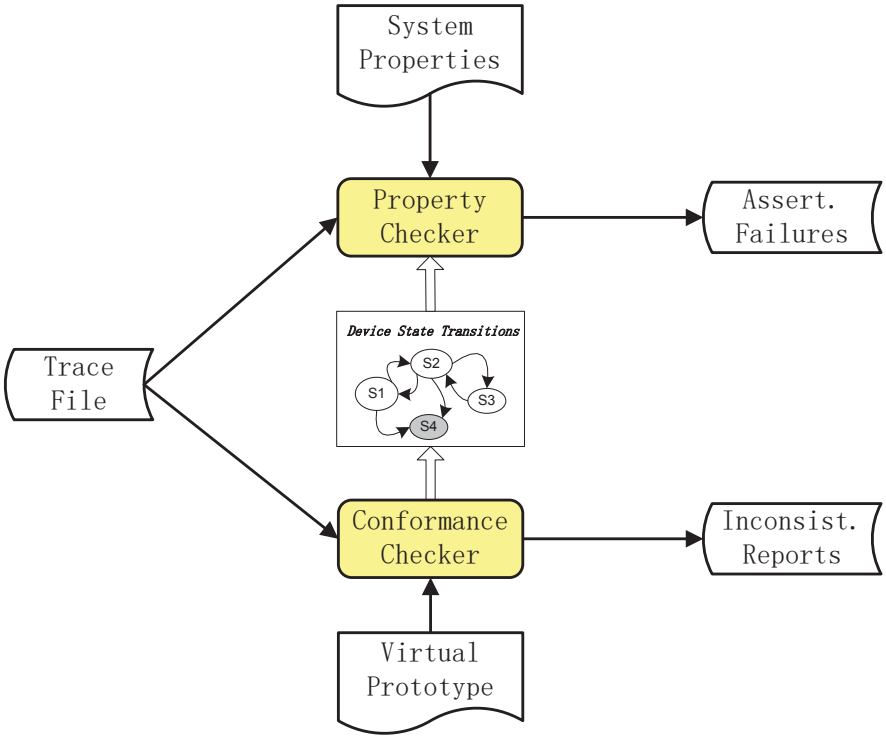


Figure 4.1: HW/SW co-validation framework for DMA interfaces

- *Conformance checker.* Conformance checking over interface registers in Section 3.2 are extended to checking the conformance of both interface registers

and DMA interface states between the device and its virtual prototype. The conformance checker detects errors not only in interface registers but also in DMA interfaces. As the conformance checker simulates the device behaviors over the virtual prototype and checks their conformance under each driver request, the virtual prototype essentially shadows the device execution trace and keeps track of the device state transitions. The device state transitions exposed by the virtual prototype provides the foundation for property checking.

- *Property checker.* Property checker is implemented in a same manner as the property checking presented in Chapter 3. However, instead of verifying the properties over the device/driver interactions across interface registers, the property checker verifies the properties related to device/driver interactions across the DMA interface. It observes the device state transitions through the virtual prototype and detects if any property violation is possible over the state transitions.

As our property checker is directly inherited from the property checking in Chapter 3, we focus on presenting our extended conformance checking infrustracture.

4.2.3 Conformance Checking over DMA Interfaces

This section presents how we extend conformance checking to support the DMA interface validation. The previous approach to conformance checking (cf. Section 3.2) cannot validate DMA interface implementations.

Limitations of previous approach

The aim of validating DMA interface implementations is to detect two types of bugs: those exactly in the DMA interface, which we refer to as **DMA interface bugs**; and those in the interface registers whose values have dependencies on the DMA interface state, which we refer to as **DMA register bugs**. The previous approach does not record the DMA interface state at runtime. So it clearly misses DMA interface bugs. When executing the virtual prototype, this approach models the DMA interface state with symbolic values. So it also misses DMA register bugs.

We use an example to illustrate how such a bug escapes. Figure 4.2 shows how eepro100 processes a DMA driver command. Function *pci_dma_read* fetches a descriptor which is stored in $s \rightarrow cu\_desc$. As labels P1 and P2 indicate, depending on different commands, the device updates the CU state with different values and fires interrupts.

The variable $s \rightarrow cu\_desc$ is assigned symbolic values during symbolic execution of the eepro100 virtual device (VD). As a result, symbolic execution of eepro100 virtual device covers all the three paths in Figure 4.2. We denote the three paths as p_1, p_2, and p_3. The path p_1 follows the code where the branch condition at P1 is true. The path p_2 follows the code where the branch condition at P1 is false and branch condition at P2 is true. The path p_3 follows the code where both of the two branch conditions are false.

Assume that there is a DMA register bug, when the device follows p_2, it fails to update the cu state with *cu_suspend*. When executing the virtual prototype, the conformance checker also explores p_3 where the cu state is consistent with the cu state in the device. According to Definition 3.4, the conformance checker does not discover this update failure. Instead, the extended approach uses concrete DMA

```
    ... ...
    pci_dma_read(cb_address, &s->cu_desc, size);
    ... ...

P1: if (s->cu_desc & COMMAND_EL) {
       // CU becomes idle, fire interrupt
       set_cu_state(s, cu_idle);
       eepro100_cna_interrupt(s);
    }

P2: else if (s->cu_desc & COMMAND_S){
       // CU becomes suspended, fire interrupt
       set_cu_state(s, cu_suspend);
       eepro100_cna_interrupt(s);
    }
    ... ...
```

Figure 4.2: DMA interface implementations of eepro100 VD

inputs, therefore, only p_2 is explored and the cu state is updated to *cu_suspend*, which is inconsistent with the device cu state. The bug is discovered. The evaluation results show that our extended approach detected several bugs both in the DMA interface and DMA registers (cf. Section 4.4).

Extended Conformance Checking

Our approach follows a similar work flow as the previous conformance checking, but makes three key extensions:

1. **Record concrete DMA interface states**. In addition to the interface registers, the trace recorder also records the DMA interface state at runtime.

2. **Extend Device State Representation**. We define a DMA interface state as a set of DMA interface variables with their values. Given $V = \langle V_I, V_N \rangle$ and $S = \langle S_I, S_N \rangle$ defined in Section 3.2.1, we extend the virtual prototype state as $V_E = \langle V_{EI}, V_N \rangle$, where $V_{EI} = \langle V_I, V_M \rangle$, V_M represents the virtual prototype DMA interface state. Similar as V_I, the values of V_M can be either symbolic or concrete. We define the extended silicon device state as $S_E = \langle S_{EI}, S_N \rangle$, $S_{EI} = \langle S_I, S_M \rangle$, where S_M represents the device DMA interface state. The values of S_M are concrete.

3. **Report inconsistent DMA interface**. The conformance checker checks the conformance between V_{EI} and S_{EI} in the same manner as Definition 3.3 and Definition 3.4. If the virtual prototype and the device do not conform, the conformance checker outputs inconsistency reports which contains inconsistencies of both interface registers and the DMA interface.

Remarks. The previous approach is sound theoretically. However in practice, it might have false positives, i.e., false alarms. The virtual prototype might have unbounded loops which make symbolic execution non-terminating. A loop bounding algorithm is used to set constant bounds for these loops dynamically. This algorithm may reduce possible behaviors of the virtual prototype; therefore, producing false positives. Our approach faces the same challenge. However, the chance of false positives is lower than the previous approach in both DMA interface and interface register conformance checking results. In the previous approach, most of false positives are caused by bounding two kinds of loops: (1) ones whose loop conditions depend on the environmental inputs; (2) the others whose loop conditions

depend on the DMA interface values. Our approach does not record environmental inputs; therefore, we may still get false positives on the first kind of loops. However, as we record concrete DMA interface values, our approach eliminates false positives caused by the second kind.

4.3 TECHNIQUES FOR CHECKING DMA INTERFACES

Our straightforwardly extended conformance checking over DMA interfaces has two major challenges in scaling to real industry designs. First, capturing DMA interfaces incurs a large runtime overhead. For example, when we evaluate our approach on Intel e1000 Ethernet adapter, the computer system hangs and cannot function normally when the trace recorder captures the DMA interface state. In Section 4.3.1 and Section 4.3.2, we present two techniques to address this problem. Second, the conformance checker may still miss DMA bugs related to handling environmental inputs as it cannot predict when environmental inputs were handled. We give an example and present our solution in Section 4.3.3.

4.3.1 Record-on-write Policy

The trace recorder records the DMA interface state before each driver request is issued. However, in the device, the DMA interface is not updated at every driver request, instead, the DMA interface state remains the same over a significant number of consecutive driver requests. Therefore, it is unnecessary to record the DMA interface state before each driver request. We develop a technique, the record-on-write policy, to record the DMA interface only when it is updated.

Identifying DMA interface updates

The DMA interface is only updated by the device and its driver. There are three scenarios where the updates occur: (1) the driver issues a command via the DMA interface; (2) the device outputs to the external environment; (3) the device receives environmental inputs. In fact, the trace recorder only needs to record the DMA interface under these scenarios. We show how to identify these scenarios respectively.

- The first and second scenarios are all triggered by issuing driver requests. Since the trace recorder intercepts all driver requests, by analyzing these driver requests, it can identify the first two scenarios.
- For the third scenarios, we use the technique presented later in Section 4.3.3 to identify when the device receives environmental inputs.

Associating DMA interface states with driver requests

Record-on-write leads to a potential problem: for some driver requests, there is no DMA interface state associated. However, when we replay the device trace on the virtual prototype, the virtual prototype may still read DMA memory even it does not update it. Therefore, for these driver requests without the associated DMA interface state, we need to provide a valid DMA memory to the virtual prototype. To address this problem, we implement a "copy-on-write" policy while replaying the device trace. The DMA interface state associated with the current driver request will be automatically inherited by the next driver request, if there is no "write" on the DMA interface occurs between these two consecutive driver requests. When there is a write operation on the DMA interface, the next driver request uses its own associated DMA interface state.

4.3.2 Partial Recording of DMA Interface

A DMA interface of a device is not a flatten memory. Instead, it is typically implemented as a "ring buffer" data structure. As Figure 4.3 illustrates, the device and the driver keep two indices called "head" and "tail". When the driver allocates a unit of memory to the device, it increments "tail". Similarly, when the device consumes a unit of memory, it increments "head". The memory between "head" and "tail" is considered as valid memory. The device fetches DMA descriptors only from the memory units between "head" and "tail".

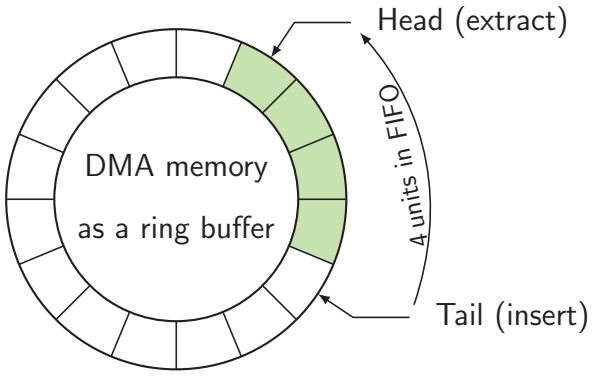


Figure 4.3: Ring buffer structure of DMA memory

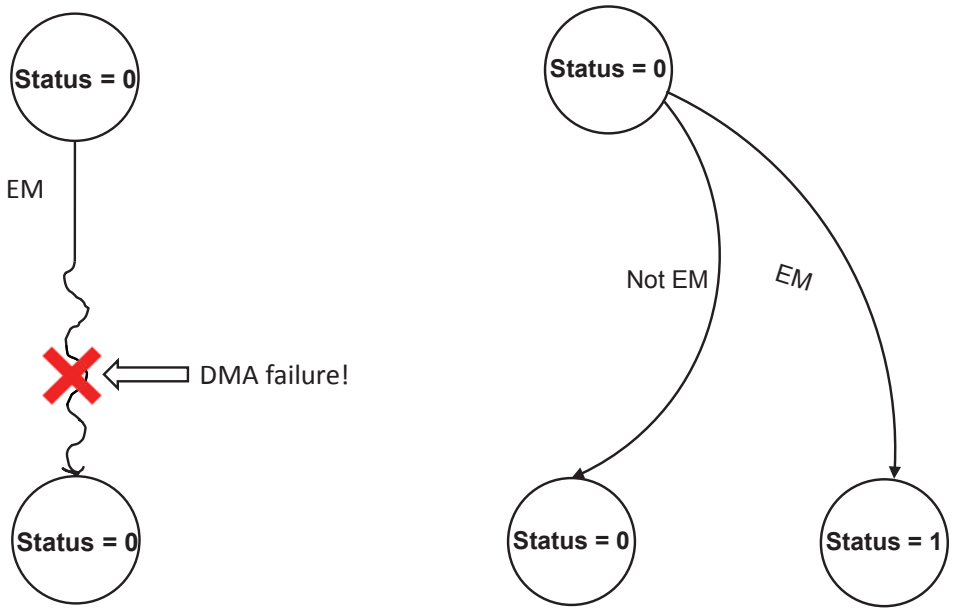
Since the device only touches the valid memory defined by "head" and "tail", when the trace recorder records a DMA memory, it does not need to record the entire memory. Instead, it only records the valid memory. This way, we further reduce the overhead incurred by DMA interface state recording.

4.3.3 Environmental Input Prediction

Motivation

As illustrated in Section 3.4.3, upon each driver request, the conformance checker uses a non-deterministic choice to decide invoking EM or not. In this way, the

virtual prototype can capture the device behaviors under two possible scenarios: (1) environmental inputs arrive; (2) no environmental input. However, there is a potential to miss DMA interface bugs. We present such a concrete example. When Intel eepro100 receives a packet from its external network, according to its specification, after processing the packet, the device will set its status bit to value 1 in the DMA interface, indicating the completeness of packet reception. Assume that the status update fails for some reason, as a result, the status bit remains 0 in the DMA interface (see Figure 4.4-(a)). However, this status bit has the same value as no external input arrivals. In the virtual prototype, as Figure 4.4-(b) shows, there are two paths including both reception (EM) and non-reception (Not EM), the conformance checker covers both paths by symbolically executing the virtual prototype. Therefore, although the DMA interface update fails, it is still considered valid. This update failure will not be discovered.



(a) Device Trace (b) Virtual Prototype Traces

Figure 4.4: DMA bugs missed w/o environment input prediction

Solution

If the conformance checker knows when the device receives environmental inputs while replaying the device trace, it can just invoke EM instead of trying both branches. The bugs will not be missed. To realize this feature, we develop a technique, environmental input prediction. Given a device trace T generated from the device, environmental input prediction determines when the device receives environmental inputs. We first summarize the typical work flow how a device receives inputs from the external environment. When environmental inputs arrives, the device processes these inputs. After the device finishes processing, it updates the corresponding status of a descriptor in the DMA interface. Moreover, it fires an interrupt to notify the driver by updating the interrupt register R_{intr} with a specific value Val_{intr}.

In a device trace T, given two consecutive driver requests D_i and D_{i+1} $(0 \leq i)$, there are two device interface states S_{I_i} and $S_{I_{i+1}}$ which are recorded before D_i and D_{i+1} respectively. If the value of R_{intr} in S_{I_i} is not Val_{intr} and the value of R_{intr} in $S_{I_{i+1}}$ is Val_{intr}, the device receives environmental inputs between D_i and D_{i+1}. We denote such a pattern of R_{intr} value change as P. When the conformance checker replays T on the virtual prototype, if P is detected in D_i and D_{i+1}, the conformance checker only invokes EM when it processes D_i; otherwise, it does not invoke EM. In this way, environmental input prediction helps avoid missing certain bugs in the DMA interface.

4.4 EVALUATION

4.4.1 Experiment Setup

We have performed our experiments on a workstation with a dual-core Intel Pentium D Processor with 4GB of RAM and Ubuntu Linux OS with 64-bit kernel version 2.6.38. We applied our framework to four Ethernet adapters and their virtual prototypes, QEMU virtual devices. Information about these devices and their virtual devices are summarized in Table 4.1. The virtual device size is measured in Lines of Code (LoC).

Table 4.1: Devices and virtual prototypes for DMA interface validation

Devices	**Virtual Device Size (LoC)**	**Basic Description**
RealTek rtl8139	3544	RealTek 10/100M Ethernet Adapter
Intel eepro100	2178	Intel 10/100M Ethernet Adapter
Intel e1000	2099	Intel Gigabit Ethernet Adapter
Broadcom bcm5751	4519	Broadcom Gigabit Ethernet Adapter

4.4.2 Bug Detection

Our framework has detected 12 new bugs summarized in Table 4.2. There are 2 device bugs, 8 virtual prototype bugs, and 2 driver bugs. Since we conducted our experiments over the stable products which have been released for many years, there are only a few device bugs. However, the virtual prototype bugs that we discovered are all common hardware design flaws. Therefore, our approach has major potential in discovering bugs in silicon prototypes including FPGAs and test devices. All the driver bugs are discovered by our property checking. Property

checking verified 26 properties in total, of which there are 9 stateless properties and 17 stateful properties.

Most of these bugs can cause serious problems. Two of the interface register bugs are related to missing interrupts, which often break down the normal driver work flow and even cause driver and system crashes. DMA interface bugs cause corrupted DMA memory, which can lead to driver misbehavior as the driver may read incorrect status. A driver input with invalid descriptors is potential to incur device misbehavior.

The results demonstrate that our framework is promising in handling the three key challenges of HW/SW integration validation presented in Section 1.2. First, our framework is effective to detect the design flaws in HW/SW interfaces. For example, the discovered bug of updating the reserved bits in the DMA interface, can be easily missed without observing HW/SW interface. Second, our framework can easily identify a HW/SW interface bug as a hardware bug or a software bug. For example, an invalid driver input often appears like a device bug as the device usually hangs under the invalid input. By detecting the invalid input through property checking, the framework clearly identifies this bug as a driver bug. Last but not the least, detecting DMA register bugs shows that our approach improves the effectiveness in validating device interface registers.

Table 4.2: Summary of device, virtual prototype, and driver bugs

No.	Bug Description	Num.	Bug Source	Bug Types	Distribution
1	Update reserved bits of the DMA interface	2	Driver	Stateless Property Violation	eepro100, e1000
2	Update reserved bits in the DMA interface	2	Device	DMA interface bug	e1000, bcm5751
3	Fail to fire required interrupt when DMA operations have errors	1	VP	DMA register bug	eepro100
4	Fail to fire required interrupt when the DMA descriptor number is low	1	VP	DMA register bug	e1000
5	Fail to check if DMA data is out-sync as specification requires	1	VP	DMA register bug	bcm5751
6	Incorrectly update the DMA interface	2	VP	DMA interface bug	bcm5751
7	Fail to simulate the concurrency of processing DMA data	3	VP	DMA interface bug	eepro100, e1000, bcm5751

4.4.3 Efficiency

In this section, we evaluate the efficiency of our recording method with record-on-write policy and partial recording, in terms of time and memory usages in the runtime recording stage of the conformance checking work flow. The test cases used in evaluation are described in Table 4.3. All these test cases heavily involve DMA I/O operations.

Table 4.3: Test cases for evaluating DMA interface validation

Test Cases	**Description**
Ping	Ping another network interface
Small transfer	Transfer a small file with size 2.4 MB
Large transfer	Transfer a large file with size 3.2 GB

The test cases are issued under three configurations: (1) No Recording (NR) mode: there is no recording conducted; (2) *Recording Everything (RE) mode:* the recording method without the two proposed techniques, which records everything in the DMA interface; (3) *Record-on-write and Partial recording (RP) mode:* the method with record-on-write and partial recording techniques. We set the NR mode as the baseline and the performance of the NR mode is normalized to 1. Figure 4.5 shows the ratios of the RE and RP modes comparing to the NR mode. In Figure 4.5, no data is provided for the RE mode in terms of e1000 and bcm5751 since the RE mode incurs a large overhead and the system hangs. By applying record-on-write and partial recording techniques in the RP mode, recording DMA interface states can be successfully and efficiently achieved.

The results demonstrate that our two optimizing techniques make recording DMA interface states scalable to the devices with complicated designs such as e1000 and bcm5751, both Gigabit Ethernet adapters; for the devices such as eepro100 and

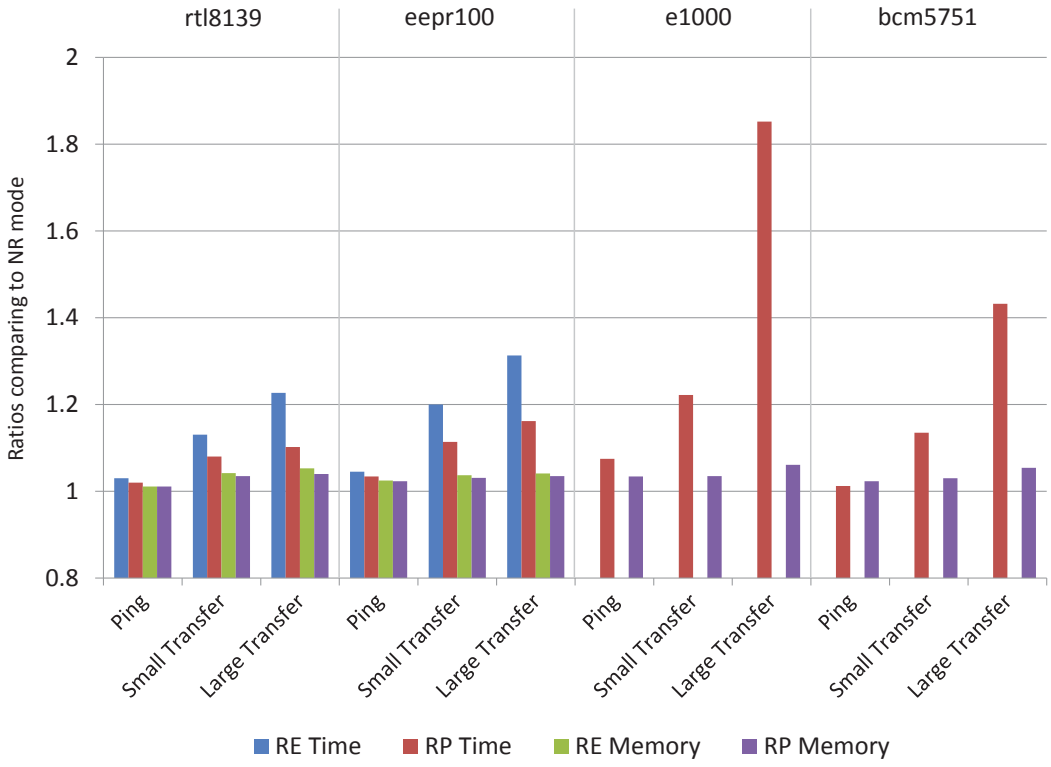


Figure 4.5: Time and memory usages of test cases under Recording Everything (RE) and Record-on-write and Partial recording (RP) modes. The usages with no recording (NR) are normalized to 1. Figure shows ratios of RE and RP comparing to NR

rtl8139, both 10/100M Ethernet adapters, record-on-write and partial recording also noticeably reduce the recording overheads.

4.5 SUMMARY

This chapter has presented a HW/SW co-validation framework to validating the DMA interface implementations. Our two-staged checking infrastructure helps detect errors in DMA interface implementations of both a device and its device driver. We discovered several bugs related to DMA interface implementations from

devices, their virtual prototypes, and their driver. Our validation framework has major potential in addressing the key challenges of HW/SW integration validation which are presented in Section 4.1. First, the framework records and validates the DMA interface state; thus the errors in the DMA interface are detected effectively. Second, the framework identifies both the device and driver errors over the DMA interface thereby it can easily attribute device/driver interface bugs as device or driver bugs. Finally, our framework only requires minimum manual efforts, which significantly saves validation human efforts.

Chapter 5

OPTIMIZATIONS FOR CONFORMANCE CHECKING

5.1 MOTIVATION AND OVERVIEW

Chapter 3 presents an approach to post-silicon conformance checking of a hardware device with its virtual prototypes. This approach symbolically executes the virtual prototypes with the same driver request sequence to the device, and checks if the interface states of the silicon and virtual prototypes are consistent. However, the internal state of a device is hard to observe and the external environment inputs to the device are also hard to capture. This approach uses symbolic execution to tackle this problem. It models internal states and environment inputs using variables with symbolic values when simulating the device behaviors on the virtual prototype. This way symbolic execution covers all the possible values of the internal state and environment inputs.

The approach presented in Chapter 3 has two major limitations.

1. **Missing internal bugs**. It checks the interface state conformance after the virtual device processes each driver request. Before processing the next driver request, it resets the internal states of the virtual device by assigning them symbolic values. This way, the internal variable values, which have already been concretized in simulation, are lost. Therefore it may miss certain internal bugs propagating to device interface registers after a few driver requests later.

2. **Incurring significant time usages**. Symbolic execution introduces a significantly overhead while exploring a large number of paths. This overhead makes the approach a time-consuming process. In post-silicon conformance checking, a driver request sequence is often composed of thousands of, even millions of driver requests, which requires a long time to process. Therefore, how to reduce time costs is a critical task to scale the conformance checking approach.

In this section, we present a thorough and efficient approach to address the two limitations above. Our proposed approach can detect the internal bugs. Rather than resetting the virtual prototype internal state to symbolic values, our approach keeps the concrete values of the internal state after the virtual prototype processes each driver request. Moreover, we propose an optimization, adaptive concretization, to reduce the symbolic execution overheads. We exploit the fact that most of virtual prototype states conforming to the device state are generated by an execution path accessing none of or only a few of symbolic values. Adaptive concretization eliminates unnecessary symbolic values to prune unnecessary paths explored by symbolic execution.

We have evaluated the approach on three Ethernet adapters and their virtual prototypes from QEMU virtual machine [9]. We discovered 25 inconsistencies, behind which there are 25 device bugs including both interface and internal bugs in either the devices or their virtual devices. Furthermore, the time usages have been reduced by an order of magnitude.

5.2 THOROUGH CONFORMANCE CHECKING

5.2.1 Problem and Motivation

As Algorithm 3.1 describes, the native approach synchronizes the virtual prototype state to the device state before processing each drive request (line 4). As the internal state variables of a device are modeled as symbolic values without any constraints, this synchronization causes the internal state variables with concrete values in the virtual device to lose their values.

We use an example to illustrate how the native approach misses internal bugs. This example is an device internal error of an Ethernet adapter as follows.

In 100 Mb/s link mode, internal clocks are slower, and access of an internal register can lead to timeout. An unknown value is returned on the PCI Express (PCIe) interface [28].

This bug happens when the internal register value propagates to the interface register (PCIe interface) upon a driver request. We give a complete scenario as follows. Let the buggy internal register be Reg_N, assume that there is a driver sequence $D_0, ..., D_i, ..., D_j, ..., D_n (0 \leq i < j \leq n)$, where D_i updates Reg_N to a concrete value val and D_j reads Reg_N and gets a value val' from an interface register Reg_I which the value of Reg_N propagates to. Essentially, the above error occurs while val of Reg_N propagates to Reg_I, val' is not equal to val. In the native approach, after the virtual prototype processes D_i and the virtual prototype conforms to the device, a new virtual prototype state V_{i+1} is created for processing D_{i+1} and Reg_N in V_{i+1} is reset to a symbolic value α instead of val. Therefore, upon the driver request D_j, the virtual prototype returns α to Reg_I. In the device, the value is val'. The native approach returns true as symbolic value α can cover val'. However, if the virtual prototype keeps the internal value val, this inconsistency

can be detected.

5.2.2 Thorough Conformance Checking Approach

We propose an approach to deal with the problem in Section 5.2.1. This approach has the same workflow with the native approach except that it avoids synchronizing the virtual prototype state to the device state in each iteration. Algorithm 5.1 illustrates the workflow.

Algorithm 5.1 thorough_replay_trace(T, F)

1: $T' \leftarrow convert\_trace(T)$
2: /*Initialize VD state V_0 to be SD state S_0*/
3: $V_0 \leftarrow S_0$
4: /* Take $\langle S_k, D_k \rangle$ from T'*/
5: **for** $k: 0 \rightarrow n$ **do**
6: $G \leftarrow sym\_exec(F, V_k, D_k)$
7: $H \leftarrow conformance\_check(G, S_{k+1})$
8: **if** $H == \emptyset$ **then**
9: $report\_incon()$
10: $V_{k+1} \leftarrow S_{k+1}$
11: **else**
12: $V_{k+1} \leftarrow construct\_next\_state(H)$
13: **end if**
14: **end for**

In the new workflow, same as the native approach, function $conformance\_check$ generates a set of virtual device states $H = \{h_i \mid h_i \neq \emptyset, 0 \leq i \leq m\}$ where $h_i = set(g_j) \cap set(S_{k+1}), 0 \leq j \leq n$. If H is an empty set, the virtual and devices do not conform, we report this inconsistency and synchronize the next virtual device state

V_{k+1} to the device state S_{k+1}. Otherwise, function *construct_next_state* constructs V_{k+1} as $V_{k+1} = \bigcup_{i=0}^{m} h_i$. As Definition 3.1 illustrates, V_{k+1} can be represented as a pair $\langle V_{I_{k+1}}, V_{N_{k+1}} \rangle$ and $\forall h_i \in H, h_i = \langle h_{i_I}, h_{i_N} \rangle, 0 \leq i \leq m$. The variables in the virtual prototype state can be denoted as var and its value can be denoted as $Val(var)$. Algorithm 5.2 shows how to compute the V_{k+1}.

Algorithm 5.2 construct_next_state(H)

1: /*Constructing interface state.*/
2: $V_{I_{k+1}} \leftarrow h_{0_I}$
3: /*Constructing Internal State.*/
4: **for each** var_j of $V_{N_{k+1}}$ **do**
5: $\quad Val(var_j)_{V_{k+1}} \leftarrow \bigvee_{i=0}^{m} Val(var_j)_{h_i}$
6: **end for**
7: **return** V_{k+1}

Notes. This approach takes the union of the conforming states, thereby all the possibilities of virtual prototype internal states are reserved. Once an internal bug occurs and propagates to an interface register in either the virtual or silicon device, our approach reports an inconsistency when all the possible states of the virtual prototype do not conform to the device.

5.3 ADAPTIVE CONCRETIZATION

5.3.1 Preliminaries

Definition 5.1 (Virtual Prototype Path)**.** *A virtual prototype path is a sequence of branch conditions, denoted as* $\pi = c_0, c_1, ..., c_{n-1}, c_n$*, where* c_i $(0 \leq i \leq n)$ *is a branch condition, a Boolean expression over device state variables and external environment inputs. We refer to virtual prototype path as path for simplicity.*

Definition 5.2 (Conforming Path)**.** *Given a virtual prototype state V_k and its next virtual prototype state sets $G = \{g_i \mid 0 \leq i \leq n\}$ (cf. Section 5.2), $\forall g_i \in G$, there exists a virtual prototype path π, V_k transitions to g_i following π, denoted as $V_k \stackrel{\pi}{\Rightarrow} g_i$. V_k is the previous state of π and g_i is the next state of π. Moreover, if g_i is a conforming state, we define π as a* **conforming path**.

5.3.2 Our Approach

Motivation. The conformance checking approach assigns symbolic values to the internal state variables and the external environment inputs. These variables with symbolic values account for a significant overhead as symbolic execution explores an enormous number of paths due to symbolic values. An intuitive idea is to assign concrete values to these variables instead of symbolic ones. We observed that a conforming path usually accesses none of, or only a small number of variables with symbolic values. In other words, variables with symbolic values do not affect the conformance checking results most of time. Therefore, we can adaptively concretize these symbolic variables to reduce symbolic execution overhead.

We present an optimization, adaptive concretization, to optimize the conformance checking approach. Figure 5.1 shows the workflow. Adaptive concretization is a two-round of conformance checking. In the first round, we concretize (1) virtual prototype internal variables and (2) external environment inputs, which all have symbolic values originally. Then we check the conformance following the same workflow of the conformance checking approach. We define this round as **concrete mode**. However, as the concrete values we assigned to the variables might not be the right values, the concrete mode may produce false alarms, i.e., false positives. To eliminate these false positives, we conduct a second around using the original virtual prototype where the internal state and external inputs all have symbolic

values. This round verifies the inconsistencies discovered in the concrete mode. We define such a round as **refinement mode**.

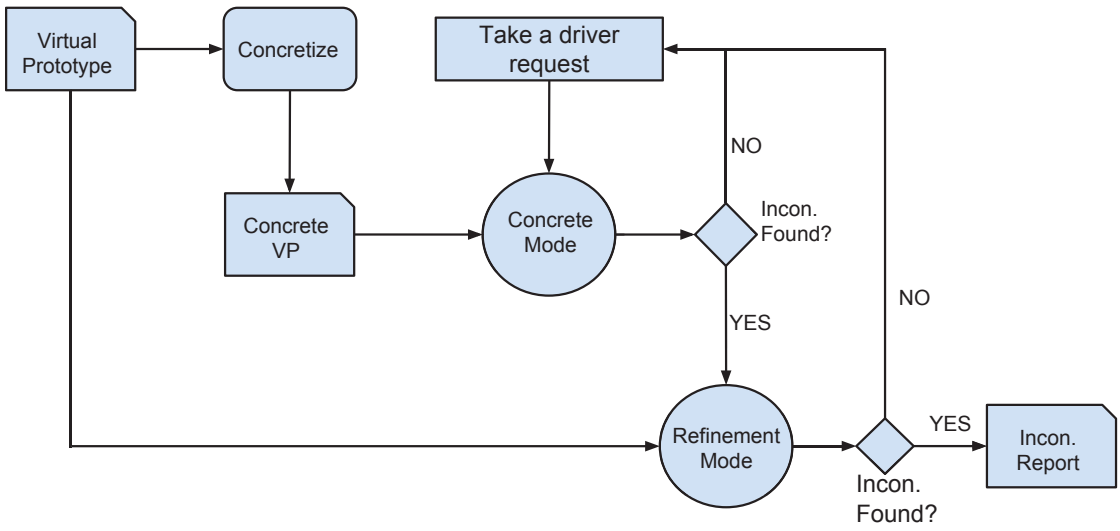


Figure 5.1: Workflow of adaptive concretization

Concrete Mode. The conformance checking algorithm of the concrete mode is shown in Algorithm 5.3. It takes a device trace T and a virtual prototype F as its inputs.

Algorithm 5.3 follows the workflow of Algorithm 3.1 except three modifications: (1) function *convert_to_conrete_trace* is applied instead of *convert_trace* to concretize device states in T; (2) function *concretize_device* concretizes the virtual prototype F; (3) when an inconsistency is discovered, the workflow enters the refinement mode rather than directly reporting an inconsistency.

Given $T = \langle S_{I_0}, D_0\rangle$, $\langle S_{I_1}, D_1\rangle$, ..., $\langle S_{I_n}, D_n\rangle$, function *convert_to_conrete_trace* converts T to $T' = \langle S_0, D_0\rangle$, $\langle S_1, D_1\rangle$, ..., $\langle S_n, D_n\rangle$, where $S_k = \langle S_{I_k}, S_{N_k}\rangle (0 \leq k \leq n)$ derived from S_{I_k}. Instead of assigning symbolic values to internal state variables of S_{N_k}, function *convert_to_conrete_trace* assigns value zero to variables of S_{N_k}.

Algorithm 5.3 concrete_mode(T, F)

1: $T' \leftarrow convert\_to\_conrete\_trace(T)$
2: $F' \leftarrow concretize\_device(F)$
3: /*Initialize VP state V_0 to be device state S_0*/
4: $V_0 \leftarrow S_0$
5: /* Take $\langle S_k, D_k \rangle$ from T'*/
6: **for** $k: 0 \rightarrow n$ **do**
7: $G \leftarrow sym\_exec(F', V_k, D_k)$
8: $H \leftarrow conformance\_check(G, S_{k+1})$
9: **if** $H == \emptyset$ **then**
10: $V_{k+1} \leftarrow refinement\_mode(F, V_k, S_{k+1}, D_k)$
11: **else**
12: $V_{k+1} \leftarrow construct\_next\_state(H)$
13: **end if**
14: **end for**

Moreover, in function *concretize_device*, external environment inputs to the virtual prototype F are also concretized to zeros. As the values of some environment input variables cannot be zero, for example, the value for modeling the received packet size cannot be zero, function *concretize_device* randomly picks up non-zero concrete values in their valid ranges.

The reason we use zero rather than other concrete values for concretization is because most of the internal state variables have zero as their initial values. By setting zero, we can largely avoid introducing false positives in the concrete mode. The zero value we use to concretize symbolic values should be treated as a special concrete value. We denote such a value as 0_{sym}, indicating this zero is concretized from a symbolic value and will be recovered to the symbolic value in the refinement

mode. As discussed above, some environmental variables are concretized into non-zero values. For these variables, we denote their random non-zero values as r_{sym}

5.3.3 Refinement Mode

The refinement mode takes the virtual prototype F, a virtual prototype state V_k, a device state S_{k+1}, and a driver request D_k as its inputs. It has the same workflow as presented in Algorithm 3.1. Additionally, it has two conversion functions $Con2Sym$ and $Sym2Con$. Function $Con2Sym$ is invoked immediately when the workflow enters the refinement mode. It replaces 0_{sym} and r_{sym} of virtual prototype variables with symbolic values. Function $Sym2Con$ is invoked in the end of refinement mode. It converts the next state V_{k+1} generated in the refinement mode from a symbolic state to a concrete state where symbolic values of internal state variables are concretized to 0_{sym} and r_{sym} again. In this way, V_{k+1} can be used in the concrete mode. By recovering 0_{sym} and r_{sym} to symbolic values, the refinement mode re-simulates the virtual prototype under the driver request leading to the inconsistency in concrete mode. The inconsistency confirmed in the refinement mode are reported as a real inconsistency.

5.4 EVALUATION

5.4.1 Experiment Setup

All experiments were conducted on a workstation with a dual-core Intel Pentium D Processor at 3.20 GHz and 4GB of RAM, running Linux with kernel version 2.6.35. We evaluated three widely used network adapters and their QEMU virtual devices as virtual prototypes. Information of these devices and their virtual devices are summarized in Table 5.1. The virtual device size is measured in Lines of Code (LOC).

Table 5.1: Devices and virtual prototypes for adaptive concretization

Devices	Virtual Device Size (LOC)	Basic Description
Intel e1000	2099	Intel Gigabit Ethernet Adapter
Broadcom bcm5751	4519	Broadcom Gigabit Ethernet Adapter
Intel eepro100	2178	Intel Megabit Ethernet Adapter

Table 5.2: Summary of virtual prototype bugs

No.	Bug Description	Num.	Distribution
1	Internal read-only register is updated	1	bcm5751
2	Reserved bits of internal registers are updated	1	bcm5751

5.4.2 Bug Detection

In this section, we demonstrate that (1) our new approach can detect all the previous bugs discovered by the native approach; (2) our approach detects several internal bugs which cannot be discovered by the native approach.

To demonstrate that our optimized approach does not reduce the capacity comparing to the native approach, we preform the test cases triggering the previous inconsistencies between devices and virtual prototypes. The results shows that our approach detects all the previous bugs.

One important improvement of our approach is to detect internal bugs. In the experiment, the approach detects 2 internal bugs in virtual prototypes. All these internal bugs cannot be caught by the native approach. Figure 5.2 shows the details of these bugs. The experiment does not find any device internal bugs. The reason is that the devices are stable products which have gone through extensive testing

and bug-fixing procedures. However, these virtual prototype bugs are still possible to show up in silicon prototypes at the early stage of hardware development, since these bugs are common violations of hardware designs. We believe that if this approach is conducted at the post-silicon testing stage before devices are released, it can also discover the device internal bugs as well.

Table 5.3: Test cases for evaluating adaptive concretization

Test Cases	**Description**
Reset Network Interface	Bring down and then bring up the network interface
Ping	Ping another network interface
Transfer files	Copy large files with total size 3.2 GB
NIC test-suite	A set of typical test cases on NIC

5.4.3 Efficiency

We evaluate the efficiency of our approach, in terms of time usages, memory usages, and false positives in the concrete mode. We issue four kinds of test cases to the network adapters to collect device traces. These test cases are all common usages of network adapters as shown in Table 5.3. "NIC test-suite" contains a family of typical test cases on network interface controllers (NIC), which manipulates a NIC in different ways, e.g., sending UDP packets and setting MTU size.

Time usages

We calculate the average time usages to process 100 driver requests in each test cases. Table 5.4 summarizes the results. The time have been reduced an order of magnitude by using the optimized approach. The time usages are reduced

Table 5.4: Time and memory usages in adaptive concretization

Devices	Test Cases	Time Usage (sec)		Memory Usage (MB)	
		Native	*Optimized*	*Native*	*Optimized*
e1000	Reset NIC	31.28	1.83	233.41	225.26
	Ping	366.28	45.10	336.21	330.24
	Transfer files	415.05	48.29	336.63	331.57
	NIC test-suite	351.13	18.36	288.79	288.33
bcm5751	Reset NIC	26.31	0.88	169.01	168.32
	Ping	305.11	42.05	284.25	279.47
	Transfer files	294.84	48.23	273.23	261.69
	NIC test-suite	261.77	23.79	225.95	225.93
eepro100	Reset NIC	28.79	0.61	251.62	243.81
	Ping	236.51	16.62	261.31	259.63
	Transfer files	210.44	16.70	262.96	258.99
	NIC test-suite	215.57	8.63	261.34	258.38

less in the test cases "Ping" and "Transfer files" than the other two test cases. The reason is that these two test cases involve receiving packets. The test case involving receiving packets has more false positives introduced in the concrete mode, as the conforming paths usually access many symbolic variables representing the environmental inputs. Therefore, in these two test cases, the approach often requires the refinement mode and the time usages are increased. (See Section 5.4.4 for the number of false positives in the concrete mode under each test cases).

Memory usages

We evaluate the memory usages in the same way as evaluating time usages. As Table 5.4 shows, the results suggest that our optimized approach has almost same memory usages with the native approach. Consider that the memory usages are not too high, the memory resource costs are acceptable.

5.4.4 False Positives of Concrete Mode

The effectiveness of adaptive concretization heavily depends on the number of false positives of the concrete mode. To explain the effectiveness of our optimized approach better, we record the number of the false positives. In this experiment, we collect the numbers of inconsistencies produced by the concrete mode and the refinement mode, denoted as α and β respectively. As a result, the false positives can be computed as α minuses β. Figure 5.2 shows the results. All the inconsistencies are counted while our optimized approach processes 100 driver requests.

As Figure 5.2 shows, our approach does not introduce too many false positives in the concrete mode. The highest case is 3 false positives out of 100 driver requests. This demonstrates adaptive concretization is efficient. In addition, we find that test cases involving receiving packets have more false positives.

5.5 RELATED WORK

Many research have been done for reducing symbolic execution overheads. A major effort is to avoid path explosions by pruning redundant paths. RWSet [10] and path subsumption [2] employ a similar heuristic whereas a path which is identical to the one previously explored can be safely pruned. Kuznetsov et al. [35] propose a method of automatically merging states to reduce the number of paths explored

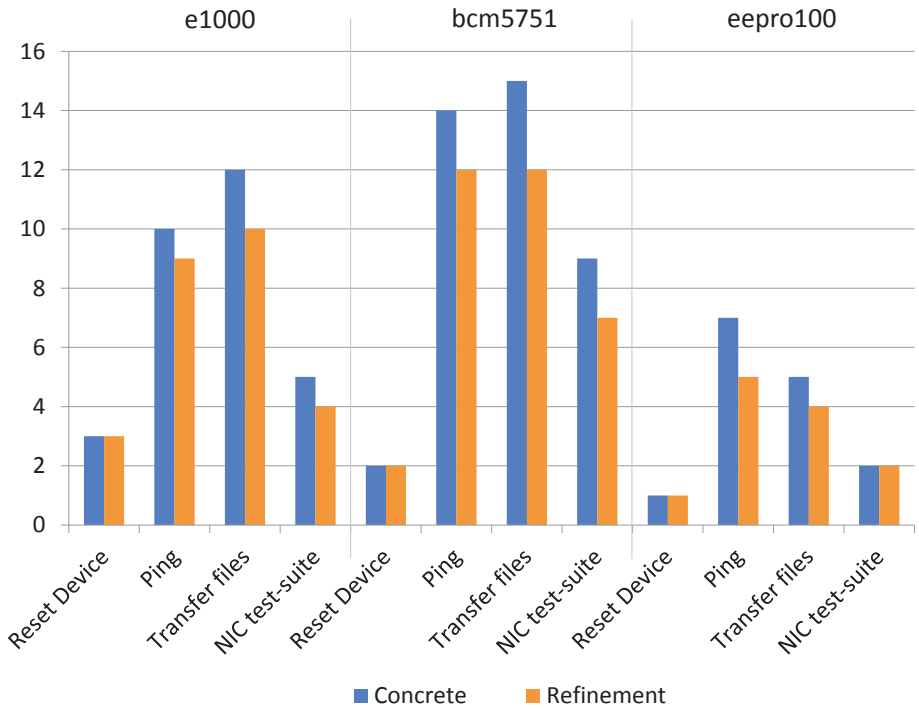


Figure 5.2: Numbers of inconsistencies under test cases

in symbolic execution. Several other approaches [21, 53, 56] leverage the benefits of concolic execution to partially concretize the target programs thereby the number of explored paths is decreased.

5.6 SUMMARY

In this section, we have presented the optimization approach to thoroughly and efficiently checking the virtual prototype and device conformance. By keeping virtual prototype internal state, the conformance checking is extended to detection of internal bugs. While employing adaptive concretization, symbolic execution time usages are reduced significantly. These optimizations make the conformance checking efficient and scalable to the hardware devices with complicated designs.

Chapter 6

HW/SW CO-MONITORING

6.1 MOTIVATION AND OVERVIEW

6.1.1 Motivation

We have presented our HW/SW co-validation framework for post-silicon stage in previous chapters. However, post-silicon validation is not sufficient: when the system is released, HW/SW interfaces are still vulnerable even after many iterations of testing and validation. In fact, assuring HW/SW interface reliability and security faces different challenges comparing to the post-silicon stage. There are three major challenges: (1) hardware transient errors are abundant in our daily life; (2) device drivers can be easily hijacked to exploit HW/SW interface vulnerabilities, sometimes, even the driver itself is malicious. (3) Recently hardware trojans are more and more prevalent, where the malicious attacks can be easily launched to target on HW/SW interfaces. Therefore, it is highly desired to develop a systematic approach which can effectively monitor HW/SW interfaces to detection of defects and malicious attacks across HW/SW interfaces.

Our post-silicon HW/SW co-validation is conducted off-line. At the deployment stage, it is not realistic to deploy the off-line validation framework. Therefore, we need to develop an on-line monitoring framework to conduct conformance and property checking. In this section, we present our on-line monitoring approach, HW/SW co-monitoring, which simultaneously monitors a hardware device and its

driver at runtime and reports device and driver errors.

6.1.2 Our Approach

We present *HW/SW co-monitoring*, a novel approach to conducting HW/SW co-verification at runtime. As shown in Figure 6.1, the foundation of this approach is a formal device model (FDM), a transaction-level, executable model which captures the device behaviors. This approach is based on the co-execution of the FDM and device where the FDM shadows the device execution. Based on the co-execution, our approach entails three major techniques to realize runtime HW/SW co-verification:

- Runtime concolic execution of the FDM and driver where the driver is running concretely while the FDM is executed symbolically;
- Runtime detection of divergence between the device and FDM, namely device checking;
- Runtime verification of system-level properties against the device and driver indirectly on the FDM and driver.

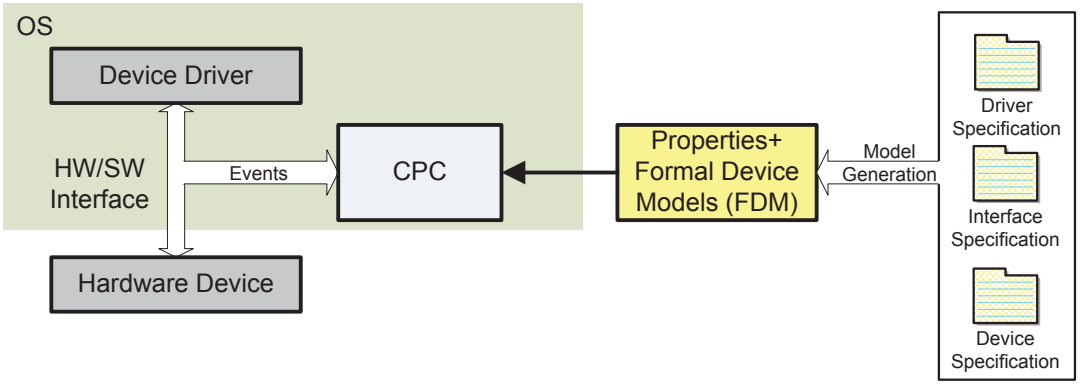


Figure 6.1: HW/SW co-monitoring of device and driver

In the runtime concolic execution, symbolic execution of the FDM helps overcome a critical challenge in runtime HW/SW co-verification. Ideally, runtime co-verification can be done by directly observing the device states, the driver states and their interactions. However, in practice, observing the device internal states is difficult. Therefore, we utilize the FDM as a reference model of the device and we symbolically execute the FDM under the same driver requests, while the device internal states are modeled as symbolic values in the FDM. As a result, symbolic execution of the FDM explores all possible device internal behaviors at runtime.

We implement device checking by examining whether the device behaviors conform to the FDM. Detecting the nonconformance between the FDM and device serves two purposes: (1) Device bugs or unintended behaviors can be discovered; (2) If no nonconformance detected, properties holding on the FDM/driver interface also hold on the device/driver interface. Essentially, co-monitoring conducts runtime verification over the device and the driver indirectly through the FDM/driver co-verification.

We carry out the runtime co-verification over the FDM and the driver by building upon previous work on automata-theoretic approach to HW/SW co-verification [40]. The previous approach verifies HW/SW interface properties over the FDM and the driver statically. The FDM and the driver are modeled as a Büchi Automaton (BA) and a Labeled Pushdown System (LPDS) respectively while their combinations are modeled as a Büchi Pushdown System (BPDS). To verify system properties, reachability analysis is carried out over the BPDS by exploring the BPDS state space and the FDM and the driver are executed symbolically. Our approach also models the FDM/driver composition as a BPDS and explores the BPDS state space to verify system properties. However, our runtime co-verification explores the BPDS state space in a concolic way: the driver is

executed concretely while the FDM is explored symbolically.

This concolic exploration is the key to adapting static co-verification to runtime monitoring. The concolic exploration of BPDS brings three benefits: (1) symbolic execution helps cover possible device behaviors at runtime; (2) concrete execution of the driver largely constrains the state space, avoiding state space explosions; (3) concrete execution eliminates the need for modeling the environment for exercising the driver.

HW/SW co-monitoring essentially provides a unified solution for detecting and analyzing HW/SW interface defects. By efficiently monitoring the device, the driver, and their interface, it can discover most types of defects in HW/SW interfaces, ranging from hardware transient errors to driver bugs to malicious exploits. Moreover, by monitoring the device as well as the driver, HW/SW co-monitoring can easily identify a HW/SW interface defect as a hardware bug or a software bug.

We evaluated our approach on four Ethernet adapters and their Linux drivers. We simulated malicious exploits from both hardware and software across HW/SW interfaces. Our approach can successfully detect these injected malicious exploits. Moreover it detected several real bugs and security vulnerabilities. By analyzing these bugs, we showed that they either cause serious system failures or bring potential security issues. The results demonstrate that hardware/software co-monitoring has major potential in improving system reliability and security.

6.1.3 Contributions

HW/SW co-monitoring makes four major contributions to improving system reliability and security.

- It realizes runtime HW/SW co-verification by leveraging concolic execution, which detects errors and unintended behaviors in both devices and drivers.

- It helps narrow down an error from a HW/SW composition into a hardware violation of specification, a software violation of specification, or an interface error.

- It provides a mechanism for early detection of malicious exploits of HW/SW interface vulnerabilities, thereby facilitating mechanisms for protection before impact from the exploits affects the rest of the system.

- It facilitates detection of and protection against transient hardware failures. Such failures will be prevented from propagating deep into the system and devices and drivers will be brought back to normal modes.

6.2 HW/SW CO-MONITORING FRAMEWORK

6.2.1 Overview

This section gives an overview of our approach. As Figure 6.2 shows, the framework of HW/SW co-monitoring consists of three major components: a wrapper driver, a symbolic execution environment (SEE), and a property monitor (PM). The wrapper driver is used to capture the device state, the driver state, and the interactions between the device and the driver. The SEE symbolically executes the FDM by taking the driver request sequence as inputs, which realizes the co-execution with the device. Based on the co-execution, the SEE conducts device checking to ensure that the FDM shadows the device execution. The PM enforces a set of system properties which specify how the device and driver interact with each other. The PM carries out runtime HW/SW co-verification, which we call property checking. Basic functionalities of device checking and property checking are described as follows.

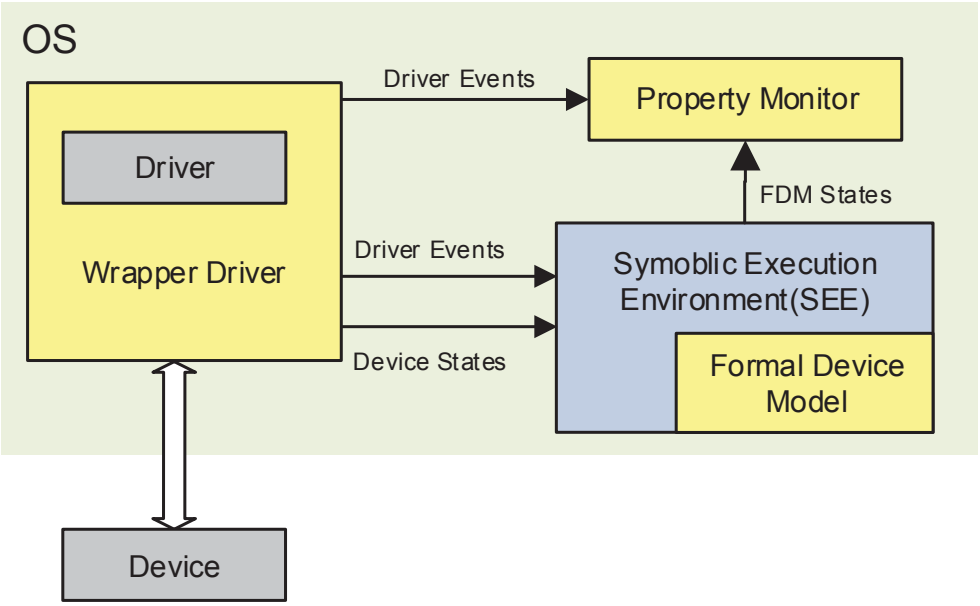


Figure 6.2: HW/SW co-monitoring framework

- *Device Checking.* The goal of device checking is to detect the divergence between the FDM and the device. We leverage a previous technique, conformance checking [38], which checks if the device behaviors conform to the FDM with symbolic execution. Through device checking, the device errors can be detected and if no divergence between the device and FDM, the FDM can be used as a model of the device for property checking.

- *Property checking.* PM carries out property checking based on device checking. It verifies system properties over the device and driver, indirectly over the composition of the FDM and driver. By property checking, invalid driver inputs or incorrect device interface states during device/driver interaction can be discovered.

6.2.2 Definitions

Based on the notion of conformance checking in previous work [38], we propose the definition of conformance between the FDM and the device. According to [38], a hardware device state is composited by a set of interface state variables denoted as R_I and a set of internal state variables denoted as R_N. We model a FDM state in the same way as the device state. A FDM state is defined as follows.

Definition 6.1. *A* **FDM state** *is denoted as* $V = \langle V_I, V_N \rangle$ *where* V_I *is the device interface state, i.e., the assignments to variables in* R_I *and* V_N *is the device internal state, i.e., the assignments to variables in* R_N.

A device state is defined as follows. Since at runtime the device internal state cannot be observed, we assign symbolic values to the device internal state variables.

Definition 6.2. *A* **device state** *is denoted as* $S = \langle S_I, S_N \rangle$ *where* S_I *is the assignments to variables in* R_I *and* S_N *is the symbolic assignments to variables in* R_N.

A FDM state or a device state can be viewed as a symbolic state if some of whose state variables values are symbolic. As Section 2.3 illustrates, a FDM can be modeled as a BA, $\mathcal{B} = (\Sigma, Q, \delta, q_0, F)$. In this way, a FDM state V can be modeled as a set of BA states $V = \{v_i \mid v_i \in Q, i{>}0\}$. Moreover, as a device state shares the same format as the FDM state. A device state S can be also modeled as a set of BA states. The conformance of a device state and a FDM state is defined as follows.

Definition 6.3 (State Conformance)**.** *A device state* S *conforms to a FDM state* V *if* $S \cap V \neq \emptyset$.

As both V and S can be treated as a set of BA states, V represents all the possible BA states that the FDM may have and S represents all the possible BA states that the device may have. The condition $S \cap V = \emptyset$ indicates that the FDM state and the device state cannot be the same, i.e., the device and FDM executions diverge at runtime.

6.2.3 Wrapper Driver

The wrapper driver captures: (1) the driver request issued to the device; (2) the device interface state before the driver request issued. Once a new driver request is issued, the wrapper driver sends a state-request pair to SEE. We denote such a state-request pair as $\langle S_I, \alpha \rangle$ where S_I is the device interface state and α is the current driver request.

Ideally, to verify HW/SW interfaces, the wrapper driver needs to capture the device state, the driver state, and the driver requests issued to the device. In our current implementation, property checking only checks the properties related to the driver requests and the device states, but not to the driver states. Therefore, the wrapper driver does not capture the driver states. For future work, we will extend our wrapper driver to capture the driver states.

Selective Capturing

The wrapper driver captures the device interface state: the device registers and their values. However, a peripheral device often has a large range of registers. Capturing all the registers heavily degrades the system performance. Previous work [38] proposes selective capturing: instead of all registers, only a small set of registers, important to device functionalities, are captured. For example, the reserved registers are typically not captured. Our wrapper driver adopts selective

capturing as our basic capturing method.

Sampling Reserved Registers

Selective capturing helps capture the registers important to the device functionalities. However, directly adopting this method does not fully meet our requirements for monitoring the device. In HW/SW co-monitoring, our framework monitors not only whether the defined behaviors are correct but also whether there are any undefined or abnormal behaviors. For example, the changes in reserved register values indicate potential malicious behaviors in the device interface. Reserved registers can be used to place the software program for code injection (see a concrete example in Section 6.3.1). As a result, the wrapper driver should capture reserved registers as well.

However capturing reserved registers faces a similar problem: the range of reserved registers is often large. To address this problem, we develop a method, namely reserved register sampling. We capture one reserved register every few registers. Our sample method works for the following reasons: (1) for large injected code, sampling can easily hit part of it; (2) for small injected code, we increase the chance of detection by sliding the sample windows every time we capture.

6.2.4 Device Checking

SEE conducts runtime device checking. It symbolically executes the FDM while continuously taking the state-request pairs from the wrapper driver. Algorithm 6.1 presents the work flow of runtime device checking. It takes a FDM $\mathcal{M}$ as inputs. Our device checking work flow has the following steps:

1. SEE gets the first request-state pair and intiaize FDM state V with the device interface state S_I;

2. SEE symbolically executes the FDM with the driver request α and the FDM state V;

3. SEE checks the conformance between the set of possible FDM next states G and the device next state S';

4. If the device conforms to the FDM, we construct the next FDM state and assign it to V, go to step 2. Otherwise, we report a device error.

The functions in Algorithm 6.1 are described as follows.

1. *Receiving Requests.* Function *receive_state_request*() is invoked to wait for and receive a state-request pair $\langle S_I, \alpha \rangle$ from the wrapper driver.

2. *Device State Construction.* Given a device interface state S_I, based on Definition 6.2, *construct_device_state* constructs a device state $S = \langle S_I, S_N \rangle$ where state variables in S_N are assigned symbolic values.

3. *Symbolic Execution.* Function *sym_exec* symbolically executes the FDM $\mathcal{M}$ and generates a set of FDM states, which is denoted as G.

4. *Conformance Checking.* As discussed above, symbolic execution of a FDM may produce a set of FDM states $G = \{g_i \mid 0 \leq i \leq n\}$. The next device state received from the wrapper driver is denoted as S'. Function *conformance_check* checks the conformance between the device and FDM. We define their conformance based on G and S' in Definition 6.4. Function *conformance_check* returns a set of the FDM states conforming to the device state, denoted as $H = \{h_i \mid 0 \leq i \leq m\}$.

5. *Device Error Report.* If the set H is empty, no conforming FDM state is produced, i.e., the FDM does not conform to the device at driver request α,

function *report_device_error* is invoked to record the device error, including the FDM execution trace, the driver request, and the state variables of the device, which are not equal to the state variables of the FDM.

6. *Next State Construction.* If H is not empty, the FDM and the device conform to each other at α. Based on $H = \{h_i \mid 0 \leq i \leq m\}$, function *construct_next_State* constructs the next FDM state V' as follows: $V' = \bigcup_{i=1}^{m} (set(h_i) \cap set(S'))$.

Definition 6.4 (Device Conformance)**.** *Given the set of FDM states* $G=\{g_i \mid 0 \leq i \leq n\}$ *and the device state* S'*, the device conforms to the FDM at* α *if* $\exists g_i \in G$ *where* $0 \leq i \leq n$*,* $S' \cap g_i \neq \emptyset$*.*

6.2.5 Property Checking

Property monitor verifies the enforced system properties over the BPDS representing the driver and the FDM. Device checking provides the foundation of property checking in two aspects. First, conformance checking between the FDM and device ensures that the FDM shadows the device execution trace. As a result, the property holding on the FDM/driver interface also holds on the device/driver interface. Second, the symbolic execution of the FDM with the concrete driver execution verifies the enforced system properties.

Runtime Verification of System Properties

Property checking conducts reachability analysis over the BPDS by leveraging our concolic exploration of the FDM and the driver. To detect if a property is violated, we implement a special method for runtime verification. In the static HW/SW co-verification, a property ψ is violated as long as there is a BPDS path where $\neg\psi$ is

reachable. However, this method cannot be used in our runtime verification since we cannot observe the device internal states and we model possible device internal states with symbolic variables. From the existence of such a BPDS path, it is insufficient to conclude it is a property violation. Therefore, we develop a method for property violation detection that is more conservative: only if the property is violated under all possible situations, we report a property violation.

As mentioned in Section 6.2.4, the symbolic execution of the FDM with the concrete driver execution explores the BPDS state space. Under a single driver request α, the execution of the FDM and the driver explores a set of BPDS paths, denoted as P. Based on this notion, Definition 6.5 gives the condition where the property is violated.

Definition 6.5 (Property violations)**.** *Given a property ψ, a set of BPDS paths $P = \{p_i \mid 0 \leq i \leq n\}$ explored under a driver request α, ψ is violated under α if $\forall p_i \in P$, $\neg\psi$ is reachable on p_i.*

The set P represents all possible device and behaviors under the current driver request α. Only if all of these possible behaviors lead to the violation of the property ψ, the PM can ensure there is an invalid driver request triggering the violation.

Implementation of Property Checking

Since the PM only maintains a relatively small set of properties, we integrate the PM into the SEE. This way, we leverage the FDM and driver executions to explore the BPDS state space and the FDM can be directly used as a validation vehicle. For property checking, we first specify the properties in assertions and then instrument the FDM with the assertions. While the device checker simulates the device behaviors with the FDM, the PM detects if any assertions fail during

the simulation. Currently we instrument the assertions manually. In future, we will develop a method that allows the users to specify assertions in a certain format and automatically instruments the assertions.

We use a system property from EEPRO100 as an example to show how we specify an assertion and instrument the assertion into the FDM. The property specified in EEPRO100 specification [27] is as follows. As Figure 6.3 shows, a special API function *comon_assert* is used to instrument the property.

Property: *If the device Command Unit (CU) is not in* `SUSPENDED` *status, the driver cannot send* `RESUME` *to the device.*

```
static void eepro100_cu_command(EEPRO100State* s,
                                uint8_t val)
{
    // Assertion to enforce the example property
    if (s->cu_state != CU_STATE_SUSPENDED)
      comon_assert(val != CU_CMD_RESUME);
    ......
}
```

Figure 6.3: Assertions instrumented in EEPRO100 FDM

Note. The system properties that the PM can verify depend on the device interface state, the driver requests, and the driver state. However, as mentioned in Section 6.2.3, in our current implementation, the PM only verifies the properties involving the driver request and device interface state. We will extend the PM to verify properties related to driver states in future work.

6.3 APPLICATIONS IN SECURITY

As mentioned in Section 6.1.2, our approach can be used in not only detecting device or driver bugs but also catching malicious exploits of HW/SW interface vulnerabilities. In this section, we elaborate on the types of malicious behaviors across HW/SW interfaces. Furthermore, for each category of malicious attacks, we present how our HW/SW co-monitoring framework detects these attacks.

6.3.1 Threat Model

Software Attack

Hardware interfaces exposed to software are vulnerable. These vulnerabilities can be exploited by malicious software. In device/driver interfaces, device specifications usually specify system rules which the driver should follow. As a consequent, if the driver issues invalid commands which do not follow the rules, the device can be easily driven to unresponsive state. For example, in Intel eepro100 Ethernet, if the device driver issues a command to activate the device when the device is already at the "active" state, the device will be driven to an unresponsive state. Malicious software can issue invalid commands to crash or hang the device.

```
while (ioread16(ioaddr + Wn7_MasterStatus))
            & 0x8000)
            ;
```

Figure 6.4: Excerpts from 3c59x driver.

Hardware Attack

Software interface to hardware is vulnerable as well. Improper handling of hardware inputs can easily cause the driver hang or even the system crash. Figure 6.4 shows an example which is illustrated by [30]. Function "ioread" reads a data from the device. If the value read from the device is not proper. The driver will loop forever. This example shows that the driver vulnerabilities can be exploited by malicious hardware. The malicious hardware can feed incorrect inputs to the driver which might crash or hang the driver.

Except the malicious hardware inputs to the driver, there are other attacks which can be done by hardware especially by hardware trojans. We list some of them as follows.

- *Denial of Services (DoS).* Besides issuing invalid inputs to the driver, hardware trojans can make the device unresponsive to any incoming data and commands.

- *Stealing user secretes.* In many systems, devices are used to encrypt password and data. Hardware trojans residing in devices can observe the encryption key. Moreover, they can reuse available hardware resources to send over the key through the network.

- *Code injection.* Hardware trojans can also have potential to injected code into the runtime system, i.e., the code injected by the hardware trojan is executed in the OS kernel. For example, DMA attack is a way to access the physical memory via DMA. Several examples have demonstrated that DMA attacks can inject code into the OS kernels. Furthermore, modern peripheral devices usually have a large piece of Memory-Mapped I/O (MMIO) registers which can be accessed by CPU as accessing the normal physical memory.

Moreover, a significant portion of MMIO registers is reserved which should be not used by either software or hardware. Therefore, hardware trojans can easilly place the injected code into the MMIO interface without affecting the normal system logic. We have implemented a DMA attack which gains the root privilege of the OS through code injection, described as follows.

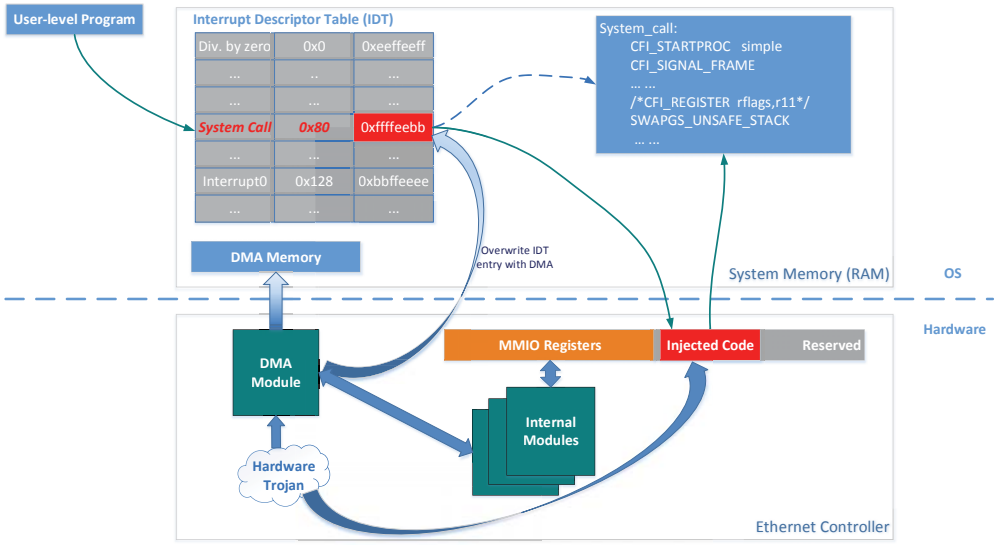


Figure 6.5: Work flow of a hardware trojan attacking OS through hooking system calls

Example. Figure 6.5 shows the work flow of this attacking scenario. In this example, a hardware trojan in an Ethernet controller takes over the OS by injecting code through DMA attacks. It has two steps: (1) Placing the code to be executed in MMIO reserved registers; (2) Hooking system calls by modifying the Interrupt Descriptor Table (IDT) through DMA. As a result, when each system call is executed, the injected code is executed first. Moreover, injected code is executed in the OS kernel space which has

root privilege. Essentially the injected can do everything. In this example, it opens the SSH port and create an account with "sudo" privilege.

6.3.2 Detecting Malicious Attacks

Our HW/SW co-monitoring infrastructure can detect the HW/SW malicious behaviors by monitoring hardware interfaces and software inputs to hardware. We explain how our approach detects each category of malicious attacks respectively.

Detecting Software Attacks

The property monitor in our framework can help detection of software attacks to hardware. In the property monitor, a set of system properties and security policies are enforced. By verifying these properties at runtime, malicious driver commands which violate the property can be detected.

Detecting Hardware Attacks

- *Detections of DoS.* When the device generates an invalid inputs to the driver or the device stops working, its interface state is incorrect. In HW/SW co-monitoring, the invalid interface state will lead to interface inconsistencies between the hardware device and the FDM. Therefore, our runtime device checking can successfully detect DoS attacks.

- *Detections of stealing user privacy.* The hardware trojan which steals encryption keys finally has to employ the network device to send over the key. In Ethernet adapters, there are several registers, called statistics registers, which record counts of packet receiving and transmitting. Hardware trojans are usually embedded in its own hijacked module, which cannot easily manipulate the Ethernet adapter to avoid recording extra packet transmission.

In this way, there are inconsistencies of statistics registers. By detecting these inconsistencies, the device checker in our framework can detect the underlying user secrete leaks

- *Detections of code injection.* As mentioned in Section 6.3.1, a large MMIO reserved registers are an ideal location for the hardware trojan to place their code to be executed, as this will not affect the normal system work flow. Once the injected code has been placed in MMIO interface, there must be an inconsistency of reserved registers between the device and the FDM. By monitoring reserved registers, our device checker can uncover the malicious behaviors over the reserved registers including code injection.

In the evaluation section, we simulate the concrete scenarios for each type of malicious attacks. The results show that our framework can detect them successfully (see Section 6.4.2 for more details).

6.4 EVALUATION

This section evaluates our approach from three aspects. First, we simulate four hacking scenario through HW/SW interfaces. By applying our framework, we demonstrated that our approach can successfully detect malicious attacks through HW/SW interfaces in reasonable delays. Second, we present several real device bugs and driver bugs, which shows that HW/SW co-monitoring detects real defects and vulnerabilities. Third, we evaluate the overhead introduced by our framework, demonstrating our approach is efficient.

6.4.1 Experiment Setup

We have performed our experiments on a workstation with a dual-core Intel Pentium D Processor with 4GB of RAM and Ubuntu Linux OS with 64-bit kernel version 2.6.38. We applied our framework to four Ethernet adapters and their FDM generated from their QEMU virtual devices. Information about these devices and their FDM are summarized in Table 6.1. The FDM size is measured in Lines of Code (LoC).

Table 6.1: Devices and FDMs for HW/SW co-monitoring

Devices	**FDM Size (LoC)**	**Basic Description**
RealTek rtl8139	2211	RealTek 10/100M NIC
Intel eepro100	1032	Intel 10/100M NIC
Intel e1000	1432	Intel Gigabit NIC
Broadcom bcm5751	2103	Broadcom Gigabit NIC

Table 6.2: Summary of software attack injection

Driver	**Property Description**	**Consequence**
rtl8139	The driver should not start new transaction when another transaction is in progress	Device hangs
eepro100	The driver should not issue START while the device is working already	system hangs
e1000	The driver should not issue command when MDIC is not clear	Device hangs
bcm5751	The driver should not start new EEPROM transaction when previous update is not finished	Device hangs

Table 6.3: Summary of detected bugs

No.	Description	Dev./Drv.	Num.
1	Driver writes a value to a read-only register.	eepro100	1
2	Driver updates the reserved register bits	e1000	1
3	Device updates reserved registers	eepro100	2
4	Device updates reserved registers	e1000	2

6.4.2 Attacks Detection

To demonstrate the capacity of catching malicious attacks, we injected several malicious attacks issued from both devices and drivers. To simulate malicious hardware, we modified the QEMU virtual devices which emulates the devices listed in Table 6.1. By using virtual devices, we can easily injected our hacking scenarios. On one hand, our virtual machine acts as a real physical machine where modified virtual devices act as malicious hardware. On the other hand, our framework utilizes the FDMs specifying the correct hardware behaviors. Malicious behaviors can be detected as there are inconsistencies between the generated FDMs and the injected virtual devices. To simulate malicious software, we directly modify the Linux drivers.

We simulate three malicious attacks which are all described in Section 6.3.1. We elaborate on more details as follows.

- **Software Violation of System Properties (SVSP).** The invalid software commands can easily hang or crash the device. We modify the drivers to issue malicious commands violating the system properties. Thereby the device or the system hangs. For each driver, we simulate one violation of a system property. The system properties violated and the consequence of

corresponded violations are summarized in Table 6.2.

- **Hardware DoS Attacks (HDoS).** We simulate a hardware trojan in QEMU virtual devices. By sending a special UDP packet to the target system, the hardware trojan is triggered, which issues an attack making the device stop working and unresponsive to any incoming driver requests.
- **Hardware Device Attacks (HDA).** As Section 6.3.1 describes, a hardware trojan can inject code through DMA attacks through device interfaces exposed to the system. In the evaluation, we simulate a hardware trojan which takes over the OS by injecting code through DMA attacks.

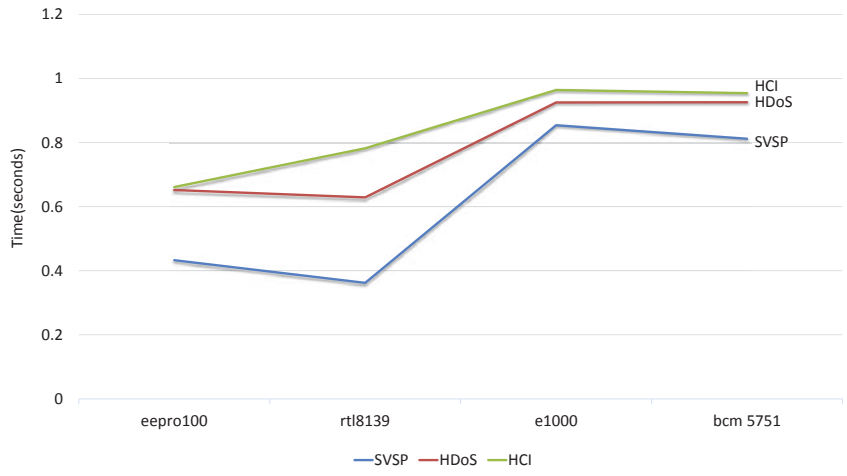


Figure 6.6: Time delayed in detecting attacks

The results show that our framework successfully detected the malicious attacks. Figure 6.6 shows the corresponded delay in seconds which means the time from the attacks occurs to the attacks have been detected. According to Figure 6.6, we can clearly see that the delay is small in eepro100 and rlt8139 while it is relatively high in e1000 and bcm5751. The reason is because eepro100 and rtl8139 are

all 10/100M adapters which have smaller interface memory, comparing to e1000 and bcm5751. As a result, the sizes of captured registers are smaller in eepro100 and rtl8139, which incur less overhead at runtime.

6.4.3 Bug Detection

Our approach can detect errors from both devices and drivers. We discuss device bugs and driver bugs respectively.

- *Driver bugs.* In the four Linux drivers with their devices, our approach detects two real driver bugs. This two bugs generally violates system properties and can cause serious problems. For example, in the second driver bug, updating the reserved registers is the same behavior as the code injection example we demonstrate in Section 6.3.1.

- *Device bugs.* We discovered four real device bugs, which are all related to unspecified hardware behaviors, e.g., the register values are changed randomly and reserved register bits are touched. This phenomenon is the same as code injection through reserved registers. As these behaviors are unknown to the system and device drivers, which should be considered as security threats.

Summary. All the bugs discovered occur on HW/SW interfaces, and involve interactions between drivers and devices. By discovering these bugs, it demonstrates the capacity of our framework in detecting the bugs and malicious exploits crossing HW/SW boundaries. Furthermore, when such a bug happen, it is hard to identify which side of device/driver goes wrong. For example, invalid software command can drive the device hang. It appears like a hardware error rather than a software defect as the device is unresponsive. However, our framework can clearly identify which side is wrong and the reason. Therefore, our approach is not only

useful in detecting bugs but also helpful in troubleshooting the bugs.

6.4.4 Performance

In this section, we evaluate the overhead introduced by our HW/SW co-monitoring framework. We mainly focuses on CPU and memory usages. We evaluate CPU and memory usages under three test cases. These test cases are common usages of an Ethernet adapter, including "load driver", "scp files", and "reset device".

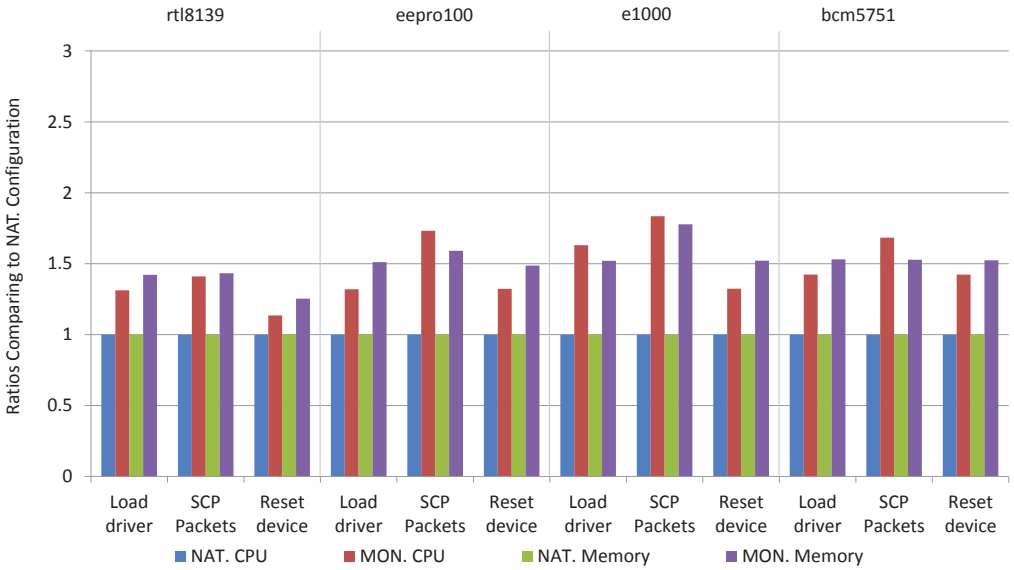


Figure 6.7: CPU and memory usages of test cases under NAT. and MON. configurations. The usages under NAT. configuration are normalized to 1.

We compare the CPU and memory usages of two configurations. The first one is the native system without our co-monitoring framework, denoted "NAT". The second one is the monitored system with our framework, denoted as "Mon". Figure 6.7 shows the results, where the usages in NAT. are normalized to 1. The results show that under most of test cases, our co-monitoring approach introduces

reasonable overhead. In the "scp files" scenario, the CPU usages are large. The reason is because under "scp files" incurs intensive data transfer as well as device/driver events, our framework is intensively executed and introducing large overhead. Moreover, as the FDMs of e1000 and bcm5751 are more complicated, the overheads on them are larger than other two devices. We discuss the potential optimizations to minimize the introduced overhead in Section 6.6.

6.5 RELATED WORK

Our approach is close to the approaches described as follows.

Lei, et al. [38, 37] propose a post-silicon conformance checking approach to checking if a virtual device conforms to its silicon device. It captures silicon device traces at runtime and checks the conformance with the virtual device by processing the captured traces offline. We apply this technique to check if the device conforms to the FDM at runtime. Our approach has two major improvements: (1) HW/SW co-monitoring is an on-line approach, monitoring device/driver interactions while the system is running; (2) we utilize a FDM as a golden reference abstracting unnecessary details, which makes runtime checking efficient.

Li, et al. [40] present an automata-theoretic approach to hardware/software co-verification. Such an approach models a hardware/software combination as a BPDS by synthesizing a BA representing hardware and a LPDS representing software. It uses model checking techniques to explore the state space of BPDS to detect property violations. In our approach, we use BPDS to represent hardware/-software interface and conduct runtime verification.

6.6 SUMMARY

Our approach can be further improved in two aspects. First, how to efficiently generate a FDM with respect to the hardware specification. Second, how to minimize the overhead of co-monitoring.

FDM generation. Recently virtual prototypes are widely used to enable early software development before silicon hardware is available [45]. The FDM and the virtual prototype of a device are both rooted in the device specification. Therefore, we can reuse the implementations of the virtual prototype to build its corresponding FDM. Building FDMs from virtual prototypes has two benefits. First, extending virtual prototypes avoids duplicated efforts. Second, in the validation stage, a virtual prototype usually has already gone through a number of conformance checking iterations and has been thoroughly validated. Reusing virtual prototype implementations can help develop high-quality FDMs. In future work, we will developed a systematic approach to deriving the FDM from the virtual prototype.

Reducing runtime overhead. As we discussed in Section 6.4.4, the major performance downgrade is caused by intensive device/driver events. As for each driver request, our framework symbolically executes the FDM under the driver request, intensive driver request traffic will lead to significant overhead incurred by symbolic execution. Therefore, if we can reduce the number of driver requests triggering symbolic execution of FDM, the runtime performance will be improved. A potential solution to address the performance issue is "caching" the driver request and FDM state transitions. Since the device and the driver often work under repeated scenarios, for example, an Ethernet adapter and its driver repeatedly receive and send packets, the same driver request and FDM state often appear many

times. For this reason, we can "cache" the driver request and FDM state transitions explored by symbolic execution of the FDM. If we encountered the same driver request and the FDM state, we can directly fetch the FDM state transitions without invoking symbolic execution. Thereby, the overhead introduced by symbolic execution will be reduced.

This chapter has presented a HW/SW co-monitoring approach, which monitors a hardware device and its driver simultaneously. By monitoring the device, the driver, and their interfaces, the bugs and malicious behaviors appear over device/driver interfaces. We evaluate our approach on four devices and drivers. The results are promising: (1) our approach detected all the malicious attacks injected in devices and drivers; (2) our approach also discovered several real bugs from both devices and drivers; (3) our approach introduced reasonable overhead into the runtime system. The results demonstrate that our approach is effective and efficient in providing an early detection of bugs and malicious exploits through HW/SW interfaces.

Algorithm 6.1 Device_Checking($\mathcal{M}$)

1: $\langle S_I, \alpha \rangle \leftarrow receive\_state\_request()$
2: $S \leftarrow construct\_device\_state(S_I)$
3: /*Initialize FDM state V to be device state S*/
4: $V \leftarrow S$
5: **while** $\alpha \neq$ `NULL` **do**
6: /*Symbolically execute FDM taking α at V state.*/
7: $G \leftarrow sym\_exec(\mathcal{M}, V, \alpha)$
8: $\langle S'_I, \alpha' \rangle \leftarrow receive\_state\_request()$
9: $S' \leftarrow construct\_device\_state(S'_I)$
10: $H \leftarrow conformance\_check(G, S')$
11: **if** $(H \neq \emptyset)$ **then**
12: $report\_device\_error()$
13: $abort()$
14: **end if**
15: $V' \leftarrow construct\_next\_State(H)$
16: $V \leftarrow V'$
17: $\alpha \leftarrow \alpha'$
18: **end while**

Chapter 7

CONCLUSIONS AND FUTURE WORK

HW/SW interfaces are pervasive in all kinds of computer systems ranging from smart phones, tablets to personal computers to cloud servers. These systems have high requirements on reliability and security. However, assuring HW/SW interface reliability and security is difficult, not only due to the intrinsic complexity of HW/SW interfaces, but also various challenges posted in different stages of the computer system life cycle. At the post-silicon validation stage, HW/SW integration validation is challenging as it is lack of effective methods for bug detection and troubleshooting. At the system deployment stage, hardware transient errors, hardware trojans, and software virus make HW/SW interfaces insecure and unreliable as well.

This dissertation research has successfully demonstrated that HW/SW interface defects and vulnerabilities can be effectively detected through the systematic co-validation over HW/SW interfaces. Moreover, our two-schemed assurance solution has major potentials in addressing the challenges of the HW/SW interface assurance over the system life cycle: (1) the HW/SW co-validation facilitates HW/SW integration testing/debugging at post-silicon stage; and (2) HW/SW co-monitoring provides the foundation for continuously protecting HW/SW interfaces after the system has been released.

7.1 SUMMARY OF CONTRIBUTIONS

This dissertation presents a comprehensive solution for validating HW/SW interfaces over the system life cycle. The dissertation makes following major contributions:

1. **Conformance checking with virtual prototypes.** Conformance checking addresses two major problems in the current industry practice: (1) lack of transaction-level validation methods for validating silicon hardware; (2) Difficulty of migrating drivers from virtual platforms to the real silicon hardware. We discuss the contribution of conformance checking from these two aspects. First, the state of the art post-silicon validation and debugging mainly focuses on the circuit level implementations which requires embedding hardware monitors into circuits. Conformance checking essentially provides an effective and light-weight method from the operating system level which does not require any hardware support. Second, due to the inconsistencies between the device and its virtual prototype, drivers developed over virtual prototypes often do not work readily on silicon devices because of either silicon device bugs or driver bugs hidden on virtual devices. By detecting the inconsistencies between the virtual prototypes and the devices, conformance checking provides a systematic and efficient way to (1) expose the virtual prototype or device errors; (2) reveal the causes of driver bugs hidden on the virtual prototypes. According to thees two contributions, conformance checking technique is expected to have a broad impact on not only hardware validations but also HW/SW integration testing.

2. **HW/SW co-validation framework.** Our HW/SW co-validation approach is built upon the conformance checking approach. It provides a comprehensive solution to address the challenges in HW/SW integration testing: (1) lack of HW/SW interface observation; (2) difficulty in attributing HW/SW interface bugs; (3) difficulty in troubleshooting HW/SW interfaces. First, our framework employs the trace recorder, which efficiently observes the HW/SW interfaces. Second, when encountering a HW/SW interface bug, based on the conformance checking result, we can clearly see if it is a hardware bug. Moreover, by analyzing the property checking result, it will identify the bug as a software property violation, i.e., a software bug or a hardware property violation, i.e., a hardware bug. Third, our co-validation framework automatically analyzes the trace produced by the HW/SW interface, largely saving human efforts. Our co-validation approach based on conformance checking can detect the bugs ranging from the devices, the virtual prototypes, and the drivers.

3. **Adaptive concretization.** Symbolic execution and other formal analysis techniques, such as model checking, often face the state explosion problem, which incurs significant overhead. To address this challenge, concolic execution and abstraction refinement are two common ways which are often used. Concolic (a portmanteau of concrete and symbolic) testing is a hybrid testing technique that integrates concrete execution with symbolic execution. Abstraction refinement is an iterative process where the target model or program is abstracted first, then it is added with more details when the counterexamples are discovered. Adaptive concretization borrows the ideas from both concolic execution and abstraction refinement: (1) we partially concretize the state variables of the model and make them symbolic again

when a counterexample is discovered. Adaptive concretization essentially provides an alternative approach in formal analysis to reducing symbolic execution overhead. It is particularly effective for dealing with the case where concretizations are correct most of time. Our research dissertation demonstrated that adaptive concretization significantly improves the efficiency of conformance checking and scales it to complicated hardware designs.

4. **HW/SW co-monitoring.** HW/SW co-monitoring is a relative new idea which realizes runtime HW/SW co-verification. The most important feature of HW/SW co-monitoring is concolic exploration of the HW/SW interface model, i.e., the BPDS. Concolic execution leverages the runtime execution trace of the system to reduce the verification overhead. We believe that concolic exploration is a practical way to realize efficient runtime verification over the system. The idea of concolic exploration can be applied to not only runtime HW/SW co-verification but also pure software or hardware runtime verification as well.

5. **Evaluation.** This dissertation research has been realized in two software tools, DCC (Device Conformance Checker) and CoMon (Co-Monitoring). These two software tools have been applied to 4 real industry hardware devices, the device drivers, and their reference models (virtual prototypes and FDMs). They discovered 42 real bugs ranging from the devices to the drivers and to the reference models. Although all these four devices have been released for several years, which have gone through rigorous testing procedures, our approach is still able to find these non-trivial bugs. All these results demonstrate that our approach has a significant potential in validating real industry hardware and software designs.

7.2 FUTURE RESEARCH DIRECTIONS

7.2.1 Pre-silicon HW/SW Co-validation

Motivation. As virtual prototypes are widely adopted for early software development, HW/SW integration validation is generally moved forward from the post-silicon validation stage to the pre-silicon validation stage where the silicon hardware has not been available yet. At the pre-silicon validation stage, HW/SW interface assurance is also challenging as due to following reasons: (1) the hardware designs including specifications, RTL designs, and virtual prototypes evolve very quickly. It is difficult for software developers to keep synchronous with hardware designs; (2) there is often a large divergence between a RTL design and its corresponded virtual prototype since they are developed by separated teams. As a result, there is always a concern that the driver developed over the virtual prototype might have hidden bugs which will be revealed in the post-silicon stage; (3) although virtual prototypes are white-boxes which might help troubleshoot HW/SW interface bugs, HW/SW interface debugging is still difficult. There are two reasons: first, driver developers or system testers do not fully understand hardware devices and virtual prototypes; second, troubleshooting an interface bug is still a time-consuming process. It requires sifting through logs recording HW/SW interface interactions and even digging into virtual prototype internals.

Solution. Adapting our post-silicon HW/SW co-validation framework for pre-silicon co-validation is a promising direction to address the above problems. The pre-silicon HW/SW co-validation entails three techniques: (1) conformance checking among the different versions of virtual prototypes and device RTL designs; (2) conformance checking between a RTL model and its corresponded virtual prototype; (3) property checking over the driver and the virtual prototype. These

three techniques address the three challenges respectively: (1) conformance checking among different versions help quickly identify the differences between each two versions of virtual prototypes and RTL designs. This helps driver developers understand the new features and the reason why the driver does not work on the new virtual prototype; (2) conformance checking between RTL and virtual prototypes help detect bugs in both RTL designs and virtual prototypes thereby discovering design defects at the early stage; (3) property checking and conformance checking together can automatically analyze HW/SW interface traces and pinpoint the root cause of a HW/SW interface bug. This will save human efforts on troubleshooting.

Research Challenges. There are several research tasks to develop the pre-silicon HW/SW co-validation: (1) the method for conformance checking among different versions of virtual prototypes; (2) the method for conformance checking among different versions of RTL models; (3) the method and algorithm for conformance checking between a RTL model and its virtual prototype. For conformance checking between two virtual prototypes, first, we can extract the simulation trace from the virtual platform where one of the virtual prototypes is executed together with its driver. Second, we replay the simulation trace to the other virtual prototype and compare their states. For conformance checking between two RTL models, we can use the same method as conformance checking of virtual prototypes. However, there are two research problems which need to be addressed: first, how to properly extract the simulation trace and what information would be included? Second, what is the algorithm to compare their states? As to conformance checking between a RTL design and its virtual prototype, a potential approach is to extract the simulation trace from the RTL simulator, replay the trace to the virtual prototype, and check their conformance. Nevertheless, a RTL model is a clock-driven model and virtual prototypes are usually modeled at transaction-level. A major

challenge is how to properly sample the RTL model simulation trace and check the conformance against virtual prototype.

7.2.2 Detecting Hardware Trojan and Malwares in Virtual Devices

HW/SW co-monitoring is demonstrated as an effective method in detecting hardware trojans at the system deployment stage. However, the hardware trojans should be discovered as early as possible. Therefore, it is also desired to detect hardware trojans at the post-silicon validation stage. However, hardware trojans manufactured in third-part IP modules are usually deeply embedded in the chip, which are quite difficult to triggered under the validation environment.

There are two major problems to be addressed for detecting hardware trojans: (1) how to generate effective test cases to trigger hardware trojans? (2) how to effectively discover hardware trojans when they are triggered? We plan to use FDMs as reference models for test case generation and trojan detection. Our approach entails following techniques.

1. **Enhancing FDM with undefined behaviors.** FDMs have already been demonstrated as effective and abstract reference models in HW/SW co-monitoring. A FDM models all the device correct behaviors defined by specifications. However, to detect hardware trojans, we should focus on the hardware undefined or even malicious behaviors which are out of boundary of the FDM. Thereby the FDM should be enhanced to include not only correct behaviors but also the undefined and malicious behaviors.

2. **Automatic test generation with FDM.** Some existing approach [15] has demonstrated that virtual prototypes can be used as references for generating high-quality test cases. These test cases can be used for testing silicon

hardware. We leverage this idea for generating test cases to trigger hardware trojans with our enhanced FDMs. As a FDM defines unknown and malicious behaviors of the device, corresponded test cases triggering these behaviors can also be generated. These test cases can be applied to trigger the hardware trojans in silicon hardware.

3. **Conformance checking for detecting hardware trojans.** To detect hardware trojans when they are triggered, we apply our conformance checking approach over the FDM and the silicon device. The FDM services as a reference model and the malicious behaviors appearing over the device interface will be detected.

Malwares in Virtual Devices

Recently hyperviors or Virtual Machine Monitors (VMM) are widely used to provide system virtualizations in cloud computing. A significant component of a hypervisor or a VMM is a number of virtual devices which emulate the hardware devices for guest operating systems. Hypervisors or VMMs can be malicious: malwares or virus in virtual devices can easily propagate into the guest systems through HW/SW interfaces. Thereby it is also highly desired to detect malwares in virtual devices. Our hardware trojan detection method can be also applied to detect malwares in virtual devices.

REFERENCES

[1] Miron Abramovici, Paul Bradley, Kumar Dwarakanath, Peter Levin, Gerard Memmi, and Dave Miller. A reconfigurable design-for-debug infrastructure for socs. In *Proceedings of the 43rd Annual Design Automation Conference (DAC)*, pages 7–12, New York, NY, USA, 2006. ACM.

[2] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. Symbolic execution with abstract subsumption checking. In *Proceedings of the 13th International Conference on Model Checking Software (SPIN)*, pages 163–181, Berlin, Heidelberg, 2006. Springer-Verlag.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HOTOS)*, pages 33–, Washington, DC, USA, 2001. IEEE Computer Society.

[4] Ran Avinun. Concurrent hardware/software development platforms speed system integration and bring-up.

[5] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 289–300, New York, NY, USA, 2007. ACM.

[6] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys)*, pages 73–85, New York, NY, USA, 2006. ACM.

[7] Mauro Baluda, Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Structural coverage of feasible code. In *Proceedings of the 5th Workshop on Automation of Software Test (AST)*, pages 59–66, New York, NY, USA, 2010. ACM.

[8] D. Becker, R. K. Singh, and S. G. Tell. An engineering environment for hardware/software co-simulation. In *Proceedings of the 29th ACM/IEEE Design Automation Conference (DAC)*, pages 129–134, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[9] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (USENIXATC)*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[10] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. RWset: attacking path explosion in constraint-based test generation. In *Proceedings of International Conference on Tools and Algorithms for the Constructions and Analysis of Systems (TACAS)*, 2008.

[11] M. Boule, J. Chenard, and Z. Zilic. Adding debug enhancements to assertion checkers for hardware emulation and silicon debug. In *Computer Design, 2006. International Conference on (ICCD)*, pages 294–299, Oct 2006.

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and

automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[13] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–278, New York, NY, USA, 2011. ACM.

[14] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 73–88, New York, NY, USA, 2001. ACM.

[15] Kai Cong, Fei Xie, and Li Lei. Automatic concolic test generation with virtual prototypes for post-silicon validation. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 303–310, Piscataway, NJ, USA, 2013. IEEE Press.

[16] Kai Cong, Fei Xie, and Li Lei. Symbolic execution of virtual devices. In *Proceedings of the 2013 13th International Conference on Quality Software (QSIC)*, pages 1–10, Washington, DC, USA, 2013. IEEE Computer Society.

[17] Semiconductor Research Corporation and Computing Community Consortium. Research needs for secure, trustworthy, and reliable semiconductors, 2013.

[18] F. M. De Paula, M. Gort, A. J. Hu, S. Wilton, and J. Yang". Backspace:

Formal analysis for post-silicon debug. In *Formal Methods in Computer-Aided Design, 2008 (FMCAD)*, pages 1–10, Nov 2008.

[19] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[20] A. Ghosh, M. Bershteyn, R. Casley, C. Chien, A. Jain, M. Lipsie, D. Tarrodaychik, and O. Yamamo. A hardware-software co-simulator for embedded system design and debugging. In *Proceedings of the 1995 Asia and South Pacific Design Automation Conference (ASP-DAC)*, New York, NY, USA, 1995. ACM.

[21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, New York, NY, USA, 2005. ACM.

[22] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[23] R. K. Gupta, C. N. Coelho, Jr., and G. De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proceedings of the 29th ACM/IEEE Design Automation Conference (DAC)*, pages 225–230, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[24] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Fault isolation for device drivers. In *Proc. of International Conference on Dependable Systems and Networks*, 2009.

[25] A. Hoffman, T. Kogel, and H. Meyr. A framework for fast hardware-software co-simulation. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 760–765, Piscataway, NJ, USA, 2001. IEEE Press.

[26] Alan J. Hu, Jeremy Casas, and Jin Yang. Efficient generation of monitor circuits for gste assertion graphs. In *Proceedings of the 2003 IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pages 154–159, Washington, DC, USA, 2003. IEEE Computer Society.

[27] Intel. *Intel 8255x 10/100 Mbps Ethernet Controller Family – Open Source Software Developer Manual*, 1.3 edition, January 2006.

[28] Intel. *Intel Ethernet Controller X540 Specification Update, Revision 2.6*, 2013.

[29] ITRS. International technology roadmap for semiconductors, 2011 edition. http://www.itrs.net.

[30] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating hardware device failures in software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)*, pages 59–72, New York, NY, USA, 2009. ACM.

[31] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[32] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[33] Robert P. Kurshan, Vladimir Levin, Marius Minea, Doron Peled, and Hüsnü

Yenigün. Combining software and hardware verification techniques. *Formal Methods in System Design (FMSD)*, 21(3):251–280, November 2002.

[34] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with DDT. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC)*, pages 12–28, Berkeley, CA, USA, 2010. USENIX Association.

[35] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 193–204, New York, NY, USA, 2012. ACM.

[36] Li Lei, Kai Cong, and Fei Xie. Optimizing post-silicon conformance checking. In *Computer Design, 2013 IEEE 31st International Conference on (ICCD)*, pages 499–502, 2013.

[37] Li Lei, Kai Cong, Zhenkun Yang, and Fei Xie. Validating direct memory access interfaces with conformance checking. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 9–16, Piscataway, NJ, USA, 2014. IEEE Press.

[38] Li Lei, Fei Xie, and Kai Cong. Post-silicon conformance checking with virtual prototypes. In *Proceedings of the 50th Annual Design Automation Conference (DAC)*, pages 29:1–29:6, New York, NY, USA, 2013. ACM.

[39] Juncao Li, Xiuli Sun, Fei Xie, and Xiaoyu Song. Component-based abstraction and refinement. In *Proceedings of the 10th International Conference on Software Reuse (ICSR)*, volume 5030 of *Lecture Notes in Computer Science*, pages 39–51, Berlin, Heidelberg, May 25-29 2008. Springer.

[40] Juncao Li, Fei Xie, Thomas Ball, Vladimir Levin, and Con McGarvey. An automata-theoretic approach to hardware/software co-verification. In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 6013 of *Lecture Notes in Computer Science*, pages 248–262. Springer, March 20-28 2010.

[41] Juncao Li, Fei Xie, Thomas Ball, Vladimir Levin, and Con McGarvey. Formalizing hardware/software interface specifications. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 143–152, Washington, DC, USA, 2011. IEEE Computer Society.

[42] Microsoft. Driver Verifier (DV). http://msdn.microsoft.com/en-us/library/windows/hardware/ff545448.aspx, 2008.

[43] Brendan Murphy and Mario R. Garzia. Software reliability engineering for mass market products. *Software Reliabilty Engineering*, 8(1), December 2004.

[44] José Augusto Miranda Nacif, Flávio Miana de Paula, Harry Foster, Claudionor José Nunes Coelho Jr., and Antônio Otávio Fernandes. The chip is ready. am i done? on-chip verification using assertion processors. In *Proceedings of 11th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SOC)*, 2003.

[45] Shannon Nelson and Peter Waskiewicz. Virtualization: Writing (and testing) device drivers without hardware. In *Proceedings of Linux Plumbers Conference*, 2011.

[46] Sung-Boem Park and Subhasish Mitra. IFRA: Instruction footprint recording

and analysis for post-silicon bug localization in processors. In *Proceedings of the 45th Annual Design Automation Conference (DAC)*, pages 373–378, New York, NY, USA, 2008. ACM.

[47] Claudio Passerone, Luciano Lavagno, Massimiliano Chiodo, and Alberto Sangiovanni-Vincentelli. Fast hardware/software co-simulation for virtual prototyping and trade-off analysis. In *Proceedings of the 34st Annual Design Automation Conference (DAC)*, pages 389–394, New York, NY, USA, 1997. ACM.

[48] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[49] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. Symdrive: Testing drivers without devices. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 279–292, Berkeley, CA, USA, 2012. USENIX Association.

[50] James A. Rowson. Hardware/software co-simulation. In *Proceedings of the 31st Annual Design Automation Conference (DAC)*, pages 439–440, New York, NY, USA, 1994. ACM.

[51] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*, pages 275–288, New York, NY, USA, 2009. ACM.

[52] Luc Séméria and Abhijit Ghosh. Methodology for hardware/software coverification in C/C++. In *Proceedings of the 2000 Asia and South Pacific*

Design Automation Conference (ASP-DAC), pages 405–408, New York, NY, USA, 2000. ACM.

[53] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, New York, NY, USA, 2005. ACM.

[54] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *ACM Trans. Comput. Syst.*, 24(4):333–360, November 2006.

[55] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23:77–110, February 2005.

[56] Aaron Tomb, Guillaume Brat, and Willem Visser. Variably interprocedural program analysis for runtime error detection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–107, New York, NY, USA, 2007. ACM.

[57] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–107, New York, NY, USA, 2004. ACM.

[58] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM*

Symposium on Operating Systems Principles (SOSP), pages 203–216, New York, NY, USA, 1993. ACM.

[59] Wikipedia. Stuxnet. http://en.wikipedia.org/wiki/Stuxnet.

[60] Fei Xie, Guowu Yang, and Xiaoyu Song. Component-based hardware/software co-verification for building trustworthy embedded systems. *Journal of Systems and Software (JSS)*, 80(5):643–654, May 2007.

[61] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. Safedrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 45–60, Berkeley, CA, USA, 2006. USENIX Association.