Coverage Evaluation of Post-silicon Validation Tests with Virtual Prototypes

Kai Cong, Li Lei, Zhenkun Yang, and Fei Xie
Department of Computer Science, Portland State University, Portland, OR 97207, USA
{congkai, leil, zhenkun, xie}@cs.pdx.edu

***Abstract*—High-quality tests for post-silicon validation should be ready before a silicon device becomes available in order to save time spent on preparing, debugging and fixing tests after the device is available. Test coverage is an important metric for evaluating the quality and readiness of post-silicon tests. We propose an online-capture offline-replay approach to coverage evaluation of post-silicon validation tests with virtual prototypes for estimating silicon device test coverage. We first capture necessary data from a concrete execution of the virtual prototype within a virtual platform under a given test, and then compute the test coverage by efficiently replaying this execution offline on the virtual prototype itself. Our approach provides early feedback on quality of post-silicon validation tests before silicon is ready. To ensure fidelity of early coverage evaluation, our approach have been further extended to support coverage evaluation and conformance checking in the post-silicon stage. We have applied our approach to evaluate a suite of common tests on virtual prototypes of five network adapters. Our approach was able to reliably estimate that this suite achieves high functional coverage on all five silicon devices.**

I. Introduction

Post-silicon validation has become a critical problem in the product development cycle, driven by increasing design complexity, higher level of integration and decreasing time-to-market. According to recent industry reports, validation accounts for a large portion of overall product cost. Post-silicon validation consumes an increasing share of the overall product development time [1]. This demands innovative approaches to speeding up post-silicon validation and reducing its cost.

To accelerate post-silicon validation, high-quality tests should be ready before a silicon device becomes available [2] in order to save time spent on preparing, debugging and fixing tests in the post-silicon stage after the device is available. Test coverage is an important metric for evaluating the quality and readiness of post-silicon validation tests. Precise coverage results are needed for engineers to judge whether existing test suites can achieve sufficient coverage on the device.

Before the first silicon prototype is ready, it is very challenging to quantify coverage of post-silicon validation tests since we do not have a silicon device to run these tests on. Even if a silicon prototype is ready, the black box nature of the silicon prototype only supports limited observability and traceability that makes post-silicon validation difficult. Recently virtual prototypes are increasingly used in hardware/software co-development to enable driver development and validation at an early stage even before silicon prototypes become available [3]. An example is how Intel used virtual prototypes to enable software development for their 40G Ethernet adapter (E40G) before the silicon prototype became available [4]. A virtual prototype for the E40G was created and used to test and validate the E40G driver being developed. Bugs were found in the driver using the E40G virtual device, even before the real E40G device became available.

978-3-9815370-2-4/DATE14/©2014 EDAA

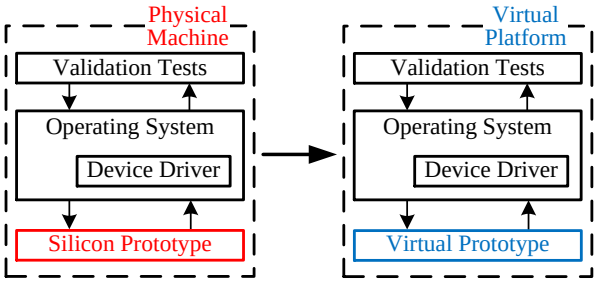

Fig. 1: From physical to virtual

As shown in Figure 1, virtual prototypes and silicon devices are running respectively in virtual platforms and physical machines. Virtual prototypes can provide the same transaction-level functionalities as silicon devices to support driver development and validation. Virtual prototypes have major potential to play a crucial role in estimating silicon device functional coverage of post-silicon validation tests. The white box nature of virtual prototypes brings complete observability and traceability that evades silicon devices. It is possible to have thorough test coverage evaluation over virtual prototypes.

This paper presents an online-capture offline-replay approach to coverage evaluation of post-silicon validation tests with virtual prototypes. We first capture necessary run-time data, including the initial device state and device requests from a concrete execution of the virtual prototype within a virtual platform under a given test. We then compute the test coverage by efficiently replaying captured data offline on the virtual prototype itself. To evaluate the coverage, we have adopted four typical software coverage metrics and developed two hardware-specific coverage metrics: register and transaction coverage. To ensure fidelity of coverage estimation on the silicon device, we further extend our approach to compute coverage after the silicon device becomes ready and check conformance with coverage estimate on the virtual prototype.

We have implemented this approach in Device Coverage Analyzer (DCA), a coverage analysis tool using virtual prototypes. We have applied our approach to evaluate a suite of common tests with virtual prototypes of five network adapters. Our approach was able to reliably estimate that this suite achieves high functional coverage on all five silicon devices.

The remainder of this paper is structured as follows. Section 2 provides the background. Section 3 presents the design of our approach. Section 4 presents the implementation. Section 5 elaborates on the five case studies we have conducted and discusses the experimental results. Section 6 reviews related work. Section 7 concludes and discusses future work.

II. Background

A. Virtual Prototypes and QEMU Virtual Devices

Virtual prototypes are fast, fully functional software models of hardware systems, which enable unmodified execution of software code. Virtual prototypes are running within virtual platforms such as Synopsys Virtualizer [5] and QEMU [6].

QEMU is a generic and open source machine emulator and virtualizer, which provides a large number of virtual devices. We adopt QEMU virtual devices as the virtual prototypes for our study due to the open-source nature of QEMU and its wide varieties of virtual devices. Technologies developed on QEMU virtual devices can be readily generalized to other open-source or commercial virtual prototyping environments due to the similarity in virtualization concepts, despite their different levels of modeling details.

```
// 1. Device state
typedef struct E1000State_st {
    PCIDevice dev; //PCI configuration
    uint32_t mac_reg[0x8000]; //Interface registers
    ......
    uint32_t rxbuf_size; //Internal variables
    ......
} E1000State;
// 2. Interface register function: write register
static void write_reg(void *opaque, uint64_t index,
     uint32_t value) {
    E1000State *s = (E1000State *)opaque;
    ......
    if(index == TRANSMIT) {
       s->mac_reg[index] = value;
       start_xmit(s); //Invoking transaction function
    }
    ......
}
// 3. Device transaction function: transmit packets
static void start_xmit(E1000State *s) {
    ......
    set_irq(s->dev.irq[0],1);//Invoking interrupt function
}
// 4. Environment function: receive packets
static ssize_t receive(NetClientState *nc, const uint8_t
      *buf, size_t size) {
    ......
    pci_dma_read(&desp);//Invoking DMA function
}
```

Fig. 2: Excerpt of QEMU E1000 virtual device

To better understand the concept of virtual prototype, we illustrate it with a QEMU virtual device for the Intel E1000 Gigabit network adapter. As shown in Figure 2, the E1000 virtual device has the following components: *1)* The device state, $E1000State$, which keeps track of the state of the E1000 device and the device configuration; *2)* The interface register functions such as $write\_reg$ which are invoked by QEMU to access interface registers and trigger transaction functions; *3)* The device transaction functions such as $start\_xmit$ which are invoked by the interface register functions to realize the functionality; *4)* The environment functions such as $receive$ which are invoked by QEMU to pass environment inputs such as a packet received to the virtual device. Both transaction functions and environment functions may access DMA data by invoking DMA functions pci_dma_read or pci_dma_write, and may fire interrupts by calling interrupt function set_irq.

B. Preliminary Definitions

In order to help better understand our approach, we first introduce several definitions.

Definition 1: A ***device state*** is denoted as $\boldsymbol{s} = \langle \boldsymbol{s}_I, \boldsymbol{s}_N \rangle$ where $\boldsymbol{s}_I$ is the interface state including all interface registers and $\boldsymbol{s}_N$ is the internal state including all internal variables. $\boldsymbol{s}_I$ can be accessed by system software (for example, device driver) while $\boldsymbol{s}_N$ can only be accessed by the device itself.

Definition 2: An ***interface register request*** is denoted as $\boldsymbol{r}_{ir}$ which is issued by drivers to access interface registers.

Definition 3: An ***environment input*** is denoted as $\boldsymbol{r}_{ei}$ which is received by the device from the environment.

Definition 4: A ***device request*** is denoted as $\boldsymbol{r}$ which is received by the device from either the system software or the environment. The request $\boldsymbol{r}$ is either $\boldsymbol{r}_{ir}$ or $\boldsymbol{r}_{ei}$.

Direct memory access (DMA) is a feature of modern computers that allows certain devices to access system memory independently of CPU. In order to process a device request $\boldsymbol{r}$, a device might read/write data using DMA.

Definition 5: A ***DMA sequence*** is denoted as $\boldsymbol{d} = \boldsymbol{d}_1, \boldsymbol{d}_2, ..., \boldsymbol{d}_n$ where $\boldsymbol{d}_i$ is the ith DMA data accessed for processing one request.

Definition 6: A ***device event*** is denoted as $\boldsymbol{e} = \langle \boldsymbol{r}, \boldsymbol{d} \rangle$ where $\boldsymbol{r}$ is a device request and $\boldsymbol{d}$ is a sequence of DMA data. For some event $\boldsymbol{e}$, $\boldsymbol{d}$ might be null since no DMA data is needed for processing $\boldsymbol{r}$.

Definition 7: A ***device event sequence*** is denoted as $\boldsymbol{seq} = \boldsymbol{e}_1, \boldsymbol{e}_2, ..., \boldsymbol{e}_n$ where $\boldsymbol{seq}$ is a sequence of events. A subsequence $\boldsymbol{seq}_k$ of $\boldsymbol{seq}$ contains the first k events of $\boldsymbol{seq}$ where $\boldsymbol{seq}_k = \boldsymbol{e}_1, \boldsymbol{e}_2, ..., \boldsymbol{e}_k$.

C. Post-silicon Conformance Checking

In previous work [7], we have developed an approach to post-silicon conformance checking of a silicon device with its virtual device. The conformance between the silicon and virtual devices is defined over their interface states. The request sequence issued to the device is first captured on the silicon device, and then replayed on the virtual device to check if the interface states of the silicon and virtual devices are consistent.

III. Online-capture Offline-replay Coverage Evaluation with Virtual Prototypes

A. Motivation

Post-silicon validation has become a bottleneck in system development cycle and is a significant, growing part of overall validation cost [8]. To speed-up post-silicon validation, some

tasks should be conducted early in the pre-silicon stage, e.g., development and evaluation of post-silicon validation tests.

Before a silicon device is ready, post-silicon validation tests can be evaluated using RTL emulation. However, emulating hardware design has certain limitations. First, RTL emulators can be very expensive. Second, RTL emulation is often slow. Third, it requires a complete working RTL design [4] to evaluate post-silicon validation tests. Recently virtual devices and virtual platforms have been used for driver development and validation before a silicon device is ready. Virtual devices are software components. Compared to their hardware counterparts, it is easier to achieve observability and traceability on virtual devices. This makes virtual devices amenable to coverage evaluation of post-silicon validation tests.

B. Online-capture

In order to compute test coverage on virtual devices, we need to collect necessary run-time data from the virtual platform. A naïve idea is to capture all necessary run-time data including execution information of virtual devices directly from the virtual platform. However, such approach has three disadvantages. First, we need to instrument virtual devices to capture execution information of virtual devices. Second, capturing detailed execution information introduces heavy overhead into the virtual platform. Third, we need to decide what kinds of information should be captured before run-time execution of the virtual platform. It is hard to guarantee that captured information is sufficient. Once a new metric is added, it is possible that we have to modify the capture mechanism and then rerun the virtual platform to capture more data.

Therefore, we developed an online-capture offline-replay approach to capture minimum necessary data at run-time, and then replay the run-time data on the virtual device itself offline to collect necessary execution information.

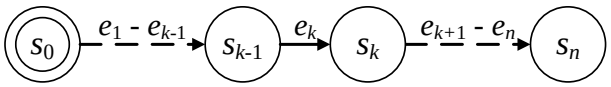

Fig. 3: Device state transition graph

A device can be treated as a state transition system. As shown in Figure 3, given a device state $\boldsymbol{s}_{k-1}$ and a device event $\boldsymbol{e}_k$, the device will transit to a new device state $\boldsymbol{s}_k$. Therefore, with the initial state $\boldsymbol{s}_0$ and the whole event sequence seq, we can infer all states and reproduce all state transitions. In other words, capturing $\boldsymbol{s}_0$ and seq from the concrete execution of a virtual device within the virtual platform should introduce the lowest overhead and deliver the most effective data.

C. Offline-replay

Our offline-replay mechanism reproduces run-time execution on virtual devices with $\boldsymbol{s}_0$ and seq, which provides flexible analysis mechanism and powerful debug capability.

1) Flexible analysis mechanism: The replay process is independent of the virtual platform/physical machine. Once run-time data is captured, users can replay the event sequence and reproduce the execution at any time. Based on different user requirements, users can generate different coverage reports from the replay process with different metrics.

2) Powerful debug capability: The replay mechanism provides capability for debugging interesting execution traces on virtual devices statement by statement, backward and forward.

Algorithm 1 REPLAY_EVENTS (s_0, seq)

```
1:  i ← 0; //loop iteration
2:  s ← s_0; //Set initial device state
3:  while i < seq.size() do
4:      e ← seq[i];
5:      ⟨t, s_next⟩ ←Execute_Virtual_Device (s, e);
6:      T.save(t);
7:      s ← s_next; //Set next device state
8:      i ← i + 1;
9:  end while
10: Generate_Report (T);
```

Algorithm 1 illustrates how to replay all events with $\boldsymbol{s}_0$ and seq to collect necessary execution information. In Algorithm 1, T is a temporary vector for saving execution information for all events. The algorithm takes the initial device state s_0 and the event sequence seq as inputs. Before replaying the event sequence, we set s_0 as the device state s. We run the virtual device with each event e in the event sequence seq and the corresponding state s to compute the execution information t and the next state s_{next}. Then t is saved in T and s_{next} is assigned to s. After replaying all events, we generate coverage reports based on T and user configuration.

D. Coverage Computation and Conformance Checking in the Post-silicon Stage

In our approach, we use coverage evaluation of virtual prototypes to estimate functional coverage on silicon devices. In order to make our approach practical and reliable, we need to address the following two key challenges:

1) Accuracy: In our approach, we capture run-time data from the concrete execution of virtual devices within a virtual platform. Events ($\boldsymbol{E}_v$) issued to virtual devices within a virtual platform can be different from events ($\boldsymbol{E}_s$) issued to silicon devices within a physical machine for the same tests. The concern is whether the coverage ($\boldsymbol{C}_v$) computed on ($\boldsymbol{E}_v$) is a good approximation of the coverage ($\boldsymbol{C}_s$) computed on ($\boldsymbol{E}_s$).

2) Conformance: Another challenge is whether coverage estimation on virtual devices can really reflect functional silicon device coverage. Although both virtual devices and silicon devices are developed according to the same specification, whether they conform to each other is still a major concern.

To address the above two challenges, we have extended our approach to support coverage computation and conformance checking after the silicon device is ready. We first reset the silicon device, and then capture run-time data, including all silicon device states $SS = \{ss_0, ss_1, ..., ss_n\}$ and the device event sequence $seq = e_1, e_2, ..., e_n$, from the concrete execution of a silicon device within a physical machine. For a silicon device, interface registers are observable while the internal registers are not observable in general. Therefore it is only possible to record all silicon device interface states $SS_I = \{ss_{I0}, ss_{I1}, ..., ss_{In}\}$ due to the limited observability. Algorithm 2 shows the extended algorithm for replaying SS_I and seq on the virtual device.

Algorithm 2 EXTENDED_REPLAY_EVENTS (SS_I, seq)

```
1: k ← 0; //loop iteration
2: s ← Reset_Virtual_Device (); //s = ⟨s_I, s_N⟩
3: while k < seq.size() do
4:    s_I ← ss_Ik; //Load captured silicon device interface state
5:    e ← seq[k + 1];
6:    ⟨t, s'⟩ ←Execute_Virtual_Device (s, e); //s' = ⟨s'_I, s'_N⟩
7:    T.save(t);
8:    Check_Conformance (s'_I, ss_I(k+1));
9:    s_N ← s'_N;
10:   k ← k + 1;
11: end while
12: Generate_Report (T);
```

In Algorithm 2, we first reset the virtual device to get the initial device state s. We assume that the internal states between the silicon device and its virtual device are the same after resetting devices. Even if both internal states are not exactly the same, a few differences should not cause a large number of functional differences according to device specifications. We take the captured device state ss_{Ik} and e_{k+1} as inputs to replay one event. The virtual device is executed with s and e_{k+1} to compute the execution information and the state s' after processing e_{k+1}. Then conformance checking is conducted between the computed interface state s'_I on the virtual device and the captured interface state $ss_{I(k+1)}$ on the silicon device to detect inconsistencies. After replaying one event, we keep the internal state and load next interface state captured to compose the device state. After replaying all events, we can get coverage reports and inconsistency report.

We utilize the coverage evaluation and conformance checking results in three aspects to assure the coverage estimation accuracy. First, we compare $\boldsymbol{C}_s$ and $\boldsymbol{C}_v$ to detect differences. If we can verify that there is no difference or few differences between $\boldsymbol{C}_v$ and $\boldsymbol{C}_s$, we can better trust that $\boldsymbol{C}_v$ can be a good approximation of $\boldsymbol{C}_s$. Second, the number of inconsistencies provides basic measurement how many differences there are between the silicon device and the virtual prototype. After analyzing the inconsistencies, we further evaluate whether these inconsistencies cause different device behaviors. If there are few inconsistencies found and there is no significant effect on the device, it can increase our confidence on coverage estimation. Third, it is easy to fix the detected inconsistencies on the virtual device so that the fixed virtual device conforms with the silicon device. Then we compute coverage again on the fixed virtual device using the same test cases. By comparing the coverage report on the fixed virtual device with that on the silicon device, we further verify that the differences in coverage caused by the inconsistencies are removed.

IV. IMPLEMENTATION

A. Coverage Metrics

Computing test coverage requires appropriate coverage metrics. In our approach, we use virtual prototype coverage to estimate silicon device functional coverage. A virtual prototype is not only a software program, but also models the characteristics of the silicon device. Therefore we have employed two kinds of coverage metrics: we have adopted the typical software coverage metrics and developed two hardware-specific coverage metrics: register coverage and transaction coverage.

1) Code Coverage: Code coverage is a typical measure used in software testing. Virtual devices are software models. We can apply all code coverage metrics to virtual devices. We select four common coverage metrics: function coverage, statement coverage, block coverage and branch coverage.

2) Register Coverage: A hardware register stores bits of information in such a way that systems can write to or read out from it all the bits simultaneously. High-level software can determine the state of the device by reading registers, and control and operate the device by writing registers. It is critical for engineers to know what registers have been accessed so they can check whether the device is accessed correctly according to the specification. Virtual devices provide complete observability, therefore we can capture accesses on both interface and internal registers. Actually in our approach, we capture all register accesses and deliver different kinds of register coverage reports according to user configuration.

3) Transaction Coverage: Devices and, therefore, virtual devices are transactional in nature: they receive interface register requests and environment inputs, and process them concurrently without interference. Thus, an interesting and useful metric is transaction coverage. For a virtual device (which is a C program), given a state s and a device request r, a program path of the virtual device is executed and the device is transitioned into a new state. Each distinct program path of the virtual device represents a distinct device transaction. When computing coverage, the impact of a test case on the virtual device in term of what transactions it hits and how often they are hit are recorded. The impact of a test suite can be recorded the same way. The coverage statistics can be visualized using pie or bar charts in term of what and how many requests were made, what and how many transactions were hit, and what percentages they account for among all requests. Moreover, the details of a transaction is recorded, such as registers accessed and interrupt status.

B. Coverage on Different Levels

To generate coverage reports, we first analyze virtual devices statically to get program information, such as the position of branches and the number of functions, and then generate all kinds of coverage reports based on the execution traces computed by the replay engine. Our approach provides flexibility to generate reports on two different levels:

1) Event Level: Given an event, a user can check what transaction is explored, what registers are accessed and whether any interrupt is fired. Moreover, the user can debug the execution trace step by step using the replay engine.

2) Test Case/Suite Level: A test case/suite issues a sequence of requests to a device. Simultaneously, the device may receive environment inputs and read DMA data. Given a test case/suite, all device events are captured. The replay engine replays all captured events and generates the code coverage, the register coverage and the transaction coverage for the test case/suite.

C. Implementation Details

We implement our approach on the QEMU virtual platform. The event capture mechanism is implemented as a

QEMU module which can be used for hooking QEMU virtual devices. Device interface functions are invoked by the QEMU framework. For instance, a driver issues a read register request, the QEMU invokes the corresponding read register function defined in the virtual device. Our module hooks all the interface functions when the virtual device registers these functions to QEMU. In this way, the module captures the device events when there is an interface register request, an environment input or a DMA access. This module provides capability to hook different virtual devices without modifying virtual devices. For capturing events on silicon devices in physical machines, we modified device drivers to achieve it.

We construct our replay engine using the symbolic execution engine KLEE [9]. We modify KLEE in three aspects. First, we implement some special function handler for loading events and DMA data. Second, we capture execution trace during execution of virtual devices. Third, we realize our own module for coverage generation.

V. Experimental Results

A. Experimental Setup

We have applied DCA to QEMU-based virtual devices for five popular network adapters: Intel E1000, Broadcom Tigon3, Intel EEPro100, AMD RTL8139 and Realtek PCNet. While our tool currently focuses on QEMU-based virtual devices, the principles also apply to other virtual prototypes. The experiments were performed on a desktop with an 8-core Intel(R) Xeon(R) X3470 CPU, 8 GB of RAM, 250GB and 7200RPM IDE disk drive and running the Ubuntu Linux OS with 64-bit kernel version 3.0.61.

B. Online-capture and Offline-replay Overhead

In order to evaluate our approach, we capture a request sequence triggered by a test suite. The test suite includes most common network testing programs, such as ifconfig and ethtool [10]. DCA needs to capture the initial device state and device events at run-time, which brings overhead to run-time QEMU environment. With the capture mechanism, both QEMU and virtual devices work normally.

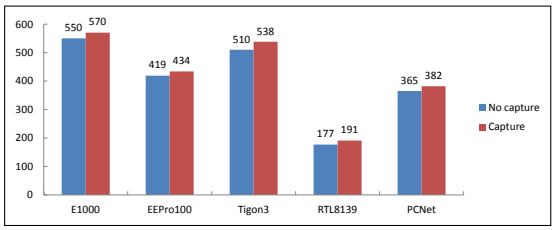

Fig. 4: Time Usage (Seconds) for Online Capture

To evaluate the overhead of online capture mechanism, we illustrate the time usage for the whole test suite under the capture configuration and no-capture configuration in Figure 4. Between the capture and no-capture configurations, there is low running time overhead introduced. For example, the overhead for E1000 is about (570 - 550) / 550 = 3.6%.

We further evaluated time and memory usages for the offline replay process. As shown in Table I, time and memory usages of the offline replay are modest. It only takes a few minutes to process tens of thousands events.

TABLE I: Time and Memory Usages for Offline Replay

	Events(#)	Time(Minutes)	Memory(Mb)
E1000	65530	10.5	268.24
Tigon3	89032	12.0	336.35
EEPro100	30112	6.0	213.18
RTL8139	43228	7.0	225.26
PCNet	54016	8.5	254.60

C. Coverage Results

We demonstrate our coverage results in three aspects: code coverage (statement/block/branch/function coverage), register coverage and transaction coverage. Due to space limitation, we only illustrate coverage results for E1000 below although we have finished coverage evaluation on all five devices.

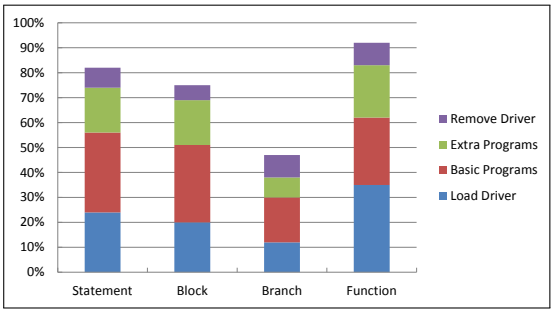

Fig. 5: Code Coverage Results for E1000

Figure 5 uses a stack to show incremental coverage of different test programs on E1000 under different code coverage metrics. We evaluate the coverage for both a test case, such as sending a ping packet, and a test suite including most common testing programs. These coverage results can give engineers basic measurement of the quality of test cases.

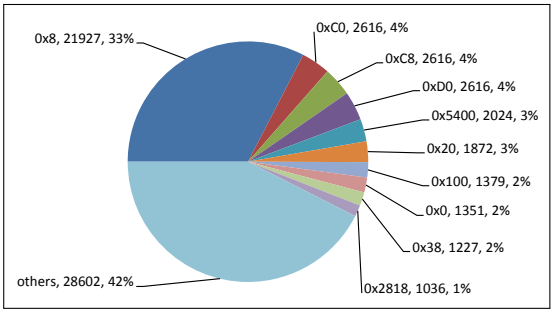

Fig. 6: Top Ten Accessed Registers for E1000

Figure 6 shows partial register coverage results for E1000. Each register is identified using the register offset, such as 0x0 and 0x8. The figure shows that how many times and how much percentage top ten registers are accessed. For instance, the most accessed register is register 0x8 (status register), which is accessed 21927 times. The system software reads this register very frequently to query the device state.

Figure 7 shows partial transaction coverage results for E1000. Each transaction is identified using a hash value, such as 0xd4e4d3ed. It shows that how many times and how much percentage top ten transactions are accessed. By analyzing transaction coverage, engineers can know what functionalities have been tested. By analyzing execution information of each transaction, engineers can further observe register accesses.

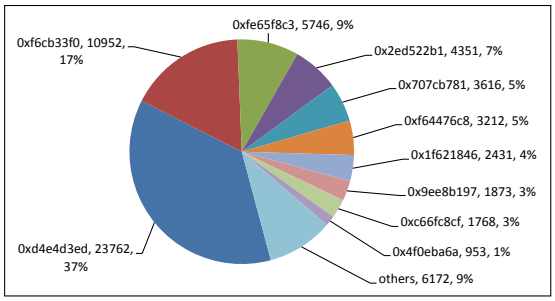

Fig. 7: Top Ten Transactions for E1000

D. Coverage and Conformance Results in Post-silicon Stage

With the same test suite, we instrumented drivers to capture run-time data on two silicon devices: E1000 and Tigon3, and computed the coverage on the corresponding virtual devices. We compare the results with these results shown in Section V-C. The coverage results are very similar for both E1000 and Tigon3 in terms of code and register coverage. One major difference is reflected on transaction coverage. Due to different speeds of physical machine and virtual platform, several transactions are affected. For example, while transmitting network packets, silicon devices can transmit more packets than virtual devices in the transmit transaction since the speed of silicon devices is much higher than virtual devices. We conclude such differences in coverage are acceptable.

We applied conformance checking to detect inconsistencies between E1000 and Tigon3 and their corresponding virtual devices. There are 13 inconsistencies discovered between the two network adapters and their virtual devices under the given tests: 7 in Intel E1000 and 6 in Broadcom BCM5751. We modified 21 lines of code in virtual devices to fix all 13 inconsistencies. Then we rerun coverage tools on fixed virtual devices to generate new coverage reports. After comparing the new reports with the post-silicon coverage reports, we found no differences except the known transaction differences.

E. Remarks

Coverage evaluation in the post-silicon stage often requires instrumenting the device driver and comes too late. Coverage evaluation on virtual prototypes can be available much earlier; therefore, it can guide improvement of post-silicon tests. From conformance checking results and coverage report comparison, it is clear the more conforming the virtual and silicon devices are, the more accurate the coverage evaluation on the virtual device. Even if there exist inconsistencies, conforming checking facilitates quick correction of coverage estimate in the post-silicon stage by conveniently detecting these inconsistencies.

VI. Related Work

One common approach to post-silicon coverage evaluation is to use in-silicon coverage monitors [11]–[13]. However, adding coverage monitors to the silicon is costly in terms of timing, power, and area [14]. In order not to introduce too much overhead, developers can only add a small number of coverage monitors in the design. Consequently, the effectiveness of coverage evaluation highly relies on what kinds of device signals are captured by in-line coverage monitors. Moreover, such approach of using coverage monitors can take effect only after silicon devices are ready. Another approach to coverage evaluation of test cases before silicon devices are available is RTL emulation. However, emulating hardware design has some limitations as we discussed in III-A. Our approach takes the obvious advantages of virtual devices: complete observability and traceability, and is applicable without silicon devices. We utilize test coverage over virtual devices to estimate silicon device functional coverage.

VII. Conclusions and Future Work

We have presented an approach to early coverage evaluation of post-silicon validation tests with virtual prototypes, which fully leverages the observability and traceability of virtual prototypes. We have applied our approach to evaluate a suite of common tests on virtual prototypes of five network adapters. We have also established high confidence in fidelity of coverage evaluation by further conducting coverage evaluation and conformance checking on silicon devices.

Our future research will explore the following directions. (1) We will research on how to define new coverage metrics which can be used for better evaluating hardware coverage. (2) We will investigate how to utilize the coverage results to measure the validation completeness and guide test generation.

VIII. 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.

References

[1] E. Singerman, Y. Abarbanel, and S. Baartmans, "Transaction based pre-to-post silicon validation," in *DAC*, 2011.

[2] S. Mitra, S. Seshia, and N. Nicolici, "Post-silicon validation opportunities, challenges and recent advances," in *DAC*, 2010.

[3] P. Sampath and B. Rachana Rao, "Efficient embedded software development using QEMU," in *13th Real Time Linux Workshop*, 2011.

[4] S. Nelson and P. Waskiewicz, "Virtualization: Writing (and testing) device drivers without hardware," in *Linux Plumbers Conference*, 2011.

[5] "Synopsys virtualizer," 2013. [Online]. Available: http://www.synopsys.com/Systems/VirtualPrototyping/Pages/Virtualizer.aspx

[6] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX ATEC*, 2005.

[7] L. Lei, F. Xie, and K. Cong, "Post-silicon conformance checking with virtual prototypes," in *DAC*, 2013.

[8] J. Keshava, N. Hakim, and C. Prudvi, "Post-silicon validation challenges: How EDA and academia can help," in *DAC*, 2010.

[9] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008.

[10] K. Cong, F. Xie, and L. Lei, "Automatic concolic test generation with virtual prototypes for post-silicon validation," in *ICCAD*, 2013.

[11] K. Balston, M. Karimibiuki, A. Hu, A. Ivanov, and S. J. E. Wilton, "Post-silicon code coverage for multiprocessor system-on-chip designs," *IEEE Transactions on Computers*, 2011.

[12] T. Bojan, M. Arreola, E. Shlomo, and T. Shachar, "Functional coverage measurements and results in post-silicon validation of Core^{TM}2 duo family," in *HLVDT*, 2007.

[13] X. Liu and Q. Xu, "Trace signal selection for visibility enhancement in post-silicon validation," in *DATE*, 2009.

[14] A. Adir, A. Nahir, A. Ziv, C. Meissner, and J. Schumann, "Reaching coverage closure in post-silicon validation," in *HVC*, 2010.