Optimizing Post-silicon Conformance Checking

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

***Abstract*—Virtual prototypes of hardware devices, a.k.a, virtual devices, are increasingly used to enable early software development before silicon prototypes/devices are available. In previous work, we presented a post-silicon conformance checking approach to detecting interface state inconsistencies between a silicon device and its virtual device. In this paper, we present an optimization, adaptive concretization, to reduce the overhead incurred by symbolic execution, a key technique used in our conformance checking approach. We have evaluated our optimized approach on three Ethernet adapters and their virtual devices. The results demonstrate that it is effective and efficient: 21 inconsistencies are discovered and time usages are reduced by an order of magnitude, comparing to the previous approach.**

Keywords—Post-silicon validation; conformance checking; symbolic execution;

I. Introduction

Virtual prototyping has emerged as a promising technique for device/driver co-development. Using virtual prototypes, i.e., virtual devices, for early driver developments has potential to shorten development cycles and reduce product time-to-market. Nevertheless to achieve these benefits, a key challenge has to be addressed. Drivers developed over virtual devices often do not work readily on silicon devices, since silicon devices often do not conform to virtual devices. Therefore, it is critical to check the conformance of a silicon device with its virtual device and discover their inconsistencies.

Our previous work [1] presents an approach to post-silicon conformance checking of a hardware silicon device with its virtual device. 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. However, the internal state of a silicon device is hard to observe and the external environment inputs to the silicon device are also difficult to capture. We use symbolic execution to tackle this problem. We model internal states and environment inputs using variables with symbolic values when simulating the silicon device behaviors on the virtual device. This way symbolic execution covers all the possible values of the internal state and environment inputs.

The approach presented in [1] has a major limitation. It incurs significant time usages. Symbolic execution introduces a significant overhead while exploring a large number of paths. This overhead makes the approach a time-consuming process. In post-silicon conformance checking, a driver request sequence is often composed of thousands of, even millions of driver requests, which requires a long time to process. Therefore, how to reduce time usages is a critical task in scaling the conformance checking approach.

In this paper, we present an efficient approach to address the above limitation. We propose an optimization, adaptive concretization, to reduce the symbolic execution overheads. We exploit the fact that most of virtual device states conforming to silicon device states are generated by execution paths accessing none or only a few of symbolic values. Adaptive concretization eliminates unnecessary symbolic values, in order to prune unnecessary paths explored by symbolic execution.

We have evaluated the optimized approach on three Ethernet adapters and their virtual devices from QEMU virtual machine [2]. We discovered 21 inconsistencies, behind which there are 21 device bugs in either the silicon devices or their virtual devices. Furthermore, the time usages have been reduced by an order of magnitude, compared to those of the unoptimized approach.

II. Post-silicon Conformance Checking

A. Definitions for Conformance Checking

In [1], the conformance 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 not observable in general. The virtual device 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.

Definition 1: A **virtual device state** *is denoted as $V{=}\langle V_I, V_N\rangle$ where V_I is the device interface state, i.e., the assignments to variables in R_I and V_N is the device internal state, i.e., the assignments to variables in R_N.*

Definition 2: A **silicon device state** *is denoted as $S{=}\langle S_I, S_N\rangle$ where S_I is the assignments to variables in R_I and S_N is the assignments to variables in R_N with symbolic values.*

Both V and S can be treated as symbolic states, which are two sets of concrete device states, denoted as $set(V)$ and $set(S)$ respectively. The conformance between a silicon device state and a virtual device state is described in Definition 3. If a virtual device state V conforms to a silicon device S, we refer to V as a **conforming state**.

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

B. Conformance Checking Workflow

As illustrated in Figure 1, the framework has two major components: a trace recorder and a conformance checker. The trace recorder records the driver request sequence to the silicon device. The conformance checker replays the sequence on the virtual device, checks the conformance, and reports the discovered inconsistencies.

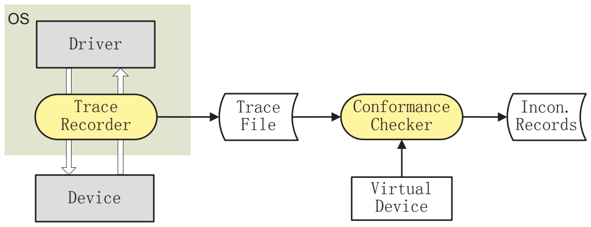

Fig. 1: Conformance Checking Workflow

1) 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**, denoted 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}.

2) Conformance checking algorithm: the conformance checker replays T on the virtual device using symbolic execution. Algorithm 1 presents this work flow. It takes a device trace T and a virtual device F as inputs. Major functions in Algorithm 1 are described below.

Algorithm 1 replay_trace(T, F)

```
1:  $T' \leftarrow convert\_trace(T)$
2:  /* Take $\langle S_k, D_k\rangle$ from $T'$*/
3:  for $k: 0 \rightarrow n$ do
4:      /*Set VD state $V_k$ to be SD state $S_k$*/
5:      $V_k \leftarrow S_k$
6:      /*Symbolically execute VD by taking $D_k$ at $V_k$ state*/
7:      $G \leftarrow sym\_exec(F, V_k, D_k)$
8:      $H \leftarrow conformance\_check(G, S_{k+1})$
9:      if $H == \emptyset$ then
10:         $report\_incon()$
11:     end if
12: end for
```

1) Given $T = \langle S_{I_0}, D_0\rangle$, $\langle S_{I_1}, D_1\rangle$, ..., $\langle S_{I_n}, D_n\rangle$, function $convert\_trace$ generates a new device trace T' = $\langle S_0, D_0\rangle$, $\langle S_1, D_1\rangle$, ..., $\langle S_n, D_n\rangle$, where $S_k (0 \leq k \leq n)$ is a silicon device state derived from S_{I_k}. (cf. Definition 2).
2) Function sym_exec symbolically executes the virtual device and generates a set of virtual device states denoted as $G = \{g_i \mid 0 \leq i \leq p\}$.
3) Conformance checking checks the conformance between G and the next silicon device state under D_k, denoted as S_{k+1}. Definition 4 defines their conformance. Function $conformance\_check$ generates a set of virtual device states $H = \{h_i \mid h_i \neq \emptyset, 0 \leq i \leq m\}$ where $h_i = set(g_j) \cap set(S_{k+1}), 0 \leq j \leq p$.
4) According to Defintion 4, if H is empty, there is an inconsistency and function $report\_incon$ reports the inconsistency.

Definition 4 (Device Conformance): Given $G = \{g_i \mid 0 \leq i \leq p\}$ and S_{k+1}, the virtual device and the silicon device conform to each other at D_k if $\exists g_i \in G$ where $0 \leq i \leq p$, $set(S_{k+1}) \cap set(g_i) \neq \emptyset$.

Notes. In the reminder of the paper, we refer to the above conformance checking approach as the native approach.

III. Optimization

This section presents our optimization applied to the native conformance checking approach. Before we describe our optimization, we introduce several preliminary definitions.

A. Preliminary Definitions

Definition 5 (Virtual Device Path): A **virtual device path** *is an execution path derived from symbolic execution of the virtual device. It can be viewed as a sequence of branch conditions, denoted as $\pi = c_0, c_1, ..., c_{n-1}, c_n$, where c_i $(0 \leq i \leq n)$ is a branch condition, a Boolean expression over device state variables and external environment inputs. We refer to a virtual device path as a path for simplicity.*

Definition 6 (Conforming Path): Given a virtual device state V_k and its set of next states $G = \{g_i \mid 0 \leq i \leq n\}$, $\forall g_i \in G$, there exists a virtual device path π that V_k transitions to g_i following π, denoted as $V_k \overset{\pi}{\Rightarrow} g_i$. V_k is the previous state of π and g_i is the next state of π. If g_i is a conforming state, we define π as a **conforming path**.

B. Adaptive Concretization

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

We apply an optimization, namely adaptive concretization, to the native approach. Figure 2 shows the work flow. Adaptive concretization includes two rounds of conformance checking. In the first round, we concretize (1) virtual device internal variables and (2) external environment inputs, which all have symbolic values. Then we check the conformance following the same work flow as the native approach. We define this first round as **concrete mode**. However, as the concrete values we assign to the variables might not be the right value, thereby the concrete mode may produce false alarms, i.e., false positives. To eliminate these false positives, we conduct a second around using the original virtual device where the internal state and external inputs all have symbolic values. This round verifies the inconsistencies discovered in the concrete mode. We define this second round as **refinement mode**.

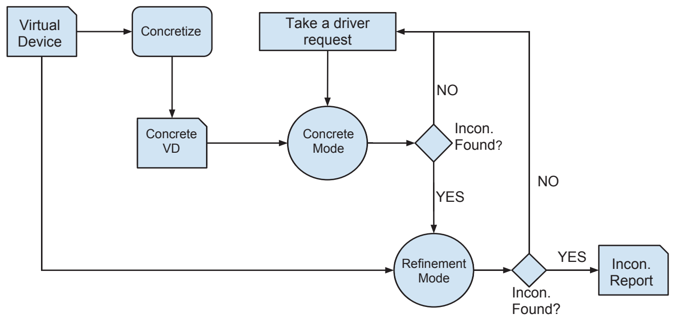

Fig. 2: Workflow of Adaptive Concretization

1) Concrete mode: the conformance checking algorithm in the concrete mode is shown in Algorithm 2. It takes a device trace T and a virtual device F as its inputs.

Algorithm 2 concrete_mode(T, F)

```
1:  $T' \leftarrow convert\_to\_concrete\_trace(T)$
2:  $F' \leftarrow concretize\_device(F)$
3:  /* Take $\langle S_k, D_k \rangle$ from $T'$*/
4:  for $k: 0 \rightarrow n$ do
5:      $V_k \leftarrow S_k$
6:      $G \leftarrow sym\_exec(F', V_k, D_k)$
7:      $H \leftarrow conformance\_check(G, S_{k+1})$
8:      if $H == \emptyset$ then
9:          $refinement\_mode(F, V_k, S_{k+1}, D_k)$
10:     end if
11: end for
```

Algorithm 2 follows the work flow of Algorithm 1 except three modifications: (1) function $convert\_to\_conrete\_trace$ is applied instead of $convert\_trace$ to concretize silicon device states in T; (2) function $concretize\_device$ concretizes the virtual device F; (3) when an inconsistency is discovered, the workflow enters the refinement mode rather than directly reporting an inconsistency.

Given $T = \langle S_{I_0}, D_0 \rangle$, $\langle S_{I_1}, D_1 \rangle$, ..., $\langle S_{I_n}, D_n \rangle$, function $convert\_to\_conrete\_trace$ converts to $T' = \langle S_0, D_0 \rangle$, $\langle S_1, D_1 \rangle$, ..., $\langle S_n, D_n \rangle$, where $S_k = \langle S_{I_k}, S_{N_k} \rangle$ $(0 \leq k \leq n)$ derived from S_{I_k}. Instead of assigning symbolic values to internal state variables of S_{N_k}, function $convert\_to\_conrete\_trace$ assigns value zero to variables of S_{N_k}. Moreover, in function $concretize\_device$, external environment inputs to the virtual device F are also concretized to zeros. As the values of some environment input variables can not be zero, for example, the value for modeling the received packet size cannot be zero, function $concretize\_device$ randomly picks up non-zero concrete values in their valid range.

We use zero rather than other concrete values for concretizing since most of the internal state variables have zero as their initial values. By setting zero, we can largely avoid introducing false positives in the concrete mode. The zero value we use to concretize symbolic values should be treated as a special concrete value. We denote such a value as 0_{sym}, indicating this zero is concretized from a symbolic value and will be recovered to the symbolic value in the refinement mode. As discussed above, some variables are concretized into non-zero values. For simplicity, when we say concretizing a variable to 0_{sym}, this also means that if the variable is an environment input variable always with non-zero value, it is concretized to a random non-zero value.

2) Refinement mode: the refinement mode takes the virtual device F, a virtual device state V_k, a silicon device state S_{k+1}, and a driver request D_k as its inputs. It has the same work flow as a single iteration presented in Algorithm 1. Additionally, it has a conversion functions $Con2Sym$. Function $Con2Sym$ is invoked immediately when the work flow enters the refinement mode. It replaces 0_{sym} of virtual device variables with symbolic values. By recovering 0_{sym} to the symbolic value, the refinement mode re-simulates the virtual device under the driver request leading to the inconsistency in concrete mode. The inconsistency produced in the refinement mode are reported as a real inconsistency.

IV. Evaluation

A. Experiment Setup

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

TABLE I: Summary of Devices for Case Studies

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

TABLE II: Types of Bugs in Virtual and Silicon Devices

No.	Bug Description	Num.	Distribution
1	Reserved Bits/Registers are updated	2	e1000
2	Generate unnecessary interrupts	2	eepro100, e1000
3	Fail to generate interrupts	1	bcm5751
4	Fail to clear interrupts	1	bcm5751
5	Fail to update registers	4	e1000, bcm5751
6	Update registers with wrong values	2	e1000
7	Model state with wrong data types	1	bcm5751
8	Registers are out of sync	2	bcm5751
9	Reset to incorrect values	3	eepro100, e1000, bcm5751
10	Fail to model concurrency of SD	3	eepro100, e1000, bcm5751

B. Bug Detection

In this section, we demonstrate that our optimized approach can detect all the bugs previously discovered by the native approach. To demonstrate that our optimized approach does not reduce the capacity comparing to the native approach, we preform the test cases triggering the previous inconsistencies between silicon and virtual devices. We summarize all the bugs derived from inconsistencies in Table II. VD indicates the virtual device bugs while SD indicates the silicon device bugs. The results shows that our approach detects all the 15 previous bugs. Moreover, by issuing more test cases to three network adapters, we detect 6 new virtual device bugs which are the ninth and tenth type of bugs.

TABLE III: 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

C. Efficiency

We evaluate the efficiency of our approach, in terms of time usages and memory usages. 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 III. "NIC test-suite" contains a family of typical test cases on network interface controllers (NIC), which manipulates a NIC in different ways, e.g., sending UDP packets and setting MTU size.

TABLE IV: Summary of Time and Memory Usages

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

1) Time usages: we calculate the average time usages to process 100 driver requests in each test cases. Table IV summarizes the results. The time have been reduced an order of magnitude by using the optimized approach. The time usages are reduced less in the test cases "Ping" and "Transfer files" than the other two test cases. The reason is that these two test cases involve receiving packets. The test case involving receiving packets has more false positives introduced in the concrete mode, as the conforming paths usually access many symbolic variables representing the environmental inputs. Therefore, in these two test cases, the approach often requires the refinement mode and the time usages are increased.

2) Memory usages: We evaluate the memory usages in the same way as evaluating time usages. As Table IV shows, the results suggest that our optimized approach has almost same memory usages with the native approach.

V. Related Work

Post-silicon validation is important but difficult. Most of previous work has been focused on improving observability of hardware internal. A notable work is "backspace" [3], which uses SAT-solving techniques to provide an execution trace to a crashed post-silicon state, thus facilitating off-line debugging. Several approaches [4], [5], [6] integrate assertions into post-silicon checking of hardware by observing its execution trace. In [7], [8], hardware monitors are introduced to ameliorate observability requirements on silicon.

Many research have been done for reducing symbolic execution overheads. A major effort is to avoid path explosions by pruning redundant paths. RWSet [9] and path subsumption [10] employ a similar heuristic where a path which is identical to the one previously explored can be safely pruned. Kuznetsov et al. [11] propose a method of automatically merging states to reduce the number of paths explored in symbolic execution. Several other approaches [12], [13], [14] leverage the benefits of concolic execution to partially concretize the target programs; thereby the number of explored paths is decreased.

VI. Conclusions and Future Work

We have presented adaptive concretization, an optimization for efficiently checking the conformance between virtual and silicon devices. By employing this optimization, time usages of conformance checking are reduced significantly. This optimization makes conformance checking efficient and capable of scaling to hardware devices with complicated designs. For the next step, we plan to apply our approach to more devices.

VII. 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] L. Lei, F. Xie, and K. Cong, "Post-silicon Conformance Checking with Virtual Prototypes," in *Proc. of DAC, 2013*.

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

[3] F. M. De Paula, M. Gort, A. J. Hu, S. Wilton, and J. Yang, "BackSpace: Formal Analysis for Post-Silicon Debug," in *Proc. of FMCAD, 2008*.

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

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

[6] 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.

[7] S.-B. Park and S. Mitra, "IFRA: instruction footprint recording and analysis for post-silicon bug localization in processors," in *Proc. of DAC, 2008*.

[8] S. Ray and W. A. Hunt, Jr., "Connecting Pre-silicon and Post-silicon Verification," in *Proc. of FMCAD, 2009*.

[9] P. Boonstoppel, C. Cadar, and D. Engler, "RWset: attacking path explosion in constraint-based test generation," in *Proc. of TACAS, 2008*.

[10] S. Anand, C. S. Păsăreanu, and W. Visser, "Symbolic execution with abstract subsumption checking," in *Proc. of SPIN, 2006*.

[11] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proc. of PLDI, 2012*.

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

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

[14] A. Tomb, G. Brat, and W. Visser, "Variably interprocedural program analysis for runtime error detection," in *Proc. of ISSTA, 2007*.