Improved Device Driver Reliability Through Verification Reuse

Leonid Ryzhyk[‡§] John Keys[¶] Balachandra Mirla[‡§] Arun Raghunath[¶]
Mona Vij[¶] Gernot Heiser[‡§]
[‡]*NICTA* [§]*University of New South Wales* [¶]*Intel Corporation*

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 discuss technical issues involved in achieving such unification and present our solutions to these issues. We report the results of a case study that applies this approach to implement a driver for an Ethernet controller device.

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.

Drivers are also important for another reason: they are the leading reliability hazard in modern OSs. Drivers are known to be responsible for the majority of OS failures [4], and have been shown to have 3–7 times the defect density of other OS code [3].

We have previously shown that the leading class of driver defects, comprising about 40 % of all driver bugs, are related to incorrect handling of the hardware-software interface [11]. These bugs include incorrect use of device registers, sending commands to the device in the wrong order, or incorrectly interpreting device responses.

Analysis of many open source drivers [11] suggests that these defects can often be attributed to a disconnect between the developers of the hardware (device) and software (driver). The interface between these developers consists of a document called the *device datasheet*, which describes device registers and behaviour from the software perspective. The datasheet is created by the hardware engineers, and typically represents the only description of the device available to the software engineers. In practice, datasheets are often incomplete and inaccurate, which results in numerous driver defects.

Another source of driver defects results from the difficulty of testing a driver as part of an OS. While the driver can issue arbitrary commands to the device, it only has partial control over its physical environment, including the I/O bus and the external medium that the device is connected to. This makes it difficult to achieve a good coverage of the state space of the driver. For example, it is difficult to test how the driver handles situations such as internal device FIFO overflow due to bus contention or packet transmission failure due to multiple network collisions. As a result, most drivers are only well tested in common scenarios and do not adequately handle various corner cases.

We argue that the underlying cause of these difficulties is the separation of two highly-related tasks: *device verification* (performed by the hardware verification engineer) and *driver development* (the job of the software engineer). As explained in more detail in the next section, 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 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 soft-

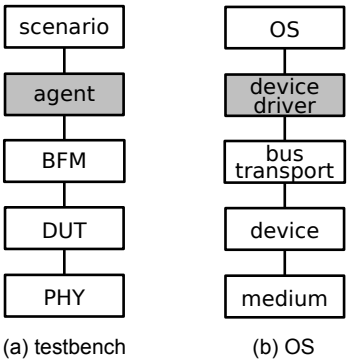

Figure 1: The conventional testbench and OS I/O stack architecture.

ware they produce, can and should be unified, and this will result in drastic improvement of device-driver quality. To support this claim, we have developed a workflow in which most of the driver code is developed and tested in the context of the device verification environment and is transferred without modifications to the OS environment. The workflow guarantees that code that works correctly in the verification environment results in a correct driver.

The rest of the paper is structured as follows. We start with an overview of the traditional hardware and driver development workflow in Section 2. Section 3 presents the proposed improved workflow. Section 4 reports on the results of a case study that applies this workflow to implement a driver for an Ethernet controller. We survey related work in Section 5 and draw conclusions in Section 6.

2 Traditional Development Process

In current industrial practice, design and testing of a device are tightly integrated. Starting from a natural-language description of the desired device functionality, the hardware engineer develops a register transfer level (RTL) model. As mistakes are expensive and time-consuming to fix once the design has been synthesised in silicon, much effort is put into verifying the design in a simulated environment prior to producing hardware.

Testbench Verification uses a *testbench*, which is a program designed to exercise the design and validate its behaviour under a wide range of operating conditions. Most modern testbenches follow the layered architecture shown in Figure 1 (a), 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. These scenarios can be directed or randomised. A directed scenario consists of a fixed sequence of commands, whereas a randomised scenario generates random command sequences subject to a set of constraints.

Finally, the bottom layer of the testbench simulates the physical medium that the device controls. For instance, if the DUT is an Ethernet MAC controller, this layer simulates an Ethernet physical transceiver (PHY) chip.

In addition to the above components, a complete testbench contains other modules, including monitors, scoreboards, and coverage points that check whether the device behaves correctly during tests and whether the tests have covered the entire device functionality.

Driver A device driver is usually developed by an individual or a team separate from the design and verification engineers who design the device and its testbench.

Figure 1(b) shows a device driver in the context of the operating system I/O stack. The latter 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.

Figure 1 illustrates that there is a lot of commonality between the agent component of the testbench and the driver produced by the software team. Both convert high-level requests into interactions with the device. Yet, the driver is developed in a much less supportive environment.

Firstly, verification engineers have complete access to all device specifications and design internals and the possess expertise to understand these specifications. 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 device 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.

3 Testbench Reuse

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, it 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.

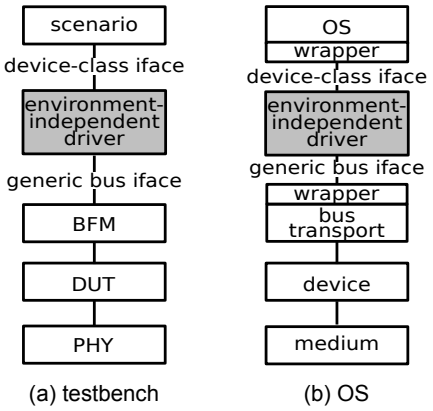

Figure 2: The testbench and OS architecture using a common implementation of the environment-independent device driver.

3.1 Overview of Approach

Instead of trying to adapt existing testbench code to work in the OS environment, we propose an approach where the testbench and the driver are *designed to share* common device management code. To this end, we unify the interfaces between the device management code and its environment, be it a testbench or an OS. A correct implementation of these interfaces is guaranteed to work correctly in both environments. We call such a reusable implementation an *environment-independent device driver*.

The resulting testbench and OS architectures are shown in Figure 2. In both environments, the device is managed by the environment-independent device driver, which interacts with the environment via two standardised interfaces: the generic device-class interface shared by all similar devices (e.g. all Ethernet controllers or all SCSI adapters) and the 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 environment-independent 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 OSs.

One complication involved in sharing the environment-independent driver implementation between the testbench and the driver is that testbenches are written in hardware description and verification languages (HDVLs) such as SystemC and SystemVerilog, whereas OS kernels can only host code written in C or C++. One possible solution is to automatically translate the environment-independent driver from HDVL to C. Alternatively, an environment-independent driver can be written in C and incorporated into a HDVL testbench using inter-language bindings. We aim to support both approaches.

3.2 Device-class contracts

Interface compatibility is not sufficient to ensure that an environment-independent 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 each interface, which 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.

The device-class contract is specified by OS experts, who have a good understanding of how device drivers interact with the OS, in collaboration with hardware verification engineers, who are responsible for incorporat-

ing the contract specification in the device testbench, as explained below.

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

> *When the controller is enabled, the client can call the packet transmission method of the driver. The driver must transmit packets received from the client over the network in FIFO order. Once a packet has been transmitted, the driver must notify the client by invoking the packet completion callback.*

In order to facilitate the testing of environment-independent drivers for contract compliance, we formulate the contract as an imperative program that can be incorporated in the device testbench as one of the testing scenarios. This program represents the most general contract-compliant environment that is capable of issuing all legal sequences of driver requests. It generates random requests to the driver according to the constraints of the contract and checks the driver behaviour for adherence to the contract.

Contract-based testing has important advantages compared to conventional driver testing. First, by randomly generating sequences of driver requests, it achieves a high degree of program coverage. Second, being integrated in the testbench, the contract can monitor not only driver responses but also device outputs and internal hardware signals. This allows it to check that requests sent to the driver result in correct device behaviours.

Of course, even such thorough testing does not guarantee correctness, since complete test coverage cannot be achieved in practice. However, it enables much higher degree of confidence than conventional driver testing.

Figure 3 illustrates how monitoring of hardware interfaces can enhance the quality of driver testing. It shows an example testbench for an Ethernet MAC controller device that uses the device-class contract as its testing scenario. Arrows represent events involved in checking the packet transmission rule specified above.

The contract module generates a random Ethernet packet and issues a `send` request to the driver. It saves a copy of the packet in its internal queue (not shown in the figure). When the driver successfully sends the packet, it appears at the PHY interface of the device and is received by the Ethernet PHY module. The PHY module notifies the scenario layer about the packet via the `tx_frame` callback. At this point, the scenario module compares the packet received by the PHY layer against the packet in its internal queue, to make sure that the packet data has not been corrupted, and marks the packet as transmitted. Eventually, the driver issues a `send_complete` notification to the contract module, which checks that the notification is not premature, i.e., the first frame in the transmit queue is marked as completed, and removes the frame from the queue.

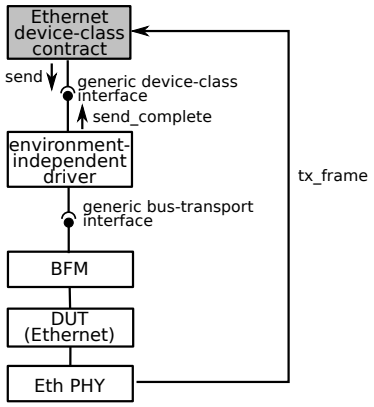

Figure 3: Ethernet controller testbench set up to test the environment-independent driver for contract compliance.

Device-class interfaces and contracts are subject to extension. Many devices support non-standard features that might not be covered by the existing interface. In such cases, a custom version of the interface and the associated contract must be produced for the device. In addition, the OS wrappers also need to be modified to support the new behaviours. Such modifications will be strictly incremental in most cases, since non-standard devices are usually compatible with all features of the standard ones. This incremental approach does not work for highly specialised devices that that cannot be attributed to an existing device class. Such devices require designing the device-class interface, device-class contract, and OS wrappers from scratch.

3.3 Discussion

The proposed methodology relies on hardware verification engineers to build their testbenches around generic driver interfaces. We believe that this is a practical approach for the following reasons. First, driver reliability is an increasingly important concern for hardware vendors, especially in the embedded space where short product development cycles put extra pressure on engineers to produce reliable drivers within a short time frame. Our approach allows implementing and testing driver code even before the device has been implemented in silicon, thus cutting down the development cost and time to market.

Second, while introducing only minor changes to the verification workflow, the proposed methodology improves the quality of hardware verification. By testing the environment-independent 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. The effectiveness of this approach has been demonstrated in the case study presented in Section 4, where we found a number of design errors in the device which had slipped through conventional verification.

4 Case Study

We applied the proposed methodology to the OpenCores Ethernet MAC 10/100Mbps controller [10] whose RTL description is publicly available. We defined the Ethernet controller device-class contract and implemented an environment-independent device driver that adheres to this contract, and constructed a testbench similar to the one shown in Figure 3. All components of the testbench, including the contract and the driver, are written in SystemVerilog.

We implemented the OS wrappers (Figure 1 (d)) for Linux. As we do not yet have a compiler from SystemVerilog to C, we perform this translation manually. This translation is still in progress at the time of writing, so we cannot yet test the driver in the target OS.

Nevertheless, initial experience has been encouraging. The contract-driven testbench not only found bugs in the environment-independent driver, but actually found defects in the DUT which the original testbench failed to expose.

Interestingly, we found that one of the defects detected in the environment-independent driver is also present in the existing Linux driver for the device. The defect was caused by an undocumented device behaviour, and manifested itself when the driver disabled the device and immediately deallocated all associated DMA buffers. Its cause was that the device, when disabled, it did not stop its transmit and receive engines until any transfer in process was complete. The defect was identified by the contract module when it received a transmit notification from the PHY module after the driver had reported that the transmitter had been disabled. This example illustrates how the effectiveness of driver testing is improved by performing such testing in the testbench environment.

The hardware-design defects we found resulted from situations which occur in a real OS, but which the original testbench failed to model. These include a hardware race condition that occurred when the device was disabled with its receive ring not empty and later reenabled, which resulted in lost and corrupted packets.

Both examples illustrate the disconnect between hardware and software engineers: software engineers often do not understand device internals enough to provide adequate software support for the device, and hardware engineers are not sufficiently familiar with the use patterns that the device must handle under OS control.

An encouraging observation is that the testbench provides a much 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.

Since our Linux driver is still incomplete, we have not been able to measure its performance. Results of the performance evaluation will be available for the final version of the paper.

5 Related Work

Previous research on device driver reliability has mainly focused on detecting [1], isolating [8], and avoiding [11] generic programming errors and errors in the interface between the driver and the OS. Few researchers have looked at errors related to interaction with the device. Kadav et al. [5] proposed an automatic code transformation tool that improves the handling of device errors in the driver. Tools like Devil [9] and Termite [12] generate a partial or complete implementation of a driver from a formal specification of the device interface. The main limitation of this approach is the availability of a correct specification.

Recently, Kuznetsov et al. [7] 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 testbench environment, it is possible to achieve better precision in detecting device protocol violations.

Bombieri et al. [2] 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.

The hardware/software co-simulation technique [6] used for early prototyping and debugging uses the same driver implementation in verification and production en-

vironments. However, this approach is not compatible with current verification practices based on HDVL testbenches. In addition, the resulting driver is tied to a single software environment and cannot be reused across different OSs.

6 Conclusions and Future Work

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 cost of driver development, as the driver is an almost-free by-product of device verification;
- driver portability, as it is developed in an OS-agnostic fashion.

Ongoing research focuses on applying this approach to a representative selection of I/O devices, demonstrating the reusability of environment-independent device drivers across different OSs, and performance analysis of the resulting drivers.

7 Acknowledgements

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

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

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 *1st EuroSys Conf.*, pages 73–85, Leuven, Belgium, Apr 2006.

[2] N. Bombieri, F. Fummi, G. Pravadelli, and S. Vinco. Correct-by-construction generation of device drivers based on RTL testbenches. In *45th DATE*, pages 1500–1505, Apr 2009.

[3] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *18th SOSP*, pages 73–88, Lake Louise, Alta, Canada, Oct 2001.

[4] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *20th LISA*, pages 101–111, Washington, DC, USA, 2006.

[5] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *22nd SOSP*, Big Sky, MT, USA, 2009.

[6] M. Kudlugi, S. Hassoun, C. Selvidge, and D. Pryor. A transaction-based unified simulation/emulation architecture for functional verification. pages 623–628, Jun 2001.

[7] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *2010 USENIX*, Boston, MA, Jun 2010.

[8] 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, Sep 2005.

[9] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *4th OSDI*, pages 17–30, San Diego, CA, USA, Oct 2000.

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

[11] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *4th EuroSys Conf.*, Nuremberg, Germany, Apr 2009.

[12] L. Ryzhyk, P. Chubb, I. Kuz, E. L. Sueur, and G. Heiser. Automatic device driver synthesis with Termite. In *22nd SOSP*, Big Sky, MT, USA, Oct 2009.