The following paper was originally published in the
Proceedings of the USENIX 1996 Annual Technical Conference
San Diego, California, January 1996

Linux Device Driver Emulation in Mach

Shantanu Goel and Dan Duchamp
Columbia University

Linux Device Driver Emulation in Mach

Shantanu Goel and Dan Duchamp

Computer Science Department
Columbia University

Abstract

We describe the design and performance of code added to the Mach microkernel (Mach 4.0, version UK02p21) that permits one to build a Mach kernel that includes unmodified Linux device drivers. We have written emulation code to support all Linux 1.3.35 network and SCSI drivers for the ISA and PCI I/O buses. Emulation increases latency, but very little. The degree depends on both device and operation, and varies from 2 microseconds for receiving small (60 byte) network packets up to 197 microseconds for writing 16KB to an ISA SCSI device.

1 Introduction

We describe the design and performance of code that permits one to build a Mach microkernel (Utah release 4.0, version UK02p21) that includes the completely unmodified source for device drivers that have been written for Linux (version 1.3.35). Our code, which consists of some changes and additions to Mach as well as run-time emulation of Linux calls, handles all of Linux's block drivers, network drivers, and SCSI host adapters for the ISA or PCI buses—53 drivers in all (block drivers for floppy, IDE hard disk, and SCSI; 30 ISA network devices; 4 PCI network devices; 10 ISA SCSI host adapters; and 5 PCI SCSI host adapters).

The motivation for this work is to improve the usefulness of the Mach microkernel on Intel x86 platforms. We are wedded to Mach because some of the research in our laboratory is dependent on its unique features. Our research also needs to incorporate new peripheral devices on a regular basis. Unhappily, because of its small user base, Mach has always had relatively few device drivers compared to more popular operating systems. Furthermore, many of these drivers are old and do not accommodate recent generations of I/O chips, either by not running them at all or else by failing to take advantage of advanced new features. In part because of developments in multimedia and wireless networking, new I/O peripherals are being invented at a remarkable rate, and Mach's set of device drivers has been steadily falling further behind the hardware base available in the PC world.

This problem eventually became acute for us. We knew we could not obtain either the time or the information to write all the Mach device drivers we wanted; writing new device drivers is often difficult because of the need to obtain access legally to proprietary hardware specifications and/or software, and because one must have a sound understanding of the hardware in order to write a high quality driver. Accordingly, we wondered whether it would be practical to implement Linux device driver emulation within Mach. Because of its relatively large following, Linux has many more device drivers and the rate at which new drivers are added to it outstrips the rate at which new drivers are added to Mach. Linux has driver call-in and call-out interfaces that are well defined and that change slowly. We thought that if we could emulate these interfaces, then we could tap into the current base of Linux drivers, and—possibly with limited further effort to update the emulation—future Linux drivers as well.

Our goal was to be able to compile completely unmodified Linux device drivers into the Mach kernel. We achieved this goal for network, block, and SCSI devices that attach to either the ISA or PCI bus. This paper reports our design, how certain aspects of both Mach and Linux constrained our design, and how our code performs.

2 Background

This section provides necessary background about how device drivers operate in both Mach 4.0 and Linux 1.3.35.

In Linux, drivers may be either statically or dynamically linked into the kernel. Mach supports only statically linked drivers. Consequently, we emulate only statically linked Linux drivers.

Linux has five classes of devices: network, SCSI, block, sound, and character. At this writing, we have tackled only network, block, and SCSI drivers that attach to the ISA and PCI I/O buses.

Linux and Mach view device access differently. Mach is a microkernel designed with portability in mind. It has a single kernel interface for all device types, and this interface is accessed by messages. Within the kernel, a device access request first passes through machine independent code and then to machine dependent code.

Linux is a monolithic operating system that was originally targeted to only the Intel x86 architecture. Its device interfaces are accessed by other parts of the operating system via procedure call, and there is no attempt at machine independence. Each type of device has its own interface.

Figures 1, 2, and 3 give the Mach device interface and Linux interfaces to network and SCSI devices, respectively.

Linux regards SCSI devices as block devices, so the interface to SCSI is the vnode interface; this explains the large number of unimplemented calls in Figure 3. Of course in both systems a device driver must also have an interface that is accessed from below: one procedure per type of interrupt. In the case of network drivers, there are two types of interrupt: `transmit done` and `packet received`. The SCSI interrupts are `command complete` and `bus reset by device`.

Network drivers have essentially no internal structure. There is one procedure to handle each entry point and a few utility routines. However, in both Mach and Linux SCSI drivers have an internal structure. The top layer is "target specific;" example targets are tape and disk/CD-ROM. The target specific layer knows about the physical structure and constraints of one type of device. This layer translates device-specific operations (e.g., read a disk block) into one or more SCSI commands. The middle layer performs bookkeeping chores like queueing and timeouts. The bottom layer is the "host adapter," which knows how to send one or more SCSI commands to a specific controller chip and return the result(s). The interface to the host adapter is given in Figure 4. This interface is the same in both Mach and Linux.

```
device_open
device_close

device_read        /* synchronous */
device_write
device_read_inband  /* small variant */
device_write_inband

/* these two replace ioctl */
device_get_status
device_set_status

/* set network packet filter */
device_set_filter

device_map
/*
 * Also, asynchronous two-message
 * (request & reply) versions of
 * each of these calls: device_open,
 * device_read, device_write,
 * device_read_inband, and
 * device_write_inband.
 */
```

Figure 1: Mach Kernel Interface to All Devices

3 Design

This section presents the details of our design. Section 3.1 discusses the relative power of the two device interfaces; i.e., is it possible to emulate the Mach device interface using Linux drivers? The remaining subsections, 3.2 through 3.6, discuss the specific modules of emulation code that we wrote. The implementation consists of about 2000 lines of C.

3.1 Device Interfaces

At the most abstract level of consideration, emulation raises two issues:

1. Emulating any procedure call or variable reference that a Linux driver might make.

2. Ensuring that the combination of emulation code plus a Linux driver are together able to implement all Mach device entry points.

```
probe
open
close
send_packet
ioctl                  /* unimplemented */
get_stats
set_multicast_list     /* effectively,
                          this is ioctl;
                          link address and
                          promiscuous mode
                          can be set too */
```

Figure 2: Linux Interface to Network Devices

```
open
release      /* same as close */
read
write
ioctl

/* obscure */
change       /* has media changed?
                e.g., CD-ROM removal */
validate     /* flush and re-read
                disk partition tables */
fsync

/* not implemented by any driver */
lseek
readdir
select
mmap
fasync       /* asynchronous sync */
```

Figure 3: Linux Interface to SCSI Devices

The first concern—how Mach can provide the facilities needed by Linux drivers—is addressed in the sections below. This section shows that Linux drivers can implement the Mach interface as well as Mach drivers implement it. In the next few paragraphs we argue that the Mach kernel interface to devices can be implemented by Linux drivers plus a small amount of simple emulation code.

Figures 2 and 3 show that Linux has obvious analogues for most Mach device entry points: `open`, `close`, `read`, `write`, and

```
detect          /* probe for device */
command         /* synchronous */
queue_command   /* asynchronous */
reset
```

Figure 4: Interface to SCSI Driver Host Adapter Layer

`ioctl`. Of course, it is possible that certain Mach `ioctl` arguments are not implemented by Linux drivers. In the case of network devices, Mach's `ioctl` calls (`device_get_status` and `device_set_status`) map not to `ioctl` but to the non-obvious `set_multicast_list`. With the right arguments, `set_multicast_list` can be used to read or write the link address, multicast addresses, and promiscuous read mode. These are the major actions of Mach's `ioctl`.

There is no Linux analogue for the Mach `device_set_filter` entry point. This entry point is used to associate a packet filter [3] with a driver. However, in Mach neither this entry point nor actual filtering of packets is implemented by device drivers. Instead, Mach has generic routines for installing and executing packet filters, and all device drivers call the generic installation routine. Once installed, a packet filter is called after the network driver has retrieved the incoming packet and copied it into a Mach message. The filter operates on the message contents. At that point, all device-specific functions have been performed, so packet filtering really takes place above the driver level.

The `device_map` entry point permits devices like frame buffers and disks to map their contents into virtual memory. The Linux analogue is `mmap`. As a practical matter, the only drivers that use these calls in either system are frame buffers, so emulation is presently a moot point. If someday it becomes important for a Linux driver to service Mach `device_map` calls, emulation would be easy because of the presence of a generic Mach routine (`blockio_map`) that converts paging activity of the mapped area into device read/write requests.

3.2 In-Kernel Device Interface

Below Mach's kernel device interface and above the device drivers is a layer of "machine independent" code. This code unpacks request messages into a generic I/O request structure, maps or moves pages between the calling process and the kernel, and sends a reply message when the device driver is finished with the operation. Also, during a `device_open` operation, this code looks up the device's name in a list called `dev_name_list`. Device drivers add entries to `dev_name_list`. Such entries include the device's name and pointers to routines in the device driver that implement the calls in the kernel's device interface. The idea is that the machine independent code performs all kernel actions that are not truly device-specific.

After acquiring some experience with the machine independent code, we decided to eliminate it because it interferes with the goal of using device drivers from other operating systems.

For example, Linux names IDE drives `ide0`, `ide1`, and so on, but Mach's UNIX server refers to IDE drives as `hd0`, `hd1`, etc. To use Linux drivers in a Mach microkernel that is running the UNIX server, it would be necessary to update the machine independent layer to translate a `device_open` kernel call with an argument of `hd0` into a call to the Linux driver. More generally, extra effort is needed to convert between several conventions defined by Mach's machine independent layer and the corresponding conventions of Linux drivers.

Since the machine independent layer would have to change anyway, and since we thought the type of translation mentioned above is more properly done by the emulation code associated with the Linux driver, we decided to eliminate the machine independent layer. The result is a new implementation of all Mach device interface calls that recognizes emulation as a possibility. For example, when a `device_open` request is received, the kernel calls the `open` routine of each emulation module. An emulation module that recognizes the device name returns a structure that contains pointers to routines that implement each call in the kernel interface. For calls other than `device_open`, the proper routine pointer in the structure is dereferenced immediately upon entry into the kernel. The emulation module is responsible for all actions needed to service a device interface call, including mapping pages and sending a reply. Three emulation modules currently exist: Linux block, Linux network, and old Mach. The old Mach module is the original Mach device code, hacked a bit to conform to the new way of doing things.

3.3 Initialization

Linux assumes that the clock is running at the time drivers configure, but in Mach this is not true. This creates a problem since Linux drivers use clock interrupts to timeout device probe commands. In particular, the initializing driver (which is the only activity running on the machine at the time) initiates a probe command and then polls the clock variable in a loop until the probed device responds with an interrupt or until the clock variable reaches some timeout value. The clock variable is incremented by the handler for clock interrupts. (Mach drivers use a similar method except that, because no time facility is available, the delay loop runs for an "estimated" amount of time.)

We solved this problem by changing Mach to start the clock earlier, and by writing a special clock interrupt handler that is used only during Linux driver configuration. We have also renamed the clock variable that Linux drivers refer to; Mach calls the variable `elapsed_ticks` while Linux calls it `jiffies`. The special handler increments this variable, and we have changed the standard Mach clock handler to do so also. Mach's clock handler could not be used during driver initialization because it manipulates the "current thread" data structure which doesn't exist at that point. We did not fix Mach device drivers after the fact to use measured time rather than estimated time in their delay loops.

3.4 Memory Usage

Mach and Linux make different use of the kernel's address space in two ways: addressing and memory allocation. Both of these differences impact driver emulation.

3.4.1 Addressing

Both Mach and Linux map the kernel into the upper gigabyte of the 32-bit address space. However, Mach sets kernel segment register values to zero, and is linked so as to generate virtual addresses in the range `[0xC0000000 .. 0xFFFFFFFF]`. In contrast, Linux sets its segment registers to `0xC0000000`, and generates virtual addresses in the range `[0x0 .. 0x3FFFFFFF]`. Linux does this to ease kernel programming: kernel virtual addresses and physical addresses are identical and interchangeable. Consequently, Linux drivers have no provision for translating between physical addresses and kernel virtual addresses, and operations that require physical addresses (e.g., DMA) would not work if a Linux driver were simply compiled and linked with Mach.

To resolve this difference, we changed Mach's machine-specific memory management module (`pmap`) and linking instructions so that Mach also generates virtual addresses in the range `[0x0 .. 0x3FFFFFFF]`.

3.4.2 Kernel Memory Allocation

An initialization call to a DMA-capable Linux driver includes two parameters which are the start and end addresses of a segment of contiguous, DMA-able[1] physical memory. The driver uses this memory for DMA buffers and for storing its data structures. In Mach, driver initialization occurs after virtual memory is enabled, so the virtual memory system cannot be avoided when searching for memory for driver initialization. The emulation code searches Mach's free page list, looking for a sequence of contiguous pages that lie below the 16MB boundary. The boundary addresses of this segment are passed to the Linux driver.

To service dynamic requests made by Linux drivers for extra DMA buffers, we implemented a new memory allocator that Linux drivers (only) use to share 64KB of DMA-able memory set aside at initialization. The reason for adding yet another kernel memory allocator is that no existing Mach facility provided the right combination of being able to run during interrupt service,[2] allocating physically contiguous DMA-able memory, and being space-efficient for small allocations.

3.4.3 I/O Blocking

Linux uses a small block size for I/O operations. For most block devices, the block size is 1KB. For CD-ROM, it is 2KB. In contrast, Mach's block size is 4KB, the page size. The negative effect of a small block size is ameliorated in part by the fact that the Linux block cache code "clusters" the pieces of a multi-block I/O operation. To cluster means to coalesce physically contiguous blocks into a single "segment," and to form a list of segments into a single I/O command. Such a list is suited to devices that have scatter/gather ability.[3] List formation is done without extra copies, since the segment list itself consists only of pointers to segments.

Clustering reduces the number of I/O operations whenever the device provides scatter/gather. (If the device does not support scatter/gather, each segment is a separate operation.) For this reason, we have ported the clustering code from Linux's block cache into the emulation code.

3.5 Synchronization

3.5.1 Between Driver and Processes

Mach and Linux embody fundamentally different design decisions regarding the (a)synchrony of interaction between device drivers and the higher level software that invokes I/O operations.

In Linux, it is assumed that some process is waiting for each I/O operation. The process formats an I/O request, initiates the I/O operation, then sleeps on the I/O request buffer. When the I/O operation completes, the device driver performs a `wakeup` on the I/O request buffer, and the process resumes. In contrast, in Mach the interaction between the driver and the process that initiated I/O is asynchronous. Mach maintains an "I/O-done list" and an "I/O-done thread." When an I/O operation completes, note is made in the I/O-done list. At some later time, the I/O-done thread is scheduled, removes the completed operation from the I/O-done list, and sends a message to the initiator of the operation.

We made the Linux synchronization method compatible with Mach as follows. The I/O request block is passed to the driver locked. When the Linux driver completes the I/O operation, it calls routine `unlock`. This routine unlocks the I/O request block and calls `wakeup`. We replaced `unlock` with code that unlocks the I/O request block then manipulates the I/O-done list to indicate that the operation is finished. It is guaranteed that the I/O-done list already has a record for the operation because the Mach I/O code placed one there before invoking the Linux driver to do the operation.

3.5.2 Between Driver and Hardware

Mach and Linux differ substantially in how they disable interrupts. Mach masks classes of inter-

[1] In the PC architecture, DMA-able memory must be below 16MB, because the ISA bus has only 24 address lines.

[2] Mach assumes that interrupts do not change the page list.

[3] A scatter/gather I/O operation consists of an indication of the direction in which the data should move and a list of segments aligned to some boundary. The device either "scatters" a write over the segments or "gathers" a read from the segments. Different devices have different limits on how many segments can be accommodated in a single I/O command.

rupts by using `spl` to set the CPU to one of 8 different interrupt priority levels. Linux does not vary CPU priority. Instead, it will disable individual devices by turning them off at the PIC (programmable interrupt controller); Linux also uses the x86 `CLI` and `STI` instructions to disable and enable interrupts entirely. Turning off individual devices is a fine grain of control not available in Mach, and changing Mach to use such control would be prohibitively difficult. Consequently, we emulate a Linux driver turning off a single device by using `spl` to turn off the entire class of devices. We have not observed any problems from this over-masking.

Linux is not designed to run on multiprocessors, so Linux device drivers are not concurrency-safe. Mach, on the other hand, contains locking to permit execution on multiprocessors. To use Linux drivers safely within Mach, the emulation code implements a per-device lock that ensures that calls to any single device are serialized.

3.6 Machine Resources

Linux implements an organized approach to the notorious problem of binding interrupt lines (IRQs), DMA channels (DRQs), and I/O port ranges to devices. There is a central allocator that keeps track of each resource. Well behaved device drivers request these resources and release them when they are finished.

Central allocation prevents conflicts, and is a good idea. Since Mach had no such facility, we simply ported this code into Mach for use by the Linux drivers. We did not bother to install similar code in the Mach drivers, so they are ill behaved with respect hardware resource allocation, and they should not be used at the same time as Linux drivers.

Another worthy facility used by Linux drivers that we ported from Linux into Mach is "automatic IRQ detection." The purpose of this facility is to automatically discover which interrupt line a jumper-configured controller card is using. It works as follows:

1. The device driver calls the "autoIRQ" facility, which installs special interrupt handlers for every unallocated IRQ.

2. The device driver forces the device to interrupt.

3. The device driver calls the autoIRQ facility to receive a report. A timeout is given as an argument.

4. If the device was configured to use one of the available IRQs, then one of the special interrupt handlers was invoked. The autoIRQ allocates this IRQ to the device and indicates so when the device reports asks for the report.

5. If the device was not configured to use one of the available IRQs, then the interrupt was handled by some other device's interrupt handler and presumably treated as an error. The device driver's call to autoIRQ for a report times out and indicates that no IRQ was allocated.

6. A side effect of autoIRQ giving its report to the device driver is that the special interrupt handlers are uninstalled.

4 Evaluation

Of the 53 emulated drivers, we tested seven and measured the performance of two, the SMC "Ultra" Ethernet controller and the Adaptec 1542C SCSI controller. Both devices attach to the ISA bus. All the remaining Linux drivers compile. This is not a trivial statement, given that compilation means that emulation code exists for every external name in all drivers.

In the two cases, we compared the performance of the emulated driver versus the native Mach driver.

4.1 Experimental Setting

Our test platform was a Pentium 90MHz system with 16MB of DRAM, ISA and PCI I/O buses, a Fast SCSI-2 disk, and an unloaded Ethernet. Artificial workloads were generated by simple programs we wrote to open a specific device (network or disk) and then access the device via direct calls to the microkernel. In fact, the exact characteristics of the hardware platform and load generation software are not important, for a few reasons. First, we are concerned with latency rather than throughput, so there is no need for the workload generator to be able to run the timed sections often. Second, we are concerned with the relative latencies of two drivers; absolute times are not important. Third, the timed code sections are short and contain few sources of variability; page faults are guaranteed not to happen during a timed section, and no I/O operation occurs in three of the four tests.[4] It is not worrisome if the system calls generated by the workload exhibit variable latency; what matters are the short timed sections deep within the kernel.

In order to generate precise timing numbers for short code sections, we used the Pentium's `RDMSR` instruction. This instruction can read any two of about 40 registers that count the number of certain actions (e.g., cache misses, time ticks) since CPU reset. The time register is a 64-bit counter that is incremented on the edge of every CPU (on-chip) clock signal; i.e., for a 90MHz processor the counter is incremented 90 million times a second. In order to interpret the counter one must know the CPU clock rate; we took code from FreeBSD 2.0.5 that figures this out. The `RDMSR` instruction itself takes 6 clock ticks. The time to move the returned value to a memory location pushes the overall time to sample the counter up to 8 clock ticks if the memory location is cached; the time could be considerably longer if the location is not cached. On a 90MHz Pentium, 8 clock ticks is about one tenth of a microsecond. Since the shortest code paths we measured is 2 microseconds, we deem the error introduced by counter sampling to be negligible.

Below are two sections, one for each comparison. In each section, we present latency figures in a table and point out and explain any results that are unexpected or interesting.

4.2 Network

DRIVER	SIZE	MIN	VAR
Linux	60 bytes	74 usec	9
Mach	60	62	3
Linux	256	159	22
Mach	256	214	3
Linux	1500	712	31
Mach	1500	582	31

Table 1: Network Transmit Latency (vs. Mach)

We measured the minimum latency and latency variance of the two performance-critical operations: network transmit and receive.

The first time sample was taken at the point where the Linux and Mach drivers first differ. Similarly, the second time sample was taken where the drivers first re-converge. For transmit, the first point is where the packet is queued and a software interrupt scheduled. The second point is the routine to deallocate a packet once transmission is complete. For receive, the first point is in the `packet received` interrupt handler, while the second point is in the routine that delivers the packet to the kernel. The transmit path includes the I/O to the Ethernet chip. I/O is finished by the time the receive path begins, but the receive path includes copying the packet from the controller.

DRIVER	SIZE	MIN	VAR
Linux	60 bytes	86 usec	12
Mach	60	84	2
Linux	256	265	9
Mach	256	295	18
Linux	1500	655	54
Mach	1500	749	185

Table 2: Network Receive Latency (vs. Mach)

For three of the six comparisons (transmit 256, receive 256, and receive 1500), we have the puzzling result that the emulated Linux driver is faster than the native Mach driver. This should not happen since, despite the differences between the drivers, the emulated Linux case always performs strictly more work than the native Mach case. In the two receive cases, we have determined that the entire time difference is due to the single instruction that copies the packet from the I/O controller to DRAM. However, we have been unable to determine what hardware effect is causing the difference[5] or why this hardware effect occurs in the native driver but not in the emulated Linux driver.

4.3 SCSI

We measured the minimum latency and latency variance of the two performance-critical operations: SCSI read and write. In these tests, two code paths were timed. Path CMD is from the `write` system call until the command is issued to the host adapter. Path INT is from the `command complete` interrupt until the I/O request block is retired from the driver's queue of commands. The I/O operation takes place between the first and

[4] SCSI read and write, and network reception. The timed segment for network transmission includes the I/O.

[5] We suspect a cache effect, but even with the considerable help of the `RDMSR` instruction we have not been able to pinpoint the cause.

DRIVER	SIZE	PATH	MIN	VAR
Linux	512	CMD	86 us	1
Mach	512	CMD	29	4
Linux	512	INT	43	4
Mach	512	INT	2	1
Linux	4K	CMD	119	2
Mach	4K	CMD	30	4
Linux	4K	INT	52	1
Mach	4K	INT	2	1
Linux	8K	CMD	141	2
Mach	8K	CMD	35	5
Linux	8K	INT	58	2
Mach	8K	INT	3	0
Linux	16K	CMD	160	2
Mach	16K	CMD	39	4
Linux	16K	INT	74	3
Mach	16K	INT	3	0

Table 3: SCSI Read Latency (vs. Mach)

DRIVER	SIZE	PATH	MIN	VAR
Linux	512	CMD	85 us	2
Mach	512	CMD	29	3
Linux	512	INT	43	4
Mach	512	INT	2	0
Linux	4K	CMD	120	1
Mach	4K	CMD	30	4
Linux	4K	INT	51	2
Mach	4K	INT	2	0
Linux	8K	CMD	143	0
Mach	8K	CMD	35	4
Linux	8K	INT	59	1
Mach	8K	INT	3	0
Linux	16K	CMD	162	0
Mach	16K	CMD	37	5
Linux	16K	INT	74	3
Mach	16K	INT	2	0

Table 4: SCSI Write Latency (vs. Mach)

second timed paths, and hence is not included in any of the timings. As with the network tests, the beginning and ending of the timed paths are the points at which code paths in the Linux and Mach drivers diverge and converge, respectively.

Unlike the network timings, the SCSI results are easily explained. First, for a common driver and data size, the times for read and write are virtually identical. This is to be expected because the two operations are the same except for the direction of data transfer. Second, the numbers for the emulated Linux driver rise with increasing data size. (In contrast, the CMD numbers for the native Mach driver do rise with data size, but very slowly; the INT numbers don't rise at all.) The explanation is that two forms of emulation processing are proportional to transfer size. One is that the emulation code tries to coalesce a multi-block transfer into "segments" with physically contiguous pages. This affects both CMD and INT paths. The other is that the CMD path checks if a DMA operation is scheduled for a memory location beyond 16MB and, if so, allocates "bounce buffers" to compensate.[6] The Mach driver makes no such check, requiring that DMA be to/from addresses below 16MB.

[6] A bounce buffer is a DMA-able region of memory used as a waystation by a DMA operation whose target address is beyond 16MB.

4.4 Conclusion

Although it is disappointing that we are presently unable to explain the network timings, one point is clear, and that is that the cost of emulation is very low. The case where emulation imposes the highest cost is writing 16KB to SCSI, and here the cost is less than 0.2 milliseconds[7] for an I/O operation that requires several milliseconds.

In fact, we added considerably more function and overhead to a re-implementation after seeing, in our initial implementation, how little cost was imposed. I/O drivers seem to be an especially appropriate place for emulation since, so long as the I/O bus is slow and the CPU is fast, a considerable number of instructions can be executed by an emulator without having noticeable impact on performance.

5 Summary

Reactions to this work at its outset included expressions that a practical emulation of Linux drivers in Mach would be quasi-miraculous. About quasi-miracles Samuel Johnson once wrote "*A dog's walking on his hind legs [is] not done well; but you are surprised to find it done at all.*" In our case, the dog actually walks quite well.

[7] 197 microseconds = 125 on the CMD path plus 72 on the INT path.

We have demonstrated the practicality of incorporating unmodified device driver source code of one operating system into another. Both Mach and Linux are intellectual descendants of UNIX, but they do not share code ancestry. Mach's device drivers are for the most part descended from 4.3 BSD, while Linux device drivers were written from scratch using different assumptions about some important aspects of the surrounding operating system. Nevertheless, the performance penalty for emulation is very limited. In the tests we did, the degree depends on both device and operation, and varies from 2 to 197 microseconds.

We are aware of no related work that shares the essential feature of our work: incorporating unmodified source code of one operating system into another. Of course within the last few years there has been a good deal of work in the UNIX community on emulating the system calls of the far more popular DOS and Windows interfaces; examples include Mach's DOS server [2], Linux "DOSemu" and "Wine" projects, and a number of commercial efforts such as "DOSMerge" and "WABI." The difference between those efforts and ours is that we include another system's source code into our kernel, so we are emulating intra-kernel variables and interfaces rather than the more standardized and carefully considered system call interface.

The obvious direction for future work in this area is to extend the emulation to include more drivers from Linux and to include drivers from other operating systems. The true mother lode of device drivers in the PC world is of course the DOS or Windows binaries that ship with nearly every piece of peripheral hardware. Emulating these would be enormously more difficult than what we have done so far because of lack of access to source code and because of the huge range of actions taken by these drivers, based on the assumption that they can control the whole mahcine. Nevertheless, a reasonable starting point might be to emulate network drivers because they are so simple and because so many conform to the NDIS specification [1].

6 Acknowledgements

This work was supported in part by the Advanced Research Projects Agency, ARPA order number B094, under contract N00014-94-1-0719, monitored by the Office of Naval Research; and in part by the Center for Telecommunications Research, an NSF Engineering Research Center supported by grant number ECD-88-11111.

References

[1] S. Dhawan.
Networking Device Drivers.
Van Nostrand Reinhold, New York, 1995.

[2] G. Malan, R. Rashid, D. Golub, and R. Baron.
DOS as a Mach 3.0 Application.
In *Proc. USENIX Mach Symp.*, USENIX, pp. 27–40, November 1991.

[3] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss.
Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages.
In *Proc. 1994 Winter USENIX Conf.*, USENIX, pp. 153–165, January 1994.

7 Author Information

Shantanu Goel is an MS candidate in the Computer Science Department at Columbia University. His research interest is operating systems.

Mailing address: 450 Computer Science Bldg., Columbia University, 500 West 120^{th} St., New York NY 10027. Email address: `goel@cs.columbia.edu`.

Dan Duchamp is an Associate Professor of Computer Science at Columbia University. His current research interest is the various issues in mobile computing. For his initial efforts in this area, he was named an Office of Naval Research Young Investigator for the period 1993-1996.

Mailing address: 450 Computer Science Bldg., Columbia University, 500 West 120^{th} St., New York NY 10027. Email address: `djd@cs.columbia.edu`.