Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Reusing I/O-Kit Device Drivers on L4

Kim-Thomas Möller

Studienarbeit

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter: Joshua LeVasseur, Gerd Liefländer

3. Mai 2005

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 3. Mai 2005

Kim-Thomas Möller

Contents

Chapter 1

Introduction

1.1 Motivation

1.1.1 Reusing Device Drivers

Device drivers can be regarded as a layer between peripheral hardware devices and software applications. They do the low level work to "run" hardware devices and relieve the applications from having to know how to access each specific device. Through the abstractions that they establish, device drivers present a uniform view onto different devices and often make it possible for several applications to use the same device in turn. In order to work efficiently, device drivers need intimate knowledge of how the processors interact with devices. Thus, in a traditional monolithic operating system they are usually implemented in the kernel.

As the number of hardware devices as well as the number of operating systems steadily increases[1], device driver reuse is becoming more and more important. Unfortunately, the very nature of device drivers makes it difficult to reuse them: They heavily depend on the abstractions and mechanisms provided by the operating system kernel that they were written for.

I/O-Kit, the device driver subsystem of the Darwin kernel[2] is designed as an object-oriented framework [5]. Frameworks represent a generic solution to a set of similar problems (in this case, device driver management) and are well suited for being reused [18]. Thus, I/O-Kit is a good starting point for constructing kernel-independent device drivers.

Encapsulation and inheritance make it unnecessary (albeit not impossible) for device drivers to directly use kernel abstractions from outside I/O-Kit. If an operating system could emulate I/O-Kit's services, any I/O-Kit device driver would run on that system without (or with only minor) modification.

[1]For example, the device driver ratio in the Linux kernel increased from slightly over 50% in 1997 to about 70% in 2001 [11]. The device driver ratio in the Darwin kernel (not counting third-party drivers) doubled from 24% in April 2001 (Darwin 1.3.1) to 55% in February 2005 (Darwin 7.8) [4].

[2]Darwin's kernel is colloquially called "xnu".

1.1.2 User-Mode Device Drivers

Running device drivers in user mode has been a goal for operating system designers for a long time [14,17,31], because running device drivers in kernel mode has several disadvantages:

- A flawed or malicious driver can distract the whole kernel or even put it out of action.
- The kernel is required to adapt to different devices, which introduces the need for either recompiling the kernel or handling loadable kernel extensions.
- In any case, the kernel becomes bigger, unsafer and more complex, making optimization (e.g. of TLB or hardware caches) more difficult.

On the other hand, running device drivers in user mode has also some drawbacks:

- When an application issues a request to a device driver, the request is channeled through the kernel, with the additional cost of at least one kernel entry and exit and usually an address space switch.
- When a driver receives data and passes it on to an application, entering and exiting the kernel and a switch of the address space are necessary again.
- The kernel has the additional burden to manage memory and device access rights for the drivers.

Maximum system dependability can only be achieved with user mode device drivers [27], whereas the modest performance drawbacks may eventually be compensated by faster kernel/user-mode and address space switching procedures.

1.1.3 Thesis Goal

In this study thesis, I examine the reuse of a device driver originally written for use within I/O-Kit (the driver for the 3COM 3C905 network interface card) on the L4 microkernel. I concentrate on I/O-Kit's static structure, leaving out dynamic driver loading. Although I/O-Kit's modular nature suggests splitting it up into a multiserver system[3], I do not explore that either. The degree of reuse is limited to reuse of source code, because the intended changes are too far-reaching to maintain binary compatibility (e.g. the ported driver will use the ELF executable format that is more common on the IA32 platform than Darwin's Mach-O format). My implementation is neither optimized for performance nor does it include I/O-Kit's huge set of facilities to communicate with application programs[4].

[3] A multiserver system is an operating system consisting of a set of hardware-protected tasks [19].

[4] In the Darwin kernel, application programs can interact with I/O-Kit through at least three mechanisms: Mach ports, BSD device files, or user space clients [5].

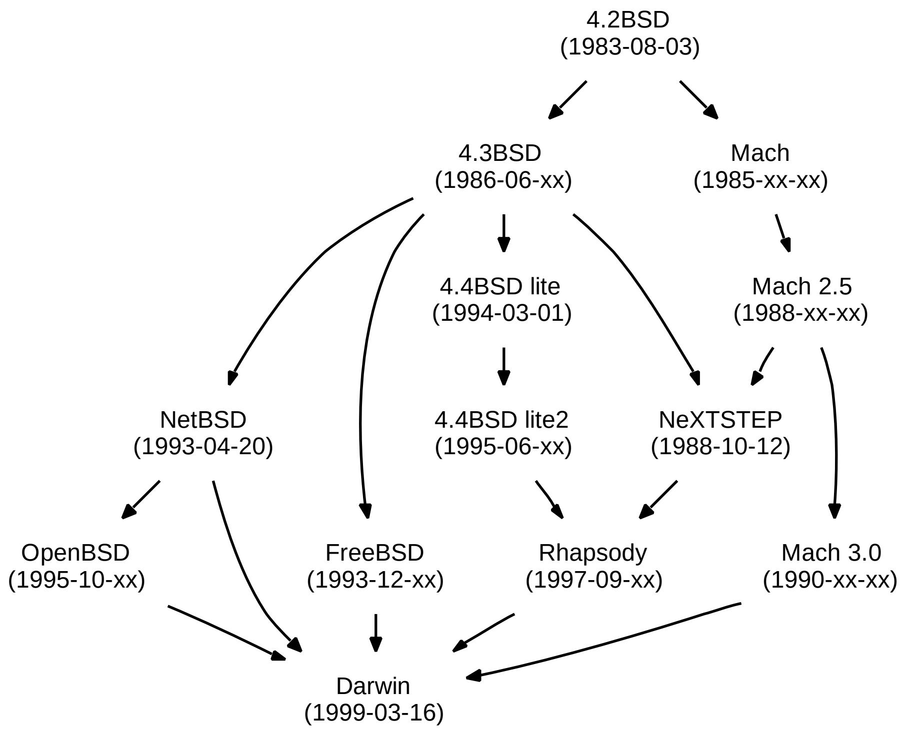

Figure 1.1: Simplified overview over Darwin's history [28,37]

1.2 Thesis Outline

For the remainder of this chapter, I will proceed by explaining the basic concepts of the Darwin kernel, I/O-Kit, and L4. The second chapter contains an overview of similar work, most notably OSKit and device driver reuse via virtual machines. In the third chapter, I will resolve which abstractions a specific driver uses by exemplification of a 3COM 3C905 network interface card (NIC). In the forth and fifth chapter I will present my design and implementation of a runtime system for the driver based on certain parts of the Darwin kernel. In the sixth chapter, I will discuss the achievements, limitations and potential improvements of my approach.

1.3 The Darwin Kernel

1.3.1 History

The provenance of the Darwin kernel is shown in Figure 1.1. The basis of the Darwin kernel is Mach, which has been developed as a first-generation microkernel at Carnegie Mellon University in the 1980s [35]. Mach is in turn based on BSD 4.2 from University of California, Berkeley. Mach 2.5 still included BSD as a so-called colocated server in its kernel, which is why it was even larger than the corresponding BSD kernel. The NeXT company licensed Mach 2.5 and used

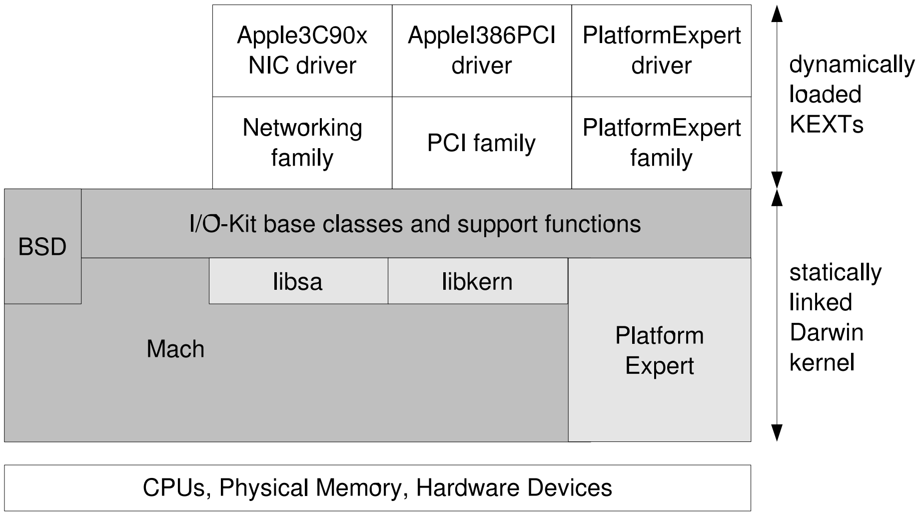

Figure 1.2: Structure of the Darwin kernel with respect to I/O-Kit

it as foundation for their NeXTSTEP operating system. When NeXT's founder Steve Jobs returned to his previous position as CEO of the Apple company, he took with him not only several chief software architects of NeXT's, but also the source code of NeXTSTEP. He pushed Apple to build their own operating system based on NeXTSTEP, which was later renamed Rhapsody. The kernel code was updated to Mach 3.0 and BSD 4.4, later on extended with features from FreeBSD, OpenBSD and NetBSD.

But neither BSD nor Mac OS 9 and its predecessors were equipped with a device driver subsystem that suited modern kernels and hardware. For example, they were neither reentrant nor safe for multiprocessing[5], and Mac OS 9 did not even have protected memory and preemptive multitasking[6] [2]. Therefore, Apple decided to develop I/O-Kit, a new object-oriented device driver subsystem [5].

1.3.2 Kernel Components

Today, the Darwin kernel consists of three major components [3] (colored dark gray in Figure 1.2):

- Mach provides base abstractions to the kernel such as memory and thread management.
- BSD provides a UNIX interface for applications to access the kernel.
- I/O-Kit provides device driver management.

[5]A reentrant piece of code may be executed concurrently by several threads of control. Operating systems for multiprocessor machines have to take special care of synchronization, because several processors can modify memory contents at once.

[6]In a system that provides protected memory, a process may only write to memory that is assigned to it. With preemptive multitasking, a process can not keep running for an infinite time: When the process's so-called time slice expires, it is forced to relinquish the CPU.

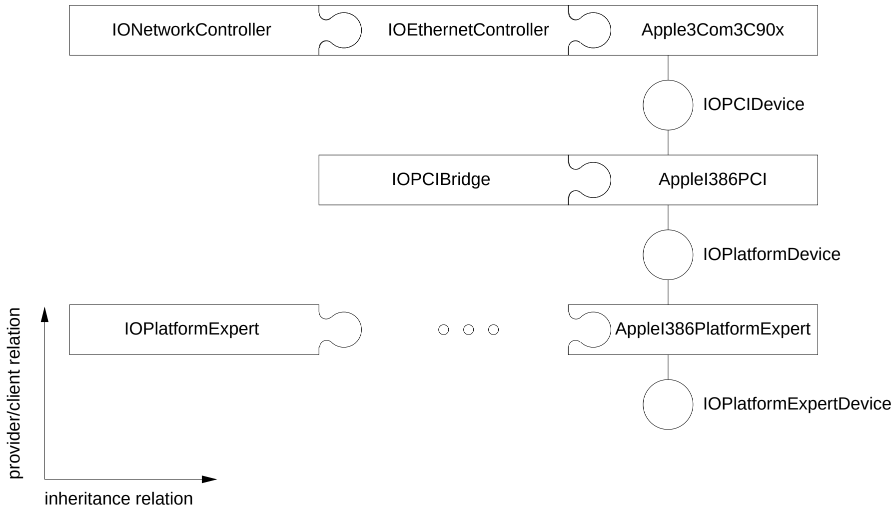

Figure 1.3: Relations between I/O-Kit classes

Besides, there are several smaller kernel components (colored light gray in Figure 1.2):

- Libkern is an Embedded C++ [12] runtime environment. It includes facilities for dynamic object allocation and some generic base classes for I/O-Kit.
- Libsa is occupied with runtime loading and unloading of modules (called kernel extensions or KEXTs).
- Platform Expert helps booting the system and interfacing with BIOS or OpenFirmware.

1.4 I/O-Kit

1.4.1 Design

When designing I/O-Kit, its developers wanted to create a device driver subsystem that would adapt neatly to the requirements of today's computer usage (such as hotplugging and power management) and today's hardware (such as diverse bus systems, multiprocessing and DMA data transfers). So they set up an object-oriented framework in which device drivers are represented as instances of device driver classes [5]. Specific driver classes are inherited from more general classes that encapsulate behavior that is common to a set of device drivers, the so-called device driver family[7]. The third fundamental I/O-Kit concept besides families and device drivers are so-called nubs, which are exported by bus drivers and represent physical hardware devices that a driver can attach to.

Figure 1.3 gives an abbreviated example of the relations between device drivers. Device drivers are shown as pieces of a puzzle, connected to other drivers

[7]In this respect, the 3COM 3C905 NIC driver is inherited from the IOEthernetController class that is part of the Networking family.

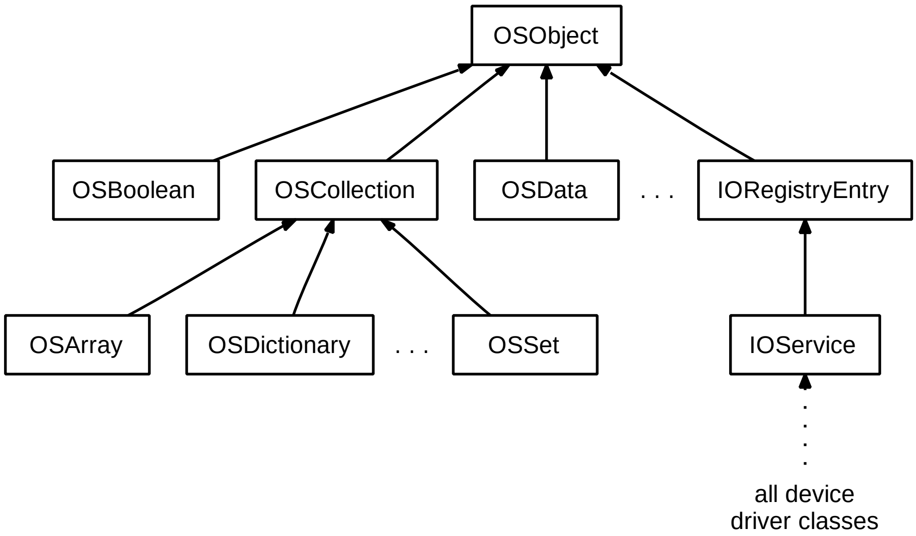

Figure 1.4: I/O-Kit's base class inheritance tree

from their family. Nubs are shown as circles connecting drivers from different families. The two fundamental dimensions of the class design are inheritance (from left to right) and provider/client (from bottom to top). The inheritance tree[8] is rooted at the IOService class (not shown in the Figure), whereas the provider/client tree is rooted at the IOPlatformExpertDevice nub.

1.4.2 Life Cycle of a Device Driver

Figure 1.4 shows I/O-Kit's base class inheritance tree. All classes in the kernel must be inherited from OSObject in order to conform to the runtime type information system. All device drivers are registered in I/O-Kit's central I/O Registry, that is why they are furthermore inherited from IORegistryEntry. The abstract behavior of a device driver is encapsulated in IOService, IORegistryEntry's sole direct child class.

IOService provides a standard life cycle for a device driver, as shown in 1.5. After constructing the driver object and calling its init() method[9], the driver is attached to a nub in the I/O Registry. Now it can be probed to drive a specific hardware device. When calling the driver's start() method, it is activated to drive the device. The semantics of the open() and close() methods are defined by its family. After stopping the driver from driving the device, it can be detached from its nub. Finally it frees its resources.

1.4.3 Implementation Language

I/O-Kit is implemented in Embedded C++, a restricted subset of C++ [12]. The restriction affects C++ features that cause either performance problems

[8] Multiple inheritance is disabled in Embedded C++.

[9] Object creation and initialization are split because exceptions (which otherwise could report creation failure) are not allowed in Embedded C++.

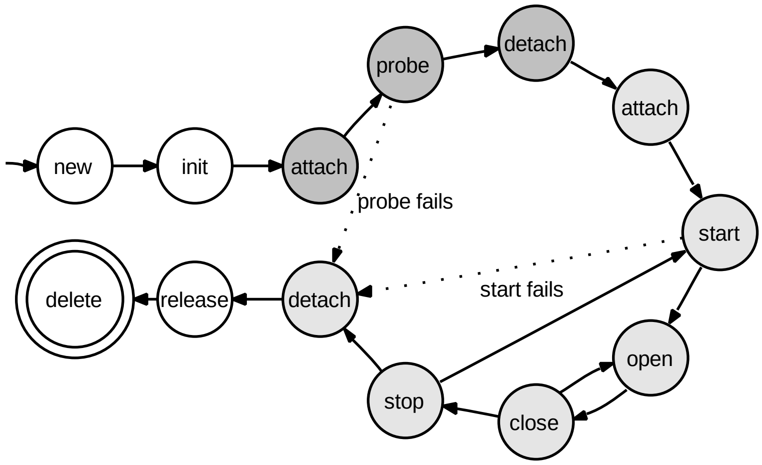

Figure 1.5: A Device Driver's Life Cycle

or problems due to code being executed in kernel mode: exceptions, multiple inheritance, templates, and runtime type information [13]. Nonetheless, Apple's I/O-Kit documentation confirms that a device driver should not depend upon these features anyway [5].

1.5 L4

1.5.1 History

L4 is a second generation microkernel that can be used as an efficient and machine independent base for virtualizing operating systems [26, 27]. It was originally developed by Jochen Liedtke [32] and is now being improved jointly by the System Architecture Group at Universität Karlsruhe, the DiSy Research Group at University of New South Wales, Sydney, and the Research Group Operating Systems at the Technische Universität Dresden.

1.5.2 Design Guideline

The design guideline for L4 is to implement a concept in the kernel only if it is impossible to implement it in user space without inevitably loosing required functionality (e.g. security and performance) [29]. From a user space application's point of view, L4 abstracts the concepts it handles and provides mechanisms to manage the abstractions.

1.5.3 Fundamental Abstractions

L4 is based on only two abstraction primitives:

- An address space is a set of virtual addresses that a user space application may access.
- A thread is an activity within a user space application, characterized by its current processor register state (including its instruction pointer) and its stack.

For both abstractions, there are operations to allow for interaction:

- Threads use inter-process communication (IPC) for information exchange.
- Subsets of address spaces (described by so-called flexpages) can be mapped or granted in order to recursively construct address spaces. A mapping allows both mapper and mappee to access the page. The mapper can revoke the mapping again. In contrast, a granting is only accessible by its receiver, and can not be revoked.

Access to hardware devices is handled by means of the two abstractions:

- Interrupts are captured by L4 and transformed into virtual interrupts, i.e. an IPC message from a virtual interrupt thread to a user-level driver thread.
- Besides virtual memory, I/O ports and I/O memory mappings are also treated by L4 as address spaces [39].

L4's specification can be found in the L4 Kernel Reference Manual [25].

Chapter 2

Related Work

2.1 Overview

Reusing device drivers is a common practice in operating system design, usually in order to avoid the tedious task of writing the drivers from scratch. For example, device drivers from the BSD operating system family have been reused many times: in Mach 2.5 [17], in SPIN [8] and in VINO [38]. BSD device drivers were also adapted to Linux and vice versa [43].

The major problem when reusing device drivers is that the drivers are embedded into a set of global functions and variables, their runtime environment. Conserving this environment can bear several problems: sometimes the environment's extension is a priori unknown, often proper documentation is missing, and potentially the environment is based on abstractions from the original system that are not available within the target system.

Reusing a device driver is easier if its original environment and the target environment are similar. This is why device drivers are usually reused within their operating system family, e.g. within Unix-like operating systems or Windows.

Although the Darwin kernel incorporates a Unix-like interface, its kernel primitives are supplied by Mach, with BSD operating on top of it [2]. Especially, I/O-Kit relies mostly on Mach's functionality and is only loosely connected to BSD. This means that not all of the insights gained from previous work on porting Unix-based drivers can be applied unalteredly to porting I/O-Kit drivers.

2.2 Avoiding to reinvent the Wheel

2.2.1 Linux Device Driver Emulation in Mach

In order to augment the Mach research operating system with more advanced device drivers from Linux 1.3.35, Goel and Duchamp designed an emulation library to run unmodified Linux device drivers on Mach [22]. Unfortunately, Linux and Mach both already incorporated I/O subsystems which were based on fundamentally different design decisions.

Goel and Duchamp identified several conflicts, which their emulation library had to circumvent:

- Both systems name devices differently and implement different ioctl calls.
- Both systems have different approaches to synchronizing I/O operations and make different assumptions about concurrency.
- Both systems have different address space layouts and use different block sizes for I/O operations. Linux furthermore distinguishes several regions within physical memory (e.g. for DMA-capable devices).
- Linux device drivers make some assumptions (e.g. that the clock is running at device initialization time).

Compared to that approach, running I/O-Kit device drivers on L4 is much simpler, because, in contrast to Mach, L4 imposes hardly any policies on systems running on top of it [30].

2.2.2 OSKit

The OSKit was developed by Ford et. al. at University of Utah in the effort to accelerate kernel prototyping by modularizing libraries that implement functionality that is commonly used, but from a design viewpoint rather bland [15,16]. The developers extracted most device drivers from BSD, Linux and Mach and encapsulated them in "glue code", which represents the original runtime environment to the device driver and exports a unified device interface to the rest of the kernel. Due to its modular design, OSKit is used within many projects.

In his Masters thesis [41], Van Maren describes his experiences in running device drivers from OSKit in user mode on the Fluke microkernel. Van Maren identifies the following issues:

- synchronization
- interrupt delivery
- physical memory allocation
- shared resources

One can expect to encounter the same issues when porting I/O-Kit drivers to the L4 microkernel.

Nonetheless, Van Maren aims at reusing device drivers from UNIX-like operating systems. I/O-Kit, in contrast, is quite different in its structure and operation. For example, traditional UNIX device drivers consist of two parts: one is executed in process context, and the other is executed in interrupt context[1]. Data is passed between these parts with the help of synchronization primitives — mutexes and condition variables. I/O-Kit takes a different approach, the Workloop[2] concept, for synchronizing data exchange. Thus, when porting I/O-Kit, the previously mentioned issues apply within a different context.

[1] BSD used the notions "top half" for the first and "bottom half" for the latter [33]. Linux seems to have reversed their meaning.

[2] A Workloop encapsulates a dedicated thread that serializes a device driver's activities that must not execute concurrently.

2.3 Increasing Device Driver Dependability

2.3.1 Nooks

Another motivation for reusing device drivers is the wish to make device drivers more dependable. The Nooks project [40] explores the reuse of device drivers within their original environment, a kernel extension, but within separate protection domains. For that approach, a transparent reliability layer that separates kernel and kernel extensions was introduced. The device drivers still execute in kernel mode. A total number of 22,266 non-comment lines of source code were changed, which is relatively high, considering that the additional code might introduce new bugs.

2.3.2 Sawmill

The objective of the Sawmill project [19] was to create a multiserver system based on the Linux kernel. Although the Sawmill project provided deep insights into design issues of multiserver systems, it turned out that the Linux kernel was weakly structured, making the transformation into a multiserver system difficult.

2.3.3 Unmodified Device Driver Reuse via Virtual Machines

Some members of my research group[3] have recently proposed a concept for device driver reuse that differs from others in its extent of maintaining the driver's environment [26,27]. They observe that usually device drivers are tightly coupled to their kernel, whereas application programs enjoy the comfort of well defined kernel APIs that greatly simplify porting them to different kernels. The authors identify five orthogonal design principles, which are indispensable for reusable and dependable device drivers, but seldom fulfilled by legacy operating systems:

- Resource delegation: The device drivers themselves are in charge of refining their resources and managing their memory.
- Separation of name spaces: Each driver must run in a separate address space.
- Separation of privilege: Device drivers must run in user mode.
- Secure isolation: Device drivers must not affect non-trusting components such as other drivers.
- Common API: The resource allocation interface must be clearly defined.

In order to accomplish these principles, the authors leave each device driver inside its native device driver OS (DD/OS) that runs on a virtual machine in a virtualized or paravirtualized[4] environment. The DD/OS is augmented by a

[3] System Architecture Group, Universität Karlsruhe

[4] Paravirtualized means that a slightly modified operating system runs on a specialized guest architecture [44].

translation module that exports its services to other clients. In this way, device drivers can be reused without being modified.

The drawbacks of this approach are the necessity for handling DMA and IO address translations, potential timing violations due to increased latency, and the resource overhead imposed by the DD/OS. The resource consumption can be mitigated by several effective techniques, including sharing memory between DD/OS's, swapping, special-caseing zero pages, memory ballooning [42] and compression.

The authors achieve a code reuse ratio of 99.9% for the researched NIC driver itself and 91% when including the virtual machine monitor and paravirtualization modifications. The paravirtualization technique has only minor drawbacks on system performance: The authors compare native Linux and paravirtualized Linux and report 3 to 8 % throughput loss for networking in the TTCP benchmark and a throughput within half of the standard deviation for the streaming disk benchmark. CPU utilization relative to native Linux increases up to about 2x depending on the respective driver and the direction of data transfer.

Although the reuse of I/O-Kit device drivers will require more modifications to the device driver's runtime environment than what LeVasseur et. al. did, several design principles are satisfied as well:

- Resource delegation and a common API are inherent to I/O-Kit device drivers due to their object oriented and well-documented design.
- Separation of privilege and secure isolation are assured by running I/O-Kit and its device drivers in user mode on top of the secure L4 microkernel.
- Separation of name spaces and secure isolation will not be implemented yet, but there are no indications that per-driver address spaces should not be possible.

Chapter 3

Analysis of I/O-Kit

3.1 Roadmap to reusing I/O-Kit Device Drivers

In the previous chapter, we have seen several examples of porting device drivers to a different operating system. The examples have in common that they leave the device drivers more or less intact, but they differ in what to consider as the driver's runtime environment: either only those functions and variables that the device driver accesses itself, the whole device driver subsystem or the entire operating system.

The question of how to define the device driver's runtime environment also emerges when porting an I/O-Kit device driver. On the one hand, I/O-Kit's design as a framework simplifies reusing it on the whole. On the other hand, I/O-Kit is only one of the Darwin kernel's components (see Figure 3.1). The other kernel components are of minor importance for device drivers. Thus, in order to take full benefit from framework reuse and avoid porting components that are not needed for the device driver system, I decided to detach I/O-Kit from the rest of the Darwin kernel. Porting I/O-Kit to L4 then consists of the following tasks:

- First, I/O-Kit will be detached from the rest of the Darwin kernel. The result is the syntactical interface between I/O-Kit and its runtime environment.
- Second, the semantics of I/O-Kit's runtime environment need to be analyzed.
- Third, the environment's functionality will be redesigned and reimplemented as an emulation library based on L4 primitives. Due to L4's proximity to hardware, porting I/O-Kit to L4 can be interpreted as porting it to a "virtual L4 machine" architecture, similarly to the L4Linux 2.6 approach [26].

The resulting system, e.g. the emulation library and the device drivers, will be named L4-I/O-Kit.

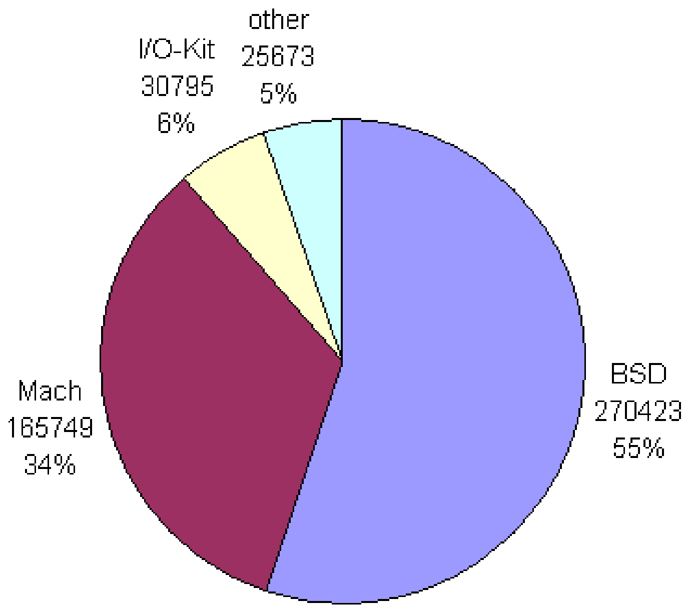

Figure 3.1: Size of the Darwin 7.8 kernel components
Absolute numbers are given in physical source lines of code (SLOC).

3.2 Identifying the Runtime Environment

Within the next sections, I examine what impact the Darwin kernel's components have on I/O-Kit and what needs to change when running I/O-Kit on L4. Following a bottom-up approach, I start with the Mach component, proceed with the other kernel components, the I/O-Kit base system, the device drivers, and conclude with the 3COM 3C905 driver itself.

3.3 The Mach Component

Although I/O-Kit drivers usually need not access the Mach kernel component directly, the device families, the I/O-Kit base classes, libkern base classes and libsa do so. Thus, some of Mach's subsystems need to be emulated.

From Mach's vast set of functions, I/O-Kit and its device drivers use only a relatively small subset. This functionality can be subdivided into the following categories:

- debugging and character string manipulation
- kernel memory allocator
- address space management
- thread management

- interrupt management
- synchronization

3.3.1 Debugging and character string manipulation

Character string manipulation functions in the Mach component are similar to those from the C standard library. There also exist some functions to help debugging the kernel: asserting that an expression holds, reporting a kernel panic and writing debugging messages to the console. All this functionality is rather self-contained and thus does not impose problems on constructing L4-I/O-Kit.

3.3.2 Kernel Memory Allocator

Allocation of small chunks of memory for the kernel's purposes is done by the kernel memory allocator. The kalloc and kfree functions reserve or free memory on the kernel's heap. Small requests result in giving out chunks from a power-of-two list, whereas large requests are satisfied by handling out entire memory pages. More information on the internal design of the kernel memory allocator used by Mach and BSD can be found in [33].

3.3.3 Thread Management

Mach has a full-featured thread management system. This system is based on several notions:

- Thread shuttles are the entity of execution within the Darwin kernel, whereas thread activations represent the resources associated to a shuttle [1].
- While in active state, a thread belongs to a run queue.
- Before executing a blocking operation, a thread can set a timer to specify that it wants to be woken up after at least the given time.
- When blocking, a thread can specify an event when it wishes to be woken up. There is a wait queue attached to each event where the thread is inserted while it blocks. When a thread signals the event, either one or all threads on the event's queue are woken up.
- For blocking operations, a continuation can be specified, e.g. a predefined state in which a thread starts when it becomes runnable again.

Although I/O-Kit provides a simplified interface for device drivers, a full-featured emulation library would have to mimic Mach's thread management.

3.3.4 Interrupt Management

The Mach component of the Darwin kernel is the starting point for interrupt handling. Mach simply passes interrupts to IOKit's platform dependent IOInterruptController instance.

3.3.5 Synchronization

Mach has four synchronization primitives:

- Simple locks are spin locks. They are the safest synchronization primitive, but their CPU utilization is high.
- Mutexes are sleep locks. A thread that tries to acquire a mutex that is already held by another thread goes to sleep until the mutex is free. Mutexes allow for SMP synchronization.
- Read-write locks (or just locks) are sleep locks that allow either an arbitrary number of readers or a single writer to access the data they protect. According to comments in the Darwin kernel source code, they are not SMP safe.
- Semaphores are objects for interprocess synchronization. They are also used by some parts of I/O-Kit.

Mutexes, read-write locks and semaphores interfere with thread management. The emulation library has to reproduce locking behavior.

3.4 The PlatformExpert Component

Aside from interfacing with the boot loader and interrupt handler installation, I/O-Kit has no dependencies on PlatformExpert.

3.5 The BSD Component

I/O-Kit drivers must not rely on the BSD kernel component, because that part of the kernel is generally considered high-level (as opposed to device drivers) and is neither multithreaded nor multiprocessing safe (which is why the so-called BSD and networking funnels[1] are needed for serialization) [5].

The only exception is that the Networking family uses BSD's networking protocol stack. Thus, the emulation library needs to implement the data link interface layer (DLIL), which is a generalization of BSD's ifnet architecture [2]. A complete L4-I/O-Kit implementation needs to enable applications to access the 3COM 3C905 driver, for example via an IDL interface [23].

3.6 The Libkern Component

Libkern contains the base of I/O-Kit's class hierarchy. It implements a custom runtime type information (RTTI) system that enables objects to be constructed by name. Furthermore, libkern provides classes for generic objects such as numbers, sets, and iterators. In its implementation, Libkern relies on Mach's memory management, string manipulation, debugging, and synchronization. Thus, Libkern can be ported to L4 without any additional requirements for the emulation library.

[1] A funnel is a synchronization primitive similar to a mutex. What makes the difference is that a funnel protects code, not data. It is automatically released when a process holding it blocks and reacquired by the process when it becomes runnable again.

3.7 The Libsa Component

Libsa is responsible for loading extensions into the kernel and setting up their runtime environment. L4-I/O-Kit does not support dynamic loading of extensions, so I do not need to detail this.

3.8 The I/O-Kit Base System

All I/O-Kit device drivers and device driver families are based on classes that are statically compiled into the Darwin kernel. Besides, I/O-Kit contains some helper functions for I/O-Kit drivers to use, which are also known as IOLib.

3.8.1 Locking

I/O-Kit locks are modeled on top of Mach's locking primitives [20]. IOLocks are simply another name for Mach mutexes, and IOSimpleLocks are Mach spinlocks. IORWLocks are the same as Mach's locks with multiple readers and one writer. IORecursive Locks can be acquired recursively by the same thread several times without causing a deadlock.

3.8.2 Thread Management

IOThreadSelf() returns a unique identifier for the currently running thread. With IOCreateThread() one creates a thread running a given function. A thread is terminated with a call to IOExitThread(). IOSleep() lets a thread sleep for a given amount of time. It is implemented with assert_wait_timeout() and thread_block() from Mach. IODelay() lets a thread turn in a spin loop for a short time, using Mach's delay() function. IOLog() and IOPanic() wrap Mach's printf() and panic() functions.

3.8.3 Memory Management

IOMalloc() and IOFree() are simple wrappers for Mach's kalloc() and kfree() functions. There are similar functions to allocate aligned memory, contiguous memory, memory with a specified physical address, or pageable memory. All these functions again resort to Mach memory management functions.

3.8.4 Workloops and Event Sources

Workloops encapsulate I/O-Kit's basic mechanism to serialize access to a device. A Workloop is simply a thread that walks event sources attached to it (e.g interrupt event sources) in order to let them do their work. If the Workloop notices that there are no more events waiting to be processed, it goes to sleep an a semaphore until it is woken up again to resume its work. With this mechanism, almost all I/O access can be done in thread context.

3.9 The Device Families and Device Drivers

Most prominently, an I/O-Kit device driver is based on its own device family. For example, the 3COM 3C905 driver is based on the Networking family. The family descends from the kernel resident I/O-Kit base classes. But as shown in Figure 1.3, the device driver also interacts with other classes and families that provide its resources: the corresponding PCIDevice nub, the AppleI386PCI driver and its family, the IOPlatformDevice nub, the AppleI386PlatformExpert driver and its family, and finally the IOPlatformExpertDevice, the root of the device tree.

The families are designed to be general, so that dependency on the hardware architecture is factored out in the device driver classes. For constructing L4-I/O-Kit, only the device drivers need to be modified.

3.9.1 AppleI386PlatformExpert device driver

The AppleI386PlatformExpert derives from IOPlatformExpert. It detects the PCI bus and the programmable interrupt controller (PIC).

3.9.2 AppleI386PCI device driver

The AppleI386PCI device driver is a subclass of IOPCIBridge from the PCI family. Its 13 data members and 25 methods are mainly occupied with reading and writing to the PCI status registers. On startup, it retrieves information about the PCI bus from PlatformExpert and enables the supported PCI configuration methods and PCI special cycle methods. Its configRead() and configWrite() methods modify the PCI configuration space by reading and writing to the IA32 processor's I/O ports.

3.9.3 3COM 3C905 device driver

The 3COM 3C905 device driver is responsible for driving one 3COM 3C905 NIC device. The Apple3Com3C90x driver class is derived from the IOEthernetController class which is part of the Networking family. The driver class's methods can be subdivided into the domains of

- MII/PHY support,
- kernel debugger support,
- media support,
- power management, and
- general methods.

The driver only accesses I/O ports directly. Its PCI configuration space is accessed by its provider, the AppleI386PCI driver. Interrupts are handled by its workloop object, and memory management is encapsulated in classes from the I/O-Kit base system.

Chapter 4

Design of an I/O-Kit Emulation Library

4.1 Structure of L4-I/O-Kit

As a result of the analysis phase, the Darwin kernel's source code can be classified to be either reused in original form, reused in modified form, or purged in order to construct an independent runtime environment for device drivers.

Figure 4.1 depicts the components and how they are reused in L4-I/O-Kit. Some parts of the 3COM 3C905 driver's original runtime environment can be reused without modifications (colored light gray in Figure 4.1):

- The libkern kernel component provides a C++ runtime environment. It is based on other kernel components and does not require any hardware resources other than main memory.
- The device driver families are virtually independent from hardware implementations.
- The 3COM 3C905 driver only accesses the IA32 processor's I/O ports directly. It controls all other resources, such as memory and interrupts, through its family and through its provider nub, the PCI driver. Although L4 restricts access to I/O ports [39], it does not abstract them. So the device driver can run unmodified on L4-I/O-Kit.

Some parts of the runtime environment need to be reimplemented (colored dark gray in Figure 4.1):

- The emulation library subsumes functionality from Mach, BSD, libsa, and PlatformExpert.
- The PCI and the PlatformExpert drivers can be derived from the respective I386 (IA32) driver classes, but they need to detect existing hardware by themselves rather than access data provided by the Darwin boot loader.

The I/O-Kit base system needs to be modified in order to access hardware through L4's abstractions. For now, L4-I/O-Kit runs as root server, which means it is trusted by L4 and does not need to request resources explicitly.

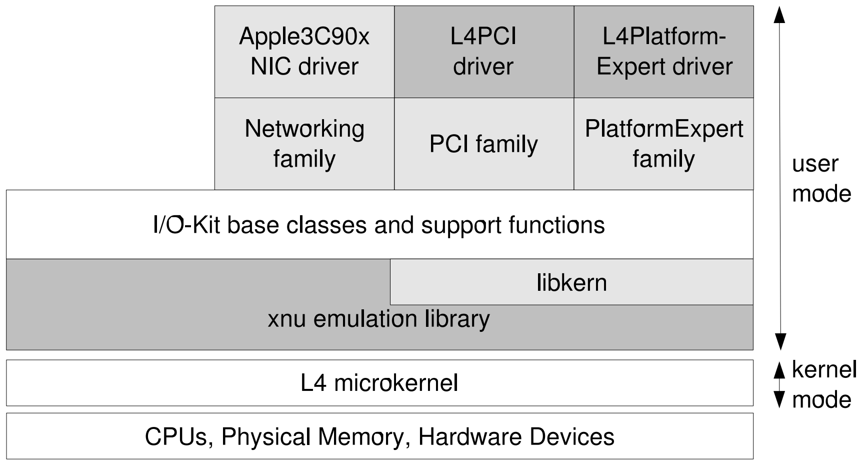

Figure 4.1: Structure of L4-I/O-Kit

4.2 The Mach Component

4.2.1 Debugging and Character String Manipulation

Some parts of the emulation library can be directly reused from Mach. This includes character string manipulation and some parts of the synchronization subsystem that are not involved in thread management.

4.2.2 Kernel Memory Allocator

I replaced Mach's kernel memory allocator with the one from Prime, the successor of the Sawmill framework [7, 19], which runs on L4. This worked as a drop-in replacement without further modifications.

4.2.3 Address Space Management

Hardware devices usually access memory by means of physical addresses. That is one of the reasons why an operating system kernel must keep track of the assignment of physical page frames to virtual pages. For now, L4-I/O-Kit runs in 1:1 physical-to-virtual mapped memory, which has the advantage that address calculations for device drivers are not required.

4.2.4 Thread Management

As I/O-Kit's work loop concept is fundamentally based on Mach threads, it was necessary to redesign Mach's thread management facilities using L4 primitives. Due to limited amount of time, I only set up a simple threading model, leaving out much functionality.

4.2.5 Interrupt Management

The machine dependent interrupt registration routine was modified to associate I/O-Kit's interrupt thread with L4's corresponding internal interrupt thread. When an interrupt occurs, L4's internal interrupt thread notifies I/O-Kit's interrupt thread which in turn runs the appropriate work loop.

4.3 BSD

The DLIL (data link interface layer) network protocol layer was decoupled from the upper protocol layers.

4.4 The Libsa component

Dynamically loading kernel extensions (KEXTs) as done by the libsa component at least requires a file system and accessing a hard disk. This is out of scope for the port of a networking device driver within three months. Thus, I purged libsa, provided stub functions instead and linked all object code into a single executable file. For simplicity and later reintegration of KEXT loading, I left libkern's runtime type information system unchanged.

Although it would be possible to build and load servers on L4 in the Mach-O binary format, it is more natural to use the ELF format to simplify development and execution. Instead of libsa, now L4-I/O-Kit's initialization code runs constructors for static objects according to information in the ELF file.

4.5 The I/O-Kit Base System

There are few modifications to the I/O-Kit base system:

- The device probing mechanism was made synchronous. Although this was not strictly necessary, following the probe steps became a lot easier.
- The IOSleep and IODelay functions were based on L4 primitives.
- The routines for allocating pageable or physical memory were modified to be independent of Mach.

Chapter 5

Implementation of the Runtime Environment

5.1 Separating the Components

Before actually starting the implementation, I spent some amount of time identifying the interdependencies of the header and source code files. In order to minimize namespace pollution of the newly created system and to clarify which structures and definitions are still needed, I purged most of the header files of the components that I had decided to emulate or replace by stub code. Only those header files remained that primarily contain type declarations. These are 22 out of 442 header files for BSD, 18 out of 33 header files from external authors, and 14 out of 479 header files for Mach.

5.2 Tools used during Implementation Phase

For browsing the header and source code files, I used the SourceNavigator [36] tool and the FreeBSD/Linux Kernel Cross Reference [43] based on the Linux Cross Reference Project [21]. For more accurate information on how the GCC compiler represents the C++ declarations, I processed the source code with GCC-XML [24]. In addition, I wrote UMLBrowser, a web based tool that uses gccxml's output to display browsable class diagrams that conform to the UML standard [34].

5.3 Families, Drivers and Nubs

In order to facilitate bottom-up implementation, I used the waterfall implementation technique: After successfully building and testing the root nub (IOPlatformExpertDevice), I pulled in the root family and driver (IOPlatformExpert, L4PlatformExpert) and the CPU and interrupt abstractions (IOCPU, L4CPU, IOInterruptController, L4PIC). Afterwards, the PCI family and driver were integrated (IOPCIBridge, IOPCIDevice, L4PCI), and finally the networking family and the 3COM 3C905 driver.

Chapter 6

Discussion

6.1 Achievements

As of this writing, L4-I/O-Kit compiles and links[1]. The 3COM 3C905 driver was left unmodified, and the changes to the I/O-Kit base system were quite minimal. L4-I/O-Kit boots on a standard L4 microkernel (Pistachio 0.4) using the GRUB boot loader [9], and detects the 3COM 3C905 NIC. Unfortunately, due to the limited amount of time[2], I was not able to do any benchmarks. As far as Apple's Public Source License permits, I can give away L4-I/O-Kit's source code by request.

6.2 Limitations

Aside from the intended limitations that were already mentioned in the introduction, L4-I/O-Kit's implementation is still incomplete. From my point of view, I encountered two fundamental problems that were in similar context already identified by others:

- Bowman et. al. state in their analysis of the Linux kernel's architecture: "[...] we learned that a concrete implementation is likely to have more dependencies than a conceptual architecture indicates." [10]
- Froehlich et. al. mention in their considerations on reusing framework that "A complex framework can require a great deal of time to learn [...]" [18].

Another problem is that I/O-Kit was probably never intended to be reused without Mach and the other kernel components.

[1] This is, at least according to Goel and Duchamp [22], a non-trivial feat.

[2] A study thesis is supposed to take three months, including design, implementation, and writing the thesis.

6.3 Potential Improvements

6.3.1 Implementing missing Functionality

For now, only the most important functions are implemented in L4-I/O-Kit. Address space management and thread management need to be refined. The design should be generalized to allow L4-I/O-Kit to run not only as root server, but as ordinary server in arbitrary address spaces, which requires a protocol for resource allocation from the root server.

6.3.2 Adapting L4-I/O-Kit to Darwin 8.0

The latest Darwin Kernel (corresponding to Mac OS X 10.4) contains several major improvements which probably have implications on L4-I/O-Kit, especially on the Networking device driver family [6]. For example, official and stable kernel programming interfaces (KPIs) were introduced so programming in the Darwin kernel becomes more sustainable. Besides, the filesystem and networking code was modified to make the BSD component fully reentrant. Furthermore, Darwin now seems to include support for ACPI. The newly introduced support of 64-bit user address spaces will not affect L4-I/O-Kit, because the Darwin kernel itself still runs in a 32-bit address space.

6.3.3 Loading Drivers dynamically and using Dynamic Libraries

The port did not address neither dynamic loading of code nor reusing the driver's binary code at run time. However, the object oriented nature of I/O-Kit would simplify both as it is already done in the Darwin kernel.

6.3.4 Improving Dependability

Until now, L4-I/O-Kit focusses only on running drivers in a single virtual machine. This does not improve dependability per se. There are basically two approaches to take advantage of the virtual machine setting:

- By adding a translation module [27], I/O-Kit would be able to communicate with other virtual machines that could for example run Linux or Windows.
- By splitting up the monolithic I/O-Kit virtual machine into separate protection domains, one would obtain a microkernel based multiserver system. A user interface and application programs would complete the operating system. Nonetheless, the split would have to be carefully designed in order to avoid serious performance degradation.

6.3.5 Supporting Arbitrary Devices

I/O-Kit's common facilities and class abstractions can even be reused when writing device drivers for L4-I/O-Kit from scratch. Nonetheless, the emulation library might need to be extended in order to support other device families such as the Storage family or the USB family. When trying to support arbitrary devices, I/O-Kit's limitations (as stated by [5]) migrate to L4-I/O-Kit:

- I/O-Kit has to support the device with an adequate driver family. There are device families (e.g. FireWire and USB imaging devices) for which such driver families do not exist. In these cases, there is not even generic device driver code that could be reused.

- Hardware quirks may contradict driver family abstractions. Sometimes, device drivers can not inherit from driver families the way it would stand to reason. The device driver may not be able to reuse a family's abstraction.

Chapter 7

Conclusion

In this thesis, I identified the issues of reusing device drivers from I/O-Kit on L4. Running the drivers at user-level is one step towards improving system dependability. I showed that I/O-Kit can be used on L4 with the help of an emulation library for Mach and other parts from the Darwin kernel. The major difficulty is to maintain the runtime environment on which I/O-Kit relies. If the semantics of the runtime environment are correctly redesigned and reimplemented, the object oriented nature of I/O-Kit device drivers enables them to be reused in a multiserver context without (or with only minor) modifications.

Bibliography

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, 1992.

[2] Apple Computer, Inc. *Inside Mac OS X: Kernel Programming*, Sept. 2003.

[3] Apple Computer, Inc. *Inside Mac OS X: System Overview*, Aug. 2003.

[4] Apple Computer, Inc. Darwin 7.6 source code. Available from http://www.opensource.apple.com/darwinsource/, Nov. 2004.

[5] Apple Computer, Inc. *Inside Mac OS X: I/O Kit Fundamentals*, Oct. 2004.

[6] Apple Computer, Inc. Darwin 8.0 source code. Available from http://www.opensource.apple.com/darwinsource/, 2005.

[7] M. Aron, L. Deller, K. Elphinstone, T. Jaeger, J. Liedtke, and Y. Park. The SawMill framework for virtual memory diversity. In *Proceedings of the 6th Asia-Pacific Computer Systems Architecture Conference*, Bond University, Gold Coast, QLD, Australia, Jan. 29–Feb. 2 2001.

[8] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 267–283, New York, NY, USA, 1995. ACM Press.

[9] E. Boleyn et al. *GNU GRUB*, 2005. Available from http://www.gnu.org/software/grub/.

[10] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: its extracted software architecture. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 555–563, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[11] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th Symposium on Operating System Principles*, pages 73–88. ACM Press, 2001.

[12] Embedded C++ Technical Committee. *The Embedded C++ specification*, 1998. Available from http://www.caravan.net/ec2plus/spec.html.

[13] Embedded C++ Technical Committee. *Rationale for the Embedded C++ specification*, 1998. Available from http://www.caravan.net/ec2plus/rationale.html.

[14] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *Proceedings of 5th Workshop on Hot Topics in Operating Systems*, pages 78–85, Washington, DC, USA, 1995. IEEE Computer Society.

[15] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: a substrate for kernel and language research. In *Proceedings of the 16th Symposium on Operating System Principles*, pages 38–51, New York, NY, USA, 1997. ACM Press.

[16] B. Ford, J. Lepreau, S. Clawson, K. V. Maren, B. Robinson, and J. Turner. The Flux OS Toolkit: reusable components for OS implementation. In *Proceedings of 6th Workshop on Hot Topics in Operating Systems*, page 14, Washington, DC, USA, 1997. IEEE Computer Society.

[17] A. Forin, D. Golub, and B. Bershad. An I/O system for Mach 3.0. In *Proceedings of the Second USENIX Mach Symposium*, Monterey, California, USA, Nov. 1991.

[18] G. Froehlich, H. J. Hoover, L. Liu, and P. Sorenson. Using object-oriented frameworks. In *CRC Handbook of Object Technology*. 1998.

[19] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The SawMill multiserver approach. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding Denmark, Sept.17–20 2000.

[20] L. G. Gerbarg. Advanced synchronization in mac os x: Extending unix to smp and real-time. In *Proceedings of the BSDCon 2002 Conference*, San Francisco, California, USA, Feb.11–14 2002.

[21] A. G. Gleditsch, P. K. Gjermshus, M. Box, and P. O. K. et. al. *LXR Cross Referencer*, 2004. Available from http://sourceforge.net/projects/lxr.

[22] S. Goel and D. Duchamp. Linux device driver emulation in mach. In *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, California, USA, Jan. 2002.

[23] A. Haeberlen, J. Liedtke, Y. Park, L. Reuther, and V. Uhlig. Stub-code performance is becoming important. In *Proceedings of the 1st Workshop on Industrial Experiences with Systems Software*, San Diego, CA, Oct. 22 2000.

[24] B. King. *GCC-XML*. National Library of Medicine, Kitware, Inc., Insight Consortium, 2004. Available from http://www.gccxml.org.

[25] The L4Ka Team, Universität Karlsruhe. *L4 Kernel Reference Manual (Version X.2)*, 2004. Available from http://l4ka.org/.

[26] J. LeVasseur and V. Uhlig. A sledgehammer approach to reuse of legacy devide drivers. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 19–22 2004.

[27] J. LeVasseur, V. Uhlig, J. Stöß, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, San Fransisco, CA, USA, Dec. 6–8 2004.

[28] E. Lévénez. Unix timeline. http://www.levenez.com/unix/history.html, Nov. 2004.

[29] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 237–250, Copper Mountain, CO, Dec. 3–6 1995.

[30] J. Liedtke. μ-kernels must and can be small. In *Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems*, Seattle, WA, Oct. 27–28 1996.

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

[32] J. Liedtke, U. Dannowski, K. Elphinstone, G. Liefländer, E. Skoglund, V. Uhlig, C. Ceelen, A. Haeberlen, and M. Völp. The L4Ka vision, Apr. 2001.

[33] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Longman, Inc., 1996.

[34] Object Management Group, Inc. *OMG Unified Modeling Language Specification*, 2003. Available from http://www.omg.org/technology/documents/modeling_spec_catalog.htm.

[35] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. B. Jones. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA, 1989. IEEE Comput. Soc. Press.

[36] Red Hat, Inc. *Source-Navigator User's Guide*, 2003. Available from http://sourcenav.sourceforge.net/online-docs.

[37] W. Schneider. The Unix system family tree. ftp://ftp.netbsd.org/pub/NetBSD/NetBSD-current/src/share/misc/bsd-family-tree, Aug. 2003.

[38] C. Small and M. Seltzer. VINO: An integrated platform for operating systems and database research. Technical Report TR-30-94, Cambridge, MA, 1994.

[39] J. Stöss. I/O-FlexPages on the x86-architecture, Study Thesis, May 2002.

[40] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th Symposium on Operating System Principles*, pages 207–222. ACM Press, 2003.

[41] K. T. Van Maren. The Fluke device driver framework. Master's thesis, University of Utah, Dec. 1999.

[42] C. A. Waldspurger. Memory resource management in VMware ESX server. *ACM Operating Systems Review*, 36(SI):181–194, 2002.

[43] R. Watson. *FreeBSD and Linux Kernel Cross-Reference*, 2004. Available from http://fxr.watson.org.

[44] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, Boston, MA, USA, Dec. 2002.