FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Language Support for Linux Device Driver Programming

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Günter Anton Khyo
Matrikelnummer 0326024

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Prof. Dr. Dipl.-Ing. M. Anton Ertl

Wien, 30.03.2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Günter Anton Khyo

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Abstract

The success of any commodity operating system is determined by the quality of device driver support. Over the last decades, the computer hardware industry has been advancing at a rapid pace, putting high pressure on device driver developers. About 52% of the Linux kernel code is comprised by device drivers, accounting for close to 7.4 million lines of code. While managing such a huge code base is a challenge on its own, Linux device driver developers have to overcome additional obstacles. The complex, multithreaded programming model of the kernel creates a high potential for bugs and many of them result in kernel crashes. Device drivers constitute the largest and most unreliable component of the kernel.
This thesis analyses the root causes of the device driver reliability problem, and demonstrates how the current driver programming model can be improved to assist programmers in creating better driver code. To examine and test feasible improvements, a prototype language (called CiD) based on a subset of C was designed with the special requirements on Linux device driver development in mind. CiD features syntactical additions for three essential device driver code aspects: concurrency, synchronization and hardware communication. The compiler is programmed with basic rules of the concurrency model of the kernel and is able to detect simple but common mistakes that lead to deadlocks. Additional consistency checks support the programmer in generating correct hardware I/O code.
Two device drivers have been converted into CiD code to test the language extensions and the implementation of the compiler. The results for concurrency and synchronization are satisfying: race conditions in the data-flow are reported with a false positive rate of 6% to 21%. The compiler also generates correct concurrency and synchronization code, thus mitigating the potential for deadlocks. The results show that hardware I/O generator leaves much room for improvement, since it generates 1.5 times more I/O operations than the original driver. Related approaches show that further optimizations can reduce the gap.
In conclusion, we find that device driver programming can be made more robust with only minor alterations to the existing programming model and little compiler complexity.

Kurzfassung

Die immer rasanter fortschreitende Entwicklung der Computerhardwareindustrie macht die Gerätetreiberentwicklung zu einer großen, wenn nicht der größten, Herausforderung in der Betriebssystementwicklung. Der aktuelle Linuxkernel umfasst mittlerweile 7.4 Millionen Zeilen Gerätetreibercode, das sind in etwa 52% des Gesamtcodes. Während die Wartung von Gerätetreibern durch die enorme Codegröße zu einer Herausforderung wird, ist schon die Entwicklung eines einzelnen Gerätetreibers oftmals eine schwierige Angelegenheit. Neben der prinzipiellen Schwierigkeit von hardwarenaher Programmierung, verkompliziert das zugrundeliegende Programmiermodell die Treiberentwicklung. So ist Gerätetreibercode unter Linux hochgradig nebenläufig, wobei Programmierer auf zusätzliche Linux-spezifische Regeln achten müssen, um häufig auftrentede Fehler, wie Deadlocks und Race Conditions zu vermeiden. Gerätetreiber sind somit die unzuverlässigste Komponente des Kernels.
Im Zuge dieser Arbeit wurde CiD, eine Erweiterung der Programmiersprache C, sowie der zugehörige Compiler, implementiert, um die Gerätetreiberentwicklung robuster zu gestalten. CiD bietet Sprachkonstrukte für Nebenläufigkeit, Synchronisation und Hardwarekommunikation. Der Compiler wurde mit Regeln des Treiberprogrammiermodells versehen, um häufig auftretende Fehler wie z.B. Deadlocks während der Entwicklung zu erkennen.
Um die Sprache und den Compiler zu testen, wurden zwei Gerätetreiber konvertiert. Die Ergebnisse sind großteils zufriedenstellend: Race conditions im Datenfluss eines Treibers werden mit einer "False-Positive"-Rate von 6% bis 21% erkannt. Durch die korrekte Generierung von Synchronisationscode werden häufig auftretende Ursachen für Deadlocks vermieden. Die Hardwarekommunikation bedarf einer Überarbeitung: Der Codegenerator erzeugt zwar korrekten Code, allerdings werden 1.5 mal mehr Hardwareoperationen im Vergleich zum ursprünglichen Code erzeugt. Verwandte Arbeiten zeigen jedoch, dass zukünftige Verbesserungen möglich sind.
Zusammenfassend lässt sich das Programmiermodell mit einfachen Mitteln robuster gestalten, wobei der Implementierungsaufwand für die Erweiterungen sehr moderat ist.

Acknowledgements

Writing this thesis has been a tremendous experience and marks the end of a very difficult but fruitful period in my life. The lessons I learnt when writing this thesis are invaluable and will guide me through the rest of my professional and personal life.
I would like to thank my supervisor Anton Ertl for his counsel and encouragement, and, especially for his unlimited patience. I could always rely on him. Also, I would like to thank Leonid Ryzhyk who encouraged me to pursue my project and who offered counsel.
Last but not least, I would like to thank my parents and grandparents for giving me continous and unconditional support for my thesis. Without them, I would have never finished my studies.

Günter Anton Khyo
Vienna, April, 2011

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Due to the enormous growth of the computer hardware industry and the massive proliferation of computer devices, device driver development advances at a very fast pace. A recent publication of the Linux Foundation reports that the code size of the kernel increased by over 70% in only 5 years (from the year 2004 to 2009) [Kroah-Hartman et al., 2009]. The latest Linux kernel version 2.6.37 now includes close to 14 million lines of code. Device drivers constitute about 7.4 million lines of code, accounting for 52% of the kernel source code.

While the number of available of Linux device drivers is rising at an enormous rate, the reliability of device drivers has been of particular concern in the OS research community. Device drivers have a bad reputation for being unreliable, and, according to numerous investigations, are held responsible for the majority of (operating) system crashes [Ryzhyk et al., 2009a, Herder et al., 2009, Renzelmann and Swift, 2009]. In 2001, Chou et al. revealed in a detailed study on operating system errors, that the bug density of Linux device drivers is up to *seven* times higher than in all other OS components [Chou et al., 2001].

Device driver programming is challenging: hardware protocols are steadily increasing in complexity, but also the kernel and its driver interfaces are under constant change. In addition, the complex multithreaded Linux device driver programming model puts a heavy burden on programmers (Chapter 2 provides a short overview of driver programming essentials). An analysis of the root causes for device driver defects reveals that support for essential driver aspects such as concurrency and hardware I/O is very limited (see Chapter 3, “The Device Driver Reliability Problem”).

This motivates the development of the CiD (C for Drivers) programming language which offers language-support for concurrency and hardware I/O (see Chapter 4). Unlike other research approaches (see Chapter 6), CiD does not change the driver programming paradigm, but shows how language-support and static verification can aid the programmer in generating device driver code. The proof of concept implementation comes with two converted CiD device drivers which demonstrate that the language extensions and the compiler are powerful enough to detect race conditions and to prevent simple, but common mistakes, that lead to deadlocks (see Chapter 5).

CHAPTER 2

Linux Device Drivers

This chapter covers the essentials of Linux device driver programming. The basic architecture of device drivers is explained with a simple model of the operating system and the underlying hardware. The main focus of the discussion lies in the programming model of Linux device drivers which is fundamentally different from that of user-space applications. Important aspects such as concurrency, synchronization and memory management are discussed. The concepts introduced in this chapter give an understanding of the special requirements on device driver programming and the inner workings of a device driver.

2.1 What Are Device Drivers?

A device driver is a software component that controls the operations of a physical or virtual device. Physical devices are actual hardware components and perform functions on behalf of the user or operating system, for example capturing audio or printing a file. Virtual devices have no physical representation and only exist in software, but are as useful as physical devices. Examples of common used virtual devices are RAM disks, software RAIDs, virtual network adapters and so on.

It is helpful to categorize devices that offer the same set of features into device classes. The actual *mechanisms* that *realize* the interface of a class are encapsulated in the corresponding device driver. This enables operating systems to treat all devices that belong to the same class in a uniform way. Applications and other parts of the operating system can then issue requests to a device via its device class interface without knowing anything about the details of the device chipset. A device driver can be seen as an interpreter which translates class-specific requests into actual device-specific commands.

Device drivers are not restricted to the handling of a single device, in fact, many device drivers are capable of handling multiple devices belonging to the same device class. For example, a single generic USB input driver may handle a wide range of (connected) input devices like mice, joysticks or keyboards at the same time.

2.2 A Simple Device Driver Model

Figure 2.1 depicts a simple model that illustrates the data flow between a UNIX-style operating system, a device driver and its associated device.

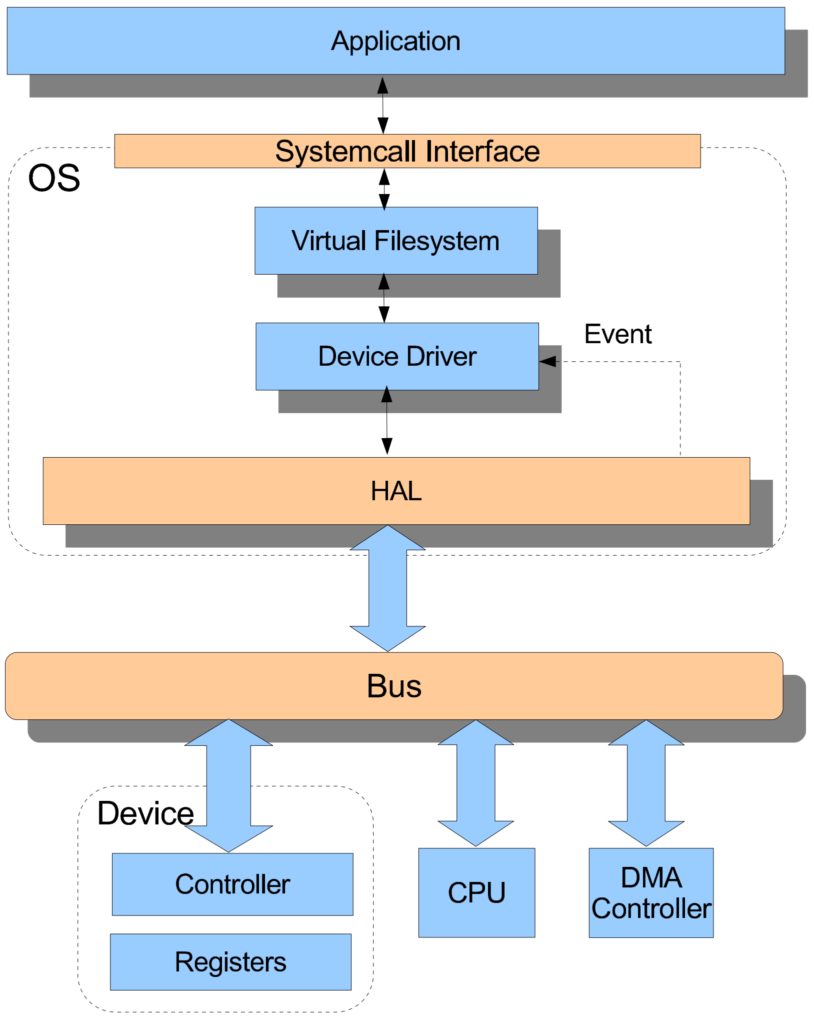

Figure 2.1: Top-down view of the data flow between an application, the underlying operating system, the device driver and its associated device.

One of the key philosophies of the UNIX operating system is that "‘everything is a file"’. Linux, being a UNIX derivate, also embodies this idea. Applications can gain access to a particular device by invoking generic I/O operations on the file that represents that device. For example, the output data of a mouse can be retrieved by reading the `/dev/input/mouse` file on any recent Linux system. In practice, the philosophy is not always obeyed. For example, network devices are not represented as files but as `interfaces`, and thus, requests do not pass the VFS. The same is also true for configuration requests that change the operational parameters

of a device. For the remainder of the discussion, we assume that requests are passed through the file system. When starting a file operation, the application has to make use of the system call interface. When issuing the system call, a mode switch into kernel mode is performed and the request is further passed to the virtual file system. Every file has an associated driver identifier (the *major number* of the device), that points to the actual driver of the device. The request is then passed to the found device driver. The device driver then translates the request into a device-specific command. The controller of a device is responsible for translating device commands into electrical signals. For example, a hard-drive-disk controller translates seek operations into mechanical movements of the disk arm(s). Access to the actual hardware is supported by the Hardware Abstraction Layer (HAL). The HAL provides support for low-level hardware facilities like DMA, device I/O and interrupt handling.
How devices are accessed is dependent on the bus[1] the device is connected to. For example, devices that connect to USB function significantly different than devices that are attached to the PCI bus family even if they belong to the same device class.
Both, the controller and the device driver, have to keep track of device state. When an application issues a request to a device, the device driver must ensure that the device is in the correct state to process the request – or issue an error. A *protocol* specifies how and when device functions can be invoked, what their parameters are and what kind of response can be expected at which time. In this context, the word function is used in a very liberal sense. The invocation of a device driver function can be as simple as reading a register or involve sending a series of well-formatted commands to the controller.
The device driver and the controller have to handle these details. Device protocols are written informally and can be found in the data-sheet of the device (if there is one available). In addition, device driver developers also have to pay attention to bus protocols. While USB defines a message-oriented communication model, communication with a PCI device is achieved by register I/O operations. The corresponding standards define message types and register layouts respectively. Therefore, device communication can be either message-based or register-based. Also, since device drivers closely work together with the kernel and use kernel services, they also have to obey protocols imposed by the operating system.
In theory, I/O operations can be implemented synchronously. However, compared to processor speed, I/O operations are very slow. In order to avoid wasting processor resources, I/O transfers are usually carried out by a DMA (Direct Memory Access) controller. Completion of an I/O operation is usually signaled via an interrupt. Similarly, network cards notify the driver when new data has arrived. A device driver also has to deal with hotplugging events which can occur at any time. Therefore, device drivers execute in a *concurrent environment* and are typically realized as multithreaded programs.
In summary, following important observations on device drivers have been made:

- Device drivers implement hardware abstractions by hiding internal device details
- Device drivers implement device and operating system protocols

[1]For simplicity, the term bus is used to denote the concept of a data-path connected to a device, not the actual topology.

- Device drivers execute in a concurrent environment

2.3 The Linux Device Driver Architecture

The Linux device driver architecture supports a huge variety of devices and busses. Due to the vastness of the architecture, it is impossible to address all important details and subsystems. However, every device driver author should have knowledge about the three basic device classes:

- Character devices
- Block devices
- Network devices

Every character and block device is represented as a file system entry in the `/dev` directory. I/O on a block or character device is performed by accessing the corresponding file and performing standard file operations such as `open`, `close`, `read` and `write`. Character devices are *stream-oriented* and carry out I/O transfers on a byte level. Examples of physical character devices are input devices, sound cards, serial and parallel ports. Well-known virtual character devices are the random number generators `/dev/rand` and `/dev/urand`.
Block devices are *block-oriented* and process data in fixed-size multiples of a byte, called a block. In contrast to character devices, I/O operations on block devices are buffered for performance reasons. The central data-structure of block devices is the request queue which is typically ordered by an I/O scheduler to satisfy performance or fairness criteria. Storage devices like hard disk drives, USB drives, card readers are examples of block devices. Useful virtual block devices are RAM disks (most Linux distributions host them under `/dev/ram[0..N]`) and virtual storage drives for mounting image files. Also, read and write requests are usually processed and (possibly) reordered by an I/O scheduler according to a
Unlike block and character devices, network, devices do not have an entry in the `/dev/` directory. Instead, they are represented as interfaces which can be accessed with tools like `ifconfig`. The fundamental data-structure of network devices is the socket buffer (`struct sk_buff`) which holds packets for transmission or reception. The basic task of a network driver is to hand over socket buffers for transmission to the controller, and to convert received packets into socket buffers. Details about the network protocol are handled by upper layers.
Apart from character, block and network devices, the Linux architecture supports all kinds of devices ranging from CPU voltage regulators to graphics cards. In addition, the Linux kernel provides APIs for accessing all common device busses (USB, PCI, I2C) and supports communication protocols such as SCSI, SATA or Bluetooth. Also, there are numerous subsystems that provide low-level drivers which handle core aspects of device classes and leave the details to high-level drivers. Examples are the input subsystem, sound subsystem and USB device class subsystems such as `usb2serial` or `usb2ethernet`. The driver architecture is being steadily improved and extended with more support for all kinds of devices.

User-Space Drivers versus Kernel-Space Drivers

In Linux, device drivers typically execute in kernel-space as part of the kernel. Writing kernel-space drivers can be considered as a challenge because the underlying programming model is rather complex, under constant change, and unforgiving to bugs. Programming faults may easily manifest as kernel crashes and debugging device drivers can be difficult.
This essentially inspired the creation of the user-space I/O framework (UIO) which targets embedded devices used in control engineering and process automation. The advantage of UIO is that the programmer can (theoretically) implement a device driver with all available tools and programming languages. However, UIO does not support performance critical devices such as network interface controllers (NICs) or storage devices. Also, bootstrapping code such as installing interrupt handlers still has to be implemented in kernel-space. Overall, the capabilities of UIO are rather limited.
Apart from UIO, programmers may also use libraries such as `libUSB` to gain direct access to devices, but again, the kernel has to offer user-space “bindings“ to the kernel subsystem that handles the corresponding device class. Thus, libraries also do not offer a complete solution.
Because user-space support is very limited, the remainder of this chapter focuses on writing kernel-space drivers.

2.4 The Linux Device Driver Programming Model

Because device drivers are part of the kernel, programmers have to be familiar with internal mechanisms that are used throughout in the kernel. Concurrency, synchronization, memory management and hardware I/O are the essential aspects and peculiarities of every device driver. This section gives an overview on how these aspects are realized in the Linux kernel.

Kernel Modules and Driver Organization

Kernel-space device drivers are typically implemented as *kernel modules*. Modules provide a dynamic plug-in mechanism for kernel code and can be loaded into kernel-space and unloaded at any time. The module concept is particularly useful in combination with hotplugging, which enables device driver code to be loaded and unloaded on demand.
A module consists of three parts,

- an interface and data objects,
- an optional list of parameters, and
- meta-information.

The interface of a module can be divided into private and public functions. By default, a module function is private and hidden from other modules. If the programmer wishes to make a function visible to other modules, the corresponding function prototype has to be *exported* to the kernel symbol table with a special macro. Data objects, i.e., global variables can be exported analogously. Once exported to the kernel symbol table, the function (or data object)

can be readily accessed by all other modules. This is useful for organizing subsystems layers of modules. In Linux terminology, this is called *module stacking*. Module stacking is also useful for splitting device drivers into two parts: a high level driver that handles generic aspects of a particular device class, and a low level driver that contains device specific code. For example, the USB-to-serial module exploits this idea and offers a dedicated interface to various device driver modules that handle different USB-to-RS232 converter chipsets.
Module parameters allow the programmer to make a module configurable. For example, a RAM disk driver may export a parameter for the capacity which can be set during loading by the user.
Every module contains a set of named attributes that describe important meta-information such as author, description and code license. Programmers have to supply these information in the corresponding source code by using macros.
Compilation and linking of modules is a simple process and just involves writing a special Makefile which can be invoked with the well-known `make` command.

Device Driver Entry Points

Every device driver implements at least one interface that defines a set of *entry points* at which control to the device driver is transferred. For example, block and character device drivers implement I/O interfaces that are invoked whenever a user performs I/O on the corresponding file node.
At the implementation level, the device driver has to register its interface implementation with the corresponding subsystem during run-time. Listing 2.1 shows a simple example that demonstrates interface implementation and registration for a character device.

In general, interfaces are overridden by passing bindings for function pointers to to the corresponding subsystem. This is a common idiom (or workaround) for interface specialization in C code. Interrupt handlers also define entry points and are handled in a similar way. The device driver simply registers the interrupt handler with the kernel by calling the `request_irq` function and passing the address of the handler.

Concurrency

The entry points of a device driver are typically invoked concurrently in an arbitrary order. Multiple processes might perform I/O on the same device, the device itself might raise an interrupt which triggers the execution of an interrupt handler and with the advent of hotplugging, a device can be disconnected at any time. In addition, since version 2.6, the kernel is now preemptive to exploit symmetric multi-processing architectures which have become standard in today's desktop computers. This means that device driver code can be preempted at any time, and may execute on different processor cores at the same time. The Linux kernel does not protect the device driver from concurrent device driver activity. It is up to the programmer to coordinate activities and to synchronize shared data objects. To this end, the Linux driver API equips the device driver programmer with numerous facilities for coordination and synchronization.

Listing 2.1: Implementation of driver entry points and device registration.

```
/* Implementation of read function */
static ssize_t
chrdev_read (struct file *file, char __user * buffer, size_t buffer_size,
    loff_t* offset);

static const struct file_operations chrdev_fops = {
  .owner = THIS_MODULE, /* Used by the kernel for reference counting */

  /* Driver entry points. */
  .read = chrdev_read,
  .write = chrdev_write,
  .open = chrdev_open,
  .release = chrdev_release,
};

/* Ready entry point */
static ssize_t
chrdev_read (struct file *file, char __user * buffer, size_t buffer_size,
    loff_t* offset)
{
  printk("chrdev:␣read\n"); return 0;
}

/* ... Implementation of read, open, and release .... */

/*
  Module initialization function
  Allocate major and minor numbers, and register as a character device
*/
static int __init init(void)
{
  if ( alloc_chrdev_region(&major_number, 0, 1, DEV_NAME) < 0)
  {
    printk("Dynamic␣allocation␣of␣major␣number␣failed:␣'%d!\n", num);
    return -1;
  }

  cdev_init(&cdev_cmos_device, &chrdev_fops);
  cdev_add(&cdev_cmos_device, major_number, 1);

  return 0;
}
```

Execution Context

Every device driver function is associated with an *execution context*. The execution context determines the types of operations that are permitted within a particular driver, or kernel, function. There are three types of execution contexts:

1. process context
2. interrupt context
3. atomic context

When a user-space program issues a device operation through the system call interface, a software interrupt transfers control to the kernel which then locates the device driver and calls the corresponding driver function (for example the `read` function of a character device driver). Because the request is being carried out on behalf of a process, the driver function executes in *process context*. In process context, the calling process can be put into sleep if the operation takes some time to finish. This happens on various occasions, for instance, when the driver calls a blocking kernel function or when the device is busy performing another operation. Once the device or driver is ready to process the I/O request, the process is woken again.
However, there are requests and events that are not associated with a particular process. For example, device interrupts occur independently from user-space processes (even if there is a casual link) and are processed in interrupt handlers without a backing process. In interrupt context, sleeping is forbidden because interrupt handlers are not part of any process and cannot be put to sleep and resumed. Calling a blocking function in an interrupt handler results in a deadlock.
Similarly, there are other code paths in a device driver that do not permit sleeping. For example, the I/O-request-queue handler of a block driver is protected with a special kind of lock (see next section) that does not allow blocking function calls. Another example is the packet transmission entry point (`start_xmit`) of a NIC driver which also executes in atomic context. The distinction between interrupt and atomic context is important for writing correct synchronization code, and will be made clear in Chapter 4.

Synchronization

The Linux kernel offers various synchronization primitives that allow the programmer to protect shared data and to coordinate concurrent driver activities.
In general, shared resources can be either protected with locks or lock-free operations. Lock-free operations are used in special circumstances, usually to protect small data objects. For example, simple objects that can be represented as numbers (such as reference counters) or bit flags can be implemented as atomic variables. The Linux kernel offers the type `atomic_t` along with operations that support concurrency-safe reading, writing and testing of single values. Atomic operations are mapped to CPU instructions that guarantee atomic behavior.
Operations such as processing the elements of a request queue are more complex and may require locking. In general, the kernel offers blocking (*semaphores*) and non-blocking locking (*spinlocks*) primitives. The choice depends on the execution contexts. If data is only shared

between functions with process context, then semaphores are the best choice. However, if the data object is also accessed within an interrupt handler or within atomic context, then the programmer has to resort to non-blocking locks. *Spinlocks* realize non-blocking behavior with busy waiting. This puts an important constraint on code that is protected with a spinlock: it has to finish quickly and it should not block. In the best case, calling a blocking operation with a spinlock leads so substantial system performance degradation, and in the worst case to a deadlock.
Another important factor that has to be considered is the performance of a locking scheme. Due to the privileged role of device driver code, a single device driver might significantly slow down the operating system with wasteful locking. The kernel features different variants of semaphores and spinlocks that are optimized for reading or writing. Analyzing lock contention and read-to-write ratio of shared data-objects is thus important in determining an optimal locking strategy.
In essence, implementing a correct locking scheme can be very tricky. Optimizations are even more difficult and dangerous but crucial for performance-sensitive device drivers. In fact, studies on operating system errors show that concurrency-related bugs such as deadlocks are very common in device driver code [Chou et al., 2001, Ryzhyk et al., 2009a]. More details about locking are discussed in Chapter 4.
A device driver also has to *coordinate* concurrent activities. To this end, the kernel offers *completions* and *wait queues*. Completions provide a mechanism to wait for the completion of an event, e.g., an I/O operation. While this can be also achieved with semaphores, there were many cases of race conditions[2], which motivated the introduction of the safer completion mechanism. Internally, completions are implemented with wait queues, which are more powerful and allow multiple processes to synchronize on events on a queue.

Interrupt Handling and Deferred Work

While an interrupt handler is executing, no other process or kernel code path can be active on the local CPU at the same time. Thus, an interrupt handler should pass control back to the kernel as fast as possible and *defer* CPU-intensive work to a later point in time.
Conceptually, an interrupt handler can be *divided* into two parts: top- and bottom-half. The code in the top-half constitutes the performance critical part and should terminate as soon as possible. The bottom-half is scheduled at a later point in time, after the interrupt handler terminates and includes code that may block or cause unacceptably high latencies. For example, the top-half of a NIC interrupt handler clears the interrupt status flags of a device, determines how the interrupt should be handled and defers the corresponding operation (such as processing received packets) to a later point in time. The Linux kernel provides three mechanisms to defer work: *tasklets*, *work queues* and *timers*.
Tasklets are essentially functions that are called in atomic context within a separate execution trace. They have higher priority than any other process and should terminate swiftly in order to avoid high latencies. In contrast, work queues do not have this constraint because they are

[2] Bovet and Cesati [Bovet and Cesati, 2005] give an example which will be paraphrased as follows: Suppose there are two concurrent code paths (A and B) in a device driver. A has to wait for B. A creates a temporary semaphore S, passes its address to B, and waits by calling `down` on S. B signals completion with `up`. A can now resume execution and delete the temporary semaphore. However, on a multiprocessor system, A and B might call `down` and `up` at the same time, with A deleting the semaphore while B is still executing the `up` function.

associated with a pseudo process and therefore execute in process context. All operations are permitted within a work queue function, which makes them less restrictive than tasklets. Work queue functions are scheduled by a worker thread which is run on each CPU. The next function on the queue can be only processed once the previous function has finished execution. This is the drawback of work queues: a blocking function stalls the entire queue.
A common use for timers is to terminate a pending operation if the device does not respond after a specified time interval. There are two important constraints on timers. First, the execution of a timer function might be delayed significantly if the system is under heavy load. Therefore, timers should not be used for real-time sensitive code. Second, timer functions execute in atomic context and must not block.

The Memory Model and Memory Management

Compared to user-space applications, the memory model used by kernel modules is rather complicated and requires careful programming. Device drivers programmers have to distinguish between different *memory zones* and *address types*. In addition, the kernel offers various memory management functions the programmer can choose from.

Memory Zones and Address Types

The Linux kernel divides memory into three zones:

- DMA-capable memory,
- normal memory and,
- high memory.

The DMA memory zone constitutes of memory that resides within the first 16 MB of main memory. DMA buffers for devices that support only 24 bit addressing (such as legacy devices that use the ISA bus or even poorly designed PCI devices) have to be allocated from this zone.
In most cases, all other memory is allocated from the normal zone [Jonathan Corbet and Kroah-Hartman, 2005]. DMA-capable memory and normal memory constitute *low memory* where kernel code and data structures reside. Low memory *pages* are always mapped to main memory and be freely accessed by kernel code (or within a kernel module). In contrast, high memory addresses are not directly accessible by kernel code and have to mapped explicitly into the kernel page table. Usually, a device driver has to deal with high memory when transferring data from user-space into kernel-space. Listing 2.2 illustrates this with a simple example: The read function of a character device has to fill a buffer provided by the user-space application with data; since user-space addresses might refer to high memory addresses, a special function has to be used to make the buffer accessible.

The exact layout of the memory zones is architecture dependent. For example, on 64-bit x86 systems, there is no distinction between high and low memory. However, a portable device driver should not make any assumptions on the underlying architecture and always use safe functions to handle different memory addresses.

Listing 2.2: Transferring memory between kernel and user space.

```
static ssize_t
chrdev_read (struct file *file, char __user * user_buffer, size_t buffer_size
    , loff_t* offset)
{
  char* device_data; int bytes_read;
  /* Read device data  ... */

  /* ... copy data to user space. */

    /* DON'T use memcpy ...*/
    /* memcpy(user_buffer, device_data, bytes_read); */

    /* ... use copy_to_user */
    copy_to_user(user_buffer, device_data, bytes_read);

  return byte_xferred;
}
```

Unlike user-space applications, which only know one type of memory address, the kernel distinguishes between four address types:

- physical addresses,
- bus addresses,
- kernel logical addresses,
- kernel virtual addresses, and
- user-space virtual addresses.

Physical addresses identify locations in main memory and are the result of MMU translations. Bus addresses are used by devices and in DMA operations. On some architectures, there is no distinction between bus addresses and physical addresses. However, architectures that are equipped with I/O memory management units (IOMMU) have a separate address space for device I/O.
Kernel logical addresses refer to a linear, physically contiguous address space (low memory) and can be translated into physical addresses by left-shifting `PAGE_SHIFT` bits. The result of mapping a page from high memory into kernel space is a kernel virtual address. No assumption on the layout of kernel virtual address should be made.
Finally, user-space virtual addresses are regular memory addresses used within user-space applications.

Memory Management

Efficient memory management is an important aspect of device driver code. Unlike user-space applications, kernel code does not use the standard C allocators `malloc` and `free` to manage memory. Instead, the kernel provides three basic memory allocator functions: `kmalloc`, `__get_free_pages` and `vmalloc`.
The most widely used allocator is `kmalloc`. It can be used to allocate memory from the normal, DMA or high memory zone. Typical uses for `kmalloc` are dynamic allocation of data structures such as ring buffers or DMA buffers. One important difference to `malloc` is that the programmer has to pass one additional argument, an allocation flag, to `kmalloc` which determines where memory is allocated from and whether the operation is allowed to block. The most common used flags are GFP_ATOMIC and GFP_KERNEL which both allocate memory from the normal zone, but the former flag forbids blocking.
The allocator `__get_free_pages` is used to allocate larger memory areas, i.e., multiples of pages (typically 4 KB). There are only a few drivers that make use of page allocation. Common uses are buffer allocation for downloading device firmware or receiving large amounts of data. Depending on the amount of main memory available, `vmalloc` can allocate up to a few gigabytes of memory. The use of `vmalloc` is also rare. For example, virtual device drivers such as RAM disks make use of `vmalloc`. In addition, the kernel also offers optimized allocator functions for frequent allocation (and deallocation) of small memory regions.

Hardware I/O

Communication with a device is either achieved by performing I/O on registers or by sending messages to device endpoints. The method of communication determined by the underlying bus. For example, the PCI bus maps device registers into (main) memory for direct access, while USB defines a message-oriented communication model with host (the USB controller) and client (the device).
There are two methods for accessing device registers: *port-mapped* and *memory-mapped* I/O. Ports are special memory regions that are accessed with dedicated CPU I/O instructions. In contrast, memory-mapped I/O locations are accessed with the same instructions as all other memory locations. The difference is important because the CPU and compiler might reorder memory instructions which may affect the correctness of the device I/O code. The solution to this problems are *memory barriers* which prevent reordering. Port-mapped I/O access does not come with these problems.
For port-mapped I/O the Linux kernel provides the functions

```
outb(u8  value, unsigned long port_number);    u8  inb(unsigned long port_number);
outw(u16 value, unsigned long port_number);    u16 inw(unsigned long port_number);
outl(u32 value, unsigned long port_number);    u32 inl(unsigned long port_number);
```

to read and write values from I/O ports at the specified port addresses. Before access, port addresses have to be claimed and released with the `request_region` and `release_region` functions, respectively. User-space applications can also gain access to I/O ports via the `/dev/ports`

file node.
I/O to memory-mapped regions is best performed with the following functions

```
iowrite8(u8  value,  void __iomem* addr);    u8  ioread8(void __iomem* addr);
iowrite16(u16 value, void __iomem* addr);    u16 ioread16(void __iomem* addr);
iowrite32(u32 value, void __iomem* addr);    u32 ioread32(void __iomem* addr);
```

Linux also exports the `/dev/iomem` file for memory-mapped I/O access within user-space.
There is no uniform interface for message-oriented communication, since the semantics are bus-dependent. For example, the USB layer supports both, synchronous and asynchronous message communication. Programmers who are used to programming client/server systems will find themselves familiar with message-oriented device I/O code. While the details differ, the concepts are essentially the same.

2.5 Further Reading

This chapter only covered the bare basics of Linux device driver programming. More information about the programming model can be found in the third edition of "Linux Device Drivers" book by Corbet, Rubini and Kroah-Hartman[Jonathan Corbet and Kroah-Hartman, 2005]. It is considered a classic in the kernel community. A hands-on approach to device drivers can be found in Venkateswaran's "Essential Linux Device Drivers" which covers a broad range of device classes with motivating real-world examples [Venkateswaran, 2008]. Cooperstein's "Writing Linux Device Drivers" is the most up-to-date book on device drivers and is a helpful yet incomplete reference [Cooperstein, 2010].
Since the kernel is under active development and driver interfaces are constantly changing and improving, the most up-to-date information can be found on the web. LWN.net provides weekly news about selected topics in kernel development [Eklektix, 2010]. The Linux Mailing List [Spaans, 2010] is the hotspot for most recent discussion and upcoming changes.
Last but not least, the Linux source code can be also a great value of information even though it might be difficult to know where to look for information. The Linux Cross Referencer (LXR) makes navigation easier and enables the user to search for identifiers and text segments the Linux kernel easier [Redpill Linpro AS, 2010].

CHAPTER 3

The Device Driver Reliability Problem

According to recent and past investigations on device driver reliability, two main sources of device driver defects can be identified [Ryzhyk et al., 2009a, Ryzhyk et al., 2009b, Renzelmann and Swift, 2009, Mérillon et al., 2009, Conway and Edwards, 2004]. On the one hand, there is **lack of formalization** of OS and device protocols which determine the correctness of every device driver. The notion of protocols was already introduced in Chapter 2 and will be refined in the following sections. On the other hand, current device driver programming models **lack support** for important aspects, notably concurrency and hardware I/O.

3.1 Device Protocol Violations

Every device driver is based on a protocol which describes the operations of its device. The device protocol is typically derived from an informal specification provided by the device manufacturer. Every nontrivial device protocol defines

- an interface,
- a state machine,
- events and
- data structures.

The interface is essentially a feature description of a device and comprises all operations (or *functions*) that the device supports. For example, every NIC protocol includes functions for transmitting and receiving packets.
How the interface functions are invoked is determined by the communication model of the underlying bus. In general, a function is either invoked by performing I/O on device registers or by sending and receiving messages to specific device endpoints.

The state machine captures the internal state of the device (usually only the part which is relevant to the device driver programmer) and possible state transitions. It puts *ordering* and *timing constraints* on the usage of interface functions and defines data formats for exchange. For example, the packet transmission function of a NIC may be only invoked once the transceiver has been put into the correct state. Usually, devices also generate events that are triggered on various conditions, for instance, on the completion of an operation or the arrival of new data. The state machine also specifies when device events are triggered and delivered.
Finally, more sophisticated device protocol also uses data-structures which are shared between the device and device driver. For example, I/O devices with high throughput rates use data structures such as ring buffers and queues to manage and process I/O requests. Other device protocols, such as the USB human interface device class, even require the implementation of parsers to marshal and unmarshall data.
A key observation is that the same device protocol usually has many instances and can be found in:

1. the device controller (as part of the firmware or an RTL specification)
2. the corresponding device specification, and
3. device driver code.

The most accurate and complete description of the functional behavior of a device can be found at the register-transfer level (RTL). Hardware modeling languages like VHDL or Verilog are used to describe the RTL of a device controller. Although the RTL specification is not important to the device driver programmer, he or she should have the same understanding of the underlying device protocol.
In general, the driver programmer obtains the device protocol from the device specification. Usually, the device specification is created by a technical writer who does not have a complete understanding of the device but has experience in the field of engineering and, ideally, writing talent. In essence, the specification is an *abstraction* of the RTL written in a natural language (typically English). Due to the nature of the writing and abstraction process, there are inherent problems with *correctness* and *completeness* [Ryzhyk et al., 2009a, Ryzhyk et al., 2009b]. Apparently minor mistakes can have large effects and lead to buggy device driver code. For example, the specification of Realtek's RTL8139C network interface controller states that reading the interrupt status register clears all interrupts [Realtek, 2002]. However, this is incorrect because the status register has to be written instead. Similarly, the revision history of Intel's specification of the 8254X gigabit Ethernet chipset shows several additions and corrections to the original document [Intel, Corp., 2009]. Interestingly, even devices that are developed according to standardized interfaces often fail to meet protocol specifications. For example, there are many controllers that do not exactly adhere to the the standard USB mass storage class specification. [Axelson, 2010, Dharm, 2010]. In consequence, a complete device driver has to deal with all these deviations to support a wide range of devices, which adds unnecessary complications.
The situation is complicated by the fact nowadays, even devices that offer simple functionality,

such as mice, have shown a drastic increase in complexity. Power management has become an increasingly important concern, adding even more complexity to an initially simple device. With the advent of hotplugging, devices can be plugged and unplugged at any time, requiring additional synchronization between concurrent device driver activities.
It comes at no surprise that device protocol violations are one of the major sources of device driver bugs. In a device driver study, Ryzhyk et al. analyzed the patch history of a selection of Linux device drivers. They found that device protocol violations account for 38% of all found defects [Ryzhyk et al., 2009a]. In conclusion, improving the creation process of device specifications and their protocols is imperative to increase the overall reliability of device drivers.

3.2 Operating System Protocol Violations

An operating system protocol describes how the operations of a device are mapped to the underlying driver model. Every device driver has to translate operating system requests into device specific commands. The underlying OS protocol determines which requests the driver has to handle and in what order they may appear. In addition, the operating system protocol may also include generic services that are needed to implement a device driver, for example hardware I/O or memory management.
The Linux operating system protocol consists of four parts:

1. core services (e.g., hardware I/O, memory management),
2. a collection of driver subsystems (e.g., networking, USB) that export interfaces,
3. their corresponding data-structures (for example, socket buffers or I/O request queues), and
4. constraints on each interface, service and data-structure.

Data-structures, interfaces and services are implemented as C functions that can be invoked within the kernel environment on behalf of the device driver. Since the C programming language does not support *design by contract*, constraints that go beyond simple type checks have to be described informally. As is the case with informal device protocols, there are inherent problems with correctness and completeness.
In their device driver study, Ryzhyk et al. have shown that OS protocol violations constitute 20% of all found device driver defects [Ryzhyk et al., 2009a]. They found that the most frequent faults are incorrect use of OS data structures, passing incorrect arguments to OS services and incorrect configuration of driver subsystems. Apart from the inherent complexity of multithreaded driver subsystems and their programming, Ryzhyk et al. find that the reason is that communication between the OS and drivers is poorly defined. Indeed, obtaining accurate information about the driver API interfaces can be difficult.
The Linux kernel comprises a huge collection of subsystems and API functions that require a lot of documentation effort which is not always properly undertaken. In fact, the main problem with the kernel documentation is that it is highly fragmented, incomplete and inconsistent. In 2007,

the Linux Foundation awarded Landley with a fellowship to improve the Linux documentation [Landley, 2008]. Landley realized that although there is a huge amount of available documentation, it is poorly structured and organized. He found that the main problem is that there is no complete, comprehensive and up-to-date documentation that can be obtained from a single location. In fact, pieces of partially redundant documentation are scattered over many different locations, and can be found in the

- kernel-source tree,
- linux-kernel mailing list (archives),
- Linux Weekly News page (`lwn.net`)
- Ottawa Linux Symposium proceedings, and in
- books and numerous other online resources.

One of the most apparent places to start looking for up-to-date documentation is the kernel source tree. However, even the kernel-source tree does not organize the contained documentation very well and contains three complementary sources of documentation: the Documentation directory, kerneldoc entries found in kernel source files and numerous help entries of the kernel configuration tool `kconfig`. Due to bad organization, neither of them prove to be a good source to get started. For example, the documentation directory contains an overwhelming number of poorly indexed plain-text files on a wide range of topics. The top-level directory is completely incoherent, containing text files about using spinlocks, the Amiga Zorro Bus, memory barriers, and so on. More specific topics are categorized and refined by subfolders. Some of these subfolders contain a central index file, others not. It is easy to get lost in the Documentation directory.

The most up-to-date information can be found in the linux-kernel mailing list. However, Landley notes: 'These days, most kernel developers consider it impossible for anyone to read all messages on linux-kernel, certainly not on a regular basis'. Indeed, the linux-kernel mailing list archive shows that over 157,000 messages were posted in 2009 [Spaans, 2010]. Although the well-known Linux Weekly News (LWN) page provides informative articles on important discussions in the kernel community, only a small fraction of all information is being covered. Attempts to provide weekly, in-depth overviews of important discussions have eventually failed [Landley, 2008].

The proceedings of the Ottawa Linux Symposium provide interesting insight into latest kernel research and the inner workings of various kernel subsystems, but the aspiring device driver programmer (or kernel hacker) has to know which topics to look for.

The arguably best source to get basic information about device driver programming and the involved protocols, is by reading one of the few available textbooks. However, many of them are already outdated and do not cover all important details of the driver API. For example, the definitive guide to device driver development 'Linux Device Drivers' by Corbet, Rubini and Kroah-Hartman [Jonathan Corbet and Kroah-Hartman, 2005] provides useful information on Linux driver programming but it is outdated and incomplete. While many sections of the driver

API are covered, many important subsystems are left uncovered.
Because kernel development moves at a very fast pace, any kind of documentation, be it books or easily updatable online resources, will inevitable face problems with incompleteness and actuality. While this can be considered a problem for any actively developed open source API, the Linux driver subsystems haven undergone many fundamental changes. Many of these changes affect the majority of the device driver code base.For example, the USB subsystem has been reworked *three* times to yield higher performance, requiring changes to all affected drivers. According to key developer Kroah-Hartman, there will be no stable API for device drivers. He argues that a stable API would make it impossible to improve kernel interfaces, and hence, also device drivers[1]. In conclusion, this makes the documentation task even more challenging.

3.3 Programming Model Weaknesses

Chapter 2 provided a short overview of the Linux device driver programming model and granted an impression of the involved complexity. In essence, tool and language support for device drivers is limited, while the requirements on driver programmers are high. The current driver programming model lacks support for two key aspects that are part of every device driver: concurrency and hardware I/O. In addition, there is no support for the separation of device and OS protocol management code which leads to readability and maintainability problems.

Concurrency and Synchronization Faults

Concurrency faults are very common in device driver code due to the complicated multi-threaded execution model of the kernel. Typically, a device driver comprises a series of entry points which are invoked simultaneously. Nowadays, almost all modern busses support hotplugging. This means that devices can be removed *at any time*, requiring additional synchronization and coordination effort between parallel driver activities. This makes device driver code very hard to read and comprehend. Another factor that leads to complications is that driver functions execute in different execution contexts.
Ryzhyk et al. conducted a detailed defect analysis based on thirteen hand selected drivers for the USB, Firewire and PCI busses [Ryzhyk et al., 2009a]. Ryzhyk et al. found that concurrency bugs account for 19% of all device drivers faults. Table 3.1 shows detailed results of their study.

As shown in Table 3.1, deadlocks comprise a significant fraction of concurrency faults. The cause for most deadlocks is surprisingly simple: “calling a blocking function in an atomic context”. There are a number of possible explanations for this. On the one hand, inexperienced programmers might not be always aware whether a function exhibits blocking behavior or if it is safe to call a blocking function. On the other hand code refactorings, changing either the behavior of a driver API function or the execution context of a function in the driver itself, might cause this kind of bug. Padieoleau et al. present a case study on this bug, in which a parameter (an allocator flag) was added to the USB messaging function `usb_submit_urb` [Padioleau et al., 2006]. The correct choice of the parameter was dependent on the surrounding execution

[1]Kroah-Hartman’s statement can be found in the documentation folder of the kernel tree. Interested readers might take a look at the file `stable_api_nonsense.txt`

Table 3.1: Types of concurrency faults in device drivers (adopted from [Ryzhyk et al., 2009a]).

Type of faults	**Occurrences**
Race or deadlock in configuration functions	29
Race or deadlock in 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 uninitialized synchronisation primitive	2
Imbalanced locks	2
Calling an OS service without an appropriate lock	1

context, and many wrong adoptions were made, leading to deadlocks. Interestingly, in some cases, programmers passed the wrong flag, even if the context should have been immediately obvious. Since the C programming model does not distinguish between different execution contexts, the compiler cannot prevent such faults, even though they are trivial to detect, as this thesis will show.
Implementation of synchronization strategies is yet another challenge device driver programmers have to face. One aspect that is particularly difficult to implement is correct locking. In general, the correctness of a locking scheme depends on the identification of all critical sections, the placement of locks, and the order in which they are taken. In Linux device driver programming, correct locking also depends on the choice of lock types. Even with the correct placement of locks, a deadlock can occur when selecting the wrong lock type. When choosing a suitable lock type, three basic rules have to be followed:

1. Never call a blocking function in atomic or interrupt context.

2. Never call a blocking function while holding a spinlock.

3. Whenever a spinlock is used within an interrupt handler, interrupts have to be disabled when taking the spinlock outside of the handler.

The first rule implies that semaphores cannot be used in interrupt handlers or in atomic context, since the lock acquisition function might block. In such cases, spinlocks have to be used instead. In particular, this means that if a shared resource is accessed within process context and interrupt or atomic context, then a spinlock has to be used in process context as well. Although sleeping is permitted in process context, doing so while holding a spinlock *may* lead to a deadlock. This might not be always immediately obvious, since code refactorings or a deep function call hierarchies might hide spinlocks. The last rule is subtle and deserves further explanation.
When using spinlocks, there are some rather subtle problems that may occur in combination with interrupts. Listing 3.1 shows a common code pattern in device drivers that uses spinlocks to protect shared data. In the sample code, there are two entry points: the interrupt handler `irq_handler` and `read` which accepts read requests from user-space processes. Both entry points perform operations on the shared resource `device_data` which is protected

from concurrent access with the spinlock `lock`. The locking code in the `read` function

Listing 3.1: Spinlocks and interrupts.

```
spinlock_t lock;

static irqreturn_t irq_handler(...)
{
  spin_lock(&lock);

    /* Manipulate device/driver data */
    some_operation(device_data);

  spin_unlock(&lock);

  return IRQ_HANDLED;
}

static ssize_t read (...)
{
  spin_lock_irqsave(&lock, irq_flags);

    /* Manipulate device/driver data */
    another_operation(device_data);

  spin_unlock_irqrestore(&lock, irq_flags);
}
```

is of particular importance in this example. The function calls `spin_lock_irqsave` and `spin_unlock_irqrestore` to acquire and release the spinlock, respectively. While the lock is taken, interrupts on the local CPU are disabled. This crucial detail ensures correctness of the locking scheme. If the `read` function used the same locking code as the interrupt handler, leaving interrupts enabled, then there is the possibility of a deadlock. If the interrupt occurs while the lock is taken in the `read` function, then the CPU will spin forever in the interrupt handler because no other activity can occur at that time. On an SMP architecture, the situation is even worse, because the code will work if the interrupt is handled on a different CPU but cause a deadlock otherwise.
Another problem that occurs in concurrent device driver code is *stack ripping*, a term coined by Adya et al. [Adya et al., 2002]. Stack ripping is common in (traditional) event-driven programming models in which computations are divided into several event handlers. For example, an asynchronous I/O operation is typically split into a function which issues the request and immediately resumes execution, and a completion handler. In this scheme, the programmer has to write glue code to establish a link between the function and its completion handler. If the I/O request operates on data of the function stack, then the contents of the stack have to be restored in the completion handler as well. The resulting code is very hard to read and maintain.
Listing 3.2 shows an excerpt from the low-performance mass storage driver **/drivers/block/ub.c**, which illustrates this problem. The purpose of the code is to (depending on the type of the

Listing 3.2: Stack ripping in the ub.c driver

```
1  static void ub_data_start(struct ub_dev* sc, struct ub_scsi_cmd* cmd)
2  {
3    /* ... */
4
5    usb_fill_bulk_urb(&sc->work_urb, sc->dev, pipe, sg_virt(sg),
6      sg->length, ub_urb_complete, sc);
7
8    if ((rc = usb_submit_urb(&sc->work_urb, GFP_ATOMIC)) != 0) {
9      /*... */
10     return;
11   }
12
13   if (cmd->timeo)
14     sc->work_timer.expires = jiffies + cmd->timeo;
15   else
16     sc->work_timer.expires = jiffies + UB_DATA_TIMEOUT;
17
18   add_timer(&sc->work_timer);
19
20   cmd->state = UB_CMDST_DATA;
21
22 }
23
24 static void ub_urb_complete(struct urb *urb)
25 {
26   struct ub_dev *sc = urb->context;
27
28   ub_complete(&sc->work_done);
29   tasklet_schedule(&sc->tasklet); /* Finish request, code not shown here */
30 }
31
32 static void ub_urb_timeout(unsigned long arg)
33 {
34   struct ub_dev *sc = (struct ub_dev *) arg;
35   unsigned long flags;
36
37   spin_lock_irqsave(sc->lock, flags);
38     if (!ub_is_completed(&sc->work_done))
39       usb_unlink_urb(&sc->work_urb);
40   spin_unlock_irqrestore(sc->lock, flags);
41 }
```

request) transmit data to or receive data from an USB mass storage device. The data transaction phase is initiated in the function `ub_data_start`. Line 8 calls `usb_submit_urb` which issues an asynchronous USB request to the device endpoint. The corresponding completion handler `ub_urb_complete` is registered at lines 5-6. While the message is being sent to the device, execution in the function resumes. In lines 13-18, the function creates a dynamic timer to

handle the event of a data transmission timeout. Upon completion of the USB request, execution continues in `ub_urb_complete` or in `ub_urb_timeout` in the case of a timeout.
The important thing to note here is that the data transfer logic is scattered over three functions (to be precise, even four, if we count the invocation of the tasklet at line 29). Note that the variable “sc” which holds information about the command and also contains a reference to the timeout handler, has to be taken from the stack and manually extracted in the timeout function and the completion handler. The resulting code suffers from readability problems due to implicit control-flow and data dependencies between different functions. The same problem also occurs in interrupt handlers which are divided into bottom and top halves.

No Separation of Concerns

The implemention of a device driver can be divided into two aspects. The first aspect implements the device protocol and communicates with the device, while the second aspect interacts with the operating system according to the rules defined in the OS protocol. While the two aspects are disjoint[2], Linux device drivers typically mix them. This has negative impact on *reusability*, *readability*, *maintainability* and *portability*.
A past study on Linux device driver code has shown that over 20 percent of Linux device driver code originates from copy&paste operations [Li et al., 2004]. According to the study, device drivers contain about 500,000 of copy&pasted code accounting for 12% of the Linux kernel 2.6.6 source code. Considering the rapid growth of the Linux kernel, it can be expected that the proportion of copy&paste code also increased significantly.
Because the programming model does not support the reuse of OS-specific code, the only options are to rewrite a device driver from scratch or to adopt, i.e., copy&paste, existing driver code. In both cases, the resulting device driver code contains a significant fraction of OS-dependent code that has to be maintained seperately along with all other device drivers. Thus, the resulting device driver code is more difficult to read. But there is another, more severe problem: if the OS protocol changes, then all affected drivers have to be (manually) modified. In 2006, Padiolaeau et al. observed that the number of referenced driver library functions per device driver of the Linux kernel version 2.6.13 has doubled since version 2.4 [Padioleau et al., 2006]. This means that device drivers contain more and more OS-specific code and are more likely to be affected by (partial) OS protocol changes. Thus, even minor changes to driver interfaces are likely to affect a large number of drivers. Padiolaeau et al. call this effect *collateral evolution* [Padioleau et al., 2006].
Until recently, either scripts that search for code patterns (based on regular expressions) or manual code editing techniques were used to make necessary modifications to the driver code base. However, with conventionally employed scripting techniques, some interface changes cannot be easily automated and require (manual) data and control-flow analysis. Also, collateral evolutions usually also introduce new bugs and slow down driver development [Padioleau et al., 2006]. It has to be mentioned however, that the recently developed semantic patcher *Coccinelle* (see Chapter 6, “Related Work”) already shows promise to solve this problem. Coccinelle has been successfully used to create a variety of kernel patches that deal with complex changes.

[2]Ryzhyk et al. demonstrated this with their device driver generator Termite [Ryzhyk et al., 2009b].

While the negative impact on maintainability can be mitigated with advanced tools like Coccinelle, there is a another problem concerning collaboration. In essence, the programming model does not support *division of knowledge and labor*. Currently, device driver programmers are required to have in-depth knowledge of device **and** OS protocols. Since the OS protocol is changing very frequently, device driver programmers have to follow discussions in the Linux community regularly. If there was a clean separation between OS and device protocols, then device experts could focus and specialize on implementing device protocols, while kernel experts could focus on implementing OS protocols. A cleaner separation of concerns has the potential to reduce costs and save time for hardware manufacturers and increase the readability and maintainability of device driver code. Also, the portability of device driver code can be vastly improved since the same device specification can be used for different operating system architectures. The usefulness of this approach has been demonstrated by Ryzhyk et al., who realize that separation of concerns is key for improvements in the device driver development process [Ryzhyk et al., 2009b]. With their prototyped device driver generator, they have shown that OS protocol implementations for Linux and FreeBSD can be interchanged without affecting the device protocol implementation.

No Support for Hardware I/O

According to Mérillon et al., code that communicates with hardware is known to be particularly error-prone. On the one hand, this is due to the addressed problems with device protocols (see Section 3.1), on the other hand this is because of the low-level nature of hardware I/O code. Hardware designers usually encode different information into a single register to efficiently use space and reduce the amount of I/O operations. Thus, individual values have to be extracted with (error-prone) bit operations such as bitmasking or shifts. Another problem is the possibility of race conditions memory-mapped I/O operations due to instruction reordering by the compiler or the CPU. Mérillon et al. found that bit operations can represent up to 30% of device driver code [Mérillon et al., 2009]. They do not, however, provide statistics on the error-rate in bit operations. Experienced device driver programmers probably write more robust I/O code. Nonetheless, what Mérillon et al. do show is that high-level language support can help the compiler to perform more accurate type checks [Mérillon et al., 2009].

Poor Support for Debugging and Testing

Debugging device drivers can be a very difficult issue. While a buggy user-space applications can be restarted after a crash, device driver bugs often result in kernel crashes. As a result, the computer (or virtual machine) has to be restarted, which can be very tedious and frustrating. Kernel debuggers such as kdb can help detecting the cause of the bug, and in many cases, the kernel prints out useful information when it encounters an error condition. However, once the system is in an illegal state, there is no way around a restart: errors like memory corruption easily affect logically unrelated subsystems or code parts. This is an inherent limitation of kernel-space drivers. Therefore, it is wise to program defensively and carefully. However, the brittle kernel environment makes experimentation with unfamiliar subsystems very difficult and in particular newcomers will have frustrating experiences eventually. A successful technique for

locating bugs is code-halving which involves removing and reinserting parts of the driver code until the bug has been found. The drawback of this method is obviously the time that has to be spent to find the bug.
Testing of device driver code is also a difficult issue because of the multithreaded programming model and the fragile kernel environment. In any moderately complex device driver, there are many interleaving code paths that might lead to race conditions and which have to be covered. Most importantly, to the best of the author's knowledge, there are no official tools and methodologies for testing Linux device driver code.

No Tolerance for Hardware Failures

In a recent study, Kadav et al. found that many drivers do not validate device data and as a result might crash if the device does not operate correctly [Kadav et al., 2009]. For example, they found that the following code fragment of the 3c59x.c NIC driver will be stuck in an infinite loop, if the device malfunctions:

```
while (ioread16(ioaddr + Wn7_MasterStatus)) & 0x8000) ;
```

At the time of writing, almost one-and-a-half years have passed and the most recent kernel version (2.6.37) still contains this line of code. Kadav et al. also discovered that drivers use unchecked device data as indices for static and dynamic arrays. Also, while some device drivers do validate the state of a device, they halt the kernel prematurely on failure instead of performing device recovery. For example, there are three cases in the RTL 8139C+ driver where the kernel is halted if a data-structure is in an unexpected state or an illegal request is passed to the driver. Semantic patches are a possible solution to this problem. In contrast, in microkernel architectures, device drivers can be simply restarted, thus, no code accomodations are necessary [Herder et al., 2009].

3.4 Towards Solving the Reliability Problem

In summary, three main causes for the device driver reliability problem can be identified:

- increasing OS and device protocol complexity,
- no formalization of OS and device protocols, and,
- high complexity of the underlying programming model.

Due to the increasing complexity of OS and device protocols, formalization has become a requirement to effectively prevent protocol violations. The lack of formal methods is a problem which affects the reliability of device drivers belonging to any operating system. With respect to Linux, it has been demonstrated that the programming model is very complex and does not provide support for crucial aspects of device driver programming. The absent support for synchronization adds additional complications to the control and dataflow in device driver code. An equally important problem is that there is no separation between device protocol and OS protocol implementation. Therefore, device driver progammers have to knowledgeable in the area of kernel development and have to be familiar with the internals of devices as well. Considering the rapid pace at which kernel development moves, keeping up-to date with the latest changes is not as easy as it sounds.

Solving the reliability problem is a complex issue. In essence, the way device drivers are written has to be completey changed. First and foremost, it is evident that there is need for a formal basis for device and OS protocols. Also, the implementation process has to be simplified, since the current programming model puts too much of a burden on driver programmers. What makes the problem particularly complex is the fast rate at which devices with new features are put onto the market, requiring increasingly complex drivers. However, even without new devices, fitting or recreating existing drivers according to a new paradigm is yet another difficult challenge.

The situation is not hopeless, however. There are already a number of promising technologies which are discussed in detail in Chapter 6. However, instead of creating a new paradigm for device drivers, the next chapters will show how the current programming model can be made more robust and which improvements can be expected for future device drivers.

CHAPTER 4

The CiD Programming Language

This chapter introduces CiD (**C** for **D**rivers), an extended subset of the C programming language which reflects the device driver programming model of the Linux kernel. CiD provides built-in support for three major device driver code aspects: concurrency, synchronization and hardware I/O. In addition, there is support for reusing code patterns in device driver code. A simple template language is used to capture these patterns.
The CiD compiler is programmed with a set of rules that define legal operations in concurrent device driver code. Because syntactical enhancements allow the compiler to derive more accurate information about the concurrent control flow of a device driver, the compiler can effectively apply kernel-specific concurrency rules to perform more rigid consistency checks. As a result, the compiler is capable of detecting race conditions and operations that lead to deadlocks.
Support for synchronization is twofold. CiD offers a Java like `synchronized` construct that can be used by the programmer to mark critical sections in device driver code in order to protect access to shared resources. The code generator infers the correct locking primitive for each of the critical sections, which reduces the potential of deadlocks. Since locking can be an expensive operation, CiD also supports the generation of lock-free atomic operations. The compiler translates arithmetic and bitwise operations on atomic variables into atomic operations.
Also, CiD supports the programmer in generating device I/O code by providing means to describe and manipulate low-level data layouts such as message descriptors and device registers. Additional consistency checks assist the programmer in writing more robust low-level code.

4.1 Basic Language Design

The design of CiD is motivated by the question how programmers can be assisted by a compiler in creating more reliable device driver code. Unlike other (promising) research approaches, CiD builds on the traditional Linux driver programming model instead of changing it. Since all kernel developers are intimately familiar with the C programming language, CiD is based on a C99 subset with extensions that provide demonstrable, practical support for (Linux) device driver programming. Thus, programmers that are familiar with C should find themselves also familiar with CiD. In order to avoid unnecessary complications, an informal description of the language will be provided. The basic subset which CiD builds on includes:

- Arithmetic, logical, relational and bitwise expressions,
- Control flow statements,
- Functions,
- User-defined types, i.e., `structs` and `typedefs`
- Unparameterized preprocessor macros

CiD features all arithmetic, logical, relational and bitwise expressions except shorthand operators. Operator precedence follows the definition found in the C standard. Control flow statements are limited to `if-else` and `while` statements. There is also support for `gotos` which are used throughout in the kernel in conjunction with error-handling.
The type system has been adapted to the needs of device driver programming. As part of the adaption process, some types have been entirely removed. Since floating point arithmetic is forbidden in kernel space, there is no support for the classical `float` and `double` types.
The handling of integer types has been simplified and made more flexible. The C99 standard defines a rather confusing set of integer types with their actual sizes varying from platform to platform. Because this makes portability of data layouts cumbersome, the Linux kernel provides its own set of integer types which are architecture-dependent `typedefs` that follow a clear and consistent naming convention. While the CiD compiler uses the Linux types to generate code, they were not explicitly integrated into the language. Instead, CiD offers arbitrary sized integers that are translated into their closest-fitting native types. This feature is useful when describing low-level data layouts and allows the compiler to perform additional consistency checks (see section 3.3).
Unlike C, CiD does not support arithmetic on pointer types to prevent potential memory access faults. Instead, pointers have been replaced with C++ like references to enable the programmer to define aliases and call-by-reference semantics for function parameters. Cases where pointer arithmetic can be useful (such as manipulating device memory) are dealt with the hardware I/O features CiD provides.
CiD offers five new user-defined types: *descriptors*, *register files*, *flags*, *templates* and *perdevice contexts*. The types will be introduced in the following sections. In addition, there are new function modifiers that denote execution context and driver entry points (see section 3.3).
The initial design of CiD also employed automatic memory management with a scope-based memory model to deal with memory leaks. This model allowed the compiler to allocate and cleanup a resource based on the life-time of the surrounding scope. While this approach seemed reasonable at first glance, the main problem is that programmer-defined error-handling for failed allocations was not supported. However, CiD offers a compromise and provides the C++ inspired `new` and `delete` operators for which the compiler infers suitable memory allocators.

4.2 Support for Code Reuse

Device drivers that handle the same device class share common code patterns which are also known as “boilerplate” code. Boilerplate code typically comprises allocation and initialization of device-class-specific data structures and resources, registration with the corresponding device class subsystem and cleanup code. However, boilerplate code may be also more complex and include device-class specific logic, such as ring buffer management for a NIC driver. It is common practice to use existing device driver code as reference and to copy, paste and adopt similar code sections. The drawback of this approach is that changes in the original code have to be propagated to all copy and pasted drivers. In contrast, CiD provides a cleaner way to reuse code with templates. Every CiD device driver includes a template which contains skeleton code for a specific device class.
Listing 4.1 shows a fragment of the CiD template declaration for a PCI-based NIC driver. Every template definition consists of a list of attributes, overridable functions (indicated by the `request`-modifier) and external functions defined in the Linux kernel. A template may also include type definitions as shown in the first line by the opaque type `sk_buff_t`, a handle to a socket buffer. Opaque types are similar to C typedefs, but they do not inherit operators from their base types, except assignment, equality and inequality operators.
Listing 4.2 shows an excerpt from the template definition, containing part of the boilerplate code. Every template is divided into sections, containing native C code mixed with template expressions. For every overridable entry point, there must be an equally named section. A template must also include an `include` and a `header` section (lines 1-12 and lines 14-22) which constitute of #include directives and function declarations, respectively. This makes it easier for the compiler to merge different template files together. The `probe` section (lines 24-52) contains boilerplate code to preinitialize the device.
Preinitialization consists of multiple steps which are the same for every NIC driver, for instance, allocating a custom data container for each device instance (we call that a perdevice context), allocating and registering a handle to the network layer, setting DMA attributes, and so on. Template code is parametrized with template expression which are delimited with matching pairs of $-tokens. During translation, the template compiler replaces applied template variables (expressions) with their bound values. Template variables can be either defined as attributes or function parameters. The template language comes with a set of predefined variables. These include code insertion variables and constants like the name of the device driver. Internally, the compiler divides the code of every overridden function into entry point `$:entry$`, the “actual” code `$:code$`, and cleanup code `$:exit$`. This division is necessary, because unlike the C89 standard, CiD allows variable declarations mixed with statements. Thus, the `$:entry$` variable contains declarations, while the remaining code resides in `$:code$`. Initialization and cleanup of embedded perdevice data structures such as tasklets (see section 4.3) resides in the variable `$:perdevice_init$` and `$:perdevice_cleanup$` (not shown in the example), respectively.
Listing 4.3 shows how the template can be used in a CiD driver. Upon inclusion of the file **pcinet.cid**, the compiler looks for the corresponding template definition in the inclusion path (in this case **pcinet.tl**), and invokes the template compiler to translate the template file. The

compiler ensures that every entry point is overridden and issues an error message otherwise. As shown in the example, it is also possible to directly “include” C code. C code sections are delimited with pairs of the ’@’-token.
One interesting feature is that compiler-defined functions can be invoked within templates. These functions can then access the AST of the program, which can be very useful. For example, the function `allocator_flag` determines the current execution context and infers a suitable allocator, i.e., `GFP_KERNEL` or `GFP_ATOMIC`. This is useful for various functions that need to allocate memory, as shown for the `dma_alloc_coherent` section (line 55).

Listing 4.1: PCI NIC template declaration

```
opaque "struct␣sk_buff*" sk_buff_t; /* Socket buffer handle */

template PCINET {
/* Template attributes */
  int product_id;
  int vendor_id;
  int tx_timeout;
  unsigned int num_registers;

/* Overridable entry points */

  /* PCI hotplugging events */
  request int  probe(perdevice& dev, net_dev_t net_dev, readonly pci_dev_t
      pdev);
  request void disconnect(perdevice& dev);

  /* NIC entry points */
  request interrupt int irq(perdevice& dev);

/* External functions */
  byte[] dma_alloc_coherent(readonly pci_dev_t pdev, int size, dma_addr_t&
      dma_addr);
}
```

4.3 Support for Concurrency and Synchronization

Execution Contexts

Every device driver function is associated with an execution context which determines the types of operations that are permitted within the function. Because execution contexts are fundamental to the concurrency model, CiD provides two function modifiers that define the context of a function.

```
atomic    void f() { /* ... */ }
interrupt irqreturn_t ISR() { /* ... */ }
```

Listing 4.2: Generic initialization code for PCI-based NIC drivers

```
include /@

  #include <linux/pci.h>
  #include <linux/pci_regs.h>

  /* Supported PCI device */
  struct pci_device_id _pcinet_id_table[] = {
          {PCI_DEVICE($vendor_id$, $product_id$)},
          {0},
  };

@/

header /@
  struct pci_driver _pcinet_pci_driver = {
          .name          = "$:device_name$",
          .id_table      = _pcinet_id_table,

          .probe         = _pcinet_probe,
          /* ... */
  };
@/

probe(netdev, pdev) /@
  $:entry$ /* Variable declarations are expanded here */

  /* Allocate perdevice context */
  if ( ($netdev$ = alloc_etherdev(sizeof(_pcinet_priv))) == NULL) {
      printk("$:device_name$␣:␣Allocation␣of␣netdev␣failed\n");
      goto alloc_ether_dev_failed;
  }

  $netdev$->netdev_ops=&rtl_ops;

  $netdev$->watchdog_timeo = $tx_timeout$;

  $:perdevice_init$

  if (register_netdev($netdev$))
  {
    printk("$:device_name$␣:␣Registration␣of␣network␣device␣failed!\n");
    goto netdev_failed;
  }

  /* User defined code goes here ... */
  $:code$

  if ( (rc=pci_enable_device($pdev$))) {
          printk("$:device_name$␣:␣pci_enable_device␣failed\n");
          goto pci_enable_failed;
  }
@/

dma_alloc_coherent(pdev, size, dma_addr) /@
  dma_alloc_coherent(&$pdev$->dev, $size$, $dma_addr$, $allocator_flag()$)
@/
```

Listing 4.3: Excerpt from the 8139C+ NIC driver, showing how to use templates

```
#include "pcinet.cid" /* Include our PCI-based NIC template declaration */

PCINET.vendor_id        = 0x10EC; /* Realtek */
PCINET.product_id       = 0x8139; /* 8139C+ */
PCINET.num_registers    = 0x100;
PCINET.tx_timeout       = 6000;

/* Data per device, automatically instantiated in the template */
perdevice {
  bool vlan_enabled;
}

request int PCINET.probe(perdevice& dev, net_dev_t net_dev, pci_dev_t pdev)
{
  dev.vlan_enabled = 0;
  @printk("rtl:␣device␣attached.\n");@
  return 0;
}
```

Functions that are not declared with a context modifier execute in process context. The compiler ensures that there are no calls to blocking functions in an atomic function since this may lead to a deadlock. The algorithm is trivial as it only needs to walk through the call graph and check if there is a function that violates the atomicity property. The same rule applies to interrupt handlers. An additional rule ensures that interrupt functions are not called by any other function.

Deferred Work

Deferring computations to a later point in time is a common idiom in Linux device drivers. The kernel provides three mechanisms that realize deferred work: *dynamic timers*, *tasklets*, and *workqueues*.

CiD **unifies** these concepts with simple-to-use *deferred blocks*. Listing 4.4 shows a fragment of CiD code that handles an interrupt. The example shows a typical pattern in device driver code. The top half of the interrupt handling routine determines what caused the interrupt, acknowledges the interrupt and defers processing to a later point in time. When translating the code into native Linux kernel code, the CiD compiler performs three steps. Listing 4.5 serves as illustration and shows the converted device driver code.

In the first step, the compiler chooses a suitable mechanism for each deferred block. In the example, the choice depends on the execution context of the function `process`. If `process` is defined as an atomic function, then the deferred block will be replaced with a tasklet. If, on the other hand, `process` may block then a workqueue will be chosen instead. If the execution of the deferred block has to be delayed by some amount of time, the block can be extended with a time parameter, e.g., `defer 100ms { ...}`. Depending on the operations inside the block, Listing 4.4: Deferring work in an interrupt handler in CiD

```
perdevice {
    int pending_tx;
    queue_t request_queue; /* queue_t defined elsewhere */
}

interrupt ISR(int irq, perdevice& dev) {
      /* Top half */
        // Read interrupt status register
        // Acknowledge interrupt

      /* Bottom half */
      defer /* 100 ms */ {
          while (dev.pending_tx > 0)
          {
               process(dev.request_queue); /* process defined elsewhere */
               dev.pending_tx = dev.pending_tx - 1;
          }
      }

      return IRQ_HANDLED;
  }
```

the compiler either chooses a dynamic timer or a delayed work queue to replace the block. In the following, we assume that that no timing parameter is present and `process` is an atomic function, and therefore, the bottom half will be executed in a tasklet.
In the second step, the compiler generates the function `deferred_ISR_instance` (lines 23-33) which contains the execution code for the tasklet, i.e., the bottom-half of the interrupt handler (lines 28-31). The original deferred block is replaced with a call to the driver library that instructs the kernel to schedule a tasklet (line 18).
In the third step, the compiler creates data-structures (containers) that hold variables that were part of the original function stack and have to be moved into a wider scope (so that the tasklet function can access them). Lines 1-4 define the container `ISR_container` which stores the reference to the perdevice instance and also holds the tasklet instance. The container is embedded into the perdevice structures as defined in lines 6-9. The container and the reference to the device context is extracted from the parameter of the tasklet function `deferred_ISR_instance` (lines 24-26). Finally, the tasklet is initialized, i.e. associated with its function and container, during device initialization (line 39). Currently, deferred blocks may only access perdevice contexts; access to the variables on the enclosing function stack is not possible (with the exception of the perdevice& parameter).
In some cases, it is necessary to cancel or wait for the (pending) execution of a deferred computation. CiD offers the `path` type which represents instances of deferred computation. Variables of type `path` can be assigned to deferred blocks. Listing 4.6 demonstrates how this can be put into use. In the example, the driver sends an asynchronous USB control request to the device. In the meanwhile, the driver waits for the operation to be finished by calling

Listing 4.5: Translated driver code

```c
typedef struct {
  perdevice* dev;
  struct tasklet_struct tasklet_instance;
} ISR_container;

typedef struct {
  int pending_tx;
  ISR_container container;
} perdevice;

irqreturn_t ISR(int irq, void* data) {
  perdevice* dev = (perdevice*) data;

  /* Read interrupt status register */

  /* Acknowledge interrupt */

  tasklet_schedule(&dev->tasklet_instance);

  return IRQ_HANDLED;
}

void deferred_ISR_instance(unsigned long data) {
        perdevice* dev;
        ISR_container* container = (ISR_container*)data;
        dev = container->dev;

        while (dev->pending_tx > 0)
        {
                process(dev->request_queue);
                dev->pending_tx = dev->pending_tx - 1;
        }
}

/* Device initialization */
int __devinit init() {
        perdevice* dev;
        /* ... */
        tasklet_init(&dev->container, deferred_ISR_instance, (unsigned long)
            dev->container->tasklet_instance)
}
```

Listing 4.6: Named deferred computations in use

```
perdevice {
  completion_t compl;
}

void func(perdevice& dev) {
  // Send USB control request
  usb_fill_control_urb(..., ub_urb_complete, &compl);

  path timeout;

  timeout = defer 500ms {
    complete(&dev.compl);
  }

  wait_for_completion(&dev.compl); // Wait for the USB transfer to finish
  cid.cancel_sync(timeout);        // Shutdown timeout handler
}
```

`wait_for_completion` on the coordination structure `compl`. Upon completion of the request, the function `ub_urb_complete` will be called, indicating completion of the operation by manipulating `compl`. If, however, the request does not finish within 500 milliseconds, the request will be terminated prematurely with `complete(&dev.compl)`. After completion, the driver terminates the pending timeout operation `timeout`.
In general, named deferred blocks should be only used when absolutely necessary. Note that the CiD compiler takes care of generating shutdown code for pending deferred computations when the driver or its device is removed.

Concurrency Protocols

A device driver contains multiple entry points which are invoked concurrently. By default, the CiD compiler assumes that all entry points can be invoked simultaneously at any time. While this assumption guarantees that every unsynchronized access will be detected, it also leads to a high number of false reports. Depending on the *state* of the device driver, some entry points might be deactivated, while others are active. For example, when a network interface driver receives a power-management suspend request, the packet request handling entry points will be deactivated and only activated after a resume request.
To make the detection of race conditions more accurate, CiD features *concurrency protocols* which allow the programmer to specify the active entry points in each state. As already mentioned, the specification of such a protocol is optional and for each device driver, there may be only one protocol. Listing 4.7 shows an excerpt of the concurrency protocol of a NIC driver. Every protocol is divided into two parts. The first part contains a programmer-defined list of device (driver) states and their active entry points. For example, in the state `DRIVER_INSERT`, the only active entry point is `Module.init`. The second part captures state transitions which

Listing 4.7: Excerpt of the concurrency protocol from the NIC driver 8139cp.cid

```
protocol {
  /* Device driver states and active entry points */
  DRIVER_INSERT : Module.init;

  PCI_PROBE     :
     global     : || PCINET.probe, || PCINET.disconnect;
     perdevice  : PCINET.probe;

  NIC_RUNNING   :
     global     : || PCINET.probe, || PCINET.disconnect, PCINET.irq, /* ...
         */ ;
     perdevice  : PCINET.disconnect, PCINET.irq, /* ... */ ;

  /* ... */

  /* State transitions triggered by function calls */
  PCINET.init              -> PCI_PROBE ;
  PCINET.netif_start_queue -> NIC_RUNNING ;
}
```

are triggered by function calls or invocation of driver entry points.
According to the protocol, after calling the function PCINET.init (which registers with the PCI subsystem), the driver switches to the PCI probing state `PCI_PROBE`. Since the driver supports multiple devices, the probe and disconnect functions (`PCINET.probe` and `PCINET.suspend`, respectively), can be invoked simultaneously. They can be also invoked in parallel to themselves which is indicated by the '||' prefix. Therefore, access to global device driver data has to be synchronized. The situation is different for perdevice-data, however. Since a disconnect request is always associated with a previous probe request, access to perdevice-data is implicitly serialized. Therefore, the protocol allows to differentiate between between global and per-device scope.
If there is a deferred code block in the control path of an entry point, then the compiler extends the protocol with an additional entry point as the deferred block. Whenever the corresponding entry point is active, then all of its associated deferred computations are active as well.

Synchronization

Locking is an essential aspect of multithreaded programming and often difficult to implement correctly. As we have seen in chapter 2, the programming model for device drivers is particularly complex because of different lock types and execution contexts.
CiD offers some relief to the programmer and provides the `synchronized` keyword to mark critical sections in device driver code. The compiler associates each synchronization block with the correct lock instance and lock type. This reduces the potential for creating deadlocks due to incorrect lock types or imbalanced locks. The compiler ensures that blocking operations are not called in synchronized blocks that are replaced with spinlocks. If a synchronization block

Listing 4.8: Comparison between traditional locking code and CiD locking code.

```
spinlock_t lock;

/* Device interrupt */
irqreturn_t irq_handler(...)
{
  spin_lock(&lock);

    /* Manipulate device/driver
         data */
    some_operation(device_data)
        ;

  spin_unlock(&lock);

  return IRQ_HANDLED;
}

/* Read request - Executes in
    process context*/
ssize_t read (...)
{
  spin_lock_irqsave(&lock,
      irq_flags);

    /* Manipulate device/driver
         data */
    another_operation(
        device_data);

  spin_unlock_irqrestore(&lock,
       irq_flags);
}
```

```
/* Device interrupt */
request interrupt irq_handler(...)
{
  synchronized {
    /* Manipulate device/driver data */
    some_operation(device_data);
  }

  return IRQ_HANDLED;
}

/* Read request - Executes in process
    context*/
request ssize_t read (...)
{
  synchronized {
    /* Manipulate device/driver data */
    another_operation(device_data);
  }
}
```

contains a `return` or `goto` statement, then the compiler releases the lock, if necessary. Listing 5.2 revisits the spinlock example from chapter 3 and demonstrates how it can be solved with synchronization blocks in CiD. When inferring the correct lock type, the compiler only distinguishes between spinlocks and mutexes even though the kernel offers specialized variants of spinlocks and semaphores which are optimized for multiple readers. Theoretically, when determining the correct lock type, factors such as read to write ratio and contention have to be considered as well. However, the CiD compiler does not try to estimate those factors. This is because a simple analysis of the device driver tree has shown that in over 10,000 device driver files, about 94% of all lock instances are made up by only two types: spinlocks and mutexes. Figure 4.1 shows the distribution of lock instances of all types in the driver tree. Also, when instantiating a lock, two scopes have to be distinguished: global scope and perdevice scope. If

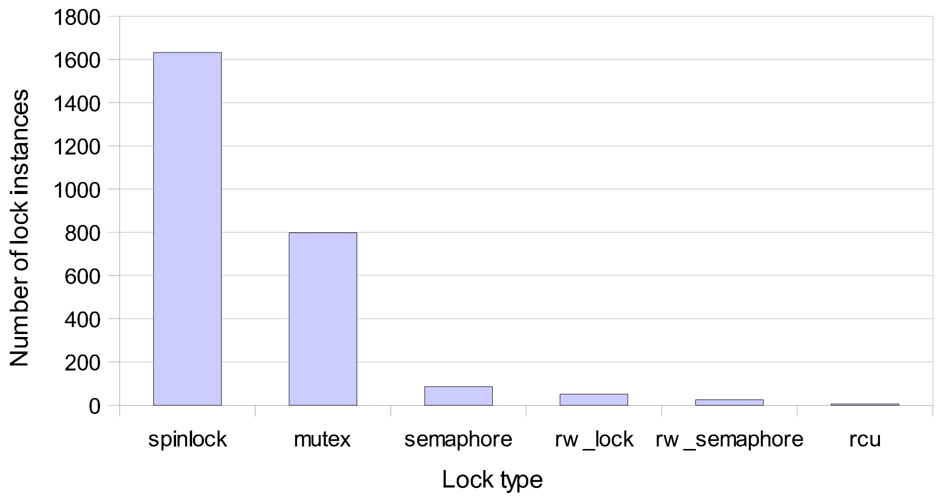

Figure 4.1: Kernel locking primitives and their uses in device drivers. Spinlocks and mutexes are the most commonly used lock types.

the protected resource is part of a device instance, then its corresponding lock is declared in the scope of the device. Although instantiating a lock in global scope is always correct, this distinction is crucial for performance.

Synchronization blocks can be also placed around function calls to protect the resources along the callgraph hierarchy, thus the following lines are perfectly legal in CiD:

```
byte shared[512];

request void read(...) { /* Read is reentrant */
  synchronized {
      f();
  }
}
void f() { /* Operate on 'shared'  */ }
```

Atomic Expressions

Atomic expressions provide an efficient way to safely operate on shared variables. Typical uses of atomic variables and expressions include keeping track of device status or counting the number of pending I/O operations. The CiD compiler is able to transform such expressions into atomic operations. The algorithm walks through all expressions in a CiD program and tries to match the corresponding operator trees with atomic operations. Table 4.1 shows all code (and operator tree) patterns the algorithm recognizes and the resulting transformations. There is one additional safety rule: there must be only a single occurrence of an atomic variable in an atomic expression.

4.4 Support for Hardware I/O

Hardware communication is an integral part of device drivers which usually involves writing (error-prone) low-level which consists of a series of bit-operations. CiD supports the programmer in writing device I/O code by providing bit *flags*, *descriptors* and *register files* that can be used to describe low-level data structures and to communicate with devices.

Descriptors

Descriptors enable the programmer to define and operate on low-level data layouts such as DMA descriptors or commands for message-oriented devices. To illustrate the use of descriptors, table 4.2 shows the data layout of the SCSI `Write(12)` command which can be used to write data to various kinds of SCSI-based storage devices. The actual meaning of the fields is not important, focus will be put onto the memory layout.

When translating a memory layout into C data-structure, three important details have to be considered: *packing*, *byte alignment* and *byte order*. Listing 4.9 shows how the `Write(12)` command might be defined in C (with Linux kernel types) and in CiD .

Although the CiD code is much more verbose, there are several benefits over the C code. Unlike ordinary C `structs`, descriptors carry more information about the actual data layout and thus, enable the compiler to perform consistency checks and to generate low-level bit code for accessing individual fields.
Every descriptor field declaration is defined in the context of a range expression which denotes the byte region (or byte position) in which the corresponding declarations reside. This allows the programmer to specify constraints on packing and byte-order. For example, the second byte of the Write(12) descriptor is occupied by the write `mode` flags and the LUN (logical unit number) of a device. The programmer can access both fields separately. In contrast, in the original C code, both fields have to be expressed with the same field declaration. Theoretically, the programmer could use bit fields to split the declaration, but the C standard does not define the ordering of bit fields. Thus, portable device drivers never use bit fields for hardware I/O transactions. In C code, when writing the variable `lun` to the LUN field, the following operations have to be performed:

```
SCSI_Write12 w12;
lun             = (lun << 5) | w12.LUN_mode;
w12.LUN_mode    = lun;
```

In CiD , the compiler is capable of generating the above bit operations, and, so the programmer can simply write:

```
w12.LUN = lun;
```

The optional bit range specifiers (at the end of a declaration) tell the compiler about the exact bit positions of packed fields.

Table 4.1: Atomic code patterns the CiD compiler recognizes, and their transformations.

CiD Code	**Transformation**
Read-and-write Operations	

int A<**atomic**>;
int num;
A = num;
num = A;
	atomic_t A;
	int num;
	atomic_set(num, &A);
	num = atomic_read(&A);

Arithmetic	

A = A + 1;
A = A - 1;
num = A + 1;
num = A - 1;
A = A + num;
A = A - num;
num = A + num;
num = A - num;
	atomic_inc(&A);
	atomic_dec(&A);
	num = atomic_inc_return(&A);
	num = atomic_dec_return(&A);
	atomic_add(num, &A);
	atomic_sub(num, &A);
	num = atomic_add_return(num, &A);
	num = atomic_sub_return(num, &A);

Test-and-set operations	

if (A=A-1, A == 0) **then**
...
end if
if (A=A+1, A == 0) **then**
...
end if
	if (atomic_dec_and_test(&A)) **then**
	...
	end if
	if (atomic_inc_and_test(&A)) **then**
	...
	end if

Bit operations	

flags(3) status_t {B1, B2, B3};
status_t status<**atomic**>;
status.B1 = 1;
status.B2 = 0;
status.B3 = ∼status.B3;
	u8 status;
	
	set_bit(0, &status);
	clear_bit(1, &status);
	toggle_bit(2, &status);

Table 4.2: Data-layout of the SCSI_WRITE12 command

↓ byte / bit →	7	6	5	4	3	2	1	0
0	Opcode = 0x2A							
1	LUN			DPO	FUA	EBP	Reserved	RelAdr
2-5	LBA							
6	Reserved							
7-8	Transfer length							
9	Control							

Listing 4.9: Comparison between C structs and CiD descriptors

```
#define RelAdr  (1 << 4)      flags(5) w12_flags {
#define EBP     (1 << 3)              RelAdr,
#define FUA     (1 << 2)              _, /* Reserved */
#define DPO     1                     EBP,
                                      FUA,
                                      DPO
                              }
typedef struct {              descriptor SCSI_Write12 {
  u8 op_code;                         0:        int(8) op_code;
  u8 LUN_mode;                        1:        w12_flags mode: 0..4;
  __le32 LBA;                                   int(3) LUN: 5..7;
  u32 reserved;                       2..5:     int(32) LBA;
  __le16 TransferLength;              6:        _; /* Reserved */
  u8 Control;                         7..8:     int(16) TransferLength;
} __attribute__((packed))             9:        int(8) Control;
    SCSI_Write12;             }
```

An important detail to consider when operating on low-level data-layouts data is byte-order. Bus protocols and device controllers do not necessarily agree with the byte order of the local CPU. Thus, byte order conversions have to be computed when communicating with a device. Programmers have to worry about byte-order only once, when defining the layout. Whenever a descriptor field is read from or written to, the compiler converts the field or the new value into the correct byte order. The compiler simply uses the Linux supplied conversion functions. The compiler uses a simple rule for determining the byte order: if the byte interval of a field declaration is ascending, its byte order is little-endian, otherwise big-endian. For example, the byte interval `2..5` of the `LUN` field denotes litte-endian byte order. The interval `5..2`, on the other hand, would denote big-endian byte order.
Finally, the compiler performs a set of consistency checks on every descriptor definition which ensure that

1. there are no overlapping byte and bit intervals,

2. there are no gaps between byte and bit intervals,
3. all fields exactly fit into their respective byte intervals,
4. the sizes of all bit intervals coincide with the bit sizes of their corresponding field data types.

These assertions ensure that every bit and byte in an descriptor is accounted for and check whether a descriptor definition is complete. If there are unused bytes or bits an descriptor, then the programmer has to use “don’t care” fields (_) to explicitly fill up unneeded space.
There is one important restriction when defining descriptors: Descriptor fields must be either scalar types or byte arrays. In particular, this means that nesting of descriptors is forbidden and there must not be reference fields.

Register files

Register files enable the programmer to define device register layouts and to communicate with devices that expose their registers to a bus. Listing 4.10 shows a CiD fragment of the register definition for the RTL8139 network interface controller. Every register is declared in the con-

Listing 4.10: CiD Register file definition for the RTL8139C NIC

```
typedef unsigned int(1) bit;

regfile RTL {
  /* Chip command register */
  0x37:
    _: 7..5;            /* Reserved */
    bit CMD_RST : 4;    /* Reset */
    bit CMD_RE  : 3;    /* Receiver enable */
    bit CMD_TE  : 2;    /* Transmitter enable */
    _: 1;
    bit CMD_BUFE: 0;    /* Receive buffer empty? */

  /* Transmit status descriptor */
  0x10..0x13, ..., 0x1c..0x1f:
    bit CRS: 31;        /* Carrier sense lost */
    /* ... */
    bit OWN: 13;        /* Own bit, 0 starts transmission */
    unsigned int(13) TX_SIZE: 12..0; /* Packet size */
}
```

text of a byte region which refers to the base address of the device. For example, the command register is mapped at offset 0x37. There is also the possibility to define register groups, which is demonstrated by the transmit status descriptor (TSD) definition. The RTL controller defines 4 identical, 4 byte status descriptors, which are mapped into a contiguous area in the register file (from 0x10 to 0x1f). Instead of defining the same set or registers all over again, programmers

can use the shorthand notation `0x10..0x13, ..., 0x1c...0x1f` to define the layout only once. The compiler performs the same consistency checks as it does on descriptors, i.e., it ensures that the register file definition is complete.
Listing 4.11 shows how the register file definition can be used to communicate with the controller. As the example shows, I/O is performed by accessing fields of the RTL namespace.

Listing 4.11: I/O interaction with the RTL controller

```
/* Initialize the NIC */
request int PCINET.start_nic(perdevice& dev, net_dev_t netdev)
{
  RTL.CMD_RST = 1;      /* Reset device */
  /* ... */
  RTL.CMD_TE = 1;       /* Enable transmitter */
  /* ... */
  RTL.CMD_RE = 1;       /* Enable receiver */
}

/* Packet transmission request */
request int PCINET.start_xmit(perdevice& dev, sk_buff_t skbuff)
{
  /* dev.cur_tx - index to current transmitter status register */

  RTL.TX_SIZE[dev.cur_tx] = PCINET.skb_data_len(skbuff);
  RTL.OWN[dev.cur_tx] = 0;
}
```

Register groups are accessed with the array index operator `[]`. The actual I/O address of the device is hidden as a field in the perdevice context, which the code generator uses as an argument to the `iowrite/ioread` functions (see chapter 2). Therefore, the register file namespaces may be only accessed within perdevice context.
How the memory address (the pointer) to the device register file is obtained depends on the bus and is realized in the corresponding code template. For example, the template for PCI NICs uses the `pci_iomap` function to create a pointer to the device registers. In order to ensure correctness of I/O code, there are three rules to which the compiler adheres. The first rule states that the size of the byte interval determines the width of the read or write access. For example, the chip command register (at offset 0x37) is accessed with the `io{write|read}8` functions, while the transmit status registers are accessed with the `io{write|read}32` functions.
The second rule states that consecutive write operations to the same register group (offset) are combined into a single write operation. In the example, the transmit function `PCINET.start_xmit` relies on this rule because the RTL specification states that the `OWN` bit and `TX_SIZE` have to be written at the same time.
The third rule ensures that unmodified register bits are preserved if the programmer writes only to a portion of a register group. The compiler issues a read-modify-write operation in this case.
One crucial detail the programmer currently has to pay attention to is the possibility of reordered device register memory accesses. This is a problem unique to memory mapped I/O, and the only

way to compensate is with memory barriers. Currently, memory barriers have to be placed by the programmer manually. Also, the programmer has to be aware of device-specific side-effects when reading or writing to registers.

4.5 Implementation

This section gives a short overview of the compiler implementation. Focus will be put onto the concurrency analysis, since it is the most elaborate and interesting part of the compiler. The other analyses are straightforward and can be browsed in the compiler source files.

Compiler Infrastructure

The architecture of the compiler is very simple and follows the classical division into front- and backend.
The frontend uses PLY [Beazley, 2010], a python port of the well-known lex and yacc tools, to scan and parse input files. The parser generates an object-oriented abstract syntax tree (AST) which is processed by analysis and transformation rules via the visitor pattern [Gamma et al., 1995].
The backend consists of the code generator and the template compiler. The code generator traverses the (transformed) AST and directly generates C statements and uses the template compiler to generate code from templates. The compiler relies on the C compiler to perform low-level optimizations such as constant folding, or common-subexpression elimination. Table 4.3 presents an overview of the most important source files of the compiler.

Table 4.3: Overview of the most important compiler source files.

Package	**Module / File**	**Description**
Core components		
	cid.py	“Executable”; wires all analysis files together
	lexer.py	Lexer
cid	parser.py	Grammar definition
cid.syntree	nodes.py	AST node classes
cid.analysis	name_analysis.py	Name analysis
cid.analysis	operator_usage_analysis	Type analysis on operators and functions
Concurrency		
cid.analysis	function_analysis.py	Calculation of root sets
	atomic_expression_analysis.py	Transformation of atomic expressions
	deferredwork_analysis.py	Replace defer blocks with tasklets, workqueues or timers
cid.analysis.concurrency	protocol_analysis.py	Preprocess programmer-supplied protocol information
	sblock_inheritance_analysis.py	Inheritance of synchronization blocks
	concurrency_analysis.py	Detection of race conditions and lock inference. Also, check for blocking calls in atomic context, or while holding spinlocks.
Hardware I/O		
cid.analysis.hwio	layout_analysis.py	Consistency checks for memory layouts, i.e., descriptors and register files
	descriptor_analysis.py	Generate bit manipulation code for field access, generate byte-order conversions
	register_file_analysis.py	Replace register file accesses with I/O code; coalesce consecutive write operations
Templates and Codegenerator		
	codegen.py	Code generator: emits C code from AST nodes
cid.codgen	template_compiler.py	Template compiler: parses template files, transforms template expressions, emits C code
	template_functions.py	List of compiler functions that can be invoked from template code

Listing 4.12: CiD example of concurrent reads and writes.

```
int a;
perdevice { int z; }

protocol {
  INIT:         || e1, e2, e3;
  NOT_E1:       e2, e3;

  stop_e1       -> NOT_E1;
}

interrupt request void e1()
{
  synchronized {
    write_a();
  }
}
```

```
request void e2(perdevice& dev)
{
  stop_e1();
  read_a();

  synchronized {
    read_z();
  }
}

request void e3(perdevice& dev)
{
  synchronized {
    read_a();
    write_z();
  }
}
```

Concurrency Analysis

The concurrency analysis has two goals: to detect and report any race conditions and to infer the correct lock instance and type for every synchronization block. Listing 4.12 shows a simple example of concurrent CiD code that will be used to explain the analysis.

The analysis comprises the following tasks:

1. Compute the set of concurrent variable accesses.
2. Based on this set, determine the set of conflicting accesses.
3. Compute which synchronization blocks belong together, and determine the correct lock type.

Instead of showing the native Python implementation of the individual steps, a more abstract language is chosen to explain the ideas behind the algorithms to avoid going into unnecessary details[1].
The first step is depicted by Algorithm 1. When the algorithm terminates, every variable is associated with a table, `concurrent_accesses`, which keeps track of read and write operations (`RWAccesses`) with respect to the current entry point `ep` and all active entry points. The set of active entry points is defined by the supplied concurrency protocol (or the default protocol with

[1] In contrast to the specified algorithms, the implementation is far more complex: On the one hand, the implementation combines all three steps into one single pass. On the other hand, the AST has to be interpreted, protocol information has to be inspected, active entry points are calculated and so on. Nonetheless, the algorithms convey the basic idea and serve as an aid to understand the reasons behind the implementation.

all entry points enabled at any time). The function `ActiveEntryPoints` computes the entry points for any given source location in the program. Table 4.4 shows the results of the algorithm for the running example. Note that since `e2` disables entry point `e1`, there are no concurrent accesses from entry point `e1` as indicated by the empty list.

Algorithm 1 Computing concurrent accesses in CiD code

```
for all $ep \in$ EntryPoints(Program) do
  for all $rw \in$ RWAccesses($ep$) do
    for all $a \in$ ActiveEntryPoints($rw$) do
      $rw$.variable.concurrent_accesses[$ep$][$a$] += [$rw$]
    end for
  end for
end for
```

Table 4.4: Results of the concurrency analysis for the program in Listing 4.12

a	e1	e2	e3
e1	[write_a]	[write_a]	[write_a]
e2	[]	[read_a]	[read_a]
e3	[read_a]	[read_a]	[read_a]

z	e1	e2	e3
e1	[]	[]	[]
e2	[]	[read_z]	[read_z]
e3	[write_z]	[write_z]	[write_z]

Algorithm 2 performs the second step and reports any race conditions. For every variable access, the algorithm checks if there is a conflict, i.e., at least one write operation happening at the same time. In this case, the access has to be synchronized. As shown in the algorithm, there are two cases of synchronization, either via a surrounding or inherited synchronization block, or if the variable is declared as atomic. Whether the atomic expression can be converted into native code is determined by the atomic expression evaluator. Note that parallel read accesses are not regarded as conflicting accesses. For example, function `e2` can safely read from `a` without synchronization.

After all synchronized accesses have been determined, the last step of the algorithm can be performed (see Algorithm 3). The basic idea of the algorithm is very simple: If the concurrent variables x and y are protected with an enclosing synchronization block S_1, then they are related. This relation is transitive: If z happens to be synchronized in S_2 along with y, then x is also related to y, i.e., S_1 and S_2 have to be replaced with the same lock instance. To keep things simple, we assume that this relation has already been computed, and the function `SyncSet` simply yields all synchronization blocks for a given variable and the current entry point information. The choice for the correct lock type is trivial. If at least one variable in the transitive closure is shared within interrupt or atomic context, a spinlock has to be used. Otherwise it is safe to use a mutex. Computing the correct scope is analogous, whereas global scope is favored over per-device scope. In the example, all synchronization blocks are replaced with a single global spinlock instance. Note that even though `e2` deactivates entry point `e1` before reading from `z`, the same lock has to be used, because `e3` synchronizes `a` along with `z`.

Algorithm 2 Detecting race conditions in CiD code.

```
for all $ep \in$ EntryPoints(Program) do
  for all $rw \in$ RWAccesses($ep$) do
    for all $a \in$ ActiveEntryPoints($rw$) do
      $acc\_list \leftarrow rw$.variable.concurrent_accesses[$a$][$ep$]

      if ($rw$ = WriteAccess $\wedge$ $|acc\_list|$ > 0) $\vee$ WriteAccess $\in acc\_list$ $\vee$ IsReentrant($a$, ScopeOf($rw$)) then
        {Conflicting access detected. Check for explicit synchronization.}
        $synchronized \leftarrow$ EnclosingSyncBlock($rw$) $\neq$ NULL
        $synchronized \leftarrow synchronized \vee rw$.variable.type = Atomic
        $synchronized \leftarrow synchronized \vee$ | $f$.inherited_sblocks | >= $f$.invocations $\wedge$ $f$.invocations > 0

        if $\neg$ $synchronized$ then
          Report race condition for $rw$
        end if

        $rw$.variable.synchronized = $synchronized$
      end if
    end for
  end for
end for
```

Algorithm 3 Compute lock instances and lock types.

```
for all $ep \in$ EntryPoints(Program) do
  for all $rw \in$ RWAccesses($ep$) do
    {Compute initial lock}
    if Modifier($ep$) = InterruptModifier $\vee$ Modifier($ep$) = AtomicModifier then
      $lock \leftarrow$ Spinlock(shared_with_irq=(Modifier($ep$) = InterruptModifier))
    else
      $lock \leftarrow$ Mutex()
    end if
    $scope \leftarrow$ ScopeOf($rw$.variable)
    {Propagate locks}
    for all $a \in$ ActiveEntryPoints($rw$) do
      for all $sblock \in$ SyncSet($rw$.variable, $ep$, $a$) do
        $sblock$.lock $\leftarrow$ ChooseLock($sblock$, $lock$)
        $sblock$.lock.scope $\leftarrow$ ChooseScope($sblock$, $scope$)
      end for
    end for
  end for
end for
```

CHAPTER 5

Experimental Evaluation

This chapter presents an experimental analysis of CiD's language extensions. In order to demonstrate the practical usefulness of the newly added language concepts, two Linux drivers have been converted: the network driver for the 8139C+ chipset [Realtek, 2002] `8139cp` and the low-performance USB mass storage driver `ub`. The two drivers have been selected because they operate on two different communication models: the NIC driver is register-oriented, while the USB mass storage driver is messaged based. The drivers serve as reference to analyse the proposed concurrency, hardware I/O and template features.

5.1 Methodology

In the course of this thesis, a broad spectrum of device drivers have been studied, ranging from simple input drivers to more complex network and block device drivers. Among these, two moderately complex drivers have been selected and converted into CiD to assess the proposed language extensions and to test the compiler implementation.
The first driver controls Realtek's 8139C+ fast-ethernet NIC chipset [Realtek, 2002] which is used in cost-effective, low-end network devices. The driver has been chosen because, unlike more complex drivers such as the widely used E1000 driver, it has a manageable code size (i.e., 2000 lines of code compared to over 10,000 lines of code) and includes all important aspects of a NIC driver. The 8139C+ driver serves as assessment for CiD's hardware I/O, concurrency and code reuse features.
The second driver that has been converted is the `ub` driver (/drivers/block/ub.c), a low-performance driver for USB mass storage devices. While this driver is simpler than the standard high-performance driver `usb_storage`, its complicated control-flow serves as a good test for CiD's concurrency and synchronisation features and their implementation.
The NIC driver has been tested with an emulated RTL8139C+ controller. For this purpose, the machine emulator QEMU (version 0.12.5) [Bellard, 2011], running the Linux kernel version 2.6.35 was used. The USB mass storage driver was tested with the 2.6.35 kernel running on a ThinkPad T60 and various USB thumb drives as test devices.

5.2 Concurrency and Synchronization

Table 5.1 shows various statistics on the converted drivers which were obtained with the CiD compiler. The remainder of this section will give a detailed analysis on the data.

Table 5.1: Statistics on concurrency and synchronization obtained with the CiD compiler

	USB Mass Storage Driver	**8139C+ NIC Driver**
Concurrency data		
Number of entry points	13	27
Reported race conditions (without protocol)	889	600
Reported race conditions (with protocol)	512	367
Falsely reported race conditions	110	40
Inferred deferred work instances	11	0
Synchronization data		
Number of critical sections	14	18
Inferred lock instances	3	1
Inferred atomic operations	8	0
Inferred allocator flags	13	5

Detected Race Conditions

The NIC and the mass storage driver have many interleaving code paths which is clearly indicated by the high number of concurrent accesses (i.e., race conditions without protocol information and synchronization blocks removed) to shared variables inferred by the CiD compiler (see Table 5.1).
Table 5.1 shows that concurrency protocols significantly improve the compiler's ability to correctly identify race conditions. With protocols, reductions of 38 to 42 % are achieved, yielding to a false positive rate of 6% to 21% for the fully synchronized driver. In the NIC driver, with all synchronization blocks in place, the compiler reports 40 false race conditions. On the one hand, these are due to 28 concurrent accesses to the netdevice handle which holds device statistics. In the current implementation, the individual statistics fields of the handle are indirectly accessed with helper functions, e.g., `PCINET.rx_ok(dev.netdev)`. Thus, the compiler does not recognize that actually separate fields are safely accessed. On the other hand, the 12 remaining race condition reports are due to corner-cases in which either race conditions do not affect correctness of the driver or are avoided by lock-free synchronization. Figure 5.1 illustrates the 12 cases with a compiler-generated data-flow callgraph. Listing 5.2 shows an example of the driver's polling routine and interrupt handler which concurrently access the status register `ISR` in a safe manner.
Accurate detection of race conditions in the mass storage driver is very difficult. This is because the driver makes heavy use of deferred work, asynchronous functions and uses implicit synchronization patterns which are hard to detect with the current implementation. Listing 5.1 shows a simple case of implicit synchronization. The compiler reports that there is a conflict for the vari-

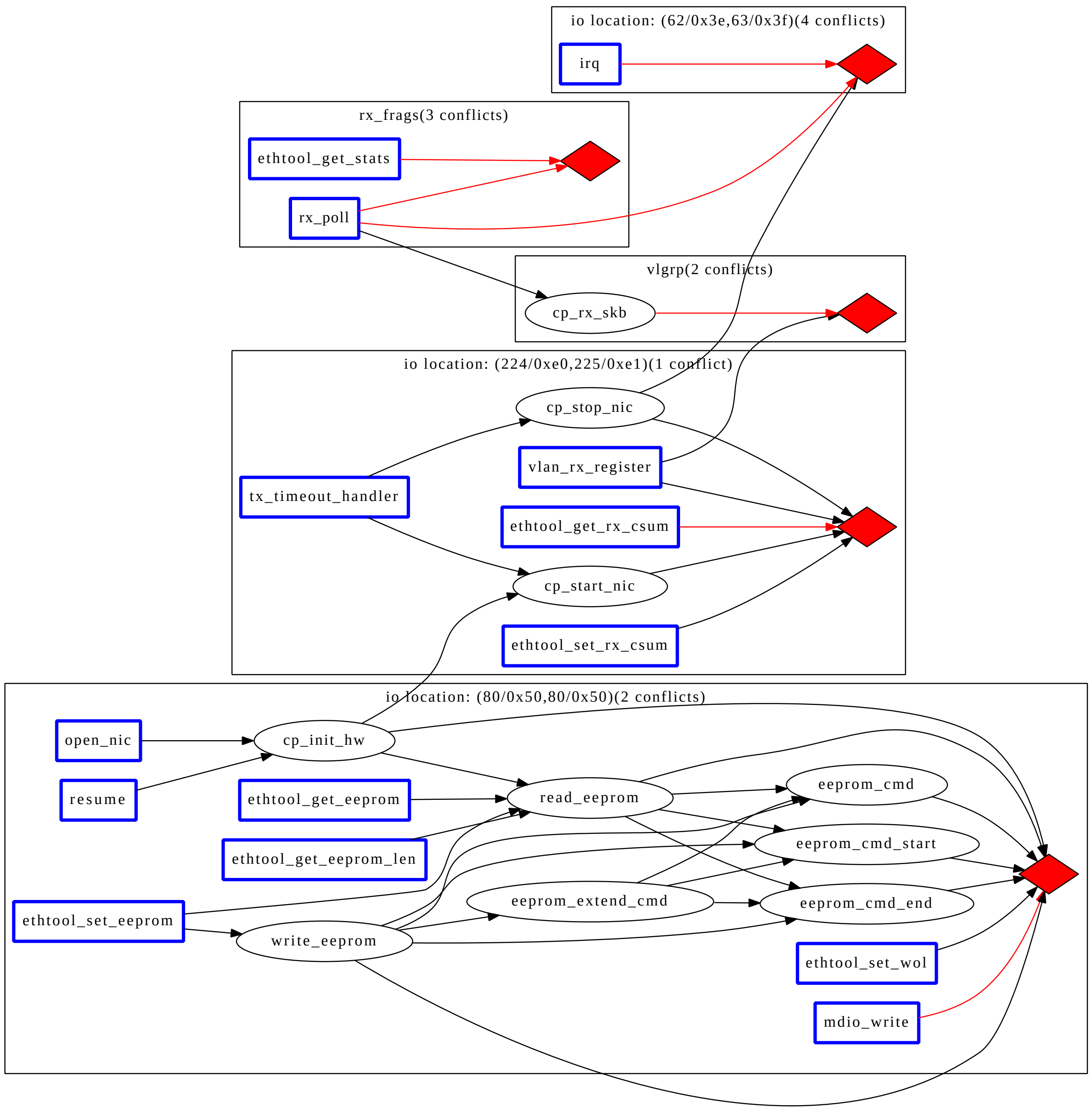

Figure 5.1: Compiler-generated conflict graph of the 8139C+ driver. Blue boxes represent driver entry points, red edges denote unsynchronized data paths.

able `dev.comp`, between the deferred computation and the waiting statement. However, the value is intended to be changed concurrently. Also, what the example does not show is that there are numerous other instances in which this pattern occurs, and, thus the number of race reports easily adds up. Theoretically, protocols could be used to express safety, but unfortunately, there is currently no way to address named deferred computations. Even if this was possible, the resulting protocol would get even more convoluted than it already is (with 16 state transitions and 5 states). However, the results could improve with language support for coordination patterns. Nonetheless, the mass storage driver represents a rather extreme case. For register-based drivers with a moderately complex control flow, the concurrency analysis results can be expected to match with the converted 8139cp driver.

Listing 5.1: A case of implicit synchronization in the mass storage driver

```
request int USB.probe(perdevice& dev, ...)
{
  /* ... */
  ub_sync_getmaxlun(dev, ...);
}

int ub_sync_getmaxlun(perdevice& dev)
{
  timeout = defer 500 ms {
    @complete@(&dev.comp);
  }

  cid.wait_for_completion(&dev.comp);
  cid.cancel_sync(timeout);
}
```

Locking

Both, the NIC and the mass storage driver, use locks as the most frequent type of synchronisation. Locking in the NIC driver is straightforward. There are 18 critical sections and all of them are protected with a single per-device spinlock. The CiD compiler infers almost all NIC driver locks correctly, with the exception of one critical section. This is due to a limitation in the expressiveness of register files, as Listing 5.3 shows. This problem can be worked around with by inducing a redundant data-flow dependency between related critical sections, i.e., reading from a shared variable or register location.
Locking in the USB mass storage driver is more complicated. Block device drivers usually use a spinlock to interact with the I/O queue and to protect critical sections. This cannot be expressed with synchronization blocks, since the programmer cannot gain access to lock instances and, for example, pass a dedicated queue lock to the blocklayer. Instead, in the converted driver code, the template takes care of instantiation of the queue lock, while the compiler infers separate lock instances for the critical sections. The compiler infers one additional lock for the per-device counter `openc` which keeps track of the number of processes which have opened the device. In the original driver, this counter is protected by the shared queue lock, but since there is not data-flow dependency to other shared variables, the CiD compiler infers a separate lock. This seems to be correct, however, in case of doubt, applying the data-flow dependency trick will work.

Listing 5.2: Implicit synchronization in the RTL8139C+ driver

```
request atomic int PCINET.rx_poll(...)
{
  int rx;

rx_status_loop:
  rx = 0;
  RTL.ISR = cp_rx_intr_mask;

  while (1) { /* Receive packets */
    if (status & DescOwn)  {
      break;
    }
    ...
    if (rx >= budget) { // Are we done
       with all packets?
      break;
    }
  }

  if (rx < budget) {
    /* Implicit synchronization: process
       remaining packets if we received
       packets in the meanwhile  */
    if (RTL.ISR & cp_rx_intr_mask) {
      goto rx_status_loop;
    }
  }
  ...
}
```

```
/* Device interrupt */
request interrupt PCINET.irq
    (...)
{
  intr_t status;

  status = RTL.ISR;

  /* Clear all interrupts */
  RTL.ISR = status & ~
      cp_rx_intr_mask;

  ...

  return 0;
}
```

Listing 5.3: Establishing data-flow dependencies to correct locking code.

```
/*
Read the contents of all registers, copy to user-space.
Currently not supported by CiD, so we have to use a workaround.
*/
request void PCINET.ethtool_get_regs(perdevice& dev, ..., void& p)
{
  synchronized {
    @memcpy_fromio(p, _perdev_instance_->__iomem__, 0x100);@ /* Can't do that
         in CiD */
    cid.nop(dev.tx_head); // Establish data-flow dependency (will not
        generate code)
  }
}
```

Atomic Operations

The mass storage driver uses the per-device atomic variable `poison` to keep track of the connection status of the device. The operations on the variable are very simple and constitute of read and set operations. In total, all 8 atomic operations have been correctly inferred by the compiler.

Deferred Work

Among the two drivers, only the mass storage driver uses deferred work. Although the NIC driver comprises an interrupt handler, it uses the NAPI (new network API) to bounce off CPU-intensive computations. The CiD distribution comes with an old-style network driver (`rtl.cid`) which demonstrates what a traditional interrupt handler looks like. Contemporary drivers, however, use the NAPI as it provides important features such as software implemented interrupt mitigation.
The mass storage driver uses two kinds of deferred work: dynamic timers and a single workqueue. Dynamic timers are used to cancel delayed, asynchronous USB transactions, while the workqueue implements the state machine for the driver and processes I/O commands. The workqueue and the dynamic timers have been successfully replaced with deferred blocks and correctly inferred by the compiler.
One problem with the current implementation is that every deferred computation is associated with a separate handle. This leads to a considerable increase in binary size as Table 5.2 shows. Future versions could easily overcome this problem by allowing the programmer to specify an identifier for each deferred block to share handles.

Deadlock Prevention

In Linux device driver code, there are numerous ways to create deadlocks. In the converted drivers, the compiler ensures that following properties hold:

- There are no blocking function calls while holding a spinlock
- There are no blocking function calls in atomic context
- Locks are always balanced

The first two properties have been verified by placing blocking calls in atomic functions and synchronization blocks that are replaced with spinlocks. Special attention has to be paid to functions whose blocking behavior is determined by an allocator flag. This is important because choosing the wrong allocator flag may lead to a deadlock. In the CiD drivers, these flags are automatically inferred by the compiler, thus preventing the chance for deadlocks. This means that the functions are now context insensitive and unaffected by future code refactorings. In the mass storage driver and the NIC driver, 13 and 5 flags, respectively, have been inferred by the compiler. Finally, in the generated code, all locks are balanced which is a seemingly trivial consequence that comes with the use of synchronization blocks. However, care has to be taken when returning from a function while holding a lock, which might not be always immediately apparent. In the mass storage driver, there are two such cases which are handled correctly by the compiler.

Table 5.2: Code statistics of the converted and original drivers. The NIC driver code size could be reduced by 14%. In contrast, the mass storage driver shows an increase in code size because CiD does not provide all concise syntax features of C, yet. Increases in binary size are due to redundant HW I/O operations and wastefully instantiated deferred work.

Drivers		
	8139C+ - C / CiD	**USB Mass Storage - C / CiD**
Physical SLOC	1607 / 1377	1584 / 1701
Binary Size	31,549 / 34,233	33,960 / 26,960

1

5.3 Hardware I/O

The hardware I/O features have been used in both, the NIC driver and the USB mass storage driver. However, since the mass storage drive is message-oriented, it only serves as a test for descriptors. The NIC driver, on the other hand, uses descriptors and register files. The main criteria of evaluation was the number of redundant operations generated by the compiler, and, of course, correctness.

Descriptors

The NIC driver uses descriptors to operate on DMA ring buffers for sending and receiving packets. In the USB mass storage driver, descriptors are used to encode data-transfer commands. When reading or writing descriptor fields, there is a penalty since the compiler inserts conversion functions. This is no issue for the network driver, but in the mass storage driver, there are 3 instances in which no such conversion is necessary. There are two causes for this. First, when comparing descriptor fields, no byte order conversions are necessary. This was not considered during implementation, but the compiler could be updated with a simple optimization rule to handle this case. Second, the command descriptor of the mass storage protocol contains a tag field that identifies the current transaction for which its byte order does not matter. This cannot be expressed with the current syntax, but descriptor field declarations could be extended with a don't-care modifier for byte-order.
However, there are other accommodations which have to be done. One feature that would be helpful is inheritance of data-layouts. For example, the packet transmission descriptor of the 8139C+ chipset changes part of its layout (the first 32 bits) depending on its state. Currently, this cannot be expressed with descriptors. Therefore, two separate and almost identical descriptor layouts have been definied. Future versions should consider inheritance to avoid unnecessary typing.

Register Files

In the original 8139C+ driver, there are 78 I/O operations. The CiD compiler generates 120 I/O operations which is a factor of 1.5 increase compared to the original code. The result was expected to be much lower because the original I/O interaction code is very simple, thus not requiring any special considerations. There are two explanations.

First, the compiler always generates read-modify-write operations from writes to individual registers of a register group, regardless of the inter-procedural data-flow. For example, during initialization phase, the transmitter and receiver of the NIC are enabled by changing the corresponding bits in the Command register. Because all other bits in the Command register are left untouched, the compiler issues a read-modify-write operation to protect their values. However, during initialization, no other execution trace touches the register, which is unknown to the compiler.

Second, the expressiveness of register files is rather limited, and not all access patterns can be easily expressed with CiD's register files. Once a register group has been divided into individual registers, it is not possible to access the entire group. However, as the 8139cp driver shows, this is necessary in some instances. The original driver uses a per-device copy of the Command register `cmd` to keep track of important changes such as enabled/disabled checksum offloading. In the converted driver, the changed bits have to be written back individually, thus increasing the number of I/O operations.

A quick performance test with `netperf` revealed no significant performance impacts. This might be a bit surprising, but this was expected since the redundant I/O operations are scattered evenly over the source code, and thus, different functionalities. However, more detailed testing might reveal performance penalities, but this was not the focus of this thesis (and the compiler implementation).

In order to eliminate redundant read-modify write operations, two steps have to be taken. First, the hardware I/O analysis has to be made aware of the results of the concurrency analysis to determine possible conflicts. Second, the register file specification has to offer some way to specify whether it is safe to concurrently access individual bits of a register. This should be considered for future versions.

Whether the resulting I/O code shows an increase in readability is debatable. The main reason is that the 8139 chipset does not define a very complicated register layout, and therefore, I/O interaction is straightforward. Also, the original driver is very well written and the I/O code is easy to read. In fact, the CiD code is even more verbose because individual register flags can be only accessed separately (with a clean register file description). However, the programmer can choose to represent individual flags as a single integer type and read or write all flags at once. The drawback is that this would prevent the compiler from performing consistency checks due to lost layout information. Also, numerical (typically hexadecimal) numbers that represent a combination of flag values are more difficult to read. From this point of view, more verbosity means also clearer code.

5.4 Code Reuse and Separation of Concerns

Four templates have been written to support the creation of the USB mass storage and the RTL8139 driver. The template `module.tl` contains bootstrap code for a typical Linux module and is included by every device driver. The blocklayer template `blocklayer.tl` contains boilerplate code to set up a block device and supports basic operations for the interaction with the blocklayer. Similarly, the `usb.tl` template includes code to interact with the USB layer. The USB mass storage driver uses all three aforementioned templates. The network driver uses the `pcinet.tl` template which interacts with the network and the PCI layer.

Code Reuse

With the PCI template approximately 14% of the original NIC driver code could be reused. The majority of code reductions is due to setup code for the PCI and network systems, including acquisition and release of I/O space, setting up the interrupt handler, and initialization of subsystem data-structures such as function pointer tables.
The result could be vastly improved by adding packet processing logic to the template. For example, the Linux kernel distribution already features a full-fledged code template, but it is intended to be used by 8139C+ based chipsets.[2] However, there are, for instance, similarities in the ring buffer management of the E1000, the 8319C+ chipset and other NICs, which could be exploited.
In fact, better results have been achieved by Conwell with his NDL language (see [Conway and Edwards, 2004]). The NDL driver for the NE2000 NIC comprises only half as many lines of code as the original Linux driver. The corresponding NDL code template is more elaborate than the CiD template and contains almost all of the OS-specific code. However, a deeper investigation reveals that the template also contains device-specific code that cannot be reused in the 8139C+ driver, for instance, EISA bus initialization, or EEPROM I/O code. In contrast, the CiD template contains only code that can be reused by all PCI-based NIC drivers.
With the USB and block layer templates, only 68 lines of code could be reused in the mass storage driver. This result is very disappointing as it was expected that at least 10% of the driver could be replaced with template code. In fact, a previous version of the blocklayer template comprised a common strategy for processing requests on the I/O request queue. Although the mass storage driver uses the same strategy, an attempt to merge the strategy with the original driver failed. The reason is that the driver relies on information of individual requests, which was abstracted away in the template. Further investigations on this matter could yield to better results, but the overall potential for code reductions is probably low.
Another aspect that should be mentioned is that code reuse in the kernel is already achieved with low-level drivers, or subsystems, that provide services for high-level drivers. This further limits the potential of code reuse.

[2]The template can be found in /drivers/net/pci-skeleton.c

Separation of Concerns

The initial motivation behind the template mechanism was to offer a way to separate device-specific from OS-specific code. With the current design this separation can only be achieved to a minor degree. While the templates take care of mundane, reoccurring tasks such as initialization of subsystems and driver resources, device driver programmers still have to possess basic knowledge about Linux device drivers and the subsystems they work with. However, instead of simply copying similar device driver code, templates provide a cleaner way to reuse code. The resulting code slightly improves in readability since all the boilerplate code resides in a separate file.
A cleaner separation between device-specific and OS-specific code is demonstrated by the recently developed driver generator system Termite [Ryzhyk et al., 2009b]. A Termite device driver specification is divided into three parts, the device protocol specification, the operating systems specification and the device-class specification. CiD does not support this kind of division, but future extensions could be considered.

5.5 Compiler Complexity

The CiD compiler features roughly 6913 physical source lines of code: 13% of the source code account for concurrency algorithms, 6% constitute of hardware I/O code and 4% are made up by the template compiler. The remaining 77% constitute of parsing, lexing, AST node definitions and building routines, name analysis and (lots of) type checks. In conclusion, the proposed language extensions and algorithms were, as expected, easy to implement.

5.6 Limitations

While the proposed language extensions and the CiD compiler can aid the programmer in detecting mistakes (or preventing them), the complexity still remains. To some degree, the programming model can be simplified by freeing the programmer from choices which can be made automatically, such as the correct lock type or deferred work mechanism. However, correct CiD code does not mean that a device driver is free from deadlocks or race conditions, the most vicious types of bugs. The CiD compiler cannot perform checks on API constraints, and for example, there is one non-critical path in the mass storage driver which (currently) leads to a deadlock. Due to the high degree of concurrency, detecting faults like race conditions and deadlocks is still very difficult. Also, once the kernel crashes, all information about the program state is lost, which makes it even more difficult to locate deadlocks.
It was hoped that templates could simplify the programming of device drivers to some degree, but the resulting drivers are still tightly coupled with the Linux driver model, as the poor code reuse results show. However, more fleshed-out templates could yield to better results.
In conclusion, CiD can only assist the programmer in managing the high complexity of the programming model. However, no significant reductions of the overall invovled complexity are likely to be expected with further improvements and extensions.

CHAPTER 6

Related Work

In the last few years, there has been increasing effort to address the device driver reliability problem in the OS research community. In general, two key approaches can be identified. The first, and more traditional, approach sees the causes of the reliability problem in weaknesses of current OS and device driver *architectures*. The second approach recognizes the role of *languages* and their contributions to the quality of device driver code.[1]

Microkernel architectures are the best-known technique to build reliable and fault-tolerant operating systems. Traditional microkernel systems, however, do not directly address the reliability of device drivers and assume that they are inherently faulty. While this assumption has been proven true for current device drivers, it may be very well invalidated in the future with new techniques such as static source code verification and new domain-specific languages [Ryzhyk et al., 2009a, Conway and Edwards, 2004, Mérillon et al., 2009, Ryzhyk et al., 2009b].

The concept of user-space drivers is not limited to microkernel-based architectures. Research prototypes have shown that monolithic kernels, such as Linux, can be accommodated to running user-space drivers [Renzelmann and Swift, 2009, Leslie et al., 2005]. Particularly notable efforts are the *Decaf Drivers* architecture and Leslie et al.'s adoption of the Linux kernel to support user-space drivers.

While user-space drivers have a lot to offer, they do not address all problems that current device drivers show. As has already been pointed out in previous chapters, most general purpose languages do not reflect special aspects of device driver development very well, or at all. Crucial aspects such as hardware I/O and communication protocols are not part of most languages. Noteworthy innovations in this area are hardware interface description languages such as *Devil* and *NDL*, which allow the programmer to specify register layouts, register I/O operations and their side-effects [Mérillon et al., 2009, Conway and Edwards, 2004].

[1] It is important to note that the distinction between architecture and (programming) languages is rather arbitrary and can get blurry. In fact, Hunt et. al demonstrate with their research OS "Singularity" that there is a synergy between programming languages and operating system architecture, one aspect shaping the other [Hunt and Larus, 2007]. Nonetheless, due to the complexity of the device driver reliability problem, it is helpful to differentiate between the two aspects.

Concurrency related bugs are very likely to be found in device drivers due to the complex programming model. Successful attempts to *simplify* the Linux concurrency model have been demonstrated: The *Dingo* architecture demonstrates the benefits of event serialization at little cost in performance [Ryzhyk et al., 2009a].
An orthogonal problem that device driver developers have to face is *collateral evolution*. The semantic patcher Coccinelle offers a powerful language to capture complex changes in the driver codebase with a concise and intuitive syntax [Padioleau et al., 2008].

6.1 Device Driver Architectures

User-space Drivers

User-space drivers offer several benefits over kernel space drivers. The arguably most important advantage is that user-space drivers can be written with the same tools and programming languages as all other user-space applications. As a result, device drivers can be more easily tested and debugged with available tools. While there are obvious gains in productivity, experimentation with already existing programming languages is encouraged. Another advantage is that faulty device drivers which are stuck in a deadlock or in an illegal state, can be recovered by simply restarting the driver process [Herder et al., 2006].
The idea of user-space drivers has also found its way into the Linux kernel with the user I/O (UIO) patch. The motivation behind the patch was to make driver programming less difficult and more productive. However, UIO provides only rudimentary support for user-space drivers and is geared towards less sophisticated embedded devices with less performance demands than for example, network interface controllers.
There are two explanations why user space drivers have received only little attention in the kernel community. First, a wide spread myth is that user-space drivers are inherently slow. While it is true that early microkernel implementation such as Mach had slow user-space drivers due to poorly designed IPC, modern microkernel architectures such as L4 have far better performance [Härtig et al., 1997]. In fact, Leslie et al. have shown that monolithic kernels such as Linux can be also accommodated to user space drivers with high performance requirements. Leslie et al.'s user space architecture is a good example for this. Second, migration to user space drivers is a difficult problem due to the enormous size of the existing code base. The DecafDrivers research architecture shows that part of this process can be automated.

Leslie et al.'s User-Space Architecture

Figure 6.1 shows an overview of Leslie et al.'s user-space architecture [Leslie et al., 2005].
Device drivers gain access to device registers by mapping portions of physical (I/O) memory into their address space. The Linux kernel already provides this capability with the `mmap` system call and the `/dev/mem` file node which represents all available physical memory.
In order to support DMA operations, the system call interface has been extended with mapping services that translate physical addresses into bus addresses and services that pin memory regions. Pinning of DMA-mapped memory regions is essential to prevent memory pages from being swapped during DMA transfers.

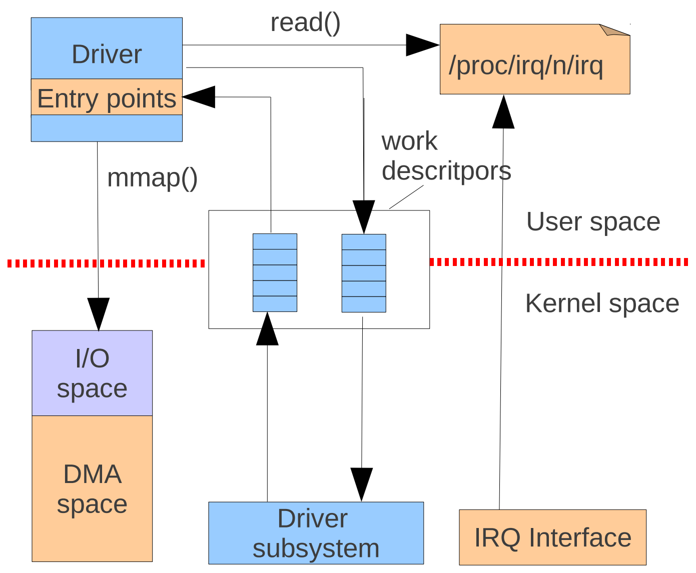

Figure 6.1: Structure of the user-space driver architecture by Leslie et al [Leslie et al., 2005].

Interrupts require special treatment because they can be only handled in kernel-space. In the proposed architecture, user-space drivers register with an interrupt by calling the `open` function on the desired entry in the `/proc/irq/n` directory, where n designates the interrupt number. Once the driver calls `close`, the interrupt handler is unregistered. The user-space driver receives interrupts by invoking the `read` system call on the file node. In the kernel, a semaphore keeps track of the number of occurred interrupts and decrements with each read operation. If there is no pending interrupt, the driver blocks until the device signals an interrupt which increments the semaphore.

Communication between the device driver and the kernel is the (performance) critical part of the architecture. Requests from the kernel to the device driver and vice versa are encoded as message descriptors which identify the operations that have to be performed. The descriptors are shared between kernel and driver in lock-free circular buffers, one for each direction. Both, kernel and user-space drivers, can directly access the descriptors without additional overhead.

Leslie et al. have evaluated their architecture with a block device driver and a NIC driver, which both have high performance requirements. The results are remarkable: the user-space drivers performed nearly as well as the kernel space drivers with only minor processing overhead and neglectable drops in throughput rates. This shows that, in theory, the Linux kernel could be accommodated to support user-space drivers at neglectable costs in performance considering the benefits in security and reliability. However, in practice, the main disadvantage of the architecture is that all existing drivers have to be converted and existing driver interfaces have to be adopted to the proposed message interface.

Decaf Drivers

Decaf Drivers is a generic device driver architecture that supports the development of user space device drivers in, conceptually, any programming language. Unlike the architecture proposed by Leslie et al., Decaf Drivers also provides a migration path that makes it easier for kernel developers to convert legacy device drivers into user-space drivers. Figure 6.2 shows an overview of the Decaf Drivers architecture.

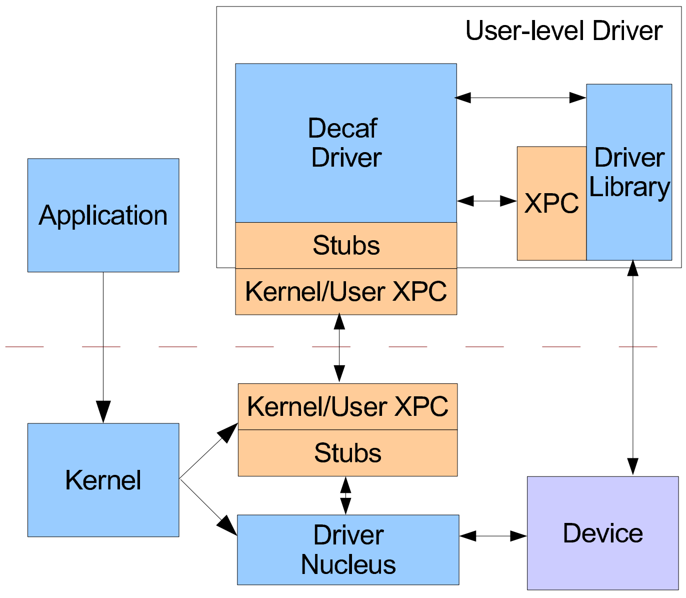

Figure 6.2: Decaf Drivers Architecture (adopted from [Renzelmann and Swift, 2009]).

A device driver is divided into three parts: a driver *nucleus*, a *decaf driver* and a driver library. The nucleus is a kernel module which contains performance critical code and makes the user-level decaf driver compatible with the kernel. The decaf driver executes in user-space and, thus, can be written in any language. The driver library contains support functions written in C. Communication between the decaf driver, the driver library and the nucleus is achieved via extension procedure calls (XPCs). XPCs enable the transition from user space to kernel space, and vice versa, and are also used to allow the decaf driver to call native functions in the driver library. XPCs are heavily optimized for run-time performance and for example, avoid unnecessary copying of data-structures. XPCs offer five services: control transfers, object transfers, object sharing and synchronization and stubs. Control transfers emulate procedure call semantics, object transfers marshal and unmarshal data structures. Object sharing, in conjunction with synchronization, enable the decaf driver and the nucleus to safely share data. Stubs contain code

for setting up and calling XPC services.
The tool DriverSlicer is a crucial part of the Decaf Drivers architecture which makes the transition from kernel-space to user-space drivers easier. The DriverSlicer tool is capable of partitioning a legacy C driver into code that has to reside in the nucleus or the decaf driver. In addition, the tool also generates XPC stubs, but the programmer has to provide some annotations for marshaling and unmarshaling kernel data structures.
The Decaf Drivers architecture was evaluated with five device drivers, including network drivers, a sound driver, a USB 1.0 host controller, and a serial mouse driver. The decaf drivers were all written in Java. It was also shown that even for performance-sensitive drivers, such as the E1000 network driver, there were overall minor performance penalties.
More interestingly, Renzelmann and Swift observe that Java device driver code benefits from high-level features such as exception handling, generic standard libraries, object-orientation and, possibly, garbage collection. However, benefits of garbage collection have not been fully evaluated because management of shared objects has to be manually done by the programmer. Error handling in Linux device drivers can be improved with exception handling. For example, in the course of rewriting the E1000 NIC driver in Java, Renzelmann and Swift found 28 cases in which errors were originally ignored in the legacy code, but reported as uncaught exceptions by the Java compiler in the new code. Also, reductions in driver code size could be achieved with code inheritance (object-orientation) and the use of Java collections.

Dingo

Dingo is a device driver architecture that addresses concurrency issues and OS protocol violations which occur within traditional device driver code [Ryzhyk et al., 2009a]. A Dingo driver consists of two parts: (1) a protocol specification that defines message exchange between a device driver and the operating system and (2) a C implementation of the driver that processes the defined messages. Unlike in a traditional Linux driver, the Dingo engine serializes all requests (messages) that are passed to a driver, and therefore, there is only one code path active at a time. Verification of the protocol is done during run-time via a module called "protocol observer".
In the Dingo model, the operating system and device drivers communicate via *ports* which represent bidirectional communication endpoints for exchanging messages. Each port is associated with a protocol that defines order and timing in which messages may arrive as well as their contents. Ports, messages and their protocols are described with a mix of textual and graphical syntax. For example, the port declaration of an USB-to-Ethernet driver (presented in the corresponding paper [Ryzhyk et al., 2009a]), is as follows:

```
component asix {
ports:
  Lifecycle lc;
  mirror Timer timer;
}
```

According to the component declaration, the Asix ethernet controller exports a Lifecycle port and uses the Timer protocol provided by the OS, as indicated by the `mirror` keyword.[2].

[2]The complete declaration contains additional ports, but they are not shown in order to keep the example brief.

Every protocol consists of a list of messages and a state machine that defines how the messages affect the state of the driver (or device). The message definition is defined textually, as the following code fragment shows:

```
protocol Lifecycle {
messages:
  in start();
  out startComplete();
  out startFailed(error_t error);
  in stop();
  out stopComplete();
  int unplugged();
}
```

The corresponding state machine is described with the Statechart language, a graphical language similar to state charts in UML, as shown in Figure 6.3. State transitions are triggered by receiving or sending messages defined in the protocol. With statecharts it is possible to decompose states into hierarchies. For example, the superstate `connected` consists of three states, `starting`, `running` and `stopping`, from which a transition to the `disconnected` state is possible.

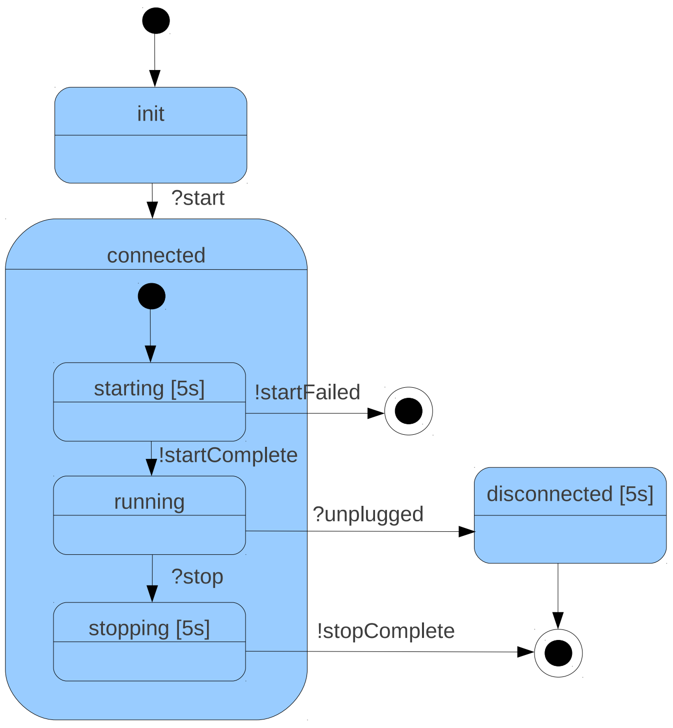

Figure 6.3: Lifecycle state machine (adopted from [Ryzhyk et al., 2009a]).

It is the job of the programmer to translate a given device protocol into actual code. Dingo features a customized C preprocessor with specialized macros for event (i.e., message) handling.

A serious drawback of event-driven models is stack ripping (see Section 3.3). Because event handlers may not block, requests that cannot be immediately handled have to be divided among event handlers, i.e., completion chains. To address this problem, Dingo offers language-extensions and compiler support to allow the programmer to write completion chains in a sequential way. For example, asynchronous function calls can be handled with the `CALL` as follows:

```
/* Repeatedly read from a device register */
do {
  /* Wait 10 milliseconds, without blocking */
  CALL (timeout, (10, &notif), notif);

   /* read status register */
  ...
} while (/* condition */)
```

The CALL macro calls an asynchronous timer service. The compiler takes care of saving the state, i.e., the local function stack and the instruction pointer. Upon completion, the state is restored, and execution continues after the CALL statement. Without this macro, the above code would significantly increase in complexity and decrease in readability. This is because the timeout event would have to be handled in a different event handler, and the loop would have to be replaced with a counter, indicating the current iteration state.
Dingo also offers additional coordination statements. For example, messages that are defined in the protocol can be received and sent with the `AWAIT` and `EMIT` macros, respectively. For example, `AWAIT(disconnected)` would suspend the current execution trace, waiting for the disconnect event.
The idea of protocols has also been introduced as part of the Singularity operating system [Hunt and Larus, 2007]. The concepts are quite similar: In Singularity, message protocols are described with *channel contracts* (a channel is just another name for a port), which define both, messages for exchange, and a state machine. Unlike Dingo, Singularity uses static verification to ensure protocol correctness.

Active Device Drivers

The Active Device Drivers architecture is a recent contribution by Ryzhyk et al. that introduces the distinction between passive and active drivers [Ryzhyk et al., 2010].
In a passive device driver architecture like in Linux, the OS passes requests to device drivers by concurrent invocation of the drivers' entry points. If the device driver is busy processing other requests, it still has to cope with more concurrent requests coming in. In contrast, an active driver is a sequential program with its own thread of control and can determine on its own when it is ready to process the next type of request.
The communication model for an active driver is as follows. Requests are put into shared memory locations called mailboxes. Every mailbox corresponds to a particular type of request (such as an I/O request or a configuration request) and may contain an arbitrary number of messages of the same type. If an active driver is ready to process a request, then it issues a blocking read operation on the corresponding mailbox. Thus, requests of the same type are *serialized*. When a driver has finished a request, it puts a response message into the corresponding reply mailbox.

Because devices may have multiple, independent communication paths, such as the send and transmit channels of a NIC, the Active Drivers also supports multithreaded request processing. To this end, an active driver may create *cooperative domains* in which threads that belong to the same domain are scheduled cooperatively. Threads from different domains may run in parallel.

Discussion

Architectural modifications such as user-space drivers and alterations of the concurrency model as in Dingo show promising potential to prevent frequent faults in device drivers and to make driver programming more productive.

User-space Drivers. Theoretically, the concept of user-space drivers is superior to *traditional* kernel-space drivers despite the performance overhead [3]. The free choice of development tools and the safer programming environment allow for significant gains in productivity and show potential to increase the reliability of device drivers. Clearly, compiler enhancements to fit the kernel programming model (as demonstrated with CiD), cannot reach the flexibility of user-space drivers. It must be stressed, however, that user-space drivers do not solve all aspects of the device driver reliability problem but "merely" provide a reliable platform to build on. Support for crucial aspects such as hardware I/O, concurrency and device protocols is complementary to user-space drivers and necessary to further increase the quality of device driver code. CiD shows potential to reduce driver faults with additional consistency checks for hardware I/O and concurrency.

Dingo and Active Device Drivers. The event-driven approach in Dingo relieves the programmer from the complex multithreaded model and also addresses the problem of stack ripping. Similarly, the active device drivers architecture also reduces the complexity of multithreaded drivers with its new communication model. With Dingo or Active Device Drivers, programmers do not have to worry about locking and intertwined driver execution paths. In contrast to Dingo and Active Device Drivers, CiD retains the multi-threaded concurrency model but provides a safety net through static compiler checks. Detection of race conditions, inference of lock types and illegal function calls (that lead to deadlocks), assist the programmer in generating correct concurrent driver code. Therefore, device drivers can be written in the classical multithreaded model with some reassurance, but the complex program structure still remains.

An important feature that Dingo has to offer is the verification of the OS to driver interaction. Dingo's protocol specification capability is not to be confused with CiD's protocol construct. CiD protocols are solely used improve the accuracy of detecting race conditions in the data path of a driver. Unlike Dingo, the CiD compiler cannot verify if the driver behaves correctly (with respect to the device and OS protocols). Therefore, future versions of CiD should incorporate more powerful protocols. Because of the multithreaded model, verification will be more difficult to implement than in Dingo.

The Migration Problem. Any technology or paradigm that replaces a previous technique faces the problem of migration. In the case of Linux device drivers, this problem is particularly serious

[3]According to Herder, the performance gap between kernel-space and user-space drivers might even narrow further with hardware support for message passing [Herder, 2010]

because of the enormous size of the code base.
Although the development of user-space drivers can be eased with code generation tools like DriverSlicer, there is still a lot of manual labor involved. Driver interfaces have to be adopted and drivers have to be written in new languages. While Dingo provides some sort of plug-in framework for the new architecture, leaving legacy drivers untouched, it requires the transition to an event-driven model. However, the changes would be less labor-intensive than for user-space drivers.
There are two possible scenarios in which new architectures could be accommodated to the Linux kernel and its drivers. In the first scenario, legacy drivers are left untouched and new drivers are written with the new architecture. In the second scenario, legacy drivers are adopted to the new architecture. None of the two scenarios seem favorable, since either legacy code has to be additionally maintained (old drivers should still benefit from kernel innovations), or the entire driver code base has to be converted eventually. Considering that driver developers are already busy keeping up with new hardware, migration is unlikely to happen until there are more powerful driver code generators available. Also, the transition to a new paradigm poses unknown risks not readily to be taken by pragmatic kernel developers.
While legacy driver code has to be converted into CiD as well, the changes are less labor intensive since no architectural modifications are required and (most importantly) the programming paradigm stays the same. Thus, CiD provides a smoother transition than any other of the proposed architecture.

6.2 Domain-specific Languages

Devil

Devil is domain-specific language for describing the hardware I/O interface of a register-based device [Mérillon et al., 2009]. Listing 6.2 shows a devil specification for a serial mouse which will be explained in the course of this section.
Every Devil specification is based on three abstractions: *ports*, *registers* and *device variables*. Ports unify port-mapped and memory-mapped I/O and enable uniform communication with a device. Device variables form the visible interface of a device (in object-oriented terminology, they are comparable to getter- and setter-methods) and are defined with Devil's *registers*, comparable to private variables of a class, to access individual registers of a device. For each variable, the Devil compiler generates a native I/O function that can be invoked in a device driver.

The first line in Listing 6.2 declares the *port* variable `base` which is used to derive the signature register, configuration register, interrupt register and the index register of the device. The device register layout consists of 4 banks, each of them being 8-bits in size, denoted with `bit[8] port @ {0..3}`. Line 4 declares the signature register, `sig_reg`, which occupies the second register bank (`base@1`). The corresponding variable `signature` encapsulates access to this register by specifying three constraints, indicated by the keywords `volatile`, `write trigger` and the type constraint `int(8)`.
The keyword `volatile` indicates that two successive reads to the device signature register may yield two different results. This information is important for the compiler, because read

Listing 6.1: Devil specification for the logitech busmouse (taken from [Mérillon et al., 2009])

```
1  device logitech_busmouse (base : bit[8] port @ {0..3})
2  {
3    // Signature register (SR)
4    register sig_reg = base @ 1 : bit[8];
5    variable signature = sig_reg, volatile, write trigger : int(8);
6
7    // Configuration register (CR)
8    register cr = write base @ 3, mask ’1001000.’ : bit[8];
9    variable config = cr[0] : { CONFIGURATION => ’1’, DEFAULT_MODE => ’0’ };
10
11   // Interrupt register
12   register interrupt_reg = write base @ 2, mask ’000.0000’ : bit[8];
13   variable interrupt = interrupt_reg[4] : { ENABLE => ’0’, DISABLE => ’1’ };
14
15   // Index register
16   register index_reg = write base @ 2, mask ’1..00000’ : bit[8];
17   private variable index = index_reg[6..5] : int(2);
18
19   register x_low = read base @ 0, pre {index = 0}, mask ’****....’ : bit[8];
20   register x_high = read base @ 0, pre {index = 1}, mask ’****....’ : bit[8];
21   register y_low = read base @ 0, pre {index = 2}, mask ’****....’ : bit[8];
22   register y_high = read base @ 0, pre {index = 3}, mask ’...*....’ : bit[8];
23
24   structure mouse_state = {
25       variable dx = x_high[3..0] # x_low[3..0], volatile : signed int(8);
26       variable dy = y_high[3..0] # y_low[3..0], volatile : signed int(8);
27       variable buttons = y_high[7..5], volatile : int(3);
28     };
29
30 }
```

operations may be cached for performance reasons. The keyword `write trigger` indicates that writing to the register induces side-effects on the controller. Similarly, `read trigger` would indicate side-effects after a read operation. Finally, the type of the variable is constrained to an 8-bit integer.

The configuration register is located at memory bank three. The contents of the device register are constrained with the bit pattern `’100100.’`. All bits of this register have a fixed value (being either 1 or 0), however, the first bit can be of an arbitrary value, as indicated by a dot `’.’` . Asterisks (`*`) can be used to specify bit positions that should be extracted from a bit vector. Lines 19-22 make extensive use of bit masks. This value is further constrained in the corresponding variable definition `config`, which expresses that bit 0 (`cr`[0]) can be either set to `CONFIGURATION` or `DEFAULT_MODE`, which are both part of a type-safe enumeration.

Access to registers may be also predicated by *preactions* (see lines 19-22). Some device designs may use the same device memory location to represent different registers. According to the devil specification, the logitech busmouse uses the first register bank to represent x and y coordinates.

The value of the index register determines which coordinate component gets stored in the first register bank. For example, before accessing register `x_low`, index is set to 0. Variables can be grouped together to form a structure (see line 24). Upon loading of the structure, the corresponding variables are loaded all at once.
Based on the detailed specification, the Devil compiler is capable to perform a series of consistency checks which a C compiler cannot perform. For example, the Devil compiler ensures statically that registers do not overlap (like the CiD compiler). If the programmer wishes, the Devil compiler also generates run-time checks. For instance, when writing a non-constant value to a variable, an optional run-time check verifies that the size is within the bounds of the corresponding type.
In conclusion, Devil assists the programmer in generating more robust code. In fact, Mérillon found that the probability of undetected errors is 1.6 to 5.2 higher in C drivers than in Devil-based drivers [Mérillon et al., 2009].

NDL

Like Devil, NDL allows the description of device register layouts. However, NDL offers additional features that simplify device driver development and is even capable of generating complete (network) device drivers from code templates.
Every NDL specification consists of three elements: an interface description, a state machine and a set of driver functions. Device register descriptions are part of a device interface declaration which may inherit other interfaces and template information. Listing 6.2 shows a fragment of the NE2000 NIC NDL driver, adopted from the most recent NDL release [Conway, 2010].

The first line declares the device and inherits the code template and interface functions from the built-in NetworkDevice declaration. The `ioports` section in the following line defines the command register of the NE2000 controller. The `stop`, `start` and `transmit` each occupy one bit and trigger a side-effect upon write access (as indicated by the `trigger` keyword). The `dmaState` compound field occupies three bits and defines constants that encode different states in the DMA engine.
The states of the NIC are defined after the `ioports` declaration. Every state can be associated with a list of actions that are performed at the transition, including transitions to other states. States that are mutually exclusive are connected with the `||` operator. In the example, the controller can be either in the start or stopped state *and* in one of the three DMA states.
Every NDL driver consists of a series of functions, in which the device specification can be used to program the driver. To this end, NDL comprises a small C-like subset with control flow constructs, arbitrary-sized integers, arrays and integer arithmetic. The source code of the NE2000 driver also reveals that there are synchronization statements which are not mentioned in the original publication, such as waiting for a device register to change its value. Also, similar to CiD's `synchronized` statement, `critical` blocks are used to mark critical sections in device driver code.
An interesting result is that the NDL driver for the NE2000 has only half of the size than the Linux driver. On the one hand, this is because NDL code is more terse than Devil, one the other hand, code reductions can be also achieved with templates that contain boilerplate code.

Listing 6.2: Excerpt of NE2000 NDL interface (adopted from [Conway, 2010])

```
device ne2k : NetworkDevice {
  ioports {
    0x00: /* At offset 0x00 */
    command = {
      stop      :       trigger except 0,
      start     :       trigger except 0,
      transmit  :       trigger except 0,
      dmaState  : {
        READING    = #001
        WRITING    = #010
        SENDING    = #011
        DISABLED   = #1**
      },
      registerPage : int{0..2}
    },

  /* State machine */
  state STOPPED {
    goto DMA_DISABLED ;
    stop = true ;
  }
||
  STARTED { start = true ; }

  state DMA_DISABLED { dmaState = DISABLED ; }
  ||
    DMA_READING { goto STARTED ; dmaState = READING ; }
  ||
    DMA_WRITING { goto STARTED ; dmaState = WRITING ; }
}
```

Coccinelle

Coccinelle addresses the problem of collateral evolutions in device driver code (see Section 3.3 in Chapter 3) and has been successfully used to create a variety of kernel patches [Muller, 2010]. Coccinelle enables the programmer to specify C code changes in a declarative language called the semantic patch language (SmPL). The syntax of SmPL resembles that of the well-known GNU patch tool [Free Software Foundation, 2010], but is far more powerful because it also reflects the structure of C code. Listing 6.3 shows an excerpt from an official Coccinelle patch that replaces direct access to the driver-specific data field (`driver_data`) of a generic device handle (`struct device`) with newly introduced getter and setter functions.

The patch consists of two matching and transformation *rules* for reading from and writing to the field, respectively. The beginning of each rule contains a header (delimited with the "@@' 'characters) followed by a list of *metavariable* declarations. The types of the variables are essentially constraints to the pattern matcher. In the example, variable `E` represents an arbitrary C expression and `T` denotes the name of a valid C type name. Similar to GNU patch, lines that

Listing 6.3: A simple Coccinelle patch that inserts a newly introduced getter function

```
@@
struct device *dev;
expression E;
type T;
@@

- dev->driver_data = (T)E
+ dev_set_drvdata(dev, E)

@@
struct device *dev;
type T;
@@

- (T)dev->driver_data
+ dev_get_drvdata(dev)
```

should be added and removed are prefixed with the minus and plus sign, respectively.
Pattern matching rules can be also combined to express more complex code changes as the patch presented in Listing 6.4 shows. The purpose of the patch is to replace access to the private data field `priv` of a NIC handle `net_device` with the getter function `netdev_priv`. Unlike the previous example, the usage of the function is more restricted than `get_drv_data`, since it can be only applied to device handles that have been allocated with either `alloc_netdev`, `alloc_etherdev` or `alloc_trdev`. The purpose of the first rule is to match these handles. The third rule performs the actual replacement, but only if the first rule matches. More precisely, the match is constrained with the variable `T` which is *inherited* from the first rule.

Listing 6.4: Metavariables and rule dependencies in Coccinelle

```
@ rule1 @
type T;
struct net_device *dev;
@@

  dev = (alloc_netdev | alloc_etherdev | alloc_trdev)
        (sizeof(T), ...)


@ rule2 depends on rule1 @
struct net_device *dev;
type rule1.T;
@@

- (T*) dev->priv
+ netdev_priv(dev)
```

When patching a large code base, the fact that the same computation can be expressed with different syntactical variations complicates the creation of accurate patches. For example, a NULL pointer condition can be expressed with an equality test or the short-hand operator `!`. To address this problem, SmPL allows the programmer to abstract syntactic and control-flow variations into so-called *isomorphisms*. An isomorphism expresses semantical equality for different syntactical constructs. For example, the isomorphism for the aforementioned NULL pointer test can be specified as follows:

```
@@
expression X;
@@
X == NULL <=> NULL == X <=> !X
```

With isomorphisms, patch authors can simply choose whatever syntax they find appropriate while Coccinelle takes care of deriving the specified syntactical alternatives. This makes the resulting patches more readable and more accurate.
The following example shows that isomorphism can be quite powerful and even capture control-flow variations:

```
@ neg_if @
expression X;
statement S1, S2;
@@
if (X) S1 else S2 => if (!X) S2 else S1
```

The Coccinelle distribution already comes with a fairly comprehensive set of isomorphism, allowing the patch author to focus on the specifics of a patch without distraction to mundane details.
While Coccinelle has been developed to address the problems with changing device driver interfaces, its program matching capabilities can be also used to detect and fix source code bugs. The official website has a quite impressive showcase of error detecting patches, for example, discovering null pointer dereferences and resource deallocation errors [Muller, 2010].

Discussion

Devil and NDL. Devil and NDL demonstrate the potential for fault prevention and fault detection in low-level hardware code through low-level code generation and more rigorous type checks. CiD's register file construct has been inspired by these languages. However, in the current state, CiD lacks advanced features to describe more complex device interfaces. For example, Devil allows the programmer to specify pre- and postactions for accessing registers. These features should be incorporated into CiD. Like NDL, CiD uses templates to generate device driver code. Judging from the latest NDL release, the template languages of NDL and CiD are equally expressive. To exploit more code reuse opportunities, CiD's template language could be extended with conditional and control-flow statements. One problem with domain-specific languages like NDL or Devil is that they make migration much more difficult than new driver architectures. Unlike architectural modifications, however, new languages do not change legacy drivers.
Coccinelle. Code reuse and collateral evolution are an orthogonal issue that device drivers have

to face. CiD uses templates to loosen the dependency between device drivers and kernel internals, but also to support code reuse. In addition, the built-in support for synchronization and deferred work allows for some resilience to code changes. Coccinelle can be considered as a powerful complementary approach that allows for more fine-grained changes than code templates and language extensions can provide.
One aspect that has to be addressed, however, is more flexibility for driver evolution. Currently, the compiler infrastructure does not support multiple template and code generator backends for different kernel versions. This should be considered for future versions.

6.3 Other Technologies

This chapter provided only a small selection of recent innovations and technologies in the area of device driver development. Other important approaches are static verification, fault isolation techniques and driver synthesis.

Static Verification

An example for static verification is Microsoft's Static Driver Verifier (SDV) tool [Microsoft, 2011]. SDV analyses the code of a Windows driver and checks if it violates any of the driver API rules. Recently, similar work has been done by Witkowski et al. with their Linux driver verification tool DDVerify [Witkowski et al., 2007]. Both tools rely on counter-example-guided abstraction (CEGAR), a technique which reports counterexamples that expose bad behavior in program (device driver) code. In contrast, the CiD compiler verifies a predefined set of rules and does not offer the flexibility to verify arbitrary API constraints. Also, the compiler does not provide counter examples, which could be considered for future extensions.
However, static verification techniques could profit from language extensions that increase the level of abstraction and thus make rule checking simpler. For example, the locks in a CiD driver are always balanced and of the correct type.

Fault Isolation

In Linux, device drivers are part of the kernel and have the potential to crash the entire system since there is no protection between the (faulty) drivers and unaffected parts of the kernel. The purpose of fault isolation is to contain faults in the corresponding subsystems, i.e., the drivers, where they can only do limited harm. Fault isolation can be achieved statically and/or during run-time. Safe kernel programming languages are an example of static fault isolation. The SPIN operating system relies on a safe subset of Modula-3 as implementation for kernel extensions. Like Linux kernel modules, these extensions can be linked during run-time into the kernel. However, unlike C, the subset features pointer-safe casting, language-based isolation of untrusted code and a secure dynamic linking. Similar to CiD, the safe Modula-3 subset features a procedure modifier that indicates whether an operation can be killed, which is important for interrupt handling in SPIN. Another example is Sing#, an extension of the C# programming language in which the Singularity operating system has been implemented [Hunt and Larus, 2007]. Unlike in conventional operating systems, processes are isolated by the memory-safe language

and not hardware protection mechanisms like MMUs.
Run-time fault isolation is achieved by architectural means. Microkernel architectures successfully build on the principle of fault isolation. A recent study by Herder et al., demonstrates the robustness of the Minix microkernel operating system against faulty device drivers [Herder et al., 2009]. It is important to note that run-time fault-isolation can be also achieved with the Linux kernel, as the Nooks architecture demonstrates [Swift et al., 2002]. Nooks introduces protection domains between the kernel and drivers, shielding them from errors such as memory corruption. Regarding to the work presented in this thesis, fault isolation can be considered as a powerful and probably necessary complementary approach. Even if language-extensions like CiD can prevent certain mistakes, there is no guarantee that a CiD driver works correctly. Also, static verification may not be able to detect all driver bugs. Another concern is that most drivers do not deal with hardware faults properly (or at all), and, for example, could be stuck in an endless loop when polling a faulty device. Therefore, fault isolation should be also considered for the Linux kernel.

Driver Synthesis

The goal of driver synthesis is to automatically generate device driver code from formal specifications, thus ensuring ”correctness by construction“. A recent contribution to driver synthesis is Termite [Ryzhyk et al., 2009b], a tool which provides a clean separation between OS-specific and device-specific code. Based on a device-class specification, the Termite engine synthesizes driver code from these specifications. Thus, when porting a device driver to a different OS architecture, only the OS specification has to be changed. Termite specifications are based on an event-driven model.
Another approach has been presented by Bombieri et al., who automatically generate simple device drivers based on register-transfer logic (RTL) testbenches [Bombieri et al., 2009].
Driver synthesis is a promising approach that could solve the device driver reliability problem. In order to make driver synthesis possible, the formal gap between devices and operating system has to be bridged with formal languages. In order to narrow the gap, hardware manufacturers and OS developers have to work closely together and develop a standardized formal language for describing device and driver models. With the advent of more expressive programming and specification languages for device and driver development the gap can be narrowed further. It might be tempting to regard CiD as a temporary solution, until fully functional driver generators are available. However, driver generators can benefit from simplified programming models (or more powerful compilers). For example, protecting critical sections in CiD is simply a matter of placing `synchronized` blocks, which simplifies code generation.

CHAPTER 7

Conclusion

The central question of this thesis was how device driver programming can be made more robust. Unlike other research approaches, this thesis investigated and demonstrated possibilities to improve the current driver programming model without revolutionary changes. The prototype language CiD shows promise and demonstrates how this can be achieved in principle.
However, there is much room for improvement and, despite all effort, this thesis just scratched the surface. Section 7.2 shows how the work can be continued.

7.1 Brief Summary and Review of Results

The current Linux programming model lacks support for:

- Concurrency and synchronization
- Hardware I/O
- Code reuse and separation of concerns

Concurrency faults, i.e., race conditions and deadlocks, are very common in device driver code: A device driver has to deal with multiple requests at the same time and synchronize concurrent activities. In addition, concurrent driver code has to satisfy concurrency model constraints such as never calling a blocking function in atomic or interrupt context. Until now, the programmer had to verify these rules by hand. With CiD, it has been successfully demonstrated that the concurrency model can be incorporated into a C compiler with only minor extensions to the programming language, i.e., concurrency protocols, synchronization blocks and function context modifiers. As demonstrated with two converted drivers, the CiD compiler always chooses the correct lock type, eliminating one potential cause for a deadlock. Also, the compiler ensures that all blocking operations are safe in the current execution trace, further mitigating the potential for deadlocks. Finally, race conditions in the data-flow of the converted device drivers can be detected with a false positive rate of 6% to 21%.

Hardware I/O is a particularly error-prone aspect of device driver code since minor mistakes such as swapping two bits cause the driver to malfunction and are undetected by the compiler. CiD's hardware I/O features assist the programmer in writing correct low-level code with the generation of bit manipulation code, automatic byte order conversions and consistency checks on data layouts. However, compared to more elaborate approaches such as NDL and Devil, CiD provides only a small hardware I/O kernel with need for optimization. In the converted NIC driver, there is a factor of 1.5 increase on the number of register I/O operations.

An important aspect that the current model does not support is separation of concerns and fine grained code reuse. As a result, device driver code includes OS-specific and device-specific code, leading to hard to maintain code [Padioleau et al., 2006]. The Termite project shows that separating those two aspects is possible [Ryzhyk et al., 2009b]. With CiD, a code template mechanism was used capture and reuse OS-specific code. As a result, the source code size of the NIC driver could be reduced by about 14%, but no code reductions were achieved for the mass storage driver. Also, a clean separation between OS-specific and device-specific code could not be achieved. NDL and Termite demonstrate that this is feasable, but at a change in the programming paradigm.

Another important aspect is that separation of concerns is key to addressing OS and device protocol violations. For example, in CiD, OS protocol constraints can be captured in templates or within the compiler as rules to the static concurrency analysis. However, CiD does not offer the flexibility to check API rule violations such as DDVerify [Witkowski et al., 2007]. More importantly, CiD does not offer a way to specify the operations of a device in a clean way such as Dingo [Ryzhyk et al., 2009a].

Despite CiD's limitations, I believe that the reliability of device drivers can be improved with the proposed language extensions. Investigations on driver reliability show that simple mistakes (such as calling a blocking function in atomic context) are surprisingly common [Ryzhyk et al., 2009a, Padioleau et al., 2006]. With CiD these faults are prevented by design. Also, compared to other (and more elaborate) approaches, the transition from C to CiD is seamless and can be automated with Coccinelle.

7.2 Future Directions

The next steps that should be taken are to incorporate the proposed language elements and the compiler modifications into GCC and to patch the driver tree with the extensions. A good starting point are CiD's function modifiers, which, together with a simple static call graph analysis, have the potential to prevent many deadlocks. Also, critical sections and deferred code in legacy drivers could be replaced with synchronized and deferred blocks, respectively. Coccinelle could be used to perform these conversions automatically on legacy device driver code.

Also, the existing compiler infrastructure should be further maintained because it provides an easy and fast way to test new ideas for language refinements and additions.

Language Improvements and Additions

Current additions to the language should focus on improving the core elements. What follows is a list for the most important improvements that should be made in the future. The individual tasks are ranked by priority in descending order.

Hardware I/O

1. Support register group aliases to allow reading or writing to all registers in the group with one statement.
2. Reduce the number of redundant read-modify-write operations by using the results of the concurrency analysis.
3. Support preactions and postactions for register accesses
4. Support overlay registers and bank switching
5. Support inheritance of descriptor layouts.

Concurrency and Synchronization

1. Make the syntax of protocols less cumbersome and verbose by adding a simple algebra for combining and reusing entry point lists. Also, support hierarchic composition of states to reduce typing amount.
2. Add first-class citizens for coordination, i.e., completions and wait-queues.
3. Add named synchronization blocks to override automatic locking instantiation (might be useful for shared locks, e.g., for block devices).
4. Add full support for closures (as demonstrated by deferred work) for asynchronous functions.
5. Add "deferred" function modifier to reduce the number of generated deferred work instances.
6. Evaluate elements from data-flow oriented languages to simplify drivers for message-based devices.

Templates

1. Improve current templates to yield better results for code reuse. Add device-independent logic to the NIC template to demonstrate feasibility.
2. Add support for different template versions to ensure backwards compatibility.
3. Extend the template language with conditional statements and loops to make template code more flexible.

Compiler Improvements

Although the current compiler implementation is simple and overall easy to maintain, more effort has to be put into making the analyses more resilient to (minor) language changes. Thus, current efforts should focus on improving the current compiler architecture. Following steps should be taken in the future:

1. Add a simplified intermediate language to make analyses more resilient to language changes.
2. Add more unit tests and test cases.
3. Document dependencies between analyses and which attributes they calculate.
4. Integrate a generic tree matcher like BURG into the compiler to simplify the atomic expressions and hardware I/O generator.

7.3 Lessons Learned

The initial goal of this thesis was to design an extensible domain specific language for the automatic generation of Linux device drivers. Unsurprisingly, this has proven to be an impossible endeavor because the prerequisite for such a language is a full-fledged domain model of computer hardware. Such a model could not be engineered, partly because of the intimidating amount of hardware specifications and different technologies, and partly due to the lack of experience in the field. The overwhelming complexity of the Linux kernel was yet another difficult obstacle that had to be faced.
The idea for simple language extensions as proposed in CiD has been developed quite early in the course of this work. At the beginning it seemed too trivial to be seriously considered. However, as time was passing by, compromises had to be made. As it turns out, it is worth to pursue even simple ideas and to think about them thoroughly.
Writing the compiler was another valuable experience. There have been two attempts. The first attempt was made with the compiler generator system Eli [University of Colorado at Boulder, 2011]. While the generator features a comprehensive set of powerful tools and specification languages, I came to the conclusion that it is difficult to work with, especially for newcomers that experiment with language design. The second attempt with PLY was an overall pleasant experience because it enabled me to write the compiler in a familiar programming paradigm.
Undoubtedly, the most challenging aspect of writing a compiler is to ensure completeness of all analyses. Even with languages like CiD that have small and simple grammars, it can be difficult to foresee every possible way a language construct can be used in a program. While today's tools and programming languages allow quick prototyping of a compiler, ensuring completeness and correctness of all analyses is the real difficult part.
Most importantly, writing kernel code has often proven to be a frustrating but also a rewarding experience. Writing kernel code takes a lot of discipline that is worth obtaining to be prepared for future challenges.

7.4 Compiler Availability

The CiD compiler and the driver files are hosted as a Sourceforge project and can be obtained at `http://sourceforge.net/projects/cdrivers`. For verification purposes, the distribution includes a snapshot, `thesis.tar`, of the compiler and driver files which were used to obtain the experimental data presented in Chapter 5. Further information on the compiler and latest updates can be found on the project website.
Contributions are very welcome!

Bibliography

[Adya et al., 2002] Adya, A., Howell, J., Theimer, M., Bolosky, W. J., and Douceur, J. R. (2002). Cooperative Task Management Without Manual Stack Management. In *Proceedings of the General Track: the 2002 USENIX Annual Technical Conference (USENIX-02)*, pages 289–302.

[Axelson, 2010] Axelson, J. (2010). USB Mass Storage Device Problems. Available online: http://lvr.com/device_errors.htm.

[Beazley, 2010] Beazley, D. (2010). PLY: Python-Lex-Yacc. Available online: http://dabeaz.com/ply/.

[Bellard, 2011] Bellard, F. (2011). QEMU: open source processor emulator. Available online: http://www.qemu.org.

[Bombieri et al., 2009] Bombieri, N., Fummi, F., Pravadelli, G., and Vinco, S. (2009). Correct-by-construction generation of device drivers based on rtl testbenches. In *Design, Automation and Test in Europe, DATE 2009*, pages 1500–1505.

[Bovet and Cesati, 2005] Bovet, D. P. and Cesati, M. (2005). *Understanding the Linux Kernel, 3rd Edition*. O'Reilly Media.

[Chou et al., 2001] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. (2001). An Empirical Study of Operating System Errors. In Ganger, G., editor, *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, volume 35, 5 of *ACM SIGOPS Operating Systems Review*, pages 73–88.

[Conway, 2010] Conway, C. L. (2010). NDL: The Network Device Language, official website. Available online: http://cs.nyu.edu/ cconway/ndl.

[Conway and Edwards, 2004] Conway, C. L. and Edwards, S. A. (2004). NDL: A Domain-Specific Language for Device Drivers. *ACM SIGPLAN Notices*, 39(7):30–36.

[Cooperstein, 2010] Cooperstein, J. (2010). *Writing Linux Device Drivers: A Guide With Exercises*.

[Dharm, 2010] Dharm, M. (2010). Observed USB Mass Storage Target Deviations from the Published Specification. Available online: http://one-eyed-alien.net/ mdharm/linux-usb/target_offenses.txt.

[Eklektix, 2010] Eklektix, I. (2010). Linux Weekly News. Available online: http://lwn.net.

[Free Software Foundation, 2010] Free Software Foundation (2010). GNU Patch. Available online: http://savannah.gnu.org/projects/patch/.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). The Visitor Pattern. In *Design Patterns. Elements of Reusable Object-Oriented Software*, pages 331–344. Addison-Wesley.

[Härtig et al., 1997] Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., and Wolter, J. (1997). The Performance of μ-Kernel-based Systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 66–77. ACM Press.

[Herder, 2010] Herder, J. N. (2010). *Building A Dependable Operating System: Fault Tolerance in Minix 3*. Vrije University Amsterdam.

[Herder et al., 2009] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. (2009). Fault Isolation for Device Drivers. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 33–42.

[Herder et al., 2006] Herder, J. N., Bos, H., and Tanenbaum, A. S. (2006). A Lightweight Method for Building Reliable Operating Systems Despite Unreliable Device Drivers. Technical Report IR-CS-018, Department of Computer Science, Vrije Universiteit.

[Hsieh et al., 1995] Hsieh, W., Fiuczynski, M., Garrett, C. D., Savage, S., Becker, D., and Bershad, B. N. (1995). Language Support for Extensible Operating Systems. Technical Report TR-95-11-02, University of Washington, Department of Computer Science and Engineering.

[Hunt and Larus, 2007] Hunt, G. C. and Larus, J. R. (2007). Singularity: Rethinking the Software stack. *Operating Systems Review*, 41(2):37–49.

[Intel, Corp., 2009] Intel, Corp. (2009). PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual. Available online: http://download.intel.com/design/network/manuals/8254x_GBe_SDM.pdf.

[Jonathan Corbet and Kroah-Hartman, 2005] Jonathan Corbet, A. R. and Kroah-Hartman, G. (2005). *Linux Device Drivers, Third Edition*. O'Reilly Media.

[Kadav et al., 2009] Kadav, A., Renzelmann, M. J., and Swift, M. M. (2009). Tolerating hardware device failures in software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009*, pages 59–72.

[Kroah-Hartman et al., 2009] Kroah-Hartman, G., Corbet, J., and McPherson, A. (2009). Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. Linux Foundation.

[Landley, 2008] Landley, R. (2008). Where Linux Kernel Documentation Hides. In *Proceedings of the Linux Symposium*, volume 2, pages 7–19.

[Leslie et al., 2005] Leslie, B., Chubb, P., Fitzroy-Dale, N., Götz, S., Gray, C., Macpherson, L., Potts, D., Shen, Y.-T., Elphinstone, K., and Heiser, G. (2005). User-Level Device Drivers: Achieved Performance. *J. Comput. Sci. Technol*, 20(5):654–664.

[Li et al., 2004] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. (2004). CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *OSDI*, pages 289–302.

[Mérillon et al., 2009] Mérillon, F., Réveillère, L., Consel, C., Marlet, R., and Muller, G. (2009). Devil: An IDL for Hardware Programming.

[Microsoft, 2011] Microsoft, C. (2011). SDV: Static Driver Verifier. Available online: http://msdn.microsoft.com/en-us/library/ff552808

[Muller, 2010] Muller, G. (2010). Coccinelle, official website. Available online: http://coccinelle.lip6.fr.

[Padioleau et al., 2008] Padioleau, Y., Hansen, R. R., Lawall, J., and Muller, G. (2008). Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *Proceedings of the EuroSys 2008 Conference*, pages 247–260. ACM.

[Padioleau et al., 2006] Padioleau, Y., Lawall, J. L., and Muller, G. (2006). Understanding Collateral Evolution in Linux Device Drivers. In *EuroSys*, pages 59–71.

[Realtek, 2002] Realtek, R. S. C. (2002). Realtek 3.3V Single Chip fast Ethernet Controller with Power Management. RTL8139C(L).

[Redpill Linpro AS, 2010] Redpill Linpro AS (2010). The Linux Cross Reference. Available online: http://lxr.linux.no.

[Renzelmann and Swift, 2009] Renzelmann, M. J. and Swift, M. M. (2009). Decaf: Moving Device Drivers to a Modern Language. In *Proceedings of the USENIX Annual Technical Conference*.

[Ryzhyk et al., 2009a] Ryzhyk, L., Chubb, P., Kuz, I., and Heiser, G. (2009a). Dingo: Taming Device Drivers. In *EuroSys*, pages 275–288.

[Ryzhyk et al., 2009b] Ryzhyk, L., Chubb, P., Kuz, I., Sueur, E. L., and Heiser, G. (2009b). Automatic Device Driver Synthesis with Termite. In *SOSP*, pages 73–86.

[Ryzhyk et al., 2010] Ryzhyk, L., Zhu, Y., and Heiser, G. (2010). The case for active device drivers. In *Proceedings of the 1st ACM SIGCOMM Asia-Pacific Workshop on Systems*, pages 25–30.

[Spaans, 2010] Spaans, J. (2010). The Linux Kernel Mailing List Archive. Available online: http://lkml.org.

[Swift et al., 2002] Swift, M. M., Martin, S., Levy, H. M., and Eggers, S. J. (2002). Nooks: an architecture for reliable device drivers. In *Proceedings of the 10th ACM SIGOPS European Workshop*, pages 102–107.

[University of Colorado at Boulder, 2011] University of Colorado at Boulder, University of Paderborn, M. U. (2011). Eli: An Integrated Toolset for Compiler Construction. Available online: http://eli-project.sourceforge.net/.

[Venkateswaran, 2008] Venkateswaran, S. (2008). *Essential Linux Device Drivers*. Prentice Hall International.

[Witkowski et al., 2007] Witkowski, T., Blanc, N., Kroening, D., and Weissenbacher, G. (2007). Model checking concurrent linux device drivers. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 501–504.