Improved Device Driver Reliability Through Hardware Verification Reuse

Leonid Ryzhyk[1,2] John Keys[3] Balachandra Mirla[1,2] Arun Raghunath[3]
Mona Vij[3] Gernot Heiser[1,2]

[1] NICTA * [2] University of New South Wales [3] Intel Corporation

leonid.ryzhyk@nicta.com.au john.keys@intel.com balachandra.mirla@nicta.com.au
arun.raghunath@intel.com mona.vij@intel.com gernot.heiser@nicta.com.au

Abstract

Faulty device drivers are a major source of operating system failures. We argue that the underlying cause of many driver faults is the separation of two highly-related tasks: device verification and driver development. These two tasks have a lot in common, and result in software that is conceptually and functionally similar, yet kept totally separate. The result is a particularly bad case of duplication of effort: the verification code is correct, but is discarded after the device has been manufactured; the driver code is inferior, but used in actual device operation. We claim that the two tasks, and the software they produce, can and should be unified, and this will result in drastic improvement of device-driver quality and reduction in the development cost and time to market.

In this paper we propose a device driver design and verification workflow that achieves such unification. We apply this workflow to develop and test drivers for four different I/O devices and demonstrate that it improves the driver test coverage and allows detecting driver defects that are extremely hard to find using conventional testing techniques.

Categories and Subject Descriptors D.4.4 [*Operating Systems*]: Input/Output; B.4.2 [*Input/Output and Data Communications*]: Input/Output Devices

General Terms Reliability, Verification

Keywords Device Drivers, Reliability, RTL Testbenches, Automated Testing, Co-verification.

1. Introduction

Device drivers are critical components of an operating system (OS), as they are the software that controls peripheral hardware, such as disks, network interfaces or graphics displays. They make up a large fraction of OS code, e.g., around 70 % in Linux.

* NICTA is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.

Drivers are also important for another reason: they are the leading reliability hazard in modern OSes. Drivers are known to be responsible for the majority of OS failures[9], and have been shown to have 3–7 times the defect density of other OS code [5].

Our recent study [21] has shown that the leading class of driver defects, comprising about 40 % of all driver bugs, are *device protocol violations*, i.e., situations where the driver incorrectly handles the software interface of the device. Common examples of such defects include incorrect use of device registers, sending commands to the device in the wrong order, incorrectly interpreting device responses, etc.

Device protocol violations are easy to introduce but hard to detect and eliminate. They are easy to introduce because driver developers often lack adequate documentation describing how the device should be controlled from software. Available documentation provided by hardware designers in the form of a device datasheet is usually incomplete and inaccurate, leading to numerous driver defects.

Device protocol violations are hard to detect due to limitations of the conventional quality assurance process based on testing. In order to thoroughly test the interaction between the driver and the device, the test harness must exercise the driver under various combinations of inputs from the OS and the device. Existing device driver testing kits [16, 25] do a good job of exercising the OS interface of the driver by generating various sequences of I/O requests that model real application scenarios. Testing different hardware operating conditions is a much harder task. For instance, the behaviour of a network controller device depends on the Ethernet line speed, data throughput, collision rate, inter-packet gap, hardware flow control, and numerous other parameters. The device driver must be able to correctly handle the device under any combination of these parameters.

The problem is that controlling the values of these parameters requires special hardware, firmware, and software support on the remote host. While such support can be provided in principle [25], this is rarely done in practice due to the high complexity of this approach. As a result, many driver defects escape even seemingly thorough testing that does not model different environments that the device can be placed into.

Other components of the system that affect the device behaviour include the CPU, the I/O bus, and the OS scheduler. All of these components influence the timing of communication between the driver and the device in ways that are beyond the control of the testing software, which further limits the achievable test coverage. For example, I/O bus contention in a system with multiple active I/O devices can reduce the device-to-memory data transfer speed and lead to internal device buffer overflow, which must be carefully

handled by the driver. Since such uncommon scenarios cannot be easily triggered in a directed fashion, they often remain untested.

In addition to the lack of control over the driver's execution environment, the conventional driver testing methodology also suffers from the lack of observability of the device behaviour. Every I/O request sent to the driver must result in certain events occurring at the external interface of the device. For instance, a request to send a network packet must result in the correct data, with valid CRC and padding fields being sent through the Ethernet interface of the device. Since the testing code cannot directly monitor this external device interface, it must rely on indirect evidence to make sure that the driver completed the operation correctly. This is not always sufficient, since an incorrect behaviour is often concealed by timing delays or by hardware reliability mechanisms.

In summary, conventional driver testing suffers from serious limitations that reduce its effectiveness in detecting driver defects. The problem is exacerbated by the inadequate device documentation, leading to the overall poor quality of driver code.

We argue that the underlying cause of these difficulties is the separation of two highly-related tasks: *device verification* and *driver development*. As explained in more detail in the following sections, these two tasks have a lot in common, and result in software that is conceptually and functionally similar. The main difference is that the hardware verification code is built around a simulated model of the device and thus has complete control over all components of the simulated environment, including the I/O bus, the external physical medium, and the thread scheduler, which enables thorough testing of the device as well as device control logic. In contrast, the driver is developed in the OS environment running on top of the real hardware, which does not allow tight control over the testing process.

The result is a particularly bad case of useless duplication of effort: the verification code is correct, but is discarded after the device has been manufactured; the driver code is inferior, but used in actual device operation.

Our central claim is that the two tasks, and the software they produce, can and should be unified, and this will result in a drastic improvement of device-driver quality. To support this claim, we have developed a workflow in which the driver developer implements and tests the driver in the context of the device verification environment. The resulting driver is then transferred without modifications to the OS environment. The workflow guarantees that code that works correctly in the verification environment results in a correct driver.

In the proposed workflow, driver testing is performed in parallel with hardware verification, months before the actual hardware is manufactured. As a result, driver testing is no longer in the critical path to product delivery, which enables shorter product development times and encourages more thorough testing.

We evaluate this workflow by applying it to develop and test drivers for 4 different I/O devices. The results are encouraging: we were able to find 9 driver defects, all of which would have been extremely hard to find using conventional driver testing techniques. Furthermore, we demonstrate that this approach allows improving the quality of hardware verification by finding two hardware design defects in one of the devices.

The rest of the paper is structured as follows. Section 2 gives an overview of the hardware verification workflow. We introduce the main ideas behind our approach to verification reuse in Section 3. Section 4 discusses potential advantages and limitations of this approach. Section 5 presents design and implementation details. We evaluate the proposed approach in Section 6, survey related work in Section 7 and draw conclusions in Section 8.

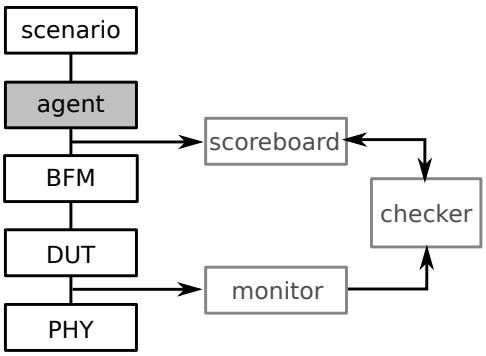

Figure 1. The RTL testbench architecture.

2. Hardware verification

This section provides an overview of the hardware verification process used in the majority of modern hardware designs.

In current industrial practice, design and testing of a device are tightly integrated. Starting from a formal or informal description of the desired device functionality, the hardware engineer develops a register transfer level (RTL) design of the device. As mistakes are expensive and time-consuming to fix once the design has been fabricated in silicon, much effort is put into verifying the design in a simulated environment prior to producing hardware. The software infrastructure for such verification, called the RTL *testbench*, is developed in parallel with the design of the device.

The testbench is designed to exercise the RTL design and validate its behaviour under a wide range of operating conditions. Most modern testbenches follow the layered architecture shown in Figure 1, which models the hardware and software structure of a real computer system. It is built around a simulated model of the device, called the *design under test* (DUT). The DUT is connected to a *bus functional model* (BFM), which simulates the I/O bus the device is connected to. It accepts bus-read and -write commands from higher layers and translates them into bus transactions at the device interface. The *agent* module consists of functions and associated state that implement high-level device transactions such as sending network packets, changing device configuration, handling interrupts, etc.

The *scenario* layer consists of test scenarios designed to thoroughly test the device in various modes. Finally, the bottom layer of the testbench simulates the physical medium that the device controls. For instance, if the DUT is an Ethernet Medium Access Control (MAC) controller, this layer simulates an Ethernet physical transceiver (PHY) chip.

In addition to the above modules that model components of a computer system, a complete testbench contains modules responsible for monitoring and validating the device operation. The *scoreboard* module keeps track of requests sent by the agent to the DUT and predicts the results of these requests. The *monitor* module records input and output signals at the physical interface of the device and groups them into high-level transactions. The *checker* compares transactions observed by the monitor against predicted ones recorded in the scoreboard. Finally, coverage points (not shown in the figure) are used to measure the progress of the testbench in fulfilling the verification plan requirements.

The testbench is usually designed to operate in the directed and randomised modes. In the directed mode, the testbench validates device responses to pre-defined sequences of input stimuli. In the randomised mode, the testbench generates random sequences of stimuli subject to a set of constraints. Randomisation applies to the ordering, timing, and content of messages sent to the DUT via

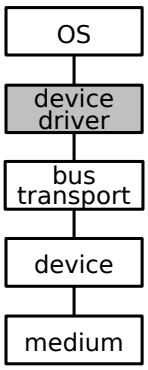

Figure 2. The operating system I/O stack architecture.

the bus and the PHY interfaces. Since the testbench has complete control over device interfaces, the test coverage is only limited by the duration of the testing run.

Constructing such a testbench involves substantial engineering effort. It is not uncommon for an RTL testbench to be larger and more complicated than the design that it is intended to test. This complexity can be somewhat reduced by using domain-specific languages such as SystemC and SystemVerilog that provide support for common tasks arising in testbench design, such as generation of constrained-random stimuli and interfacing with RTL. In order to further reduce the effort involved in testbench development, electronic design automation (EDA) tool vendors provide verification class libraries that facilitate the construction of layered designs similar to the one shown in Figure 1. Examples of such libraries include the Verification Methodology Manual (VMM) [2] library from Synopsys and the Open Verification Methodology (OVM) [17] library from Mentor and Cadence.

3. A co-verification approach to driver reliability

This section presents our proposed approach to driver quality assurance. In particular, we argue that the hardware verification ecosystem described in the previous section can be reused for driver development and testing.

We observe that the agent component of the testbench (Figure 1) provides similar functionality to a device driver and could in principle use the same implementation.

The agent accepts high-level I/O and configuration requests from the scenario layer and turns these requests into sequences of device-register read and write operations. Examples of requests handled by the agent include sending a network packet, performing a USB bus transaction, or writing a block of data to the disk. The agent also receives interrupt notifications from the DUT, reads device status registers, and informs the scenario layer about the completion status of requested operations.

For comparison, consider the operating system I/O stack architecture shown in Figure 2. It consists of the hardware device connected to the physical medium, the I/O bus transport comprised of the physical I/O bus and the OS bus framework, and the device driver providing services to the rest of the OS.

Similarly to the agent component of the testbench, the driver converts high-level requests from the OS into low-level interactions with the device over the bus transport. Yet, the agent is developed in a more supportive environment than the driver. Firstly, verification engineers have complete access to all device specifications and design internals. As such, they are in a good position to implement device control logic correctly. In contrast, driver developers only have access to the (frequently incorrect) datasheet.

Secondly, the device-control logic can be tested in the context of the RTL testbench more exhaustively than in the context of a driver. The testbench is specifically designed to expose various corner cases in the device behaviour. It has complete control over all components of the simulated environment and can model unusual situations like bus contention, network collisions, etc. As a side effect, such testing also exposes defects in other testbench components, including the agent, since such defects are likely to cause failures in verification scenarios.

Given the conceptual similarity between testbench and driver code, it seems promising to reuse the former for developing the latter. However, achieving this in practice faces significant challenges.

While the agent and the device driver serve a similar purpose in their respective environments, the exact interface they implement can be different. Adapting the agent implementation to the driver interface enforced by a particular OS may involve significant refactoring, e.g., splitting one operation into several or adding calls to OS-specific services. In addition, such adaptation requires writing device-specific glue code to translate OS requests into calls to agent functions. The need for these time-consuming and error-prone steps compromises the purpose of code reuse.

Instead of trying to adapt existing testbench code to work in the OS environment, we propose unifying the device driver interface across the OS and the testbench. A correct implementation of this interface is guaranteed to work correctly in both environments. We call such a reusable implementation an *environment-independent device driver*.

One way to achieve driver interface unification is to simulate the existing device driver API defined by the OS in the testbench environment. This way, a driver developed and tested in the context of a testbench can be reused directly in the OS kernel without the need for any modifications or wrapper code. Moreover, it is also possible to incorporate an existing OS driver in the testbench environment. This is useful for testing drivers developed using the conventional methodology.

The downside of this approach is that the resulting driver is only compatible with one OS. This is often acceptable, in particular for embedded devices that are designed for use with a specific system.

An alternative approach that allows producing reliable drivers for multiple OSes is to define a single, unified driver interface to be supported by the testbench as well as by each target OS. The resulting testbench and OS architectures are shown in Figure 3. In both environments, the device is managed by the environment-independent device driver, which implements two standardised interfaces: the *generic device-class interface* shared by all similar devices (e.g. all Ethernet controllers or all SCSI adapters) and the *generic bus interface* that provides generic methods for access to a specific bus type (PCI, USB, etc). The environment-independent driver is developed and tested in the context of the testbench and is reused without modifications inside the OS.

In the testbench environment, the driver interacts directly with the scenario layer and the BFM, which are designed to be compatible with the generic device-class and bus interfaces. In the OS environment, additional wrappers are required to translate between OS-specific driver interfaces and the corresponding generic interfaces. These wrappers only need to be implemented once for each OS and each generic interface. This way the environment-independent driver can be reused not only between the testbench and the OS environment but also across different OSes.

This architecture enables driver verification reuse across multiple operating systems at the cost of having to provide interface wrappers for each supported OS. In addition, it is not compatible with existing OS-specific device drivers.

In this paper we experiment with both OS-specific and OS-independent driver interfaces and demonstrate that both approaches

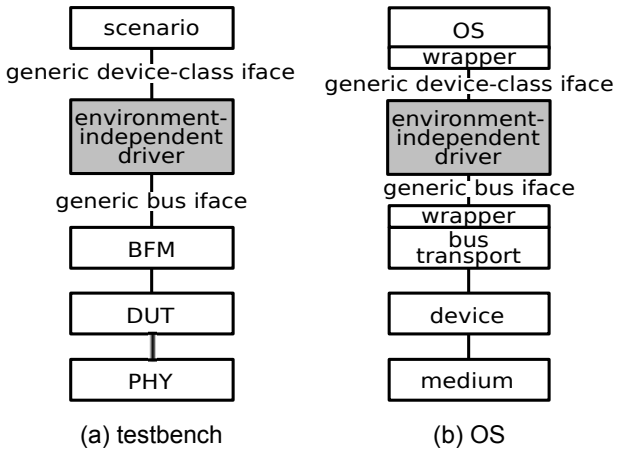

Figure 3. The testbench and OS architecture using a common implementation of the environment-independent device driver.

work in practice. The choice of the optimal solution is left to the user and depends on whether they require support for multiple OSes as well as how much effort they are prepared to invest in the driver verification infrastructure.

Interface compatibility is not sufficient to ensure that a driver developed in the testbench environment will work correctly in the OS environment. In particular, some of the implicit assumptions about the environment incorporated in the driver code may not hold inside the OS. The OS may perform various operations in a different order than the testbench, or issue concurrent calls to operations that the testbench executes sequentially. As a result, the driver based on this code may contain defects even though the code was thoroughly tested in the testbench environment.

In order to avoid such defects, we associate a detailed behavioural *contract* with the device-class interface. The contract defines constraints on the ordering of driver requests and responses as well as operations that the device must complete to satisfy each request. A correct driver must handle any sequence of requests permitted by the contract. Any valid environment must use the driver in a way that satisfies all constraints of the contract.

In case the driver implements an OS-specific interface, the contract must capture constraints imposed by the OS on driver behaviour. Such a contract can be derived from OS documentation and source code. Alternatively, if an OS-independent interface is used, the contract must be defined together with the interface. Section 5.2 discusses contracts in more detail.

In order to test the environment-independent driver against constraints of the contract, we implement a *contract conformance tester* as an extension of the testbench environment. The tester consists of two parts. The first part is a testing scenario that randomly generates valid sequences of requests to the driver. The second part consists of monitoring and checking components that keep track of outstanding requests, driver responses, and operations performed by the device and validate them against constraints of the contract.

In summary, our proposed driver development and testing workflow consists of the following steps.

1. The driver developer implements an environment-independent driver for the device in question. The driver must comply with the appropriate device-class and bus interface specifications. These interfaces can be OS-specific or OS-independent.
2. The verification engineer constructs a co-verification environment for the driver. To this end they either modify an existing device testbench or, if one does not exist, build a new testbench for use as a co-verification environment from the ground up. Refactoring an existing testbench that has not been designed for co-verification involves replacing the agent component of the testbench with the environment-independent driver. It also requires modifying the scenario layer and the BFM to support appropriate driver interfaces. The rest of the testbench infrastructure, including PHY, DUT, monitor, checker, and scoreboard modules, are reused with minimal or no changes.
3. The verification engineer extends the resulting testbench with the contract conformance tester, which provides additional testing scenarios to check that the driver correctly implements the device-class contract. The contract conformance tester needs to be implemented once for a class of devices.
4. Once the driver has been tested in the co-verification environment, it undergoes final testing in the target OS environment.

Note that this workflow does not attempt to shift the task of driver development from system programmers to hardware verification engineers, but rather enables them to work together in order to improve the quality of both the driver and the device.

4. Analysis

The hardware/software co-verification technique introduced in the previous section has the potential to significantly improve driver reliability by performing driver testing earlier in the product life cycle and by achieving better test coverage. On the other hand, the need to change current hardware verification practices may impede the practical application of this technique. In this section we analyse potential advantages and limitations of the proposed approach.

4.1 Improved driver test coverage

We start with analysing how hardware/software co-verification improves the likelihood of detecting various types of driver defects. To this end, we adopt the taxonomy of driver defects developed in our earlier work [21]. We distinguish four categories of defects:

1. *Device protocol violations* occur when the driver incorrectly handles the device interface by issuing an invalid sequence of requests to the device or incorrectly interpreting data received from the device.
2. *OS protocol violations* occur when the driver violates the ordering, content, or timing of interactions with the OS.
3. *Concurrency defects* occur when a driver incorrectly synchronises multiple threads of control executing within it, causing a race condition or a deadlock.
4. *Generic programming faults* include common coding errors, such as memory allocation errors and typos.

Device protocol violations account for about 40% of all driver defects, with the remaining defects distributed evenly across the other three groups [21].

4.1.1 Device protocol violations

As discussed in Section 1, the conventional driver testing environment has insufficient control and visibility of the device behaviour, which limits its ability to detect situations where the driver incorrectly uses the device interface.

In contrast, the device testbench built around the simulated model of the device can control device inputs and observe its outputs and its internal state at the clock-cycle granularity. In order to exhaustively test the device under a wide range of operating conditions, the testbench is equipped with stimuli generators that

feed various patterns of input signals to the device, monitors that interpret device outputs, and checkers that validate these outputs.

Importantly, testing the driver requires the same generators and monitors as testing the device. This allows leveraging the significant effort invested in the hardware-verification infrastructure for driver testing. For example, the Ethernet PHY model used in testing the Ethernet MAC controller device (Figure 1) should be able to simulate network traffic to the device with a broad range of parameters. On the one hand, this is necessary for testing the controller operation under various network traffic conditions. At the same time, this also allows thoroughly testing how the driver handles data exchange with the device.

The fine-grained visibility of the device behaviour enables end-to-end validation of driver-device interactions. This validation is carried out by the contract conformance tester, which checks that every I/O request issued to the driver is successfully completed by the device. For example, it monitors the network interface of the device to make sure that packets sent to the driver appear at the network interface in the right order and that none of the packets are dropped or delayed. Note that such validation cannot be achieved during conventional driver testing, where the test harness can only observe driver's responses to I/O requests but not the matching device behaviours.

4.1.2 OS protocol violations

Device-class contracts are also the main means of testing OS protocol compliance of a device driver. The contract conformance tester is designed to randomly simulate all possible sequences of requests that the driver can get from the OS and to validate driver responses against contract requirements.

4.1.3 Concurrency defects and generic programming errors

The co-verification methodology that we present here does not provide special means to detect generic programming errors and concurrency defects. However, generic programming errors, such as bugs in bit-level arithmetic, often lead to device or OS protocol violations, in which case they can be detected using mechanisms described above.

In contrast, concurrency defects are unlikely to be detected in the co-verification environment. In order to achieve deterministic execution, device testbenches are usually designed around the co-operative threading model. As a result, most thread interleavings that may cause race conditions in the OS kernel environment do not occur in the testbench environment.

Concurrency defects and generic programming errors can be mitigated using complementary techniques, including static analysis [12] and software fault isolation [26].

4.1.4 Hardware defects

While the co-verification approach is primarily intended to detect device-driver defects, it also has the potential to improve the quality of hardware testing.

Most device testbenches derive their testing scenarios from the specification of the device being tested. These scenarios represent the hardware designer's idea of how the device is going to be used, rather than the actual usage patterns that occur in a real system.

Contract-based testing closes this gap. By testing the device driver for adherence to the contract, we simultaneously test the device under a wide range of scenarios that model how the device will be used in a real OS. This helps uncover hardware-design defects missed by other testing scenarios before the device is implemented in silicon.

4.2 Impact on the testbench architecture

The proposed co-verification methodology relies on hardware verification engineers to incorporate environment-independent device drivers and device-class contracts in their testbenches. The need to change well-established verification practices may complicate the industrial adoption of this methodology.

For the approach to be practical, such changes must be kept to a minimum. In particular, modifications required to the existing verification infrastructure to make it compatible with environment-independent driver interfaces only affect the BFM and the scenario layer, which comprise a small fraction of the testbench code. We argue that such modifications are not only acceptable, but that they improve the testbench architecture: interface unification helps avoid reinventing the same interface for every device and facilitates code reuse across testbenches.

4.3 Impact on driver development and maintenance

In current OSes, driver development follows one of two dominating models. In the first model the driver is created and maintained by the hardware-device vendor. This approach is standard for Windows drivers. The co-verification methodology fits well into this model, as both the initial version of the driver and all subsequent releases can be tested by the hardware vendor in the co-verification environment.

In the second model, used for the majority of Linux drivers, driver maintenance is the responsibility of OS developers. The initial version of the driver is often provided by the hardware vendor. However, subsequent support, including bug fixing and adaptation to kernel API changes is performed by one of the kernel developers. In this model the driver maintainer does not have access to the device testbench; hence co-verification can only be performed by the device vendor on the initial version of the driver.

In the worst-case scenario, the device vendor does not provide even the initial driver implementation. The driver is written by OS developers based on the device datasheet or by reverse engineering an existing driver for another OS. In this scenario, the driver developer does not have access to the device RTL and its testbench; hence co-verification is not applicable to such drivers.

The use of OS-independent driver interfaces allows improving this process. Instead of implementing drivers for multiple OSes, the device vendor can publish the co-verified OS-independent implementation of the driver. OS developers only need to build and maintain interface wrappers for relevant interfaces.

4.4 Simulation speed

The quality of testing is related to its duration: longer test runs detect more defects and result in more reliable drivers. One limitation of testing in the simulated testbench environment is that simulated devices typically run three to four orders of magnitude slower than real hardware. As a result, fewer tests can be run in the given time frame.

Low simulation speed is compensated for by higher testing precision: while conventional driver testing ends up hitting the same common-case execution paths most of the time, the testbench uses its fine-grained control over device interfaces to drive the device into various corner-case situations. To this end, it relies on randomisation and careful choice of testing scenarios.

The problem can be further mitigated using techniques for improving simulation speed. These include FPGA-based testing [3] and replacing low-level RTL device models with more abstract models that simulate faster [27]. Finally, testing in the co-verification environment can be complemented by faster, but less accurate, conventional OS-based testing.

4.5 Unified driver-OS interface

Generic device-class and bus interfaces enable the reuse of verification results across multiple OSes: a driver tested in the testbench environment should work correctly in the context of any OS that provides wrappers for the appropriate generic interfaces.

Unfortunately, previous attempts to introduce a unified driver interface [20] were not successful, because major OS vendors were reluctant to give up the competitive advantage of having better hardware support than less popular systems. In addition, open-source systems like Linux continuously evolve their driver interfaces, so that drivers are not even portable across Linux kernel releases. Cross-OS interface unification looks problematic in these settings. Therefore, in the short term the use of OS-specific interfaces in co-verification is likely to be a more practical approach.

4.6 Handling non-standard device features

The co-verification methodology described so far assumes that all devices of the same type provide equivalent functionality and can be accessed via a common interface. This allows the reuse of the entire co-verification infrastructure, including the contract conformance tester and OS wrappers, for all devices of the same class.

In practice, many I/O devices support non-standard features that are not covered by the existing device-class interface. Often these features distinguish the product among competitors; therefore reliable software support for them is important.

In order to support such non-standard devices, a custom version of the device-class interface and the associated contract conformance tester must be produced for the device. In addition, the OS wrappers also need to be modified to support the new behaviours. We expect such modifications to be strictly incremental in most cases, since non-standard devices are usually compatible with all features of the standard ones.

Thus, the reuse of the verification environment can be extended even to non-standard devices, although additional per-device effort is required in such cases.

Note that the problem is not unique to the co-verification methodology. Inherently, devices with non-standard features require special testing code. However, with our approach, existing device-class contract testers can be immediately reused to test the generic functionality of non-standard devices, since such devices must still support all the features present in simpler devices.

5. Design and implementation

5.1 Interface specifications

Before starting on the implemention of the driver and its co-verification environment, one must obtain specifications of the driver interfaces. In case OS-specific interfaces are used, their specifications can be extracted from OS header files. Alternatively, if the co-verification environment is constructed around generic OS-independent interfaces, these interfaces need to be defined first.

Ideally, generic driver interfaces should be standardised across the industry, which will facilitate the reuse of the associated testbench components. In practice, however, different vendors are likely to define their own generic interfaces and develop the entire co-verification infrastructure in-house. In either case, these interfaces must be designed by system programmers, who have in-depth understanding of how the OS interacts with hardware.

Generic interfaces resemble analogous driver interfaces defined by an OS. Unlike OS-specific interfaces, however, generic driver interfaces must allow implementation in any OS, i.e., it should be possible to build an efficient wrapper to translate between the OS-specific and the generic interface.

```
1  virtual class ethernet_mac;
2      virtual task mac_enable (...);
3      virtual task mac_disable (...);
4      virtual task tx_queue_packet (...);
5      ...
6  endclass: ethernet_mac
7
8  virtual class ethernet_mac_cb;
9      virtual task tx_packet_sent (...);
10     ...
11 endclass: ethernet_mac_cb
```

Figure 4. A fragment of the Ethernet controller device-class interface specification in SystemVerilog.

We start defining a generic device-class interface by identifying the set of I/O operations that this particular device class must support. These include data transfer requests (e.g., sending and receiving packets), configuration requests, power management requests, etc. We analyse how each of these operations is implemented in several existing OSes in order to come up with a design that is efficient, e.g., avoids unnecessary data copying and blocking, and that can be mapped onto existing OS interfaces. Based on this analysis, we determine whether the given operation should be implemented as a single, potentially blocking, method or as a request-completion chain and define the list of its arguments and their types.

A generic interface specification can be written in either a system programming language, e.g., C or C++, or in a hardware-verification language, e.g., SystemVerilog or SystemC. It can then be automatically translated into any other supported language. To this end, language subsets that allow mapping to other languages must be used.

Figure 4 shows a fragment of the generic interface specification for the Ethernet MAC controller device class, written in SystemVerilog. This interface must be implemented by any environment-independent Ethernet device driver. The first part of the specification (lines 1 through 6) lists methods that an Ethernet driver must provide to the OS, including methods to enable and disable the controller, add a new packet to the device transmit queue, etc. The second part (lines 8–11) defines the callback interface that the OS must provide to the driver. The driver uses this interface to report device status changes and I/O request completions to the OS. For instance, the `tx_packet_sent` callback notifies the OS about successful transmission of a packet.

In addition to generic device-class interfaces, one must also specify generic bus interfaces (see Figure 3) for various types of I/O buses as well as generic interfaces to common OS services such as timers, synchronisation, and DMA buffer management.

In particular, design of the DMA buffer management interface raises some interesting issues due to the different ways in which the testbench and the OS manage devices' access to physical memory. In the OS environment, access to memory is mediated by address-translation hardware, including the MMU and the IOMMU. In contrast, in the testbench environment the driver and the device both have direct access to a simulated RAM.

In order to abstract away the differences between the two models, we designed a generic DMA buffer management API that allows efficient implementation in the testbench environment as well as in any OS kernel. The API exports two simple abstractions: the *iospace* abstraction that represents the device's view of the physical RAM, and the *iobuffer* abstraction that represents a data buffer. All mapping, translation, and coherency issues are handled transparently by the internal implementation of the API.

5.2 Device-class contracts

A device-class contract extends the device testbench with a specification of how the OS interacts with the device. It defines constraints on the ordering of requests and responses exchanged by the driver and the OS. It also defines the semantics of each request in terms of its effect on the device behaviour.

The following is a simplified example of a rule defined by the Ethernet controller device-class contract:

1. When the controller is enabled, the OS can call the `tx_queue_packet` method of the driver to transmit a network packet.
2. The driver must transmit packets received from the OS over the network in FIFO order.
3. Once a packet has been transmitted, the driver must notify the OS via the `tx_packet_sent` callback.

The second clause of the rule is the most interesting one, as it relates OS requests to the expected device behaviour. By enforcing this rule at run time, the co-verification environment makes sure that the driver correctly completes all transmit requests. This is in contrast to conventional driver testing where the test program cannot directly observe whether the device has really performed the requested operation.

A device class contract must be sufficiently restrictive to rule out any illegal behaviours, yet sufficiently liberal to allow a range of possible valid behaviours. For example, the network controller is not allowed to drop an incoming packet when its receive buffer is empty and the packet is received without an error. A failure to deliver such a packet to the OS indicates a defect in either the device or the driver. On the other hand, the controller may or may not drop the packet if the inter-packet gap is too short, the receive buffer is full, or the packet is longer than the currently configured threshold. To complicate things further, if several such packets arrive in a row, the controller can accept any subset of them.

Error handling is another source of complexity in contract specifications. The driver must correctly report the status of completed I/O requests to the OS. For example, if hardware CRC checking is enabled and the device receives a packet with invalid CRC field, the driver must report a CRC error.

Finally, a contract must capture various interleavings of hardware and software events. For example, when the network controller is disabled, it must drop all incoming packets. After the OS enables the controller, it must deliver incoming packet to the OS. However, depending on the implementation of the driver and the device, they can keep receiving packets while the disable or enable request in in progress.

At the moment, device-class contracts are written as informal documents. Based on this document, the verification engineer implements a contract conformance tester that is integrated into the device testbench and checks the driver's contract compliance. The tester consists of three modules shown in Figure 5. The *contract scenario* module simulates the most general contract-compliant environment that the driver can be placed into by randomly issuing requests to the driver according to the constraints of the contract. It is designed in such a way that any valid sequence of requests will be generated eventually with probability one.

The *contract scoreboard* module keeps track of outstanding requests issued to the driver and the driver's responses to these requests. It operates at a higher level of abstraction than conventional scoreboards used in hardware verification (see Figure 1). While the hardware scoreboard records individual bus transactions, the contract scoreboard records high-level I/O requests to the driver.

Finally, the *contract checker* module observes the device behaviour and validates it against requests recorded in the scoreboard.

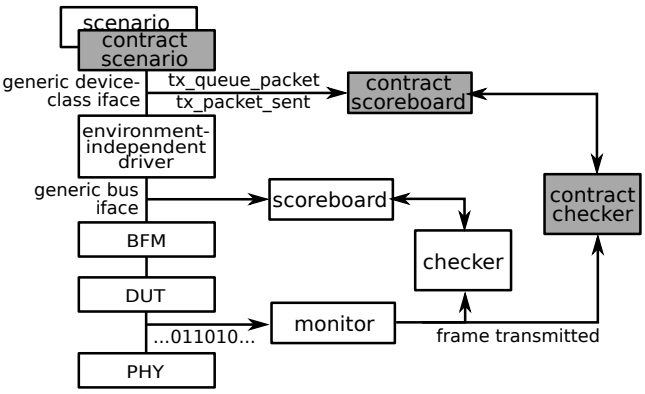

Figure 5. Ethernet controller co-verification environment with the contract conformance tester.

As a concrete example, consider how the rule cited above is enforced by the Ethernet device-class contract conformance tester. The contract scenario module implements the first clause of the rule by randomly generating transmit requests to the driver (along with other types of requests) whenever the controller is enabled. The contract scoreboard records each packet sent to the driver in its internal queue. The contract checker receives notifications about transactions that occur at the PHY interface of the device from the monitor module. In particular, it is notified whenever the device transmits a complete Ethernet frame on the wire. It checks that packets are transferred in FIFO order by comparing the payload of this frame with the packet at the head of the internal queue maintained in the scoreboard (clause 2). The checker also gets notified by the contract scoreboard when the driver invokes the `tx_packet_sent` callback and verifies that this only happens *after* the successful transmission of the packet (clause 3). Finally, it makes sure that the driver does not lose packets by checking that every packet recorded in the scoreboard is sent on the network and acknowledged by the driver within a certain time period.

The contract scoreboard and checker modules are incorporated in the device testbench alongside the conventional scoreboard and checker used in device verification (Figure 5). Likewise, the contract scenario module is used along with potentially many other scenarios developed by verification engineers to test the device functionality.

Implementing a contract conformance tester is a complex task; therefore it is desirable to reuse the same implementation with multiple testbenches for different devices of the same type. To this end, all interfaces between the tester modules and the rest of the testbench must be standardised. Inputs to the contract scoreboard module are already standardised as part of the the device-class interface specification. The only remaining interface that needs to be defined is the one between the monitor and the contract checker. For Ethernet devices, for example, this interface consists of methods that notify the checker about frames sent and received by the controller, network collisions, aborted transfers, etc.

5.3 Environment-independent drivers

The task of implementing an environment-independent device driver is similar to that of writing a normal driver. One complication involved in sharing the driver implementation between the testbench and the OS environment is that testbenches are written in hardware verification languages such as SystemC and SystemVerilog, whereas OS kernels can only host code written in C or C++. One possible solution is to implement the environment-independent driver in C or C++ and incorporate it into a SystemVer-

ilog or SystemC testbench using inter-language bindings. Alternatively, the driver can be written in the same language as the rest of the testbench and automatically translated into C. We expect the former approach to be the more common one, as C and C++ are currently languages of choice for driver developers. If the latter approach is taken, the use of the verification language must be restricted to a subset that allows automatic translation to C. In case of SystemVerilog, this mainly implies avoiding features that require automatic garbage collection.

Detailed investigation of automatic SystemVerilog-to-C translation is beyond the scope of this research. Only one of the case studies presented in Section 6 uses a SystemVerilog implementation of the driver. For this driver, the translation to C was performed manually.

5.4 OS wrappers

OS wrappers implement the translation between OS-specific and generic driver interfaces. This translation must be implemented efficiently to avoid performance degradation. In particular, operations that involve data copying and synchronisation should be avoided in the performance-critical data path.

Avoiding copying means that data buffers passed by the OS to the driver should be converted from the OS-specific representation into the generic iobuffer representation in place, i.e., without copying their data payload. To this end, we provide several implementations of the iobuffer interface on top of different data buffer primitives supported by the OS. For example, Linux has separate primitives for simple buffers, scatter-gather buffers, and network buffers; hence three different iobuffer implementations are required.

6. Evaluation

In this section we evaluate the proposed co-verification methodology with respect to the following criteria:

- *Feasibility.* We demonstrate that this methodology enables the creation of device drivers that can be used without modifications in both the OS and the testbench environment. Furthermore, implementing a device driver using this methodology is comparable in terms of required time and effort to conventional driver development.
- *Reliability.* We show that hardware/software co-verification helps improve driver reliability by detecting driver defects that are extemely hard to find during conventional driver testing.
- *Performance.* We provide evidence that the improved driver reliability does not come at the cost of performance degradation.

6.1 Case studies

Our evaluation was performed as a series of case studies where we implemented and tested drivers for four different devices. The choice of devices was limited to open-source designs for which RTL specifications are publicly available, and a limited selection of designs used inside Intel.

In one of the case studies we built a co-verification environment around an existing Linux driver. In the three other case studies we defined OS-independent driver interfaces for respective device classes and implemented drivers and their co-verification environments around these interfaces.

Three of the case studies were accomplished by refactoring existing device testbenches, whereas in the fourth case study we implemented a testbench from scratch.

6.1.1 Case study 1: Ethernet controller

In the first case study we implemented and tested a driver for the 100Mb/s Ethernet controller device from the OpenCores project [18]. Our co-verification environment is based on an existing open-source testbench for this device developed using the Synopsys VMM methodology [2].

This is the only case study where the environment-independent driver was implemented in SystemVerilog and then manually translated into C. The translation was performed in a mechanical way that mimicked the work of an automatic translator. In the other case studies drivers were implemented in C from the beginning and integrated in the verification environment using SystemVerilog-to-C bindings.

6.1.2 Case study 2: High-speed UART

The device used in this case study is a high-speed Intel UART controller with an integrated DMA engine.

Our co-verification environment for this device is based on the existing proprietary testbench, developed using the OVM methodology [17].

6.1.3 Case study 3: USB host controller

The third case study is based on an existing Linux driver for the OpenCores USB 1.1 host controller device [19]. We implemented the co-verification environment for this device on top of the existing custom testbench that does not rely on any standard verification methodology.

6.1.4 Case study 4: USB slave controller

Our final case study is based on an Intel USB slave controller device . Since we did not have access to an existing device testbench, we implemented our own testbench from scratch.

6.1.5 Summary of case studies

Table 1 summarises the four case studies. For each case study it shows:

- The size of the native Linux driver for the given device.
- The size of the environment-independent driver used in co-verification.
- The size of the co-verification infrastructure, excluding the driver, the DUT, and the contract conformance tester (see Figure 5).
- The size of the contract conformance tester.

The last two columns represent the reusable part of the co-verification environment that can be applied to test multiple devices of the same type and their drivers. As can be seen from the table, this reusable part is an order of magnitude larger than the driver that is being tested. The only exception is the last case study, where we only implemented a minimal rudimentary testbench. Another observation that can be drawn from the table is that the size of environment-independent device drivers is on par with equivalent native Linux drivers.

6.2 Feasibility

We start with evaluating the feasibility of developing and testing device drivers using the co-verification methodology.

Our case studies did not produce any surprises here: writing an environment-independent driver was quite similar to writing a regular OS-based driver, and involved a comparable amount of labour.

Refactoring an existing testbench into a co-verification environment also proved a straightforward task. The main effort involved in such refactoring is implementing various OS services required by the driver, which took several days in our case studies. This is

#	Case study	Lines of code			
		Linux driver	Env-independent driver	Co-verification environment	Contract conformance tester
1	Ethernet controller	1,189	957	10,032	1,794
2	High-speed UART	1,554	2,077	18,418	5,622
3	USB host controller	1,147	1,052	7,909	1,892
4	USB slave controller	3,333	2,705	427	1,230

Table 1. Summary of case studies.

a one-off effort, since the resulting implementation can be reused when building a co-verifciation environment for a new device.

The next step is to introduce the contract conformance tester into the testbench in order to thoroughly test the driver functionality against OS requirements. Implementing this tester is the most complicated and time consuming step in the co-verification workflow. Modelling and checking the outcomes of I/O requests under various combinations of hardware and software inputs requires involved logic. This logic is particularly difficult to get right given that it must handle events from three concurrent sources: the driver, the DUT, and the PHY module. In addition, it must reflect the nondeterminism in the driver and the device behaviour where the same sequence of inputs can cause multiple valid outputs. The contract conformance tester must be carefully implemented to allow all legal outputs, while detecting any illegal ones.

The effort of developing a device-class contract conformance tester could hardly be justified it it had to be done afresh for each device. Fortunately, this work is amortised across many drivers of the same class that can be tested using the common tester.

The effectiveness of the co-verification methodology can be further improved by automating the task of writing contract conformance testers. As mentioned in Section 5.2, device-class contracts are currently written as informal documents. If contract constraints are specified formally, one can in principle generate a contract conformance tester automatically from this specification. Our ongoing work is focusing on languages and tools for automatic generation of contract conformance testers.

In our case studies, we implemented co-verification environments on top of VMM and OVM testbenches, which represent the two most popular hardware verification methodologies nowadays, as well as on top of a custom testbench. This gives us confidence that our approach to the reuse of the hardware verification infrastructure in driver testing applies to the majority of current device testbenches.

Another encouraging observation is that the testbench provides a better debugging environment for drivers than real hardware. Driver developers are only too familiar with the frustrating situation where a device simply refuses to behave as expected, a situation which frequently requires a trial-and-error approach to resolve. It is a consequence of the inability to inspect the internal state of the device. In the testbench environment, the programmer has complete access to the source code and the runtime state of the simulated device, which helps tremendously in diagnosing difficult problems.

Driver debugging is further simplified due to the fact that the testbench executes as a user-level program. This enables the use of program debugging and analysis tools that are not available in the kernel environment.

6.3 Reliability

For evaluating the impact of co-verification on driver reliability, we set out to answer the following questions. Firstly, what types of defects that are hard to find during conventional driver testing can be detected using co-verification? Secondly, what types of defects does co-verification fail to detect?

In order to answer the first question, we invert the normal co-verification workflow: we start with developing and testing the driver in the OS environment before moving to the co-verification environment. Additional defects found during co-verification are analysed to determine why they escaped detection during the initial testing. In some cases, the analysis showed that the defect could have potentially been discovered in the OS environment by running additional tests. In other cases, even extensive additional OS-based testing would have been unlikely to find the defect. These defects represent the classes of defects that can be eliminated or substantially reduced using co-verification.

We followed this approach in case studies 2 and 4. Case study 3 takes a similar approach, but instead of implementing and testing our own Linux driver we used an existing Linux driver for the device that had been used in several working designs. We ported this driver to the co-verification environment with minimal changes.

Finally, case study 1 implemented the normal co-verification workflow, i.e., it started with developing and testing the driver in the co-verification environment before moving to the kernel environment. As we discovered defects in our implementation of the driver, we compared them against the existing Linux driver for the same device. This way we found a potentially dangerous runaway DMA defect in the Linux driver.

Case study 1 also revealed two hardware-design defects in the Ethernet controller RTL. These defects were triggered in situations which occur in a real OS, but which the original device testbench failed to model, such as disabling the controller while there is a packet transfer in progress. As a result, they had not been detected during conventional hardware verification. This is a common problem in hardware verification: verification engineers do not have a good understanding how the device under test must interact with the OS and as a result fail to implement relevant test cases. These defects also had not come up during conventional driver testing, because their effect was masked by TCP/IP error recovery mechanisms.

Table 2 summarises software and hardware defects that were discovered during our case studies and that would have been difficult to detect using conventional testing techniques. These include device and OS protocol violations, generic programming defects, and hardware design errors.

These defects can be further partitioned into three groups. The first group includes defects that depend on system parameters that cannot be controlled by the testing software, such as I/O bus speed or BIOS-defined device configuration (defects #4, 6). The second group contains defects that escape detection during testing because their effect cannot be directly observed by the test program (defects #1, 2, 3, 5, 9, 10). The third group is comprised of defects triggered by software request sequences that are difficult to generate in a directed way in the OS environment (defects #7, 8, 11).

As discussed in Section 1, the fundamental problem that prevents these defects from being detected during conventional OS-based testing is that the OS lacks means to directly control and observe the device behaviour. In contrast, the co-verification environment can generate arbitrary input signals to the device and observe

#	Description	Type	Comments
	Case study 1: Ethernet		
1	Runaway DMA: the device continued accessing DMA buffers after they have been deallocated by the driver. The defect was caused by an undocumented device behaviour where the device, when disabled, did not immediately stop its transmit and receive engines.	dev	The defect was identified by the contract checker when it received a packet transmission notification after the driver had reported that the transmitter had been disabled. It was not found during conventional testing, since runaway DMA cannot be detected by the OS until it causes a memory corruption and even then tracing down the root cause of the problem is difficult.
2	Hardware race condition: disabling the device and later re-enabling it causes the hardware to skip the first DMA descriptor in the transmit queue.	h/w	These hardware design defects are triggered in situations which occur in a real OS, but which the original device testbench failed to model, such as disabling the controller while there is a packet transfer in progress. As a result, they were not detected during conventional hardware verification.
3	Hardware race condition: disabling only the receiver part of the device caused corruption of the next transmitted packets.	h/w	
	Case study 2: High-speed UART		
4	The HS UART implements an extended register set that is only available in the memory-mapped device configuration. The driver, however, attempted to access these extended registers in both the memory-mapped and the I/O-space mapped mode.	dev	The driver was tested on a platform that configured the device in the memory-mapped mode. In contrast, the co-verification environment simulated both modes, which allowed detection of the error.
5	The driver did not disable software flow control interrupts when running with hardware flow control enabled. This did not affect its functional correctness, but resulted in an increased CPU load due to excessive flow control interrupts generated by the device.	dev	The defect did not cause any functional errors and therefore could not be easily detected during OS-based testing. In the co-verification environment, the UART device-class contract defines constraints on UART interrupt generation, which were violated when the defect was triggered during co-verification.
	Case study 3: USB host controller		
6	The driver is required to wait for a few microseconds after issuing a write to the device reset register and before reading or writing any other device registers. The Linux driver was missing this delay. As a result, the device failed to initialise correctly.	dev	This defect did not come up during conventional driver testing due to the slow I/O bus on the development board, which introduced sufficient delay between register accesses for the device to complete the reset. It would, however, prevent the device from working correctly in a system with a faster bus.
7	A typo in bit-vector arithmetic prevented the driver from correctly setting the USB hub port speed during port power-up.	gen	This defect was not detected while testing the driver in Linux, because Linux usually resets the hub port immediately after powering it up. The reset routine set the port speed correctly, thus undoing the effect of the power-up routine. The defect was discovered in the co-verification environment by the contract checker, which checks the connection status after port power-up.
8	The driver incorrectly assumed that the OS never submits more than one request to a USB control endpoint at a time. When such a situation occurred, the driver's USB transfer abort function aborted the wrong transfer.	os	Multiple outstanding control transfers are uncommon in USB devices, which is why this defect was not discovered during driver testing. Our USB host-controller contract scenario simulates such behaviour and therefore was able to detect this error.
	Case study 4: USB slave		
9	Upon a USB disconnect, the driver generated numerous bogus disconnect, suspend, and resume notifications for 300ms. The problem was caused by the USB suspend signal in the device status register, which started floating after a disconnect, causing fake interrupt notifications.	dev	The defect was not detected during conventional testing, because the OS is unable to distinguish a bogus disconnect notification from a real one.
10	When aborting a partially transferred packet, the driver returned incorrect count of transferred bytes. The problem was caused by the driver not properly updating the transfer length field of the packet in the abort path.	gen	This defect could not be detected during conventional testing, because the OS does not know the actual number of bytes transferred and is therefore unable to validate the value returned by the driver.
11	The driver violated the USB endpoint abort protocol expected by the OS by failing to mark the endpoint as stopped before aborting the first transfer associated with it. As a result, the OS could immediately requeue the transfer, potentially entering an infinite loop.	os	The infinite loop scenario is possible, but rarely occurs in practice and was never encountered during conventional driver testing.

Table 2. Driver defects discovered using co-verification that would have been difficult to detect using conventional testing techniques. The third column describes the type of defect: *dev* – device protocol violation, *os* – OS protocol violation, *gen* – generic programming error, *h/w*–hardware design error.

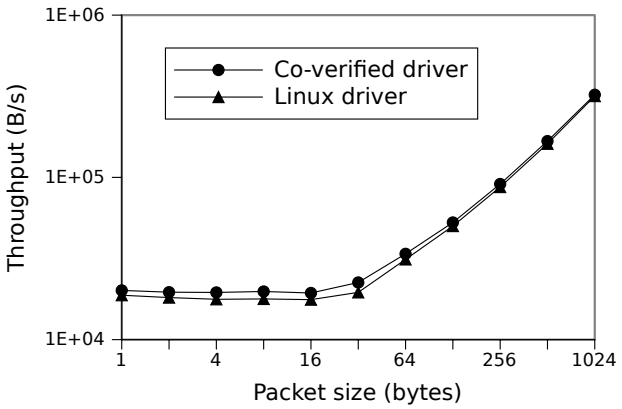

Figure 6. OpenCores Ethernet controller UDP throughput benchmark results.

device output signals at the wire level, which makes the detection of this kind of defects straightforward.

To answer the second question about faults missed by co-verification, we place the Ethernet controller driver from case study 1, which has been tested in the co-verification environment, into the Linux kernel environment and keep testing it there. As a result, we uncovered two additional defects that were missed in the co-verification environment.

The first defect was a race condition between the packet transmission function and the interrupt handler. As mentioned in Section 4.1.3, our methodology is not effective against concurrency errors; therefore the discovery of this kind of defect in a co-verified driver did not come as a surprise.

The second defect was caused by the driver not initialising one of its state variables to 0, which prevented it from correctly sending network packets. The defect could have been detected during co-verification if our environment had randomised the content of dynamically allocated memory.

Overall, these results confirm our analysis presented in Section 4: the co-verification methodology is effective in dealing with most types of driver defects, including defects that are difficult to detect using conventional testing techniques.

6.4 Performance

For our performance evaluation, we use the Ethernet controller driver from case study 1. The controller hardware runs on a 25MHz FPGA board with an OpenRISC CPU. On such a slow platform, driver performance can have critical impact on both I/O througput and CPU utilisation, especially under heavy network traffic.

We compare the performance of the driver developed using the co-verification methodology against the native Linux driver for this device. To this end we measure network throughput and CPU utitlisation when receiving streams of UDP packets of various sizes.

In all experiments, both drivers generated CPU utilisation of about 100%. Throughput results are shown in Figure 6. The two drivers achieved similar throughput for all packet sizes. In fact, the co-verified driver sustains up to 8% higher throughput than the Linux driver. Our analysis showed that the additional overhead in the Linux driver was caused by an extra data copy operation that it performed in the receive path.

These results indicate that our approach to driver development does not result in a noticeable performance overhead.

7. Related Work

Previous research on device driver reliability has mainly focused on detecting [1, 6, 7], isolating [8, 10, 13, 14, 24, 28, 29], and avoiding [21, 23] driver defects. Another approach, implemented in tools like Devil [15] and Termite [22] reduces the number of defects in drivers by generating a partial or complete implementation of a driver from a formal specification of the device interface.

All of these techniques, however, suffer from serious limitations. Existing static analysis tools are capable of detecting a limited subset of errors, such as common OS API rule violations [1] and certain memory allocation and synchronisation errors [7]. Stronger correctness properties, such as memory safety, race-freedom, and correct use of device interfaces, are currently beyond the reach of these tools. Runtime isolation architectures are capable of detecting broader classes of errors; however many of these systems incur intolerably high overhead. In addition, isolation doesn't remove the need for testing, as a bug in an isolated driver can still lead to a complete of partial system failure. Finally, automatic code generation tools are limited by the availability of correct device specifications.

Due to these limitations, driver testing remains the most important technique for ensuring driver correctness.

Recently, Kuznetsov et al. [11] proposed a symbolic execution technique for improving the test coverage of device drivers. Their approach allows detecting generic programming errors, concurrrency errors and, potentially, OS protocol violations; however it is not effective against device protocol violations, which is the most common type of driver defects. We believe that our techniques are complementary: by using symbolic execution rather than randomisation to steer drive testing in the co-verification environment, it is possible to achieve better precision in detecting device protocol violations.

In the hardware design world, virtual prototypes [27] are widely used for software and hardware testing. A virtual prototype is a fully functional model of a complete hardware platform, including the CPU and I/O devices, capable of running the complete system software stack. It enables testing of device drivers as part of the OS before the actual device hardware is available. Virtual prototypes are primarily intended for integration testing and do not provide mechanisms for fine-grained control and monitoring of the device behaviour, which are necessary for thorough testing of device drivers.

Bombieri et al. [4] propose a method for generating a device driver from an existing device testbench by converting the testbench to a state machine representation and manually identifying sequences of commands that correspond to driver operations. One problem with this approach is that the required manual step is error-prone. More importantly, as discussed in Section 3, testbench code that is not designed for reuse in the OS environment is unlikely to be reusable without substantial modification.

8. Conclusions

We argue that the lack of cooperation between hardware and software designers is a major source of driver reliability problems and propose an architectural framework to enable such cooperation. This framework allows most of the driver code to be developed and tested at the hardware verification stage, before the device is implemented in silicon.

The benefits are:

- significant improvement of driver reliability, since the driver is identical to the code used to debug the hardware, and is more thoroughly exercised than feasible in an OS environment;

- reduced development cost due to code reuse across the device driver and the device testbench;
- reduced time to market as driver development and testing proceed in parallel with device verification.

Acknowledgments

We would like to thank Julius Baxter for his help in getting the OpenCores development board up and running. We also thank Anton Burtsev, Thomas Sewell, Simon Winwood, and the anonymous reviewers for helpful comments.

NICTA is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

References

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st EuroSys Conference*, pages 73–85, Leuven, Belgium, Apr. 2006.

[2] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale. *Verification Methodology Manual for SystemVerilog*. Springer-Verlag, Inc., 2005.

[3] Bluespec, Inc. Emulation: enabling it on every desktop, 2008.

[4] N. Bombieri, F. Fummi, G. Pravadelli, and S. Vinco. Correct-by-construction generation of device drivers based on RTL testbenches. In *Proceedings of the 45th ACM/IEEE Conference on Design, Automation and Test in Europe*, pages 1500–1505, Apr. 2009.

[5] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, Lake Louise, Alta, Canada, Oct. 2001.

[6] A. Chou, B. Fulton, and S. Hallem. Linux kernel security report, 2005.

[7] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, San Diego, CA, Oct. 2000.

[8] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 75–88, Seattle, Washington, Nov. 2006.

[9] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *Proceedings of the 20th USENIX Large Installation System Administration Conference*, pages 101–111, Washington, DC, USA, 2006.

[10] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM Operating Systems Review*, 40(3):80–89, July 2006.

[11] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *Proceedings of the 2010 USENIX Annual Technical Conference*, Boston, MA, June 2010.

[12] S. K. Lahiri, S. Qadeer, and Z. Rakamari'c. Static and precise detection of concurrency errors in systems code using SMT solvers. In *Proceedings of the 21st International Conference on Computer Aided Verification*, pages 509–524, June 2009.

[13] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. R. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, Sept. 2005.

[14] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay. Two years of experience with a μ-kernel based OS. *ACM Operating Systems Review*, 25(2):51–62, Apr. 1991.

[15] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, San Diego, CA, USA, Oct. 2000.

[16] Microsoft Corporation. Network Driver Interface Specification Test. http://www.microsoft.com/whdc/DevTools/tools/NDIStest.mspx.

[17] OVM. OVM class reference. Version 2.1.1, Mar. 2010.

[18] Project OpenCores. 10/100 Mbps Ethernet MAC core. http://www.opencores.org/project,ethmac,.

[19] Project OpenCores. USBHostSlave IP core. http://www.opencores.org/project,usbhostslave,.

[20] Project UDI. UDI core specification. Version 1.01, Feb. 2001.

[21] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th EuroSys Conference*, Nuremberg, Germany, Apr. 2009.

[22] L. Ryzhyk, P. Chubb, I. Kuz, E. L. Sueur, and G. Heiser. Automatic device driver synthesis with Termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, Oct. 2009.

[23] L. Ryzhyk, Y. Zhu, and G. Heiser. The case for active device drivers. In *Proceedings of the 1st Asia-Pacific Workshop on Systems*, New Delhi, India, Aug. 2010.

[24] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing (Lake George), New York, USA, Oct. 2003.

[25] usbtest. USB testing on Linux. http://www.linux-usb.org/usbtest/.

[26] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, USA, Dec. 1993.

[27] M. Willems and F. Schirrmeister. Virtual prototypes for software-dominated communication system designs. *IEEE Communications Magazine*, 48:37–43, June 2010.

[28] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 241–254, San Diego, CA, USA, Dec. 2008.

[29] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 45–60, Seattle, WA, USA, Nov. 2006.