Post-silicon Conformance Checking with Virtual Prototypes

Li Lei
Dept. of Computer Science
Portland State University
leil@cs.pdx.edu

Fei Xie
Dept. of Computer Science
Portland State University
xie@cs.pdx.edu

Kai Cong
Dept. of Computer Science
Portland State University
congkai@cs.pdx.edu

ABSTRACT

Virtual prototypes are increasingly used in device/driver co-development and co-validation to enable early driver development and reduce product time-to-market. However, drivers developed over virtual prototypes often do not work readily on silicon devices, since silicon devices often do not conform to virtual prototypes. Therefore, it is important to detect the inconsistences between silicon devices and virtual prototypes.

We present an approach to post-silicon conformance checking of a hardware device with its virtual prototype, i.e., a virtual device. The conformance between the silicon and virtual devices is defined over their interface states. This approach symbolically executes the virtual device with the same driver request sequence to the silicon device, and checks if the interface states of the silicon and virtual devices are consistent. Inconsistencies detected indicate potential errors in either the silicon device or the virtual device. We have evaluated our approach on three network adapters and their virtual devices, and found 15 inconsistencies exposing 15 real bugs in total from the silicon and virtual devices. The results demonstrate that our approach is useful and efficient in facilitating device/driver co-validation at the post-silicon stage.

Categories and Subject Descriptors

B.6.2 [**Logic Design**]: Reliability and Testing—*Error-checking*

General Terms

Design, Reliability, Verification

Keywords

Post-silicon validation, conformance checking, virtual prototypes

1. INTRODUCTION

Virtual prototyping has emerged as a promising technique for device/driver co-development. A notable example is how Intel utilizes virtual devices to enable early driver development for their new generation, 40 Gigabit Ethernet (40GbE) adapter, before the FPGA prototype is available [13].

DAC 2013 May 29 - June 07 2013, Austin, TX, USA

The use of virtual prototypes, i.e., virtual devices, has potential to shorten device/driver development cycles. Nevertheless to achieve this benefit, a key challenge has to be addressed. As silicon devices often do not conform to virtual devices, drivers developed over virtual devices often do not work readily on silicon devices due to either silicon device bugs or driver bugs hidden on virtual devices. Unfortunately, troubleshooting these bugs today heavily depends on ad-hoc and time-consuming system testing/debugging at the post-silicon stage. By detecting the inconsistencies between the virtual and silicon devices, conformance checking provides a systematic and efficient way to (1) expose the virtual or silicon device errors; (2) reveal the causes of driver bugs hidden on the virtual device.

We present a novel approach to post-silicon conformance checking of a hardware device with its virtual device and discovery of their inconsistencies. The conformance of a silicon device and its virtual device is defined between their interface states (cf. Section 3.1). This approach symbolically executes a virtual device with the same driver request sequence to its silicon device, and checks if the interface states of the silicon and virtual devices are consistent. There are three major steps: (1) recording the driver requests to the silicon device; (2) symbolically executing the virtual device by taking the recorded request sequence; (3) checking if the silicon and virtual device interface states are consistent after executing the virtual device on each driver request.

Post-silicon validation often suffers from the limited observability of hardware, a critical problem that our approach needs to address. A silicon device has internal states that are difficult to observe. Moreover, the outside environment inputs to the silicon device are hard to capture. We use symbolic execution to address this problem. Instead of observing the internal states and environment inputs, our approach models them using variables with symbolic values when simulating the silicon device behaviors on the virtual device. This way symbolic execution covers their possible values.

We implement our approach using the virtual devices from the QEMU virtual machine [2] as virtual prototypes, and discover 15 inconsistencies behind which there are 15 real bugs in the silicon and virtual devices. These bugs can cause severe problems, e.g., system crashes. Detecting these bugs can significantly facilitate hardware/software co-validation at the post-silicon stage.

The rest of this paper is organized as follows. Section 2 introduces related background. Section 3 presents our approach. Section 4 shows how we implement the approach. Section 5 reports our experiment results. Section 6 presents the related work. Section 7 concludes and discusses the future work.

2. BACKGROUND

In this section, we introduce three related concepts: QEMU vir-

tual devices which we adopt as our virtual prototypes, non-deterministic interleaving which we utilize in generating execution harnesses for virtual devices, and symbolic execution with which we replay driver requests on virtual devices.

2.1 QEMU and Virtual Devices

QEMU [2] 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. Such virtual devices are widely used for device driver developments.

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 shown in Figure 1, the e1000 virtual device has the following major components:

- PCI device state, as defined by `E1000State`, which keeps track of the state of the PCI device;
- PCI device module functions, which simulate the basic functionalities of the PCI device. As Figure 1 shows, function `e1000_mmio_writel` simulates how the e1000 device responds to the driver write request, and `e1000_receive` simulates how the e1000 device receives network packets and notifies QEMU via interrupts.

```
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];
    ... ...
}E1000_state;

static void
e1000_mmio_writel(void *opaque,
          target_phys_addr_t addr, uint32_t val)
{
    ... ...
    if (index < NWRITEOPS && macreg_writeops[index])
    {
      macreg_writeops[index](s, index, val);
    }
     ... ...
}

static ssize_t
e1000_receive(VLANClientState *nc,
const uint8_t *buf, size_t size)
{
  ... ...
  //Fire an interrupt after receiving packets
  set_ics(s, 0, n);
}
```

Figure 1: Excerpts from the e1000 QEMU virtual device.

2.2 Non-deterministic Interleaving

Non-deterministic interleaving [11] is a transaction-level modeling technique for hardware concurrency. Hardware devices are concurrent in nature. For example, a network adapter processes driver requests and receives data concurrently. To model this concurrency using non-deterministic interleaving, there are three steps: (1) identify the concurrent modules (e.g., processing driver requests, receiving data, etc.) of the target hardware device; (2) specify the modules using separate C functions, which we refer to as module functions; and (3) non-deterministically invoke these module functions in a loop. When the loop is executed multiple times, these module functions are executed in a non-deterministic sequence. The possible effects of hardware concurrency can be captured by the set of hardware states after non-deterministic many executions of the loop. For example, a network adapter concurrently processing a driver request and receiving a packet can be captured by either processing a driver request followed by receiving a packet, or invoking these two modules in the reverse order. Since a QEMU virtual device already implements the first and second steps, we only need to complete the third step before executing it symbolically (cf. Section 4.2).

2.3 Symbolic Execution

Symbolic execution [10] 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.

3. CONFORMANCE CHECKING

This section presents the basic workflow of our conformance checking framework. As illustrated in Figure 2, the framework has two major components: a trace recorder and a conformance checker. The trace recorder records the driver request sequence to

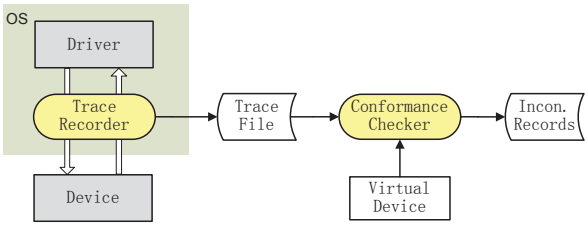

Figure 2: Workflow of conformance checking

the silicon device. The conformance checker replays the sequence on the virtual device 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 device execution trace under the driver request.

3.1 Preliminaries

Before discussing the details of this workflow, we first introduce our notion of conformance, which is defined between the states of

the silicon and virtual devices. The state of the silicon device is determined by the values of its interface and internal registers. The interface registers of the silicon device are observable while the internal registers are generally not observable and are sometimes even unknown. The virtual device is a model of the silicon device. It models interface registers of the silicon 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 silicon device. We define a virtual device state as follows.

DEFINITION 1. *A* **virtual device 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 silicon 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 2. *A* **silicon 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, we give a conformance definition between a silicon device state and a virtual device state, as described in Definition 3.

DEFINITION 3. *A silicon device state* S *and a virtual device state* V *conform to each other if* $set(S) \cap set(V) \neq \emptyset$.

To compute $set(S) \cap set(V)$, we denotes 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 Trace Recorder

The trace recorder captures: (1) each driver request issued to the silicon device; (2) the silicon device interface state before each driver request is issued. A sequence of such state-request pairs captured on the silicon 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 silicon device interface state S_{I_k}.

3.3 Conformance Checking Algorithm

The conformance checker replays T on the virtual device using symbolic execution. To compare the virtual and silicon device states, given S_{I_k} of T, the conformance checker constructs its silicon device state S_k (cf. Section 3.1). In this way, the conformance checker converts T to 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 silicon device state derived from S_{I_k}. 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'.

As discussed above, symbolic execution of a virtual device may produce a set of virtual device states $G = \{V_i | 0 \leq i \leq n\}$. The next silicon device state under D_k is denoted as S_{k+1}. We define the conformance between G and S_{k+1} as follows.

DEFINITION 4. *Given* G *and* S_{k+1}*, the virtual device and the silicon device conform to each other at* D_k *if* $\exists V_i \in G$ *where* $0 \leq i \leq n$, $set(S_{k+1}) \cap set(V_i) \neq \emptyset$.

3.4 Discussions

Our conformance definition is essentially the conformance between the interface states of the silicon and virtual devices 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 device state to the silicon device state after each drive request (Step 4). This may miss inconsistencies that only surface after several driver requests. (How to check for such inconsistencies will be discussed in a future paper.) Under this conformance definition, our approach is sound theoretically as symbolic execution explores all possible interface states of the virtual device. Nevertheless in practice, for practicality and efficiency, our approach may introduce false negatives due to optimizations of symbolic execution (cf. Section 4).

4. IMPLEMENTATION

4.1 Selective Capturing

The trace recorder captures values of the interface registers of the silicon 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 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 device: symbolically execute the virtual device 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. As selective capturing has a critical impact on conformance checking results, we will develop a systematic method that balances overhead and effectiveness in future work.

4.2 Harness Generation for Virtual Devices

A QEMU virtual device is not a stand-alone program, which is executed as part of the QEMU virtual machine. Therefore, we need an execution harness for symbolically executing the virtual device. 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 silicon device, we assign symbolic values to these input variables so that symbolic execution can cover the possible inputs from the outside environment.

Example. We illustrate harness generation using the e1000 network adapter. Figure 3 shows an excerpt from the harness we generate for the e1000 virtual device. 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.

```
        ... ...

    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: Excerpts of execution harness of e1000 virtual device

4.3 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 device. We leverage runtime behaviors of the virtual device in the QEMU virtual machine to decide the loop bound for each unbounded loop. The method contains three steps:

1. Statically analyze the virtual device: symbolically executing the virtual device using symbolic inputs, to identify the unbounded loops.

2. When the virtual device 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 silicon 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 negatives since it potentially reduces the virtual device behaviors. However, we argue that the false negative 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 device interface state. Therefore, the conformance checking result will not be affected most of the time. Second, the loop bounds cover most virtual device behaviors if the runtime test cases for identifying loop bounds have a high coverage of the virtual device (herein we use the code coverage metrics such as statement coverage). Moreover, a discovered false negative 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 negatives cannot be completely eliminated. Therefore, the user may need to search for a "sweet spot" to achieve minimum false negatives with reasonable symbolic execution costs. Our evaluation results give more details (cf. Section 5).

4.4 Implementation Details

We implement our approach 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.

We construct our conformance checker using the symbolic execution engine KLEE [4]. We modify KLEE in two aspects. First, we set the loop bounds during symbolic execution. Second, we realize our own module for conformance checking.

5. EVALUATION

This section evaluates our approach from two aspects: usefulness and efficiency. Regarding usefulness, we present the inconsistencies and real bugs we discovered in three network adapters and their QEMU virtual devices. Furthermore, we evaluate our framework in terms of time usages and memory usages, demonstrating that our approach is efficient.

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. Information of these devices and their virtual devices are summarized in Table 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 and Intel eepro100 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 interface.

Table 1: Summary of Devices for Case Studies

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

5.2 Inconsistencies and Bugs

We discovered 15 inconsistencies between the three network adapters and their virtual devices under test: 7 in e1000, 6 in bcm5751, and 2 in eepro100. By analyzing the inconsistency records generated by the conformance checker, we also discovered 13 bugs from the virtual devices, and 2 bugs from the silicon 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 silicon devices are stable products which have gone through extensive testing and bug-fixing procedures; on the other hand, their virtual devices are not heavily tested through any rigorous testing procedures. However, these virtual device 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 testing stage before devices are released, it can also discover many inconsistencies caused by the bugs of silicon devices/prototypes.

5.2.1 Types of device bugs

We summarized the bugs which cause the inconsistencies. As shown in Table 2, there are 7 types of device bugs we discovered by analyzing the inconsistencies. VD indicates the virtual device bugs while SD indicates the silicon device bugs. 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 silicon device bugs and the virtual device bugs respectively.

- *Silicon device bugs.* The bugs of the first type are silicon 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.
- *Virtual device bugs.* The bugs of second to fourth types are all related to interrupts. The bugs of the fifth type and sixth type can cause the driver to read incorrect values. These bugs often cause serious driver and system errors or even crashes, and similar silicon device errors have been reported [9].

5.2.2 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.

Table 2: Types of Bugs in Virtual Devices and Silicon Devices

No.	Bug Type	Num.	Devices
1	Update reserved register bits which not allowed	2	SD
2	Generate unnecessary interrupts	2	VD
3	Fail to generate necessary interrupts	1	VD
4	Fail to clear the interrupt when the driver requests	1	VD
5	Fail to update registers when necessary	4	VD
6	Write incorrect values to registers	4	VD
7	Incorrect data types used for modeling device states	1	VD

```
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 4: Excerpt of e1000 virtual device

However, Figure 4 shows how the virtual device 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 device always generates an interrupt by invoking the interrupt function `set_ics`. Due to this feature, the driver developed on the virtual device may always expect an interrupt after the device finishes transferring data. However, the silicon 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 silicon device, and report an exception by mistake. The driver's normal work flow will be disrupted on the silicon 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.

5.3 Efficiency

We evaluate the efficiency of our approach, in terms of time usages, memory usages, and false negative 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. "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: Summary of Test Cases

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

5.3.1 Time and memory usages

We evaluate the time and memory usages of conformance checking. Table 4 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 4: Time/Memory Usages and False Negatives

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

5.3.2 False negative ratios

To assess the number of false negatives introduced by our optimizations, we verified all the inconsistencies discovered. In the "Inconsistency" column of Table 4, we show the numbers of discovered inconsistencies and verified inconsistencies.

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

6. RELATED WORK

Recently formal methods have been increasingly used for facilitating post-silicon validation. Some of these work focus on improving observability and traceability of hardware at the post-silicon stage. A notable work is "backspace" [6], it uses SAT-solving techniques to provide an execution trace to a crashed post-silicon state, thus facilitating off-line debugging. Several approaches [3, 8, 12] integrate formal specifications into post-silicon checking of hardware by observing its execution trace. In [14], hardware monitors are introduced to ameliorate observability requirements on silicon. It uses pre-silicon RTL models to construct hardware monitors.

Symbolic execution is widely used for software testing. SAGE [7], KLEE [4], and S2E[5] use symbolic execution to test software systems that intensively interact with environments. Other tools [16, 15, 1] also employ symbolic execution to generate test cases for testing software programs.

7. CONCLUSIONS AND FUTURE WORK

We have presented an approach to conformance checking of a hardware device with its virtual prototype. Preliminary evaluation shows that our approach is useful and efficient. In three network adapters, we discover 15 inconsistencies caused by 15 bugs in the silicon and virtual devices while incurring low memory and time usages. This demonstrates our approach's major potential in facilitating hardware/software co-validation at the post-silicon stage. In future work, we plan to use virtual prototypes to estimate silicon hardware functional coverage and to validate hardware at runtime.

8. ACKNOWLEDGMENT

This research received financial support from National Science Foundation (Grant #: 0916968). A pending patent filed on this research by Portland State University has been licensed to Virtual Device Technologies (VDTech) where Fei Xie is a partner.

9. REFERENCES

[1] M. Baluda, P. Braione, G. Denaro, and M. Pezzè. Structural coverage of feasible code. In *Proc. of AST, 2010*.

[2] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of ATEC, 2005*.

[3] M. Boule, J. Chenard, and Z. Zilic. Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug. In *Proc. of ICCD, 2006*.

[4] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of OSDI, 2010*.

[5] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proc. of ASPLOS, 2011*.

[6] F. M. De Paula, M. Gort, A. J. Hu, S. Wilton, and J. Yang. BackSpace: Formal Analysis for Post-Silicon Debug.

[7] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *ACM Queue*, 10(1), 2012.

[8] A. J. Hu, J. Casas, and J. Yang. Efficient Generation of Monitor Circuits for GSTE Assertion Graphs. In *Proc. of ICCAD, 2003*.

[9] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *Proc. of SOSP, 2009*.

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

[11] J. Li, F. Xie, T. Ball, V. Levin, and C. McGarvey. Formalizing hardware/software interface specifications. In *Proc. of ASE, 2011*.

[12] J. A. M. Nacif, F. M. de Paula, H. Foster, C. J. N. C. Jr., and A. O. Fernandes. The chip is ready. am i done? on-chip verification using assertion processors. In *VLSI-SOC*, 2003.

[13] S. Nelson and P. Waskiewicz. Virtualization: Writing (and testing) device drivers without hardware. In *Proc. of Linux Plumbers Conference*, 2011.

[14] S. Ray and W. A. Hunt, Jr. Connecting Pre-silicon and Post-silicon Verification. In A. Biere and C. Pixley, editors, *Proc. of FMCAD, 2009*.

[15] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. of ESEC/FSE, 2005*.

[16] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proc. of ISSTA, 2004*.