Dingo: Taming device drivers

Conference Paper · January 2009

DOI: 10.1145/1519065.1519095 · Source: DBLP

4 authors:

Leonid Ryzhyk

48 PUBLICATIONS **693** CITATIONS

SEE PROFILE

Peter Chubb

The Commonwealth Scientific and Industrial Research Organisation

35 PUBLICATIONS **527** CITATIONS

SEE PROFILE

Ihor Kuz

National ICT Australia Ltd

48 PUBLICATIONS **843** CITATIONS

SEE PROFILE

Gernot Heiser

UNSW Sydney

246 PUBLICATIONS **9,404** CITATIONS

SEE PROFILE

Dingo: Taming Device Drivers

Leonid Ryzhyk[12] Peter Chubb[12] Ihor Kuz[12] Gernot Heiser[123]

[1]NICTA* [2]The University of New South Wales [3]Open Kernel Labs
Sydney, Australia
leonid.ryzhyk@nicta.com.au

Abstract

Device drivers are notorious for being a major source of failure in operating systems. In analysing a sample of real defects in Linux drivers, we found that a large proportion (39%) of bugs are due to two key shortcomings in the device-driver architecture enforced by current operating systems: poorly-defined communication protocols between drivers and the OS, which confuse developers and lead to protocol violations, and a multithreaded model of computation that leads to numerous race conditions and deadlocks.

We claim that a better device driver architecture can help reduce the occurrence of these faults, and present our Dingo framework as constructive proof. Dingo provides a formal, state-machine based, language for describing driver protocols, which avoids confusion and ambiguity, and helps driver writers implement correct behaviour. It also enforces an event-driven model of computation, which eliminates most concurrency-related faults. Our implementation of the Dingo architecture in Linux offers these improvements, while introducing negligible performance overhead. It allows Dingo and native Linux drivers to coexist, providing a gradual migration path to more reliable device drivers.

Categories and Subject Descriptors D.4.4 [*Operating systems*]: Input/Output; D.3.2 [*Language Classifications*]: Specialized application languages

General Terms Languages, Reliability, Verification

Keywords Concurrent Programming, Device Drivers, Domain-Specific Languages, Fault Avoidance, Reliability.

* 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

EuroSys'09, April 1–3, 2009, Nuremberg, Germany.

1. Introduction

While accounting for about 70% of OS code, drivers typically contain several times more errors per line of code than other system components [Chou 2001] and, according to recent studies, are responsible for up to 70% of system failures [Ganapathi 2006, Murphy 2004]. With the introduction of advanced hardware capabilities such as hot-plugging, power management, and vectored I/O, device drivers have increased in complexity and hence become even more error-prone.

This paper explores the factors that contribute to driver complexity and lead to buggy drivers. In analysing bugs found in real Linux drivers, we discover and demonstrate quantitatively that a large proportion of these factors result from the way drivers interface with the OS, and can be eliminated or mitigated by a better design of the driver-OS interface.

Specifically, we identify two shortcomings of the driver architecture common in modern operating systems: poorly-defined communication protocols between drivers and the OS, which confuse developers and lead to protocol violations, and a multithreaded model of computation that leads to numerous race conditions and deadlocks. To address these issues, we developed Dingo[1]—a device-driver architecture aimed at simplifying development and reducing the number of software defects in drivers.

In order to reduce protocol errors, driver protocols in Dingo are specified using a state-machine-based formal language called Tingu.[2] Tingu allows a clear and unambiguous description of requirements for driver behaviour, providing intuitive guidelines to driver programmers. The primary purpose of Tingu specifications is to serve as documentation helping driver developers avoid errors; however they can also be used as properties against which driver implementation can be formally validated either statically or at runtime. Presently we only support runtime validation by compiling driver protocol specifications into a runtime observer that detects protocol violations committed by the driver.

[1] A Dingo is an Australian wild dog.

[2] Tingu is an Australian aboriginal name for a Dingo cub.

Dingo addresses concurrency issues by defining an event-driven model of computation for drivers, where the driver interacts with the system through a series of atomic operations. This model avoids the synchronisation complexity of a multithreaded model, and eliminates many concurrency bugs.

We have implemented the Dingo architecture and several device drivers in Linux. We show that Dingo eliminates most concurrency errors and reduces the likelihood of protocol errors, while introducing negligible performance overhead.

Our work complements previous research on device driver reliability. Most existing approaches offer static [Engler 2000, Chou 2005, Ball 2006, Fähndrich 2006] or runtime [Mérillon 2000, Swift 2002, Herder 2006, Zhou 2006] techniques to detect and isolate driver faults. As we will see in Section 10, while improving driver reliability, these techniques only deal with certain types of faults and come at the cost of significant performance overhead and increased design complexity. In contrast, the goal of Dingo is to eliminate the root causes that lead to faults rather than to deal with their consequences.

Section 2 presents an analysis of driver bugs in Linux. Section 3 then introduces the Dingo device driver architecture and Sections 4 and 5 provide details of our event-driven model and the Tingu language. We present the implementation of Dingo and its evaluation in Sections 8 and 9, review related work in Section 10, and conclude in Section 11.

2. Analysis of driver bugs

In this paper we deal with drivers in general-purpose systems such as Linux and Windows. In particular we are concerned with the factors that lead to software faults in drivers. To this end we have analysed the code of existing drivers to determine the types and causes of the most common faults. The findings from this study help to direct our efforts in improving driver reliability to where they are likely to achieve most. Before discussing the study further, we introduce the key concepts related to device drivers.

2.1 The place of drivers in the OS

A device driver is the part of the OS that is responsible for directly controlling hardware devices. On the one hand it communicates with the hardware device itself through device registers, DMA, and interrupts. On the other hand it communicates with the operating system, which provides the driver with access to other drivers and support services such as synchronisation primitives and resource management, and uses the driver through higher layers of the system software stack, including file systems, network stacks and so on.

A driver's communication with the hardware and OS must follow a protocol that determines correct timing, order, and format of interactions. We distinguish between the device protocol, which regulates communication with the hardware device, and the software protocol, which regulates communication with the OS. Each device has a unique device protocol defined by the manufacturer. Drivers conceal this device diversity from the OS by implementing standard software protocols, common to a family of devices, e.g. Ethernet, audio, etc. These protocols are defined by the OS. The OS also defines protocols for accessing the support services that it provides. These protocols and services are collectively referred to as the operating system's driver framework.

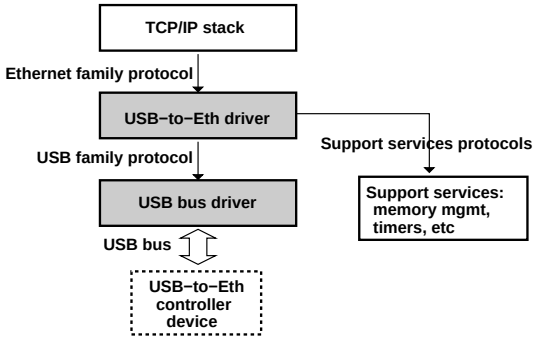

Figure 1. Example USB-to-Ethernet controller driver and its software environment.

Figure 1 shows an example of a USB-to-Ethernet controller driver and the software protocols that it implements and uses. The driver implements the Ethernet family protocol that exports the network interface functionality to the OS network stack. Since the device in question is connected to the USB bus, it cannot be accessed by the driver directly. Instead, device protocol messages must be encapsulated inside USB data packets submitted to the USB bus driver via the USB family protocol. Such driver stacking is common in operating system I/O frameworks and reflects the structure of the underlying I/O bus hierarchy.

2.2 A study of driver bugs in Linux

In this study, we analysed a sample of real defects found in a selection of Linux drivers. Our analysis made use of the accessibility of a complete kernel development history, including detailed descriptions of every change (`http://linux.bkbits.net`). This allowed us to easily identify and categorise a large number of driver bugs. A similar study of Windows drivers would enable an interesting comparison, since Windows defines a substantially different driver architecture than UNIX-like systems. Unfortunately, we were not able to conduct such an investigation, because information about bugs found in Windows drivers is not publicly available.

For our study, we selected 13 drivers for different types of devices (Table 1).[3] We selected drivers for devices on different buses (USB, IEEE 1394, and PCI) since we expected each to pose different issues for driver development.

[3] Among the 13 drivers, there are 4 USB-to-Ethernet controller drivers (top 4 entries in Table 1). We required statistics for several similar drivers for the evaluation in Section 9. Otherwise, we made our selection of drivers maximally diverse.

We then built a bug database for these drivers by analysing all changes made to the drivers during the six-year period from 2002 to 2008. In all we recorded 498 defects in this database.

In order to identify the main sources of complexity in device drivers, we distinguish between errors caused by the complexity of interacting with the device, errors caused by the complexity of interacting with the operating system, and generic programming errors. Specifically, we distinguished the following categories of driver software faults:

Device protocol violations occur when the driver behaves in a way that violates the required hardware protocol, and typically result in a failure of the hardware to provide its required service. These include putting the device into an incorrect state, mis-interpreting device state, incorrectly parsing or generating data exchanged with the device, issuing a sequence of commands to the device that violates the device protocol, specifying incorrect timeout values for device operations, and endianness violations. Device protocol violations constitute 38% of the overall defects (Table 1).

According to our study, at least one third of the faults in device-control logic are caused by poorly documented device behaviour. Such faults are particularly common when device documentation is not readily available, and the driver is produced by reverse engineering a driver from another OS.

A portion of these faults are also caused by devices whose behaviour deviates from the hardware interface standards that they are meant to implement. Similar faults are due to devices that violate their documented behaviour. In both these cases, drivers that expect hardware to behave according to the standards or documentation will function incorrectly and must be fixed by adding appropriate workarounds.

Software protocol violations occur when the driver performs an operation that violates the required protocol with the OS. This includes all violations of expected ordering, format or timing in interactions between the OS and the driver. These faults are particularly common in error-handling paths and code paths handling uncommon situations such as hot-unplug and power management requests, which are often insufficiently tested.

Examples of ordering violations include forgetting to wait for a completion callback from an asynchronous data request (data protocol violation), trying to resume a suspended device before restoring its PCI power state (power management protocol violation), and forgetting to release a resource or releasing resources in the wrong order (resource ownership protocol violation). Examples of format violations include incorrectly modifying a data structure shared with the OS, incorrectly initialising a driver descriptor before passing it to the OS, and falsely returning a success status from an operation that failed.

Software protocol violations constitute 20% of the overall driver defects. Statistics for the frequencies of different types of protocol violations are shown in Table 2.

Type of faults	#
Ordering violations	
Driver configuration protocol violation	16
Data protocol violation	9
Resource ownership protocol violation	8
Power management protocol violation	8
Hot unplug protocol violation	5
Format violations	
Incorrect use of OS data structures	29
Passing an incorrect argument to an OS service	19
Returning invalid error code	7

Table 2. Types of software protocol violations.

Concurrency faults occur when a driver incorrectly synchronises multiple threads of control executing within it, causing a race condition or a deadlock.

Unlike the previous bug categories, concurrency bugs are not related to a particular aspect of the driver functionality, but rather to the model of computation enforced by the OS on device drivers. Any non-trivial device driver is involved in several concurrent activities, including handling I/O requests, processing interrupts, and dealing with power management and hot-plugging events. Most operating systems are designed to run these activities in separate threads that invoke the driver in parallel. This multithreaded model of computation requires the driver to protect itself from race conditions using a variety of synchronisation primitives. In addition, a driver in the kernel environment has to keep track of the synchronisation context in which it is invoked. For instance, a driver running in the context of an interrupt handler is not allowed to perform any potentially blocking operations.

Concurrency management accounts for 19% of the total number of bugs. In Figure 2 we see that the rate of concurrency bugs is higher in USB drivers (26.5%) and IEEE 1394 drivers (23.5%) than in PCI drivers (9%). USB and IEEE 1394 buses support hot-plugging, which introduces a device disconnect event to the driver interface. Disconnect happens asynchronously to all other activities, causing race conditions in all USB and IEEE 1394 drivers covered by our study. In addition, since these buses are not memory mapped, communication with the device is based on asynchronous messages, which adds another degree of concurrency to the driver logic.

Statistics for different types of concurrency faults are shown in Table 3. From this we see that concurrency faults are mostly introduced in situations where a sporadic event, such as a hot-unplug notification or a configuration request, occurs while the driver is handling a stream of data requests.

Generic programming faults This category of bugs includes common coding errors, such as memory allocation errors, typos, missing return value checks, and program logic errors. These errors account for the remaining 23% of defects.

Name	Description	Total faults	Device prot. violations	S/W protocol violations	Concurrency faults	Generic faults
USB drivers						
rtl8150	rtl8150 USB-to-Ethernet adapter	16	3	2	**7**	4
catc	el1210a USB-to-Ethernet adapter	2	**1**	0	**1**	0
kaweth	kl5kusb101 USB-to-Ethernet adapter	15	1	2	**8**	4
usb net	generic USB network driver	45	**16**	9	6	14
usb hub	USB hub	67	**27**	16	13	11
usb serial	USB-to-serial converter	50	2	17	13	**18**
usb storage	USB Mass Storage devices	23	7	5	**10**	1
IEEE 1394 drivers						
eth1394	generic ieee1394 Ethernet driver	22	**6**	**6**	4	**6**
sbp2	sbp-2 transport protocol	46	**18**	10	12	6
PCI drivers						
mthca	InfiniHost InfiniBand adapter	123	**52**	22	11	38
bnx2	bnx2 network driver	51	**35**	4	5	7
i810 fb	i810 frame buffer device	16	4	**5**	2	**5**
cmipci	cmi8338 soundcard	22	**17**	3	1	1
	Total	498	189 (38%)	101 (20%)	93 (19%)	115 (23%)

Table 1. Classified counts of driver faults. The maxima in each row are in **bold face**. The highlighted cells summarise the types of faults that we focus on in the rest of the paper.

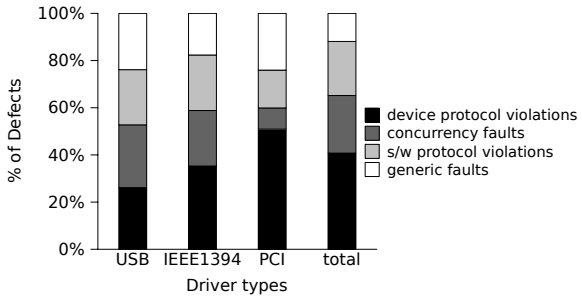

Figure 2. Summary of software faults by driver type.

Type of faults	#
Race or deadlock in configuration functions	29
Race or deadlock in the hot-unplug handler	26
Calling a blocking function in an atomic context	21
Race or deadlock in the data path	7
Race or deadlock in power management functions	5
Using uninitialised synchronisation primitive	2
Imbalanced locks	2
Calling an OS service without an appropriate lock	1

Table 3. Types of concurrency faults.

3. Dingo device driver architecture

Our driver defect study has revealed areas where better OS support could improve driver reliability. In particular two categories of faults are directly related to how the driver interacts with the OS: concurrency faults and software protocol violations. Together, these faults constitute 39% of the defects in our study, and are clearly a significant source of problems for drivers.

To address these issues, we developed Dingo, a new architecture for device drivers that simplifies interaction with the OS and allows driver developers to focus on the main task of a driver: controlling the hardware. Dingo achieves this via two improvements over traditional driver architectures. First, Dingo reduces the amount of concurrency that the driver must handle by replacing the driver's traditional multithreaded model of computation with an event-driven model. This model eliminates the majority of concurrency-related driver faults without impacting the performance. Second, Dingo provides a formal language for describing driver software protocols, which avoids confusion and ambiguity, and helps driver writers implement correct protocols.

Dingo does not attempt to provide solutions to deal with the other types of defects identified (i.e., device protocol violations and generic programming faults) since these are provoked by factors that lie beyond the influence of the OS and should be eliminated by complementary means such as those surveyed in Section 10.

Overview of Dingo Dingo specifies a model for communication between a driver and its environment. Communication occurs over ports, which are bidirectional message-based communication points. In a typical implementation, ports are represented by function tables and messages are delivered by invoking the corresponding functions. Dingo guarantees atomic message delivery resulting in a strict ordering of all messages exchanged by drivers.

Each port is associated with a protocol, which specifies a behavioural contract between the driver and the framework. It defines the messages that can be exchanged over that port as well as constraints on the ordering, timing and content of those messages. Every port has exactly one protocol asso-

ciated with it, however, a driver typically has several ports over which it communicates.

The Dingo architecture can be implemented as a self-contained OS driver framework or it can be built on top of another existing driver framework, providing an improved interface for developing device drivers within that framework. The latter approach allows Dingo drivers to coexist with legacy drivers.

4. An event-driven architecture for drivers

The concurrency problems highlighted earlier are not unique to device drivers. In a multithreaded environment, concurrent activities interleave at the instruction level, leading to non-determinism and state explosion. As a result, many programmers are generally ineffective in dealing with threads, which makes multithreading the leading source of bugs in a variety of applications, including OS kernels.

An alternative to multithreaded concurrency is event-driven concurrency. In the event model, a program executes as a series of event-handlers triggered by events from the environment. Events are strictly serialised, thus replacing instruction-level interleaving with event-level interleaving. Serialisation guarantees that the state of the program observed at the start of an event can be modified only by the current event handler. This simplifies reasoning about the program behaviour and reduces the potential for race conditions and deadlocks.

Comparison of threads versus events has been the subject of lasting debate in the systems community [Lauer 1978, Adya 2002, von Behren 2003]. One point of consensus in this debate is that different applications may favour different models. We argue that in the context of device drivers the event model eliminates most concurrency-related bugs and can be implemented in a way that neither complicates driver development nor incurs a performance penalty. Thus it should be the preferred model.

One observation in favour of an event-driven approach is that modern device drivers are already partially event-driven for performance reasons. In particular, all performance-critical I/O requests are completed asynchronously: upon receiving a request, the driver adds it to the hardware queue and immediately returns control to the caller. Later, it receives a completion notification from the device and invokes a completion callback provided by the OS. Asynchronous handling of requests enables improved performance by parallelizing I/O and computation. This interaction pattern of splitting long-running operations into request and completion steps is typical for event-driven systems. Thus, while current drivers do not fully exploit the advantages of the event-driven model, this style of programming is already familiar to driver developers.

Event-driven architecture of Dingo As described in Section 3, a Dingo driver interacts with the OS via messages. A message is delivered to a driver by invoking a corresponding message-handler function exported by the driver through a port. Likewise, a driver can send messages by invoking corresponding entry points exported by the OS. The main distinction between Dingo message interfaces and procedural driver interfaces in conventional systems is that message handlers are executed in an *atomic* and *non-blocking* manner. The atomicity guarantee means that no new message can be delivered to a driver while a previous message handler is running. This prohibits simultaneous invocations of a handler by different threads, as well as recursive calls from the same thread. Note that the atomicity constraint does not prevent the driver from being invoked from different threads or different CPU cores, as long as all invocations are serialised. In return for the luxury of being invoked atomically, the driver is required to handle messages "quickly". Specifically, it is not allowed to block or busy wait, because such behaviour would delay the delivery of subsequent messages.

The event-based architecture affects driver development in two ways. First, since Dingo serialises execution of the driver at the message level, there is no need for synchronisation among concurrent message handlers. Therefore, Dingo drivers do not use spinlocks, mutexes, wait queues, or other thread-based synchronisation primitives. However, the driver may have to synchronise tasks that span multiple messages. For example, when handling two long-running I/O requests that use the same hardware resource, the driver must ensure that execution of the second request begins after the first request completes. This is typically achieved by tracking the status of the shared resource using a state variable. The number of cases where such synchronisation is required is much smaller than in multithreaded drivers (see Table 4 in Section 9). The event-driven architecture also simplifies the use of I/O memory barriers. In paticular, barriers that order accesses to I/O memory from different CPUs can be moved from the driver into the framework. On architectures that require barriers to order I/O memory accesses on a single CPU, the programmer is still responsible for correctly placing such barriers.

Second, since message handlers are not allowed to block, there is no way for the driver to freeze its execution waiting for an event, such as a timeout or a hardware interrupt. Instead, the driver has to complete the current message handler and later resume execution in the context of a new message. Splitting a single operation into a chain of completions leads to complex and unmaintainable code—the effect known as stack ripping [Adya 2002]. The cause of the problem is that by splitting a function into multiple fragments we lose compiler support for automatic stack management and mechanisms that rely on it, such as control structures and local variables.

Fortunately, one can combine an event-driven model of computation with automatic stack management. One way to achieve this has been demonstrated by Tame [Krohn 2007], a C preprocessor providing event-driven programs with a se-

quential style of programming. We have implemented a similar approach in Dingo. Our preprocessor provides several macros that enable driver control logic to be expressed in the natural sequential way, avoiding stack ripping. To illustrate this solution, consider the following excerpt from a Linux driver, which polls a device status register until its value satisfies a certain condition. The `msleep` function causes the current thread to block for ten milliseconds before retrying the read:

```
do {
    msleep (10);
    /* read status register */
    ...
} while (/*condition*/);
```

Corresponding code in a Dingo driver looks like this:

```
do {
    CALL (timeout (10, &notif), notif);
    /* read status register */
    ...
} while (/*condition*/);
```

The `CALL` construct, when expanded by the preprocessor, calls the `timeout` function to start a timer, then saves the state of the current function, including arguments, local variables and the instruction pointer in a continuation structure, registers this structure as the handler for the `notif` event and returns from the current message handler. When the timer triggers the `notif` event after 10 milliseconds, the continuation fires, restores the state of the function and continues from the next statement following `CALL`. The key to this solution is that the state of the execution stack is stored in continuation structures and is automatically reconstructed by `CALL` statements. The programmer can rely on automatic stack management and can use local variables and C control structures in the usual way.

Other constructs supported by the preprocessor include `EMIT`, which triggers an event, and `AWAIT`, which suspends the program until one of a set of events occurs.

Selectively reintroducing multithreading One limitation of the event-driven model is that it prevents the program from exploiting multiprocessor parallelism. This is not an issue for the vast majority of drivers, which are I/O-bound rather than CPU-bound. As we will see in Section 9, a careful implementation of the event model enables event-based drivers to achieve the same I/O throughput and latency as multithreaded drivers. However, there exist devices, such as 10Gb Ethernet or InfiniBand controllers, designed for very high throughput and low latency, whose performance could potentially suffer from request serialisation. Although in our experiments event-based drivers perform well even for these devices, we would like to allow driver developers to use multithreading when absolutely necessary.

We observe that high-performance devices are designed to minimise contention and avoid synchronisation in the data path. As a result, the synchronisation complexity in their drivers is concentrated in the control path, whereas the data path is free of synchronisation operations. Based on this observation, we introduce a hybrid model in Dingo, which allows concurrency among data requests but not control requests. In this model, all control messages are serialised with respect to each other and to data messages. However, multiple data messages are allowed to be handled concurrently. The driver implementer can choose whether the driver should run in the fully serialised mode or in the hybrid mode. Drivers running in the hybrid mode benefit from the advantages of the event-driven model without experiencing any added overhead of serialisation. The distinction between data and control messages is drawn by the protocol designer who labels messages that can be sent or received concurrently with a special keyword.

We have implemented both modes for the InfiniBand driver described in Section 8. Our original implementation was fully serialised. We found that no changes to the driver were needed to run it in the hybrid mode, since the data path of the driver did not require any synchronisation.

5. Tingu: describing driver software protocols

In Section 2.2 we showed that 20% of driver defects are violations in the ordering or format of messages exchanged with the OS. A closer study of driver protocols in Linux shows that these protocols are not particularly complicated. We maintain that it is not the protocol complexity that causes bugs, but rather the fact that they are not adequately documented, forcing driver developers to guess correct behaviour. For example, details of how to react to a hot-unplug notification in the driver's different states, or how to handle a shutdown request that arrives during a transition to the suspend state (and whether such a situation is even possible), are not easy to find in documentation.

In Dingo, we address this problem by specifying the communication protocols between drivers and the OS using a formal language. While informal descriptions tend to be incomplete and can easily become bulky and inconsistent, a well-chosen formalism can capture protocol constraints concisely and unambiguously, providing driver developers with clear instructions regarding the required behaviour. Additionally, by providing a specification of driver protocols, we enable formal checking of driver correctness, both statically and at runtime. This is further discussed in Section 7.

The challenge in designing the protocol specification language is to satisfy both expressiveness and readability requirements. In order to be useful, driver protocol specifications must be easily understood by driver developers. This encourages the use of simple visual formalisms such as finite state machines (FSM) or UML sequence diagrams. Unfortunately many aspects of driver protocols cannot be expressed using these simple notations.

One such aspect is the dynamic structure of driver interfaces. For instance, the USB family protocol allows client drivers to create multiple data connections, called pipes, through the USB bus to their associated devices at runtime. In the USB-to-Ethernet controller example (Figure 1), such functionality is used by the Ethernet driver to open several parallel data and control connections to its associated device. Each such connection operates in parallel with the others, and behaves according to its own protocol.

This example also serves to highlight another complication common to driver protocols, namely protocol dependencies. In the USB driver, the behaviour of each individual pipe is dependent on the state of the main USB bus protocol. For instance, no data transactions can be issued through pipes after the bus has been switched to a low-power mode. Given that pipes behave according to their own protocols, and that this behaviour is dependent on the behaviour of the USB bus as specified by its own protocol, we require a means to describe dependencies between different protocols.

Our search for a formalism that supports both the required expressiveness and readability has led to the development of a new software protocol specification language called Tingu. The design of Tingu is driven by our experience specifying and implementing real driver interfaces. In particular, we introduce a construct to the language only if it has proven necessary for modelling the behaviour of several types of drivers and can not be expressed easily using other constructs. We give a brief overview of Tingu below; more details can be found in [Ryzhyk 2007].

Tingu has both a textual and visual component. The textual component is used to declare elements of a protocol, such as ports and messages. The visual component is used to specify protocol behaviour using a subset of the Statecharts [Harel 1987] syntax extended with several new constructs that provide support for dynamic port spawning and protocol dependencies. With dynamic spawning one can specify a behaviour that leads to creation of a new port at runtime. The new port is created as a subport of an existing port and is associated with its own protocol. Protocol dependencies define message ordering constraints across multiple protocols. This is achieved by allowing several protocols to constrain occurrences of the same message: the message can only be sent when it is permitted by all involved protocols.

We illustrate the syntax and semantics of Tingu using the USB-to-Ethernet driver example introduced earlier. The following Tingu specification fragment declares the ports and associated protocols of the driver.

```
component asix {
ports:
    Lifecycle lc;
    PowerManagement pm;
    EthernetController eth;
    USBInterfaceClient usb;
    mirror Timer timer;
}
```

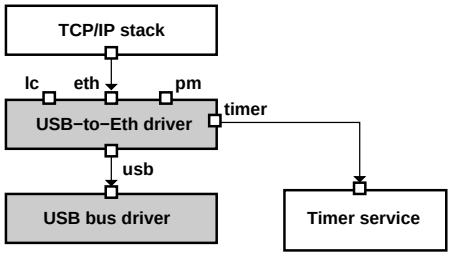

Figure 3. Ports of the USB-to-Ethernet adapter driver.

The driver provides services to the OS via `Lifecycle`, `PowerManagement` and `EthernetController` protocols. It uses the `USBInterfaceClient` protocol exported by the USB bus driver, and the `Timer` protocol exported by the OS timer service. The `mirror` keyword indicates that the driver implements the client side of the protocol. Figure 3 represents this specification visually and can be viewed as a refinement of Figure 1.

The following listing declares the `Lifecycle` protocol and its messages:

```
protocol Lifecycle {
messages:
    in  start();
    out startComplete();
    out startFailed(error_t error);
    in  stop();
    out stopComplete();
    in  unplugged();//hot-unplug event
}
```

Figure 4 shows the statechart of the `Lifecycle` protocol. State transitions represent legal sequences of protocol messages, with question marks (“?”) in trigger names denoting incoming messages and exclamation marks (“!”) denoting outgoing messages. A compact representation of complex protocols is achieved by organising states into a hierarchy—a feature provided by Statecharts. Several primitive states can be clustered into a super-state. A transition originating from a super-state (e.g., the `?unplugged` transition in Figure 4) is enabled when the state machine is in any of its internal states.

When the driver is created, the protocol state machine is in its initial state, denoted by a dot and an arrow. The protocol terminates, i.e., no more messages of this protocol are allowed, when it reaches one of its final states, denoted by a circled dot. When all protocols of the driver terminate, the driver is destroyed.

In Figure 4 some states include timeout annotations in square brackets. A protocol is violated if, after entry into such a state, the given amount of time passes without the triggering of a transition leading to a different state. For instance, the driver is not allowed to stay in the `starting` state indefinitely. It must either complete initialisation or fail within five seconds after entering the state.

Some protocol state information is inconvenient to model using explicit states and is more naturally described by vari-

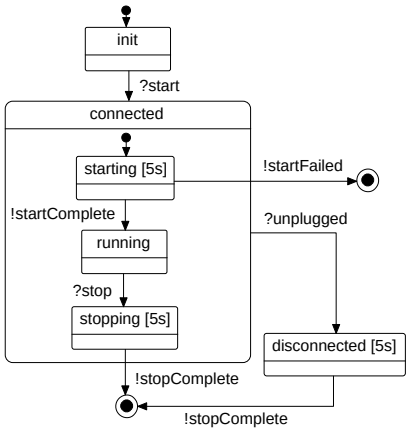

Figure 4. The `Lifecycle` protocol state machine.

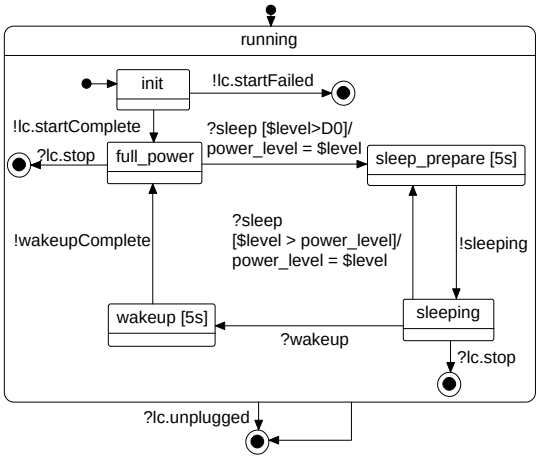

Figure 5. The `PowerManagement` protocol state machine.

ables. For example, the `PowerManagement` protocol in Figure 5 models a device's current power level using an integer variable called `power_level`. Variables are declared in a separate textual section inside the protocol specification.

A state transition label may include a guard, indicating whether the transition is enabled, and an action that updates protocol variables upon triggering of the transition. For example, consider the state machine of the `PowerManagement` protocol in Figure 5. The transition from state `full_power` to `sleep_prepare` is triggered by a `sleep` message. The guard expression in square brackets specifies that the `level` argument of the message must be greater than the `D0` constant, corresponding to the zero power saving mode; the action associated with the transition updates the value of the `power_level` variable to reflect the new power state.

Figure 5 also illustrates the use of protocol dependencies. When dealing with power management, a device cannot be transferred to a low-power mode until it has completed initialisation. This rule can be expressed as a dependency between the `Lifecycle` and `PowerManagement` protocols: The `PowerManagement` state machine may accept `sleep` messages only after a `startComplete` message, which is an

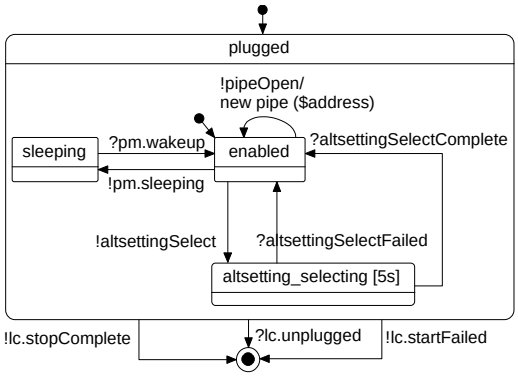

Figure 6. The `USBInterfaceClient` protocol. Message directions are shown relative to the driver using the USB bus driver. For example, an `altsettingSelect` message is sent to the bus driver in order to select and alternative interface configuration. The bus driver replies with an `altsettingSelectComplete` message.

output message of the `Lifecycle` protocol. This is shown in Figure 5 with the transition from `init` to `full_power`.

Finally, we illustrate the use of dynamic spawning with the example of the `USBInterfaceClient` protocol, which describes the service provided by the USB bus driver. As mentioned above, USB data transfers are performed via USB data pipes. The behaviour of an individual pipe is specified by the `USBPipeClient` protocol. Since the USB bus allocates these pipes dynamically, the driver determines which pipes it will use at runtime. A new pipe is created when a `pipeOpen` message is sent, as shown in Figure 6. This message takes a pipe address and a pointer to a port as arguments. In response to this message, the USB transport allocates a pipe and binds it to the provided port, so that the driver can immediately start using the pipe through this port.

One aspect of the driver interface currently not captured by Tingu protocols is I/O buffer management. For instance, Linux defines a complex API for manipulating network packet descriptors, including operations for cloning, merging, padding packets, etc. These interfaces do not fit well into the state machine framework of Tingu. Rather they can be formalised using abstract data types (ADT) or a related formalism. While Tingu does provide limited support for ADT's, a full description of such interfaces written using the present version of the language would lead to bulky unintuitive specifications, which would defeat the purpose of Tingu. As such, these APIs continue to be specified using C header files and informal documentation.

6. From protocols to implementation

Tingu protocols specify the externally visible driver behaviour and do not enforce any particular internal structure. In practice, however, the driver developer will typically closely follow the structure of the specification, maintaining correspondence between the driver code and protocol states.

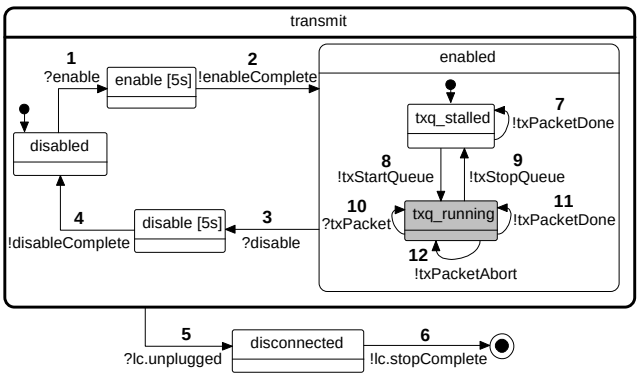

Figure 7. A fragment of the `EthernetController` protocol. Numbers above transition labels are for reference only and are not part of the protocol specification.

In this approach, driver protocol specifications are viewed as the first approximation of the driver design, which is refined into the implementation by adding device interaction code.

Figures 7 and 8 show a fragment of the `EthernetController` protocol that describes the packet transmission interface of an Ethernet driver and a simplified version of the corresponding fragment of the AX88772 driver code. Specifically, we focus on the state labeled `txq_running`. According to Figure 7, in this state the driver must be prepared to handle one of the following messages from the OS: `txPacket` instructing the driver to queue a packet for transmission, `disable` requesting the driver to disable the device receive and transmit circuitry, and `lc.unplugged` notifying the driver about a hot-unplug event. It is allowed to send one of the following messages: `txPacketDone` to notify the OS about successful transmission of a packet, `txPacketAbort` to report an error that occurred while sending a packet, and `txStopQueue` to prevent the OS from sending new packets until more buffer space becomes available in the controller.

In accordance with this specification, when the driver arrives in state `txq_running`, it pauses, waiting for one of the enabled external messages using the `AWAIT/IF/ELIF` construct (line 4 in Figure 8). The first three events listed in the `AWAIT` statement correspond to the `EthernetController` protocol events summarised above. Since the driver participates in several different protocols, it must be prepared to handle messages belonging to all its protocols. In this example, the last event (`pipeXferComplete`) in the `AWAIT` statement corresponds to a `USBPipeClient` protocol message, which is generated by the USB data pipe when it has completed transferring packet data to the controller.

Note that events given as arguments to `AWAIT` are language-level entities that are internal to the driver, and are distinct from protocol messages, which comprise the external interface of the driver. A message is delivered to the driver via a function invocation. It can then be transformed into an event using the `EMIT` construct, which instantiates an

```
1 void ax88772_txLoop(ax88772 * drv)
2 {
3   ...
4   AWAIT(txPacket,disable,unplugged,
5         pipeXferComplete)
6   {
7     IF(txPacket){ /*transition #10*/
8       /* start packet transfer over USB*/
9       ...
10
11      if(/*out of buffer space?*/) {
12        /*transition #9*/
13        eth->txStopQueue(eth);
14      };
15    }
16    ELIF(disable){ /*transition #3*/
17      /*abort all outstanding USB transfers*/
18      pipe->abort(pipe);
19
20      /*wait for the abort to complete*/
21      AWAIT(pipeAbortComplete){
22        /*transition #4*/
23        eth->disableComplete(eth);
24      };
25    }
26    ELIF(unplugged){ /*transition #5*/
27      /*wait for the USB pipe to abort all
28        outstanding transfers*/
29      AWAIT(pipeAbortComplete);
30
31      /*transition #6*/
32      lc->stopComplete(lc);
33    }
34    ELIF(pipeXferComplete){/*USB xfer complete*/
35      if(/*transfer successful?*/)
36       /*transition #11*/
37       eth->txPacketDone(eth,
38                         pipeXferComplete.pkt);
39      else
40       /*transition #12*/
41       eth->txPacketAbort(eth,
42                          pipeXferComplete.pkt);
40    };
43  };
44  ...
45};
```

Figure 8. A fragment of the AX88772 driver.

event and resumes execution of any `AWAIT` or `CALL` statement waiting for this event.

Figure 8 shows how the driver handles each input event and indicates the correspondence between messages exchanged by the driver through the `EthernetController` protocol and state transitions in Figure 7.

This example illustrates how protocol specifications can be mapped into driver implementation with the help of the event-based constructs supported by the preprocessor.

Presently, this mapping must be done manually by the driver developer. Automating it is part of the ongoing work.

7. Detecting failures at runtime

Tingu specifications help driver developers avoid protocol violations, but do not eliminate them completely. These faults can be dealt with using static or runtime verification. The Tingu compiler fully automates runtime verification by generating a driver *protocol observer* from the Tingu specification of its ports. The generated observer can be attached transparently to the driver. It intercepts all messages exchanged by the driver and keeps track of the state of all its protocols. Whenever the driver or the OS sends an illegal message or fails to send any messages within the time interval specified by a timeout state, the observer notifies the OS about the failure and outputs the current state of all driver protocols and the sequence of events leading to the failure.

Protocol observers have proved useful in testing and debugging device drivers during the development cycle. They can also be combined with any of the failure isolation and recovery solutions described in Section 10 to enhance the resilience of a production system to driver failures.

Static verification of drivers against protocol specifications is currently not implemented. One way to achieve this is to translate Tingu into a language supported by an existing model checker [Engler 2000, Chou 2005, Ball 2006]. Such translation is possible because these languages incorporate similar concepts to Tingu, but using textual rather than visual syntax. However we do not currently have experimental evidence proving or disproving the feasibility of this approach. In particular, it is unclear whether existing model checkers are sufficiently powerful to validate complex behaviours captured by Tingu protocols.

8. Implementation

Dingo on Linux We have implemented the Dingo driver architecture on Linux by constructing adapters between the multithreaded driver protocols defined by Linux and the event-driven protocols expected by Dingo drivers. This approach allows Dingo and native Linux drivers to coexist in the same system, offering a gradual migration path to more reliable device drivers.

Since the Linux driver interface is multithreaded, the Dingo adapter must serialise calls from Linux into the driver. While this could be achieved using a mutex, this solution would negatively affect performance on multiprocessor machines, where all CPUs trying to access the driver would have to wait for the current call to complete. Our solution is to use a request queue per driver. Calls that arrive while the driver is executing a message handler are queued, allowing the caller to continue execution. Requests in the queue are processed after the message handler completes execution. The request queue is protected by a spinlock. To support the hybrid model discussed in Section 4, adapters can be configured to disable serialisation for selected messages.

	# sync. objects		# crit. sections	
	Linux	Dingo	Linux	Dingo
AX88772	8	2	19	2
InfiniHost	24	6	51	10

Table 4. The use of synchronisation primitives by Linux and Dingo drivers.

Messages sent by the driver to the OS are translated to corresponding Linux calls. To ensure non-blocking semantics of messages, calls that may block are scheduled for delayed execution in the context of a kernel worker thread.

We have implemented adapters for the USB, Ethernet, and InfiniBand protocols as well as for generic protocols such as lifecycle, power management, and the timer service. Since our driver protocols are closely modelled after the corresponding Linux protocols, adapter implementation is fairly straightforward. In order to explore the reliability and performance implications of the Dingo driver architecture, we built two drivers for Linux: an AX88772 100Mb/s USB-to-Ethernet adapter driver and a Mellanox InfiniHost™ III Ex 10Gb/s dual-port InfiniBand controller driver.

Dingo on OKL4 We have also implemented the Dingo driver architecture on top of the commercial OKL4 microkernel. This implementation serves as a research vehicle for further exploration of a formal approach to device drivers. On OKL4, we have implemented drivers for the PCI-based RTL8139 Ethernet controller, the OHCI USB host controller driver, and the USB root hub.

9. Evaluation

We evaluate the Dingo architecture with respect to three characteristics: complexity of driver development, impact on driver reliability, and performance.

9.1 Code complexity

Our Dingo drivers for the AX88772 and InfiniHost devices are based on the corresponding Linux drivers, which allows us to directly compare the two implementations. The main difference we identified is that Dingo dramatically reduces the amount of synchronisation code in drivers. As pointed out in Section 4, event-driven drivers still need to synchronise activities that interleave at the message level, but situations where such synchronisation is necessary are uncommon, compared to multithreaded drivers. Table 4 summarises the use of synchronisation primitives in Dingo and Linux drivers. It shows the total number of synchronisation objects used by the Dingo and Linux versions of the AX88772 and InfiniHost drivers, as well as the total number of code sections protected by these objects.

We also found that our event-based preprocessor was effective in addressing the stack-ripping problem. Whenever a Linux driver performs a blocking call to wait for an I/O

completion or a timeout, the Dingo driver achieves the same effect using the `CALL` construct. Thus, Dingo drivers implement the complete functionality of Linux drivers without increase in code size or complexity.

Finally, we found that formally modelling driver protocols leads to simpler protocols and hence simpler drivers. For instance, in Linux drivers it is possible to receive a shutdown request while the driver is trying to put the device to a low-power state, which is a situation that requires complex code and increases the likelihood of introducing bugs. In Dingo many such situations are simply ruled out in the protocol and can, therefore, never occur. For example, this specific situation has been ruled out in the `PowerManagement` state machine in Figure 5. Such protocol restrictions are enforced by the Dingo runtime framework, which relieves the driver developer from handling these tricky corner cases and, as we will see in the following section, prevents bugs, without sacrificing useful functionality.

9.2 Reliability

In order to measure the effect of the Dingo architecture on the rate of software faults in drivers, we evaluated the AX88772 and InfiniHost drivers against a sample of bugs found in similar Linux drivers. For every bug studied, we determined whether an analogous bug could be reproduced in a Dingo driver. Some bugs simply cannot occur in Dingo, for example, most race conditions cannot be reproduced due to the event-atomicity guarantee. Likewise, Dingo protocols rule out some corner-case situations along with bugs that occur when handling them.

For those bugs that can be reproduced in Dingo, we established whether the incorrect behaviour caused by the bug is explicitly forbidden by driver protocols. We note that, while Dingo does not eliminate bugs caused by incorrect implementation of a protocol, the probability of introducing such bugs in Dingo is smaller than in Linux due to the presence of a clear and complete specification of the protocol. If, however, a protocol violation bug slips into the driver implementation, it can be detected using runtime or static verification, as discussed in Section 7.

We used bugs from the four USB-to-Ethernet adapter drivers from Table 1 and analysed them against the Dingo implementation of the AX88772 driver. We also used the 123 bugs found in the Linux InfiniHost driver and analysed them against the Dingo version of the same driver. Of the 201 bugs found in these drivers, we selected the 61 that belonged to the types of bugs that we are targeting, namely concurrency faults (29) and software protocol violations (32) and that were applicable to the AX88772 and InfiniHost drivers (some Ethernet driver bugs were not applicable to the AX88772 driver due to differences in the device interface).

The results of the evaluation are summarised in Table 5. Of the 61 selected faults, 36 fell in the category of faults not expressible in Dingo. Of the remaining possible faults 13 were protocol violations whose likelihood is reduced in Dingo. Being manually introduced in the corresponding Dingo driver, these faults could be identified by the runtime failure detector during testing. Finally, 12 bugs were deemed equally likely to occur in Dingo drivers and native Linux drivers. These bugs were software protocol violations that were not captured by Dingo protocols. They included situations where incorrect data was passed to the driver, and where the driver returned invalid data. While the appropriate format restrictions could be expressed in the current version of Tingu, this would compromise the clarity of the specification and defeat the goal of defining driver protocols formally.

	Eliminated by design	Reduced likelihood	Unchanged likelihood
Concurrency faults	27	2	0
S/W prot. violations	9	11	12
Total	36	13	12

Table 5. Categorisation of faults based on their potential occurrence in Dingo.

9.3 Performance

We evaluated the performance overhead of the Dingo driver model using the AX88772 and InfiniHost drivers. Since our implementation is based on equivalent Linux drivers, this enables a fair comparison of the Dingo and Linux models. All benchmarks were run using a Linux 2.6.27 kernel on an 8-way (4 physical CPUs with 2 hardware threads each) Itanium 2 1.7GHz with 8GB of RAM.

For the AX88772 driver we measured throughput and latency for a varying number of concurrent network connections using the Netperf benchmark suite. Figure 9 shows results of the latency test. The Dingo driver achieved the same latency as its Linux counterpart, while introducing a small CPU overhead due to the protocol translation and request queuing inside the Dingo framework. Importantly, this overhead does not increase while going from 1 to 32 clients on a multiprocessor system. The throughput benchmark showed no difference in performance between the drivers. We omit these throughput results due to space limitations.

We used the InfiniHost driver as our second example due to its extreme performance requirements. The InfiniBand interconnect architecture is designed for very high throughput and low latency. Despite the use of zero-copy techniques, it still puts substantial pressure on the CPU, especially for small transfers. Furthermore, InfiniBand supports traffic isolation among multiple concurrent connections; therefore the InfiniBand stack in Linux is designed to avoid synchronisation among data streams.

We compared the performance of the native Linux driver and the Dingo driver running in the fully serialised and hybrid modes. We used the IP-over-InfiniBand Linux module to send IP traffic through the InfiniBand link, and measured throughput and latency with Netperf. To achieve traffic isola-

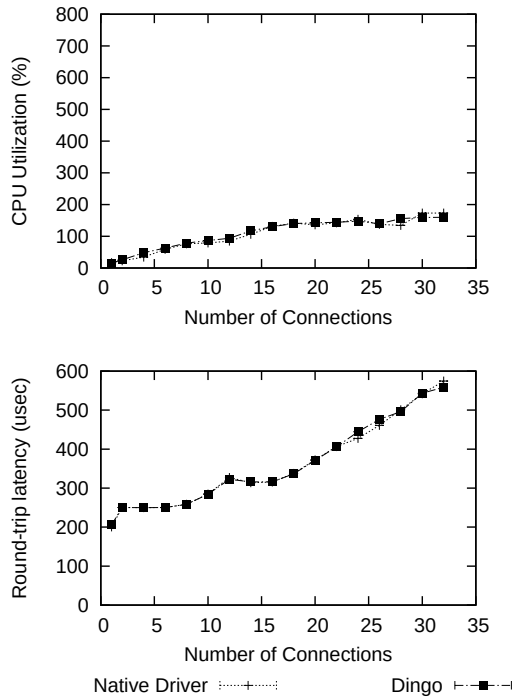

Figure 9. AX88772 UDP latency results. The top graph shows aggregate CPU utilisation over all connections (ranging from 0% to 800% on the 8-way system). The bottom graph shows average UDP echo latency across all connections.

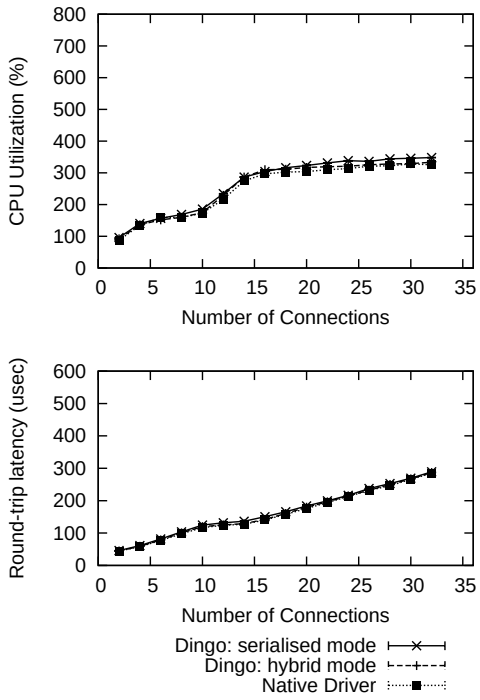

Figure 10. InfiniHost UDP latency benchmark results. The top graph shows aggregate CPU utilisation; the bottom graph shows average UDP echo latency.

tion, we configured 32 independent network interfaces, one for each client, on top of the InfiniHost controller.

As shown in Figure 10, all three versions of the driver achieve the same latency. The serialised Dingo driver shows a small increase in CPU utilisation due to request queuing. In throughput benchmarks (Figure 11), the Dingo driver in serialised mode showed 10% throughput degradation in the worst case, and less than 3% throughput degradation and no CPU overhead in the hybrid mode (in the points where the hybrid driver consumes more CPU than the native one, it sustains proportionally higher throughput). In all cases the performance of Dingo drivers scaled as well as the native Linux driver. This shows that the Dingo hybrid mode allows drivers to take full advantage of multiprocessing capabilities.

These experimental results indicate that the reliability improvement offered by the Dingo architecture does not come at the cost of performance.

10. Related work

Research on driver reliability Drivers have long been recognised as the biggest threat to system stability. A variety of techniques have been proposed for dealing with driver bugs. These techniques can be classified as either runtime or static techniques.

Runtime techniques detect, isolate and recover from driver failures at runtime, rather than eliminating the bugs that cause these failures. Most existing runtime failure detection techniques focus on detecting and isolating memory access violations. One way to deal with these failures is to encapsulate drivers in user-level processes and rely on the memory protection hardware to identify illegal memory accesses. This approach was pioneered in microkernel-based systems [Forin 1991, Liedtke 1991] and has more recently made its way into mainstream systems [Leslie 2005]. In particular, Linux, Windows, and Mac OS X allow many types of drivers to be implemented at the user level [Nakatani 2002, Microsoft 2007, Apple Inc.]. A variation of this approach was implemented in Nooks [Swift 2002], which provides memory protection for in-kernel driver.

An alternative to hardware-based isolation is software-based isolation. SafeDrive [Zhou 2006] enforces type and memory safety for in-kernel drivers written in C with source annotations through a combination of runtime and compile-time checks. Using binary rewriting, XFI [Erlingsson 2006] enforces coarse-grained protection for arbitrary modules similar to that provided by hardware-based mechanisms.

Success achieved in detecting other types of failures has been more modest. In particular, detecting device protocol violations is difficult, because formal specifications of device protocols are rarely available. Languages like Devil [Mérillon 2000], NDL [Conway 2004], and HAIL [Sun 2005] offer a compromise solution. They allow the driver developer to create a formal specification of the device interface based on the informal description found in

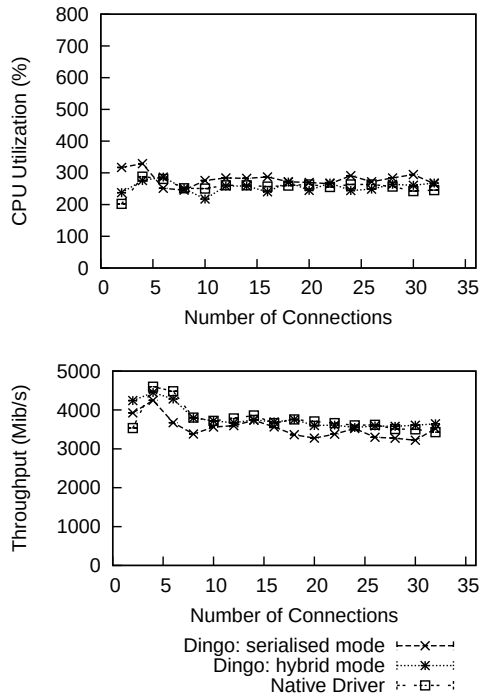

Figure 11. InfiniHost TCP throughput benchmark results. The top graph shows aggregate CPU utilisation; the bottom graph shows average UDP echo latency.

the device datasheet. This specification can be enforced via a combination of code generation, compile-time and runtime checks. This research is complimentary to Dingo, which focuses on formalising the software interface of the driver.

Verifying software protocol conformance at runtime requires a formal specification of the interface between the driver and the OS. Such specifications do not exist in current systems, which has led Nooks and MINIX 3 [Herder 2006] to implement a limited form of checking by validating only some call arguments and detecting non-responsive drivers. As discussed in Section 7, Dingo enables runtime detection of a much broader class of protocol violations.

Once a failure has been detected, a recovery mechanism must take compensatory action to hide it from the rest of the system. While Nooks has demonstrated the feasibility of driver recovery, a systematic solution applicable to many types of drivers does not currently exist. MINIX 3 avoids this problem by delegating recovery to the client of the driver.

Static techniques aim to eliminate driver bugs by analysing the source code of the driver. Unlike runtime techniques, analysis is performed at build time, and incurs no runtime overhead. Model checking tools like SLAM [Ball 2006], the Stanford Checker [Engler 2000] and Coverity [Chou 2005] have been used to find hundreds of defects in Windows and Linux drivers. While this approach has achieved much success in finding bugs, many of the more complex faults are still beyond its reach. In addition, existing techniques are ineffective in finding most concurrency faults—to find such faults requires dealing with a state space explosion caused by the multiple threads of control. The use of a model checker requires defining a set of rules that a correct driver should obey. In particular, Tingu protocol specifications could be used as such rules, which provides a bridge between our work and research into model checking.

The Singularity [Fähndrich 2006] project takes a no-compromise approach, building from scratch an OS that is amenable to static analysis. In particular, it allows the formal specification of driver software protocols and the static enforcement of protocol compliance. These improvements are tightly coupled with the Sing# language and cannot be used for drivers written in other languages or for other OSs.

Neither runtime nor static solutions provide full protection against driver bugs. Therefore, an approach, such as Dingo, that helps driver developers produce better code, containing fewer bugs, has the potential to improve both driver and overall system reliability.

Software protocol specification languages A number of protocol specification languages already exist, most noticeably UML Protocol State Machines (PSM) [OMG 2005]. Our motivation for developing yet another language was that existing languages proved inadequate for expressing realistic driver protocols clearly and compactly. In particular, neither PSM nor other languages we investigated allow the expression of protocol dependencies, and dynamic port creation.

SLAM and Singularity have both used formal languages to specify driver interfaces. They pursue different goals than Dingo and, as a result, make different language design choices. SLAM is intended to find bugs in existing drivers. Its specification language formalises individual driver API rules but does not provide means to describe the entire driver-OS interaction in a structured way. The Singularity channel specification syntax is intended to describe the complete driver behaviour; however its main design objectives are integration with the implementation language (Sing#) and compile-time verification, rather than expressiveness or readability. As a result, it does not support features like protocol dependencies and protocol variables, which we found essential for modelling driver behaviours, and does not provide the means to structure complex protocols. In contrast, Tingu separates specification from verification. Tingu specifications aim to serve as guidelines for driver developers, providing maximum information about the required driver behaviour. At the same time, they have a well-defined formal semantics and can be used as properties against which driver implementations can be formally verified.

11. Conclusion

Faulty drivers are a major cause of instability in operating systems. Our study has shown that concurrency faults and violations of the protocol between the driver and OS are significant sources of failures in a variety of drivers. We presented a driver architecture that improves OS support in these ar-

eas and thus enables simpler and more reliable drivers. We demonstrated that this architecture can be implemented efficiently in a commodity OS and is effective in reducing real driver bugs.

Acknowledgments

We would like to thank Herbert Bos, Nicholas FitzRoy-Dale, Godfrey van der Linden, Sergio Ruocco, our shepherd Petros Maniatis, and the anonymous reviewers for comments on earlier versions of this paper.

References

[Adya 2002] A. Adya, J. Howell, M. Theimer, W. Bolosky, and J. Douceur. Cooperative task management without manual stack management. In *2002 USENIX*, pages 289–302, Monterey, CA, USA, Jun 2002.

[Apple Inc.] Apple Inc. Introduction to I/O Kit fundamentals, Nov 2006.

[Ball 2006] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *1st EuroSys Conf.*, pages 73–85, Leuven, Belgium, Apr 2006.

[Chou 2005] Andy Chou, Bryan Fulton, and Seth Hallem. Linux kernel security report, 2005.

[Chou 2001] Andy Chou, Jun-Feng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *18th SOSP*, pages 73–88, Lake Louise, Alta, Canada, Oct 2001.

[Conway 2004] Christopher L. Conway and Stephen A. Edwards. NDL: a domain-specific language for device drivers. In *LCTES'04*, pages 30–36, Washington, DC, USA, Jun 2004.

[Engler 2000] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *4th OSDI*, pages 1–16, San Diego, CA, Oct 2000.

[Erlingsson 2006] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: software guards for system address spaces. In *7th OSDI*, pages 75–88, Seattle, Washington, Nov 2006.

[Fähndrich 2006] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *1st EuroSys Conf.*, pages 177–190, Leuven, Belgium, Apr 2006.

[Forin 1991] Alessandro Forin, David Golub, and Brian Bershad. An I/O system for Mach 3.0. In *USENIX Mach Symp.*, pages 163–176, Monterey, CA, USA, Nov 1991.

[Ganapathi 2006] Archana Ganapathi, Viji Ganapathi, and David Patterson. Windows XP kernel crash analysis. In *20th LISA*, pages 101–111, Washington, DC, USA, 2006.

[Harel 1987] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3): 231–274, Jun 1987.

[Herder 2006] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *Operat. Syst. Rev.*, 40 (3):80–89, Jul 2006.

[Krohn 2007] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *2007 USENIX*, pages 1–14, Santa Clara, CA, USA, Jun 2007.

[Lauer 1978] H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *2nd Int. Symp. Operat. Syst.*, pages 3–19, Rocquerncourt, France, Oct 1978.

[Leslie 2005] Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting (Rita) Shen, Kevin Elphinstone, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, Sep 2005.

[Liedtke 1991] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two years of experience with a μ-kernel based OS. *Operat. Syst. Rev.*, 25(2): 51–62, Apr 1991.

[Mérillon 2000] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for hardware programming. In *4th OSDI*, pages 17–30, San Diego, CA, USA, Oct 2000.

[Microsoft 2007] Microsoft. Architecture of the user-mode driver framework, 2007.

[Murphy 2004] Brendan Murphy. Automating software failure reporting. *ACM Queue*, 2(8):42–48, Nov 2004.

[Nakatani 2002] Bryce Nakatani. User mode drivers, 2002.

[OMG 2005] OMG. UML 2.0 specification, 2005.

[Ryzhyk 2007] Leonid Ryzhyk, Ihor Kuz, and Gernot Heiser. Formalising device driver interfaces. In *4th PLOS*, Stevenson, Washington, USA, Oct 2007.

[Sullivan 1991] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability – a study of field failures in operating systems. In *21st IEEE Int. Symp. Fault-Tolerant Comput.*, pages 2–9, Montreal, Canada, Jun 1991.

[Sun 2005] Jun Sun, Wanghong Yuan, Mahesh Kallahalla, and Nayeem Islam. HAIL: a language for easy and correct device access. In *5th EMSOFT*, pages 1–9, Jersey City, NJ, USA, Sep 2005.

[Swift 2002] Michael M. Swift, Steven Marting, Henry M. Levy, and Susan G. Eggers. Nooks: An architecture for reliable device drivers. In *10th SIGOPS Eur. WS*, pages 101–107, St Emilion, France, Sep 2002.

[von Behren 2003] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *9th HotOS*, pages 19–24, Lihue, Hawaii, USA, May 2003.

[Zhou 2006] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *7th OSDI*, pages 45–60, Seattle, WA, USA, Nov 2006.