Validating Direct Memory Access Interfaces with Conformance Checking

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

***Abstract*—Direct Memory Access (DMA) interfaces are a common and important component of Hardware/Software (HW/SW) interfaces between peripheral devices and their device drivers. We present a HW/SW co-validation framework to validate DMA interface implementations of a device and its driver. This framework employs a virtual prototype of the device as a reference model and performs co-validation in two stages: (1) conformance checking which checks the DMA interface conformance between the device and its virtual prototype; (2) property checking which checks device/driver interactions across the DMA interface. In conformance checking, the virtual prototype infers the device state transitions by taking the same driver request sequence to the device. Property checking verifies system properties over the device state transitions exposed through the virtual prototype. This framework assists HW/SW integration validation by detecting DMA interface bugs in both devices and drivers. Furthermore, we have developed three key techniques: capture-on-write policy, partial capture, and environmental input prediction, to address two major challenges in scaling the framework: DMA capture overhead and imprecise environmental input simulation. We have applied this approach to four Ethernet adapters, discovering 12 serious DMA interface bugs from the devices, their virtual prototypes and their drivers. The results demonstrate that our approach has major potential in facilitating HW/SW co-validation.**

I. Introduction

Post-silicon validation is a critical stage in the development cycle of computing systems. In this stage, not only hardware silicon validation is conducted, but also hardware/software (HW/SW) integration validation. A recent study [2] indicates that the cost of HW/SW integration validation has increased significantly. As the complexities of systems grow, there are several key challenges in the post-silicon integration validation:

1) **Lack of HW/SW interface observation.** HW/SW integration validation often relies on testing the entire system with high-level application test scenarios. However, HW/SW interfaces are often not sufficiently observed and certain interface bugs escape detection.
2) **Difficulty in attributing HW/SW interface bugs.** When a bug is discovered in HW/SW integration validation, it is often unclear if it is a hardware bug or a software bug due to the close involvement and interaction of both hardware and software.
3) **Difficulty in debugging HW/SW interfaces.** Hardware interacts with its control software frequently, producing a huge number of I/O events. To troubleshoot, the engineers usually have to sift through thousands of I/O events and analyze them manually.

Given the ubiquity and seriousness of these challenges, it is highly desired to develop systematic methods to validate HW/SW interfaces and automatically detect and analyze interface bugs. For most peripheral devices, I/O interfaces based on Direct Memory Access (DMA) are a critical part of their HW/SW interfaces. For example, in Intel EERPO100 Ethernet adapter specification [11], 25% of all pages describe the DMA interface implementations. Therefore, DMA interface validation is a critical task in HW/SW co-validation.

Recently, virtual prototyping has emerged as a promising technique to enable early software development. A virtual prototype is a system-level, executable software model of a hardware device with full observability. The device interface modeled by the virtual prototype is required to be functionally equivalent to that of the silicon device. Thus, virtual prototypes have major potential in facilitating HW/SW co-validation.

In this paper, we present a HW/SW co-validation framework for validating the DMA interface of a device and its driver. We utilize the virtual prototype of the device as a reference model in validating the DMA interface. The framework conducts co-validation in two stages: conformance checking and property checking. Conformance checking checks the DMA interface conformance between the device and its virtual prototype, thereby validating the DMA interface implementations of both. The general work flow of conformance checking has three steps: (1) recording the driver requests to the device and the device interface state before each request; (2) executing the virtual prototype with the recorded driver request sequence; (3) checking if there are any inconsistencies in interface states between the device and the virtual prototype. Through conformance checking, the virtual prototype shadows the device execution. Property checking leverages the virtual prototype to expose the device state transitions and verifies the properties related to the device/driver interactions across the DMA interface. By checking the properties, invalid driver inputs to the DMA interface are detected.

In general, a device interface includes interface registers and the DMA interface. Previous work [15] present an approach to conformance checking over device interface registers. In our framework, we extend this approach to conformance checking over both interface registers and the DMA interface. Our extended approach detects not only DMA interface bugs but also new bugs in interface registers whose values have dependencies on DMA interface state. However, the straightforwardly extended conformance checking is not scalable to complicated device designs due to two limitations:

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

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

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

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

We have applied our framework to four Ethernet adapters and their drivers using their virtual prototypes from QEMU [4]. Our approach has discovered 12 bugs in DMA interface implementations of the devices, their virtual prototypes, and their drivers. Moreover, the techniques for reducing recording overheads make our framework applicable to two devices with complicated designs.

In summary, this paper makes following key contributions:

1) The paper presents a HW/SW co-validation framework for DMA interface implementations of devices and their drivers using their virtual prototypes.
2) Besides validating the DMA interface implementations, our extended conformance checking further validates interface registers related to the DMA interface (see details in Section III-C).
3) The three key optimizing techniques make our framework scalable and effective on real industry designs.

Outline. The balance of this paper is organized as follows. Section II introduces the relevant background concepts. Section III and Section IV present our approach and three optimizations. Section V reports our experiment results. Section VI discusses related work. Section VII concludes and discusses future work.

II. Background

A. QEMU Virtual Devices

A virtual prototype is the software implementation of a hardware peripheral device and is usually integrated into a virtual platform, e.g., QEMU virtual machine [4]. In this paper, we use QEMU virtual devices as virtual prototype examples; nonetheless, other virtual prototypes have similar structures.

A QEMU virtual device is composed of the device state and device module functions. The device state models the interface and internal registers of the device. The device module functions simulate the device functionalities. Figure 1 shows excerpts from the QEMU virtual device of Intel eepro100 Ethernet adapter. Function $eepro100\_write1$ and function $nic\_receive$ are module functions, modeling how the device responds to the driver request and receives packets respectively.

```
typedef struct EEPRO100State_st {
  PCIDevice    dev;
  uint8_t      mem[PCI_MEM_SIZE];
  ......
} EEPRO100State;

static void
eepro100_write1(EEPRO100State * s,
                uint64_t addr, uint8_t val)
{
   s->mem[addr] = val;
   eepro100_write_command(s, val);
   ......
}

static ssize_t
nic_receive(VLANClientState *nc,
            const uint8_t *buf, size_t size)
{
   ... ...
}
```

Fig. 1: Excerpts from QEMU eepro100 virtual device

Categories of Device Modules. The modules in a virtual prototype fall into two categories: (1) the modules responding to driver requests; (2) the modules handling external environment inputs. We refer to them as Command Module (CM) and Environment Module (EM) respectively, e.g., function $eepro100\_write4$ is a CM and function $nic\_receive$ an EM.

B. Conformance Checking over Interface Registers

This section reviews a previous approach [15] to conformance checking over device interface registers between a device and its virtual prototype. Our new framework follows a similar work flow and extends it to checking DMA interfaces.

1) Symbolic execution: Conformance checking utilizes symbolic execution [13] to simulate the virtual prototype. It leverages symbolic execution to overcome the limited observability of hardware silicon. In HW/SW integration validation, the device internal registers are usually not observable. Conformance checking models them using variables with symbolic values when replaying the recorded driver request sequence on the virtual prototype. In this way, symbolic execution covers all the possible values of internal registers.

2) Conformance definitions: The device interface state is defined as a set of interface registers with their values and the device internal state is defined as a set of internal registers with their values. A virtual prototype state V is modeled as a pair $\langle V_I, V_N \rangle$, where V_I is the interface state and V_N is the internal state. A device state S has a same structure: $\langle S_I, S_N \rangle$, where S_I and S_N are the device interface and internal states. The registers in S_N and V_N are assigned as symbolic values since their values are unknown. The values in S_I are all concrete while the values in V_I can be either symbolic or concrete.

Definition 1 (Interface state conformance): a device interface state S_I and a virtual prototype interface state V_I conform to each other if $S_I \in V_I$ where S_I is concrete while V_I symbolic and is considered as a set of concrete states.

3) Conformance checking framework: As illustrated in Figure 2, the conformance checking framework has two stages, runtime recording and offline checking. In the runtime recording stage, the trace recorder records a **device trace**: a driver request sequence to the device and the device interface state before each driver request is issued. In the offline checking stage, the conformance checker symbolically executes the virtual prototype by taking the recorded device trace, checks the interface register conformance after processing each driver request, and reports the interface register inconsistencies.

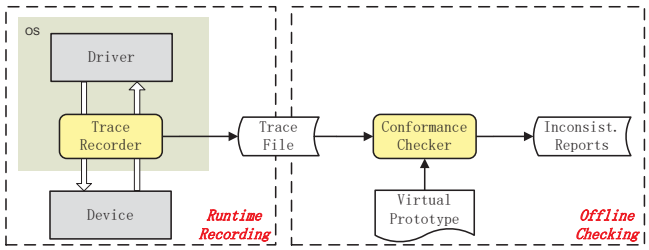


Fig. 2: Conformance checking framework

While processing a driver request D with symbolic execution, the conformance checker produces a set of virtual prototype states. We extract their interface states and denote the set of interface states as $G = \{g_i \mid 0 \leq i \leq n\}$. Conformance checking checks the conformance between G and the corresponding device interface state S_I produced under D. Definition 2 defines their conformance. After checking the conformance, the conformance checker assigns the device register values to the corresponding registers in the virtual prototype, then moves to the next driver request.

Definition 2 (Device Conformance): Given $G = \{g_i \mid 0 \leq i \leq n\}$ and S, the virtual prototype and the device conform to each other at D if $\exists g_i \in G$ where $0 \leq i \leq n$, $S_I \in g_i$.

4) Non-deterministic scheduling: When the driver sends a sequence of requests to the device, there might be environmental inputs arriving at the device between two consecutive driver requests. Therefore, while replaying the device trace, the conformance checker must explore the device behavior under the environmental inputs on the virtual prototype to avoid reporting false alarms. To achieve this goal, the conformance checker makes a non-deterministic choice which has two branches: invoking EM and not invoking EM. Symbolically executing the virtual prototype captures both possibilities: environmental inputs arrive and no environmental input arrives. Non-deterministic scheduling eliminates false alarms but might miss DMA interface bugs. We present environmental input prediction which addresses this shortcoming in Section IV.

III. Overview

A. Preliminaries

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

1) The driver builds a descriptor d which contains a command c. The driver puts d into the DMA interface and updates a special interface register Reg of the device to notify the device that there is a command in the DMA interface.
2) Once Reg is updated, the device reads the descriptor d from the DMA interface and executes the task specified by c.
3) When the device completes the task, it updates the status of d and writes d back to the DMA interface. It may also update some relevant interface registers.

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

B. DMA Interface Validation Framework

As Figure 3 shows, our HW/SW co-validation framework is built on the conformance checking work flow. It takes the virtual prototype and a trace file generated from the trace recorder, and outputs an inconsistency report and a property failure report. It consists of two major components as follows.

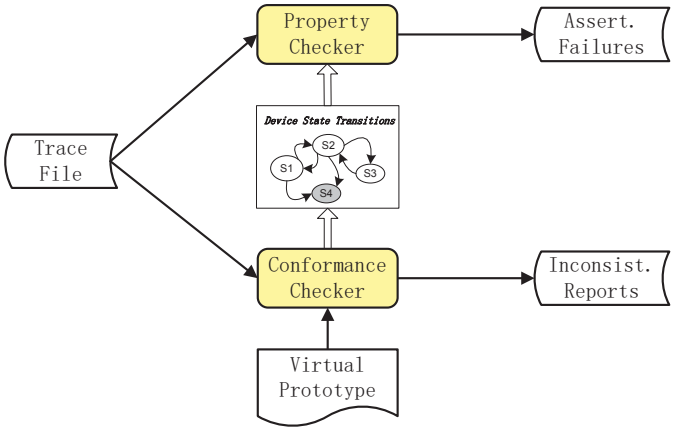


Fig. 3: HW/SW co-validation framework for DMA interfaces

- *Conformance checker.* Conformance checking over interface registers in II-B are extended to checking the conformance of both interface registers and DMA interface states between the device and its virtual prototype. The conformance checker detects errors not only in interface registers but also in DMA interfaces. As the conformance checker simulates the device behaviors over the virtual prototype and checks their conformance under each driver request, the virtual prototype essentially shadows the device execution trace and keeps track of the device state transitions. The device state transitions exposed by the virtual prototype provides the foundation for property checking.
- *Property checker.* Property checker verifies the properties related to device/driver interactions across the DMA interface. It observes the device state transitions through the virtual prototype and detects if any property violation is possible over the state transitions. A property failure often indicates that there is an invalid DMA driver input (see more details in Section III-D).

C. Conformance Checking over DMA Interfaces

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

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

We use an example to illustrate how such a bug escapes. Figure 4 shows how eepro100 processes a DMA driver command. Function pci_dma_read fetches a descriptor which is stored in $s \rightarrow cu\_desc$. As labels P1 and P2 indicate, depending on different commands, the device updates the CU state with different values and fires interrupts.

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

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

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

Fig. 4: DMA interface implementations of eepro100 VD

The variable $s \rightarrow cu\_desc$ is assigned symbolic values during symbolic execution of the eepro100 virtual device (VD). As a result, symbolic execution of eepro100 virtual device covers all the three paths in Figure 4. We denote the three paths as p_1, p_2, and p_3. The path p_1 follows the code where the branch condition at P1 is true. The path p_2 follows the code where the branch condition at P1 is false and branch conditon at P2 is true. The path p_3 follows the code where both of the two branch conditions are false.

Assume that there is a DMA register bug, when the device follows p_2, it fails to update the cu state with $cu\_suspend$. When executing the virtual prototype, the conformance checker also explores p_3 where the cu state is consistent with the cu state in the device. According to Definition 2, the conformance checker does not discover this update failure. Instead, the extended approach uses concrete DMA inputs, therefore, only p_2 is explored and the cu state is updated to $cu\_suspend$, which is inconsistent with the device cu state. The bug is discovered. The evaluation results show that our extended approach detected several bugs both in the DMA interface and DMA registers (cf. Section V).

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

1) **Record concrete DMA interface states**. In addition to the interface registers, the trace recorder also records the DMA interface state at runtime.
2) **Extend Device State Representation**. We define a DMA interface state as a set of DMA interface variables with their values. Given $V = \langle V_I, V_N \rangle$ and $S = \langle S_I, S_N \rangle$ defined in Section II-B2, we extend the virtual prototype state as $V_E = \langle V_{EI}, V_N \rangle$, where $V_{EI} = \langle V_I, V_M \rangle$, V_M represents the virtual prototype DMA interface state. Similar as V_I, the values of V_M can be either symbolic or concrete. We define the extended silicon device state as $S_E = \langle S_{EI}, S_N \rangle$, $S_{EI} = \langle S_I, S_M \rangle$, where S_M represents the device DMA interface state. The values of S_M are concrete.
3) **Report inconsistent DMA interface**. The conformance checker checks the conformance between V_{EI} and S_{EI} in the same manner as Definition 1 and Definition 2. If the virtual prototype and the device do not conform, the conformance checker outputs inconsistency reports which contains inconsistencies of both interface registers and the DMA interface.

Remarks. The previous approach is sound theoretically. However in practice, it might have false positives, i.e., false alarms. The virtual prototype might have unbounded loops which make symbolic execution non-terminating. A loop bounding algorithm is used to set constant bounds for these loops dynamically. This algorithm may reduce possible behaviors of the virtual prototype; therefore, producing false positives. Our approach faces the same challenge. However, the chance of false positives is lower than the previous approach in both DMA interface and interface register conformance checking results. In the previous approach, most of false positives are caused by bounding two kinds of loops: (1) ones whose loop conditions depend on the environmental inputs; (2) the others whose loop conditions depend on the DMA interface values. Our approach does not record environmental inputs; therefore, we may still get false positives on the first kind of loops. However, as we record concrete DMA interface values, our approach eliminates false positives caused by the second kind.

D. Property Checking

Through conformance checking, the virtual prototype shadows the device execution and exposes the device state transitions to the property checker. The property checker verifies properties over the device state transitions. By verifying such properties, the property checker detects the invalid DMA inputs from the driver. According to device specifications, there are two types of properties related to DMA driver inputs: (1) stateless properties, the properties without involving device states; (2) stateful properties, the properties related to the device state. As examples, we present two properties specified in the eepro100 specification [11] as follows.

Property 1*: When the driver issues a DMA descriptor, it should always clear the completion C bit of the descriptor.*

Property 2*: When the Receive Unit (RU) is in the status of no resources, the driver should always set the S bit of the DMA descriptor to 1, starting to discard incoming packets.*

Property 1 is a stateless property and Property 2 is a stateful property. We use them as examples to present the design and implementation of the property checking infrastructure.

1) Virtual prototype instrumentations: As we leverage the virtual prototype to infer the device state transitions, the virtual prototype can be directly used as a validation vehicle. For property checking, we instrument the virtual prototype with assertions generated from specified properties. While the conformance checker simulates the device behaviors on the virtual prototype, the property checker detects if any assertion fails during the simulation. Currently we instrument the assertions manually. In future, we will develop a method that allows the users to specify assertions in a certain format and automatically instruments the assertions. Figure 5 shows the two assertions corresponding to Properties 1 and 2 respectively. A special API function dcc_assert is used to specify these assertions.

```
... ...
pci_dma_read(address, &s->ru_desc, size);

// Assertion to enforce property 1
dcc_assert (!s->ru_desc & STATUS_C);

// Assertion to enforce property 2
if (s->ru_state == ru_no_resource)
  dcc_assert(s->ru_desc & COMMAND_S);
... ...
```

Fig. 5: Assertions instrumented in eepro100 virtual device

2) Detecting assertion failures: The property checker evaluates the assertions when the conformance checker executes the virtual prototype. Symbolic execution of the virtual prototype usually explores multiple program paths. By exploring each path, a virtual prototype interface state is generated. If a path leading to a virtual prototype interface state which conforms to the device state, we denote such a path as a conforming path. Definition 3 defines the condition when the property checker detects a property violation.

Definition 3 (Property violations): Given a property ψ, a set of conforming paths $P = \{p_i \mid 0 \leq i \leq n\}$ explored under a driver request D, ψ is violated under D if $\forall p_i \in P$, the assertion failure of ψ is reachable on p_i.

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

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

Remarks. Our current implementation of property checking has one limitation. When dealing with the stateful properties, we only allow specifying the properties related to the device interface state. Since the conformance checker only checks the interface state conformance between the device and its virtual prototype, there might be divergences in their internal states. As a result, without ensuring the internal state conformance, the driver violation related to the device internal state cannot be deemed as a true violation. In future work, we will develop algorithms to eliminate the internal divergence, so we can check the properties related to the internal state.

IV. Techniques for checking DMA interfaces

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

A. Record-on-write Policy

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

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

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

2) Associating DMA interface states with driver requests: Record-on-write leads to a potential problem: for some driver requests, there is no DMA interface state associated. However, when we replay the device trace on the virtual prototype, the virtual prototype may still read DMA memory even it does

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

B. Partial recording of DMA interface

A DMA interface of a device is not a flatten memory. Instead, it is typically implemented as a "ring buffer" data structure. As Figure 6 illustrates, the device and the driver keep two indices called "head" and "tail". When the driver allocates a unit of memory to the device, it increments "tail". Similarly, when the device consumes a unit of memory, it increments "head". The memory between "head" and "tail" is considered as valid memory. The device fetches DMA descriptors only from the memory units between "head" and "tail".

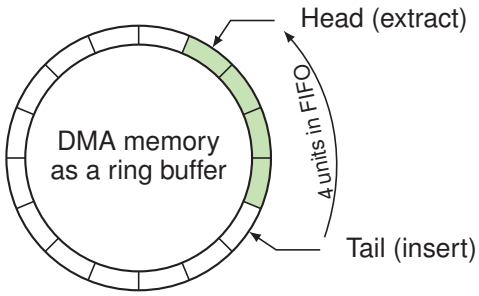
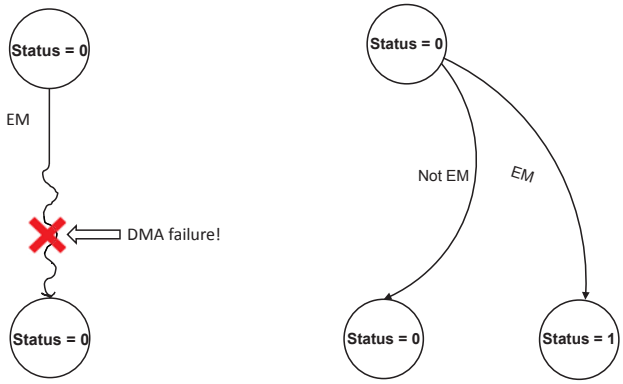


Fig. 6: Ring buffer structure of DMA memory

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

C. Environmental input prediction

1) Motivation: As illustrated in Section II-B4, upon each driver request, the conformance checker uses a non-deterministic choice to decide invoking EM or not. In this way, the virtual prototype can capture the device behaviors under two possible scenarios: (1) environmental inputs arrive; (2) no environmental input. However, there is a potential to miss DMA interface bugs. We present such a concrete example. When Intel eepro100 receives a packet from its external network, according to its specification, after processing the packet, the device will set its status bit to value 1 in the DMA interface, indicating the completeness of packet reception. Assume that the status update fails for some reason, as a result, the status bit remains 0 in the DMA interface (see Figure 7-(a)). However, this status bit has the same value as no external input arrivals. In the virtual prototype, as Figure 7-(b) shows, there are two paths including both reception (EM) and non-reception (Not EM), the conformance checker covers both paths by symbolically executing the virtual prototype. Therefore, although the DMA interface update fails, it is still considered valid. This update failure will not be discovered.



(a) Device Trace (b) Virtual Prototype Traces

Fig. 7: DMA bugs missed w/o environment input prediction

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

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

V. Evaluation

A. Experiment Setup

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

B. Bug Detection

Our framework has detected 12 new bugs summarized in Table I. There are 2 device bugs, 8 virtual prototype bugs, and 2 driver bugs. Since we conducted our experiments over the

TABLE I: Summary of Device, Virtual Prototype, and Driver Bugs

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

TABLE II: Summary of Devices for Case Studies

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

TABLE III: Summary of Test Cases

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

stable products which have been released for many years, there are only a few device bugs. However, the virtual prototype bugs that we discovered are all common hardware design flaws. Therefore, our approach has major potential in discovering bugs in silicon prototypes including FPGAs and test devices. All the driver bugs are discovered by our property checking. Property checking verified 26 properties in total, of which there are 9 stateless properties and 17 stateful properties.

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

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

C. Efficiency

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

The test cases are issued under three configurations: (1) No Recording (NR) mode: there is no recording conducted; (2) *Recording Everything (RE) mode:* the recording method without the two proposed techniques, which records everything in the DMA interface; (3) *Record-on-write and Partial recording (RP) mode:* the method with record-on-write and partial recording techniques. We set the NR mode as the baseline and the performance of the NR mode is normalized to 1. Figure 8 shows the ratios of the RE and RP modes comparing to the NR mode. In Figure 8, no data is provided for the RE mode in terms of e1000 and bcm5751 since the RE mode incurs a large overhead and the system hangs. By applying record-on-write and partial recording techniques in the RP mode, recording DMA interface states can be successfully and efficiently achieved.

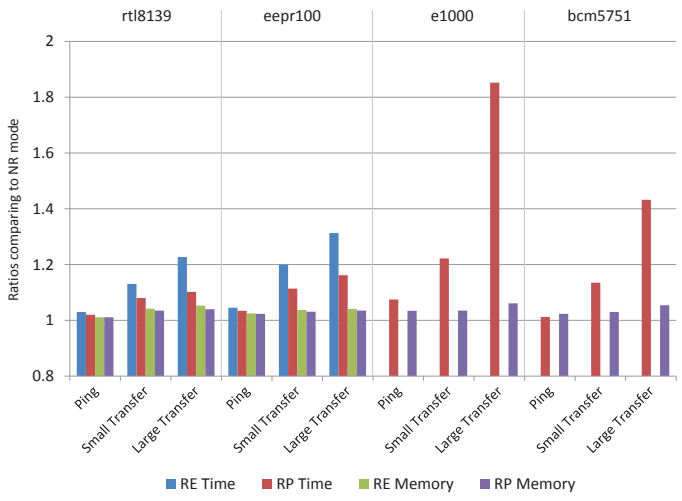


Fig. 8: Time and memory usages of test cases under Recording Everything (RE) and Record-on-write and Partial recording (RP) modes. The usages with no recording (NR) are normalized to 1. Figure shows ratios of RE and RP comparing to NR

The results demonstrate that our two optimizing techniques make recording DMA interface states scalable to the devices with complicated designs such as e1000 and bcm5751, both Gigabit Ethernet adapters; for the devices such as eepro100 and rtl8139, both 10/100M Ethernet adapters, record-on-write and partial recording also noticeably reduce the recording overheads.

VI. Related Work

Our work is related to post-silicon validation, HW/SW interface assurance, and DMA validation. Post-silicon validation is performed on silicon prototypes and test devices. A significant amount of research has focused on detecting and localizing bugs in silicon chips. Several approaches [1], [18], [20] have built hardware on-chip monitors to collect hardware execution traces with internal signals. Assertion-based verification [5], [10] and formal method [6] have been used to analyze and debug the execution traces from on-chip monitors. Our approach also works on detecting and troubleshooting post-silicon bugs. Instead of validating internal implementations of silicon hardware, we focus on validating HW/SW interfaces.

There has been a lot of work on HW/SW interface assurance at pre-silicon stage. HW/SW co-verification and HW/SW co-simulation are two main streams of techniques. In HW/SW co-verification, model checking [14], [16], [25], [17] is widely used. It verifies properties by analyzing the interface implementation statically; however, it often encounters the state explosion problem. Our co-validation framework conducts verification over the execution trace generated at runtime which largely avoids the state explosion problem. Research on co-simulation [3], [7], [8], [9], [19], [21], [23] typically utilizes design models of the hardware and does not directly work with the implementation of the hardware/software implementation.

We are not aware of work closely related to DMA interface validation on the device side. Nevertheless, some research on driver testing/verification [12], [22] validates DMA interfaces from the driver side. A previous work [24] detects malwares residing in DMA interface by verifying specific DMA operation properties. All these methods validate DMA interfaces without considering device behaviors. Our approach is the first attempt to validate the DMA interface implementations of both hardware and software.

VII. Conclusions and Future Work

We have presented a HW/SW co-validation framework to validating the DMA interface implementations. Our two-staged checking infrastructure helps detect errors in DMA interface implementations of both a device and its device driver. We discovered many bugs from devices, their virtual prototypes, and their driver. Our validation framework has major potential in addressing the key challenges of HW/SW integration validation. First, the framework records and validates the DMA interface state; thus the errors in the DMA interface are detected effectively. Second, the framework identifies both the device and driver errors over the DMA interface thereby it can easily attribute device/driver interface bugs as device or driver bugs. Finally, our framework only requires minimum manual efforts, which significantly saves validation human efforts.

Our future work will explore the following directions: (1) extending property checking to validate both DMA driver inputs and driver requests updating interface registers; (2) adapting our framework to be an on-line monitoring approach, instead of the offline checking approach that we have so far.

VIII. Acknowledgment

This research received financial support from National Science Foundation (Grant #: 0916968). A patent (No. 8,666,723) filed on this research by Portland State University has been licensed to Virtual Device Technologies (VDTech) where Fei Xie is a partner.

References

[1] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs," in *Proc. of DAC, 2006*.

[2] R. Avinun, "Concurrent hardware/software development platforms speed system integration and bring-up," White Paper. [Online]. Available: https://www.cadence.com/rl/resources/white_papers/system_dev_wp.pdf

[3] D. Becker, R. K. Singh, and S. G. Tell, "An engineering environment for hardware/software co-simulation," in *Proc. of DAC, 1992*.

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

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

[6] 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*.

[7] A. Ghosh, M. Bershteyn, R. Casley, C. Chien, A. Jain, M. Lipsie, D. Tarrodaychik, and O. Yamamo, "A hardware-software co-simulator for embedded system design and debugging," in *ASP-DAC, 1995*.

[8] R. Gupta, C. Coelho, and G. D. Micheli, "Synthesis and simulation of digital systems containing interacting hardware and software components," in *Proc. of DAC, 1992*.

[9] A. Hoffmann, T. Kogel, and H. Meyr, "A framework for fast hardware-software co-simulation," in *Proc. of DATE, 2001*.

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

[11] *Intel 8255x 10/100 Mbps Ethernet Controller Family – Open Source Software Developer Manual*, 1st ed., Intel, January 2006.

[12] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Fine-grained fault tolerance using device checkpoints," in *Proc. of ASPLOS, 2013*.

[13] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[14] R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün, "Combining software and hardware verification techniques," *Formal Methods in System Design (FMSD)*, vol. 21, no. 3, pp. 251–280, November 2002.

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

[16] J. Li, X. Sun, F. Xie, and X. Song, "Component-based abstraction and refinement," in *Proc. of ICSR, 2008*.

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

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

[19] C. Passerone, L. Lavagno, M. Chiodo, and A. L. Sangiovanni-Vincentelli, "Fast hardware/software co-simulation for virtual prototyping and trade-off analysis," in *Proc. of DAC, 1997*.

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

[21] J. A. Rowson, "Hardware/software co-simulation," in *DAC, 1994*.

[22] L. Ryzhyk, J. Keys, B. Mirla, A. Raghunath, M. Vij, and G. Heiser, "Improved device driver reliability through hardware verification reuse," in *Proc. of ASPLOS 2011*.

[23] L. Semeria and A. Ghosh, "Methodology for hardware/software co-verification in C/C++," in *Proc. of ASP-DAC, 2000*.

[24] P. Stewin, J.-P. Seifert, and C. Mulliner, "Poster: Towards detecting DMA malware," in *Proc. of CCS, 2011*.

[25] F. Xie, G. Yang, and X. Song, "Component-based hardware/software co-verification for building trustworthy embedded systems," *JSS, 2007*.