Automatic Concolic Test Generation with Virtual Prototypes for Post-silicon Validation

Kai Cong
Department of Computer Science
Portland State University
Portland, OR 97207, USA
congkai@cs.pdx.edu

Fei Xie
Department of Computer Science
Portland State University
Portland, OR 97207, USA
xie@cs.pdx.edu

Li Lei
Department of Computer Science
Portland State University
Portland, OR 97207, USA
leil@cs.pdx.edu

***Abstract*—Post-silicon validation is a crucial stage in the system development cycle. To accelerate post-silicon validation, high-quality tests should be ready before the first silicon prototype becomes available. In this paper, we present a concolic testing approach to generation of post-silicon tests with virtual prototypes. We identify device states under test from concrete executions of a virtual prototype based on the concept of device transaction, symbolically execute the virtual prototype from these device states to generate tests, and issue the generated tests concretely to the silicon device. We have applied this approach to virtual prototypes of three network adapters to generate their tests. The generated test cases have been issued to both virtual prototypes and silicon devices. We observed significant coverage improvement with generated test cases. Furthermore, we detected 20 inconsistencies between virtual prototypes and silicon devices, each of which reveals a virtual prototype or silicon device defect.**

I. Introduction

Post-silicon validation is a crucial stage in the system development cycle. According to recent industry reports [1], validation accounts for nearly 60% of overall development cost; post-silicon validation is a significant, fastest-growing component of the validation cost. This demands innovative approaches to speed-up post-silicon validation and reduce cost.

To accelerate post-silicon validation, tests should be ready before a silicon device is available [2]. The time-to-market after the device is first available can be as short as several weeks. Therefore, it is highly desired to avoid spending this precious time on preparing, debugging, and fixing tests. There should be high-quality tests available before the first silicon prototype is ready.

Currently, tests for post-silicon validation mainly include random tests, manually written tests, and end-user applications [3] [4]. Random testing can quickly generate many tests and is easy to use while facing major challenges in achieving high coverage of device functionalities and avoiding high redundancy in tests. Manually written tests are efficient in testing specific device functionalities. However, developing manual tests is labor-intensive and time-consuming. Furthermore, humans make mistakes when they write tests manually and it is difficult to check correctness of these tests until they are applied to a silicon device. End-user applications are convenient and easy to deploy; however, it is often difficult to quantify what device functionalities are covered. In addition, end-user applications are generally not device-specific, therefore often leading to insufficient coverage of device functionalities.

Recently virtual prototypes are increasingly used in hardware/software co-development to enable early driver development and validation before hardware devices become available [5], [6]. Virtual prototypes also have major potential to play a crucial role in test generation for post-silicon validation.

This paper presents a concolic testing approach to automatic post-silicon test generation with virtual prototypes. This work is inspired by recent advances in concolic testing [7], [8]. Concolic (a portmanteau of concrete and symbolic) testing is a hybrid testing technique that integrates concrete execution with symbolic execution [9]. In our approach, we first identify device states under test from concrete executions of a virtual prototype using a transaction-based selection strategy, and then symbolically execute the virtual prototype from these states. Concrete tests are generated based on the symbolic path constraints obtained. We apply the generated test cases to both the silicon device and the virtual prototype, and check for inconsistencies between the real and virtual device states. Once an inconsistency is detected, we can replay the test case on the virtual prototype through symbolic execution to see whether it is a silicon device bug or a virtual prototype defect. The combination of virtual and silicon device execution brings three major benefits: (1) helping developers more easily and better understand a silicon device using its virtual prototype, (2) checking for defects in the silicon device, and (3) detecting bugs in the virtual prototype.

We have implemented our approach in a prototype post-silicon test generation tool, namely, ACTG (Automatic Concolic Test Generation). We have applied ACTG to virtual prototypes for three widely-used network adapters. ACTG generates hundreds of unique tests for each device. These tests lead to significant improvement in coverage. When applying the generated test cases to the silicon devices, ACTG detects 20 inconsistencies between the virtual and silicon devices.

Our research makes the following key contributions:

- *Concolic testing for post-silicon validation.* Our approach to post-silicon device test generation not only integrates concrete and symbolic execution, but also combines virtual and silicon device executions. The observability and controllability of virtual prototypes are fully leveraged while generated tests are compatible with silicon devices.

- *Transaction-based test selection.* A transaction-based test selection strategy is developed to select device states under test and eliminate redundancy in generated tests. This strategy not only generates test cases with high functionality coverage in modest time, but also produces efficient test cases with low redundancy.

The remainder of this paper is structured as follows. Section 2 provides the background. Section 3 presents our approach to post-silicon test generation. Section 4 discusses implementation details. Section 5 elaborates on the case studies we have conducted and discusses the 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. QEMU is a generic, open source machine emulator and virtualizer [10], [11]. 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. Technology 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.

To better understand the concept of virtual prototype, we illustrate it with a QEMU virtual device for the Intel E1000 Gigabit network adapter. The E1000 adapter is a PCI (Peripheral Component Interconnect) device which communicates with its control software through interface registers and interrupts. The E1000 virtual device has corresponding functions to support such communication, for instance, interface register functions and interrupt functions. In order to realize the functionalities of silicon devices, the E1000 virtual device also needs to maintain the device state and implement functions that virtualize device transactions and environment inputs. As shown in Figure 1, 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 and may fire interrupts by calling interrupt function set_irq;
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 and may also fire interrupts by calling interrupt function set_irq.

B. Symbolic Execution and KLEE Engine

Symbolic execution 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 a function 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 formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy for the symbolic execution to follow the particular path. The program counter points to the next statement to execute. 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.

KLEE [12] is a symbolic execution engine built on the LLVM [13] compiler infrastructure. Given a C program, KLEE executes the program symbolically and generates constraints that exactly describe the set of values possible on a path.

```
// 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) {
    ......
    set_irq(s->dev.irq[0],1);//Invoking interrupt function
}
```

Fig. 1: Excerpt of QEMU E1000 virtual device

C. 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. The interface state $\boldsymbol{s}_I$ can be accessed by both a high-level software (e.g., driver) and the device while $\boldsymbol{s}_N$ is only accessed by the device itself.

As shown in Figure 1, the structure $E1000State$ represents the E1000 device state and includes interface registers mac_reg and an internal variable $rxbuf\_size$.

Definition 2: A ***device request*** is denoted as ***r*** which is issued by high-level software to control and operate the device.

As shown in Figure 1, the parameters $address$ and $value$ of interface register function $write\_reg$ can be treated as a request ***r***, which is issued by the driver to modify the interface register and trigger the transaction function.

Definition 3: A ***sequence of device requests*** is denoted as $\boldsymbol{seq} = \boldsymbol{r}_1, \boldsymbol{r}_2, ..., \boldsymbol{r}_n$. A subsequence $\boldsymbol{seq}_k$ of ***seq*** contains the first k requests of ***seq*** where $\boldsymbol{seq}_k = \boldsymbol{r}_1, \boldsymbol{r}_2, ..., \boldsymbol{r}_k$.

Definition 4: A ***test case*** is denoted as $\boldsymbol{tc} = \langle \mathbf{seq, r} \rangle$, where ***seq*** is a request sequence and ***r*** is a device request.

Definition 5: A ***state under test*** is denoted as $\boldsymbol{s}_{ut}$ where $\boldsymbol{s}_{ut}$ is the device state on which test cases are generated.

Devices are transactional in nature: device requests are processed by device transactions. For a virtual device (which is a 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.

Definition 6: A ***device transaction***, denoted as ***t***, is a program path of a virtual device.

III. Concolic Test Generation with Virtual Prototypes

A. Motivation

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 post-silicon test generation.

A naive approach to test generation with a virtual device is to apply symbolic execution directly to it. A virtual device can be treated as a request-driven program. The virtual device processes a possibly unbounded sequence of requests. To execute a virtual device symbolically, we first set the device reset state as the initial state s_0 of the virtual device, which is a concrete state. Then we symbolically execute the virtual device from s_0 with a sequence of symbolic device requests. Such execution can easily lead to a path explosion [14]. Indeed as we tried this approach, it caused a path explosion only after processing a few symbolic device requests. However, most functionalities of the virtual device are only triggered by long, well-formed sequences of requests from the reset state. Therefore, the above naive approach cannot generate deep test sequences that sufficiently cover device functionalities.

B. Concolic Test Generation Algorithm

In order to address the challenge in section III-A, we develop a concolic testing scheme that integrates both concrete and symbolic execution. Concrete execution is first carried out on the virtual device and a sequence seq of concrete requests issued to the device by the driver is captured. With seq, a set of device states can be computed on the virtual device, as shown in Figure 2. The virtual device starts from the initial state s_0 which is the state after resetting the device.

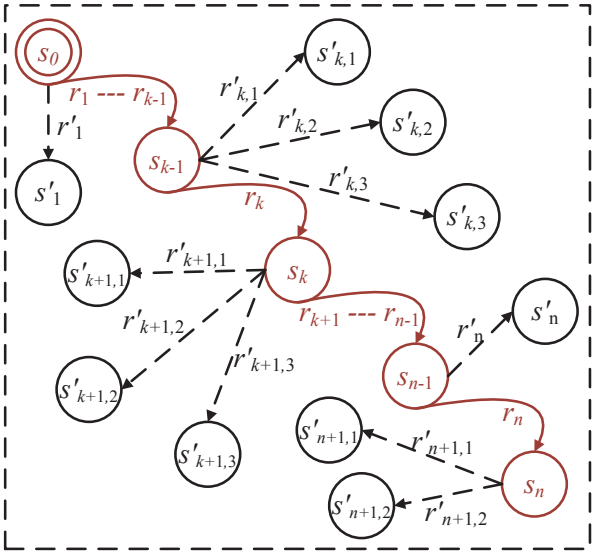

Fig. 2: Concolic test generation using virtual devices. Solid arrows denote a concrete device execution while dashed arrows denote generated test cases

With different subsequences of seq, the device is triggered to different states, for example, with the request sequence $seq_k = r_1, r_2, ..., r_k$, the device is brought to the device state s_k from s_0. With the set $\{s_0, \ldots, s_n\}$ of reproducible device states, we can apply symbolic execution to each of these states with a symbolic request. For each symbolic path explored, symbolic path constraints are recorded and a concrete request satisfying these constraints are generated. As shown in Figure 2, on the state s_{k-1}, we can generate three test cases as follows:

$$\langle seq_{k-1}, r'_{k,1} \rangle, \langle seq_{k-1}, r'_{k,2} \rangle, \langle seq_{k-1}, r'_{k,3} \rangle$$

Algorithm 1 illustrates how to generate test cases. Here, C is a temporary set for saving all constraints for each path computed by symbolic execution, and TC saves all generated test cases tc. We set the given s_{ut} as the state of virtual device s_V and create a symbolic request r_V, and then execute the virtual device. For each explored path c, we can get its symbolic path constraints. Then a concrete request r is generated for c based on its constraints. A test case tc consists of a request sequence seq_k leading the device to s_{ut} and a newly generated request r_k. For each s_{ut}, our approach generates a set of test cases TC.

Algorithm 1 GENERATE_TEST_CASE (s_{ut}, seq, k)

```
1: C ← ∅, TC ← ∅;
2: s_V ← s_ut;
3: r_V ← Create_A_Symbolic_Request ( );
4: C ← Symbolic_Execution (s_V, r_V);
5: for each path c ∈ C do
6:     r ← Generate_Concrete_Request (c);
7:     tc ← ⟨seq_k, r⟩;
8:     TC ← TC ∪ {tc};
9: end for
10: return TC;
```

There can be a large number of subsequences $\{seq_k\}$ in seq. To generate test cases from all $\{seq_k\}$ may entail prohibiting overheads. We allow the user to select $\{seq_k\}$ via assertions on device states and requests. Then a selected seq_k is replayed on virtual prototypes to get the state under test s_{ut}. After replaying a set of selected sequences, we can obtain a set of states S_{ut} where S_{ut} = $\{s_{ut1}, s_{ut2}, ..., s_{utn}\}$. To help users select states more efficiently, we provide an automatic mechanism in Section III-C.

C. Transaction-based Test Selection Strategy

In order to make our concolic testing approach practical and efficient, we need to address the following two key challenges:

1) State selection problem: For a virtual device, we can get a vast number of states under test by replaying a long sequence of device requests. Applying test generation to all these states is impractical. How to select states under test is a critical challenge. Even if we allow users to select states with filters, it can still be a laborious process.

2) Test case redundancy problem: Even if we only generate test cases on states selected, we can still get a large number of test cases. Applying all such test cases on a silicon device takes much time. However, certain test cases trigger the same behavior on a silicon device, i.e., they cover the same transaction. Therefore, to improve efficiency, we should eliminate such redundant test cases.

We develop a transaction-based test selection strategy to address the above two challenges. First, states under test are selected based on device transactions. To select states, we replay a sequence seq of device requests on the virtual device. For each state transition $s_i \overset{r_{i+1}}{\Longrightarrow} s_{i+1}$, we compute the corresponding transaction. If a new transaction t is found, we select s_i as a state under test. Based on analyzing virtual device executions, we observed that such states have good chances of triggering new transactions with different requests.

Algorithm 2 SELECT_STATES_UNDER_TEST (seq)

```
1: StateIndices ← ∅, T ← ∅;
2: i ← 0; //loop iteration
3: s_0 ← Reset_Device ();
4: while i < seq.size do
5:    r_{i+1} ← Get_Request (seq, i + 1);
6:    s_{i+1} ← Compute_Next_State (s_i, r_{i+1});
7:    t ← Compute_Transaction (s_i, r_{i+1});
8:    if t ∉ T then
9:       T ← T ∪ {t};
10:      StateIndices ← StateIndices ∪ {i + 1};
11:   end if
12:   i ← i + 1;
13: end while
14: return StateIndices;
```

Algorithm 2 illustrates how to select states under test in detail. $StateIndices$ is a temporary set for saving indices of all selected states, and T saves all unique transactions invoked. We set the state after resetting the device as the initial state s_0. Then we run the virtual device with each request in the request sequence seq. For each request, if there is a new transaction t found, we save it in T and save the corresponding state index in $StateIndices$. After all requests are executed, we get a set of state indices. The corresponding states are the selected states under test.

Second, we apply transaction-based test selection strategy to eliminate redundant test cases. In the process of selecting states as discussed above, we can get a set of unique transactions T. The set T can be further used for eliminating redundant test cases. When we conduct test generation, every time a transaction t is explored by symbolic execution, we determine whether it is a new transaction that is not in T. If it is new, the corresponding test case is saved and t is added into T. If it is not a new transaction, it's also possible that a particular sequence of transactions might have not been observed and can trigger an untested functionality in the physical device. Therefore, we save the test case as a redundant test case so that the user can utilize this test case if time permits.

IV. IMPLEMENTATION

A. Overview

As illustrated in Figure 3, our automatic concolic test generation (ACTG) framework includes three key components:

1) Device Request Recorder: The recorder captures device requests from a concrete execution of the virtual device in the virtual machine. Any user or kernel level test case may be issued in the guest OS. The request recorder fully hooks the virtual device entries so that all device requests are intercepted and recorded in the request sequence seq.

2) Symbolic Engine: The symbolic engine replays a subsequence seq_k of seq to trigger the desired state under test and then symbolically executes the virtual device from this concrete state with a symbolic request. Among the transactions explored, a transaction of interest is selected, its symbolic path constraints are recorded and a concrete device request r satisfying the constraints is generated. A new test case tc is composed of the request sequence seq_k and the newly generated request r.

3) Test Manager: The test manager is a kernel-level software module residing on the test machine with the silicon device. It applies a test case to the silicon device by issuing the sequence of requests included in the test case.

B. Harness Generation for Symbolic Execution

For symbolic execution of QEMU virtual devices, we adapt KLEE to handle the non-deterministic entry function calls and symbolic inputs to device models. Since the virtual device by itself is not a stand-alone program, in order for symbolic engine to execute a virtual device, a harness must be provided for the virtual device. A key challenge here is how to create such a harness. This harness has to be faithful so that the symbolic execution of the virtual device will not generate too many paths infeasible in the real device. On the other hand, it has to be simple enough so that symbolic engine can handle the symbolic execution efficiently. To an extreme, the complete QEMU with the guest OS can serve as the harness which, however, is impractical for the symbolic engine to handle. We discussed details about harness generation in our previous work [15].

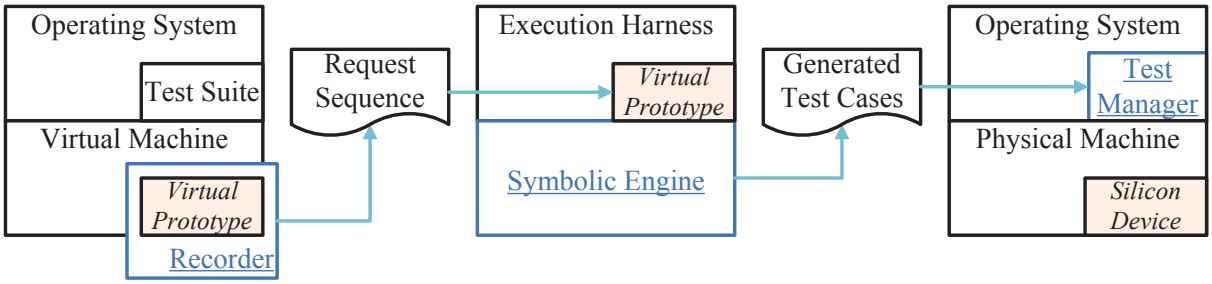

Fig. 3: Automatic Concolic Test Generation Framework

The harness includes the following parts as shown in Figure 4: (1) Declarations of state variables and parameters of entry functions; (2) Code for loading the concrete state and making parameters of entry functions symbolic; (3) Non-deterministic calls to virtual device entry functions; (4) Stub functions for virtual machine API functions invoked by virtual devices.

```
//Declarations of necessary variables
E1000State state; //Device state
target_phys_addr_t address; //Address
......

int main() {
    //Load the concrete state
    load_state(&state, sizeof(state), "state");

    //Make parameters symbolic
    make_symbolic(&address, sizeof(address), "
        address");
    ......

    //Non-deterministic calls to entry functions
    switch(svd_deviceEntry) {
    case MMIO_WRITE:
        write_reg((void *)&state, address, value);
        break;
    case MMIO_READ:
        read_reg((void *)&state, address);
        break;
        ......
    }
}

//Stub functions
uint16_t net_checksum_finish(uint32_t sum) {
    ......
}
```

Fig. 4: Excerpt of E1000 virtual device harness

C. Symbolic Execution

To improve efficiency of symbolic execution, we modify KLEE to address two key technical challenges for symbolic execution of virtual devices.

1) Path Explosion Problem: Path explosion is a major limitation for symbolic execution to thoroughly test software programs. The number of paths through a program is roughly exponential in program size. The problem also exists with executing virtual devices symbolically.

We apply two constraints when executing the virtual device to combat the path explosion problem. First, we add a loop bound to each loop whose loop condition is a symbolic expression. With the loop bound, the user controls the depth of each loop explored. Currently, we add the loop bounds manually in virtual devices. This is practical since there are only a few loops in our analysis of three virtual devices. Second, we can add a time bound to ensure that symbolic execution will terminate in a given amount of time. If the symbolic execution does not completely finish within the given time bound, there may be unfinished paths. For such paths, we still generate test cases with path constraints obtained so far. More details and experimental results are available in our previous work [15].

2) Environment Interaction Problem: A virtual device is a software component and may invoke outside API functions to interact with its environment. We divide such interactions into two categories based on whether a function call affects the values of variables in virtual devices. We detect whether the function has any pointer argument, accesses global variables, or returns a value. If so, this function potentially affects the values of variables in virtual devices. We then use two different mechanisms to handle functions in these two categories: (1) If the function call doesn't affect the values of variables in virtual devices, we instruct KLEE to ignore it; (2) If the function call may affect values of variables in virtual devices, we implement this function in our stubs. As there are not many such function calls for a category of virtual devices, such manual effort is acceptable.

D. Testing with Generated Test Cases

After generating test cases, our approach can then apply a generated test case to both real and virtual devices.

1) Application of test cases: A real (or virtual, respective) device interacts with the high-level software in a real (or virtual) machine, on which a test case tc can be applied using Algorithm 3. In order to apply tc, we first reset both real and virtual devices so that we can keep their states consistent. Our approach employs a test manager (a kernel-level module) to issue a tc in both real and virtual machines. Then we capture concrete states of both real and virtual devices after applying a tc. For a real device, it is difficult to capture the internal state. Hence, we only capture the interface state for the real device. Finally, we conduct consistency checks on the captured states between silicon devices and virtual prototypes. Our approach compares interface states of the real and virtual devices to

Algorithm 3 APPLY_TEST_CASES (TC)

```
1: for each tc ∈ TC do
2:    i ← 0; //loop iteration
3:    num ← number_of_requests_in_tc;
4:    s_{R,0} ← Reset_Real_Device ();
5:    s_{V,0} ← Reset_Virtual_Device ();
6:    while i < num do
7:       r_{i+1} ← Get_Request (tc, i + 1);
8:       s_{R,i+1} ← Compute_Next_State (s_{R,i}, r_{i+1});
9:       s_{V,i+1} ← Compute_Next_State (s_{V,i}, r_{i+1});
10:      Check_State (s_{R,i+1}, s_{V,i+1});
11:      i ← i + 1;
12:   end while
13: end for
```

detect any inconsistency. Such an inconsistency often indicates divergence between real and virtual device states, reflecting an error in either the real or virtual device.

2) Test case replay on virtual device: Upon detecting an inconsistency or a hardware error, the triggering test case can be replayed on the virtual device so that the user can better understand the exercised transaction. The symbolic engine is employed for replaying a test case $tc = \langle seq, r \rangle$. The engine first brings the device to the state under test s_{ut} and then replay the request r from s_{ut}. The engine follows the same code path that it followed while generating r, since r is generated by instantiating symbolic variables to concrete values that satisfy the constraints of that path.

The power of the symbolic engine enables full controllability and observability while replaying a test case. Our approach is sufficiently responsive to support interactive replay. It enables the user to navigate backward and forward, step by step through the execution path induced by a concrete test case. Our approach can help the user better observe what variables are changed where along the path, what inputs and initial state trigger the path, and inspect values of variables at any step.

V. EXPERIMENTAL RESULTS

A. Experiment Setup

We apply ACTG to QEMU-based virtual devices for three popular network adapters as shown in Table I. While our tool currently focuses on QEMU-based virtual devices, the principles also apply to other virtual prototypes.

TABLE I: Summary of three virtual prototypes

	Virtual Prototype		Harness	
	Lines	Functions	Lines	Entry Functions
Intel E1000	2099	53	74	4
Broadcom Tigon3	4648	34	80	4
Intel EEPro100	2178	70	85	7

To execute virtual devices symbolically, we manually created a simple harness for each virtual device. We also created a common library of stub functions for all three virtual devices. The stub library has 481 lines of C code. More details about device models and their harnesses are given in Table I. All device models are non-trivial in size ranging from 2099 lines to 4648 lines. All harnesses are relatively easy to create, having about 100 lines only. Only several hours are needed to create and fine-tune each harness and the stub library.

In order to evaluate our approach, we capture a request sequence triggered by a test suite from concrete executions of virtual devices in QEMU. The test suite includes common network testing programs. As shown in Table II, we give a partial list of programs in the test suite due to space limitation. For each virtual prototype, we have applied this test suite.

TABLE II: Summary of our test suite

Category	***Commands***	***Descriptions***
Driver Load/Unload	insmod	Load driver module
	rmmod	Remove driver module
Basic Programs	ifup	Bring a network interface up
	ifdown	Take a network interface down
	ifconfig	Configure a network interface
	ping	Send ICMP ECHO_REQUEST
	scp	Copy files between network hosts
Extra Programs	ethtool	Query or control network driver and hardware settings
	scapy	Manipulate network packets

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. Evaluation of Transaction-based Test Selection Strategy

We have applied transaction-based test selection strategy to select states and eliminate test case redundancy.

TABLE III: Comparison of different strategies

	Requests in Trace	Transaction Strategy		Random Strategy			
		States	Tests	States	Tests	States	Tests
E1000	64836	60	774	60	48	180	60
Tigon3	19157	52	175	52	46	156	54
EEPro100	41849	54	357	54	116	162	116

1) State selection: As shown in Table III, we captured a large number of requests in the request sequence, for example, 64,836 requests for the E1000 virtual device. With our transaction-based test selection strategy, only a small number of states are selected, for instance, 60 states for the E1000 virtual device. In order to evaluate the efficiency of our test selection strategy, we compare it with the random strategy. With the random strategy, we select states under test randomly. Here, we select two sets of states with the random strategy. It can be observed from Table III that with the same number of states under test selected, our strategy can generate many more useful tests, i.e., tests triggering distinctive device transactions.

TABLE IV: Time usage of transaction-based selection strategy

	States	Time (Minutes)		
		Selection	Generation	Overall
E1000	60	3.5	26.5	30
Tigon3	52	2	17	19
EEPro100	54	2	91	93

TABLE V: Summary of coverage improvement

Virtual Prototype	Statement					Block					Function					Branch				
	#	Test Suite		Generated Tests		#	Test Suite		Generated Tests		#	Test Suite		Generated Tests		#	Test Suite		Generated Tests	
		#	%	#	%		#	%	#	%		#	%	#	%		#	%	#	%
E1000	3256	2602	79.91%	2835	87.07%	298	214	71.81%	252	84.56%	42	39	92.86%	42	100%	264	165	62.5%	210	79.55%
Tigon3	1791	1496	83.53%	1689	94.3%	138	104	75.36%	128	92.75%	25	23	92%	25	100%	120	70	46.67%	97	80.83%
EEPro100	2369	1767	74.59%	2089	88.18%	266	170	63.91%	222	83.46%	44	39	88.64%	42	95.45%	150	77	51.33%	115	76.67%

To further evaluate the efficiency of our approach, we evaluate the time usages of the transaction-based selection strategy as shown in Table IV. This strategy requires spending time on both selecting states and generating test cases. The overall time for E1000 is 30 minutes which includes 3.5 minutes for state selection and 26.5 minutes for test generation.

Moreover, we applied test generation to 6000 states of the E1000 virtual device selected using the random strategy. It takes 1 day, however only two new test cases are generated. If we were to apply test generation on all 64836 states, it would have taken 10 days. This results show that (1) it is not cost-effective to apply test generation to all captured virtual device states and (2) our transaction-based strategy is efficient: order-of-magnitude reduction on time usage and effective: only missing a few tests found with much higher time usage.

2) Test case redundancy elimination: As shown in Figure 5, our transaction-based strategy is very effective in eliminating redundant tests. For each virtual device, we have achieved order-of-magnitude reduction in the number of tests that need to be applied to the virtual device.

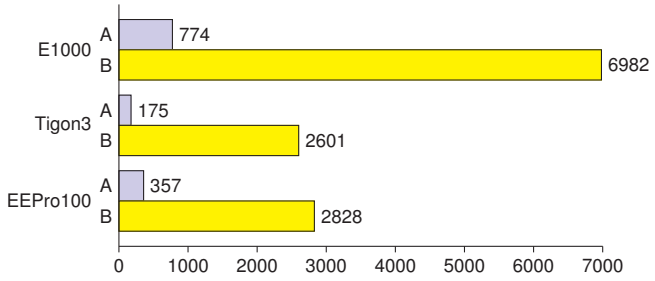

Fig. 5: Number of generated tests (A: After elimination; B: Before elimination)

C. Coverage Improvement

As shown in Table V, the generated test cases improve test coverage significantly. Hereby, we utilize test coverage over the virtual device to estimate the functional coverage over the silicon device. Because the virtual device is software, we utilize four different code coverage metrics to measure the coverage improvement. Although the test suite we use has already been able to get reasonable coverage on three virtual devices, the coverage can still be significantly improved using our generated test cases. Particularly, for E1000 and Tigon3, the function coverage can be improved to 100%. For Tigon3, the branch coverage can be improved by more than 30%.

D. Inconsistencies

As we apply the test cases on virtual and silicon devices, we collect both virtual and silicon device states. We then conduct consistency checking between the virtual and silicon device states. Our test cases have uncovered several inconsistencies between the real devices and their virtual devices. In our study, even though all the devices are popular devices which have gone through years of thorough testing and their virtual devices are created after fact, we still detected inconsistencies. The inconsistencies detected by our test suite and generated test cases are shown in Figure 6.

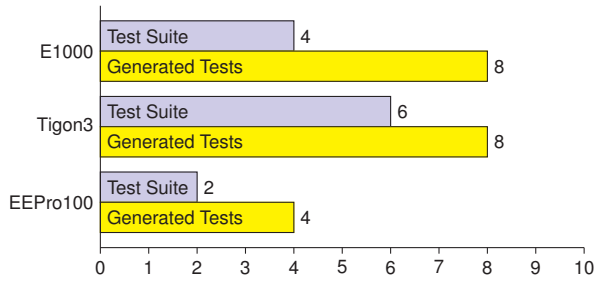

Fig. 6: Number of inconsistencies detected by test suite and generated tests

One common inconsistency is that after certain special requests, one or several device registers are modified in the real devices while in the virtual devices, they are unchanged. This inconsistency was introduced assuming the drivers would well behave and would not issue such special requests. Two types of inconsistencies detected are caused by silicon devices: (1) devices are not initialized according to the device specifications and (2) devices update registers that are specified as reserved in the device specifications. We believe that if such tests are conducted on a newly designed silicon device prototype, our approach can discover more silicon device bugs.

VI. Related Work

Post-silicon validation has become a bottleneck in system development cycle and is a significant, growing part of overall validation cost [16]. There has been much research on post-silicon validation to reduce cost and improve observability [4], [17]–[21]. However, many challenges remain in post-silicon validation, such as coverage metrics, failure reproduction, and test generation [2]. One approach to post-silicon test generation is Automatic Test Pattern Generation (ATPG) [22], [23], which targets exposing electrical and manufacturing defects rather than functional errors. There has also been efforts on reusing pre-silicon validation tests in post-silicon validation [24], [25]. Our approach shares the same goal of bridging the gap between pre-silicon and post-silicon validation, while fully leveraging the white box nature of virtual prototypes to efficiently generate high-quality functional tests.

There has been much recent work on using symbolic execution to automatically generate test inputs, leading to software testing tools such as Java PathFinder [26], CUTE and jCUTE [27], CREST [28], BitBlaze [29], DART [7], and SAGE [30]. These tools basically follow the same approach as KLEE in solving a path's constraints to generate a test case and differ in the specifics of symbolic execution and test case generation. Concolic execution [7], [30], combining concrete and symbolic execution, has also been used to optimize symbolic execution efficiency. In our approach, we applied symbolic execution to a special type of programs, virtual devices, utilized characteristics of virtual devices to improve symbolic execution effectiveness, generated test cases characterizing paths (i.e., transactions) through virtual devices, and provided facilities for applying the tests to real devices and replaying the tests on virtual devices to assist debugging.

VII. Conclusions and Future Work

We have presented an approach to generation of post-silicon tests with virtual prototypes, which fully leverages the observability and traceability of virtual prototypes. We have evaluated our approach on virtual devices for three popular network adapters. Our ACTG approach was able to generate effective test cases in a modest amount of time using the transaction-based test selection strategy. We have evaluated this strategy from two aspects: state selection efficiency and redundancy elimination. The results show that our strategy performs significantly better than random selection of states under test and has significant reduction on the number of tests applied. We have also measured test coverage and found that ACTG led to major improvement in coverage of device functionalities. Moreover, we applied generated test cases to both virtual and silicon devices and conducted consistency checking between their states. We have detect 20 inconsistencies between virtual and silicon devices, each of which reveals a defect in either virtual or silicon device.

Our future research will explore the following directions. (1) We will investigate how to use the requests captured on silicon devices to guide test generation. (2) We will research how to better utilize the results of symbolic execution. Based on the results, we will develop test coverage metrics on virtual prototypes that are specific for post-silicon validation.

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

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

[3] T. Hong, Y. Li, S.-B. Park, D. Mui, D. Lin, Z. Kaleq, N. Hakim, H. Naeimi, D. Gardner, and S. Mitra, "QED: quick error detection tests for effective post-silicon validation," in *Proc. of ITC*, 2010.

[4] I. Wagner and V. Bertacco, "Reversi: Post-silicon validation system for modern microprocessors," in *Proc. of ICCD*, 2008.

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

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

[7] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proc. of PLDI*, 2005.

[8] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *ACM Queue - Networks*, 2012.

[9] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, 1976.

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

[11] B. Fabrice, "QEMU," http://wiki.qemu.org/Main_Page, 2013.

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

[13] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proc. of CGO*, 2004.

[14] "Symbolic execution," 2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Symbolic_execution&oldid=543171109

[15] K. Cong, F. Xie, and L. Lei, "Symbolic execution of virtual devices," in *Proc. of QSIC*, 2013.

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

[17] T. Hong, Y. Li, S.-B. Park, D. Mui, D. Lin, Z. Kaleq, N. Hakim, H. Naeimi, D. Gardner, and S. Mitra, "QED: quick error detection tests for effective post-silicon validation," in *Proc. of ITC*, 2010.

[18] X. Liu and Q. Xu, "On signal selection for visibility enhancement in trace-based post-silicon validation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2012.

[19] S.-B. Park, T. Hong, and S. Mitra, "Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA)," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2009.

[20] H. Ko and N. Nicolici, "Automated trace signals identification and state restoration for improving observability in post-silicon validation," in *Proc. of DATE*, 2008.

[21] L. David, H. Ted, F. Farzan, H. Nagib, and M. Subhasish, "Quick detection of difficult bugs for effective post-silicon validation," in *Proc. of DAC*, 2012.

[22] X. Lin, K.-H. Tsai, C. Wang, M. Kassab, J. Rajski, T. Kobayashi, R. Klingenberg, Y. Sato, S. Hamada, and T. Aikyo, "Timing-aware ATPG for high quality at-speed testing of small delay defects," in *Proc. of Asian Test Symposium*, 2006.

[23] R. Drechsler, S. Eggersgluss, G. Fey, A. Glowatz, F. Hapke, J. Schloeffel, and D. Tille, "On acceleration of SAT-Based ATPG for industrial designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2008.

[24] A. Nahir, A. Ziv, M. Abramovici, A. Camilleri, R. Galivanche, B. Bentley, H. Foster, A. Hu, V. Bertacco, and S. Kapoor, "Bridging pre-silicon verification and post-silicon validation," in *Proc. of DAC*, 2010.

[25] A. Adir, S. Copty, S. Landa, A. Nahir, G. Shurek, A. Ziv, C. Meissner, and J. Schumann, "A unified methodology for pre-silicon verification and post-silicon validation," in *Proc. of DATE*, 2011.

[26] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with java pathfinder," *SIGSOFT Softw. Eng. Notes*, 2004.

[27] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for c," in *Proc. of ESEC/FSE*, 2005.

[28] M. Baluda, P. Braione, G. Denaro, and M. Pezzè, "Structural coverage of feasible code," in *Proc. of Workshop on Automation of Software Test*, 2010.

[29] D. Bethea, R. A. Cochran, and M. K. Reiter, "Server-side verification of client behavior in online games," *ACM Transactions on Information and System Security*, 2008.

[30] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. of NDSS*, 2008.