An I/O System for Mach 3.0

Alessandro Forin
David Golub
Brian Bershad

{af,dbg,bershad}@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue Pittsburgh, PA 15213

Abstract

The Mach 3.0 I/O system represents a radical departure from its predecessor – Mach 2.5, which relied on the BSD Unix model of device management. The I/O interface in Mach 3.0 supports device drivers that are largely *device-independent*, implemented at user-level, and location-independent. Our approach to device management significantly reduces the size of the kernel's machine-dependent code, enables us to reduce the length of the I/O path, and permits us to transparently manage remote devices on non-shared memory multiprocessor architectures such as the Hypercube. This paper describes the structure and performance of Mach's I/O system.

1. Introduction

This paper describes the design of the I/O system for the Mach 3.0 kernel [Rashid et al. 89]. Mach's I/O system is novel in several respects. First, it supports the notion of "device independent" device drivers. The I/O system separates out generic driver code common to a class of devices such as a screen, an Ethernet controller, or a disk, from code which is only dependent on the device controller *chip* itself, and from the code which is specific to a given processor architecture. Second, the Mach I/O system supports user-level device management of mapped devices, enabling application programs, such as an operating system server, to directly control device activity. Finally, the Mach kernel provides for location-transparent device management which can be accessed through Mach's interprocessor communication (IPC) facilities.

1.1. Device Management for Small-Kernel Operating Systems

Mach 3.0 is an operating system kernel which is intended to be freely portable across a large number of processor architectures, offer network transparency, support efficiently a variety of operating systems implemented as user-level applications, and provide a scheduling interface suitable for the needs of concurrent, Real-Time and parallel programs.

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035.

Previous versions of the Mach 3.0 I/O system made it difficult for us to meet these goals. We describe why this was so, and how the new I/O architecture addresses the problems, in the rest of the introduction.

A Smaller Kernel

The Mach 3.0 kernel had originally inherited the I/O management structure of Mach 2.5, which in turn derived its I/O system from BSD UNIX. Under BSD, devices could only be described as character-oriented or block-oriented devices. This gross characterization made it difficult to share code across functionally equivalent devices which happened to exist on different system platforms. The I/O system simply had no structure to allow similar devices, for example, a monochrome and a color display, to share code, even though the functions provided by the similar devices are nearly identical. We tried, whenever possible, to use the vendor-supplied device drivers when porting Mach to a new system architecture. While this could sometimes reduce the time to port, it resulted in a large amount of nearly duplicated code within the kernel because one vendor's interpretation of how best to drive a device differed from another. Since the drivers were not just device, but also processor dependent, we had no easy way to exploit the similarities. Worse, the drivers from the vendors themselves were nearly always "cloned" from a pre-existing version that handled a similar device, resulting in even more code duplication.

Device-independent device drivers decrease the amount of Mach's machine-dependent code, thereby decreasing the size of the operating system and the amount of time required to port the system to a new system architecture. The generic structure of the new Mach device drivers allows us to wean ourselves from the vendor-supplied drivers, thereby reducing the amount of vendor-owned code in our system. Moreover, maintenance of the system is greatly simplified because changes to the I/O system, for example to increase performance, need only be applied to each class of device, not to each device for each processor architecture for each system.

Real-Time

A second problem with earlier versions of the I/O system was that it was designed to run entirely in kernel mode, generally at high priority in ignorance of other scheduling requirements. This ignorance was because the drivers were often supplied by vendors in the context of a bundled UNIX kernel, which itself has no support for Real-Time computing. The Mach kernel, though, does provide for Real-Time support, so it is necessary to bound the amount of time spent within kernel interrupt handlers down to the "negligible" range.

User-level device management reduces the amount of code that runs in privileged kernel mode, and increases the predictability of the kernel's scheduling algorithms.

High-Performance I/O Devices

While I/O systems of the last 20 years have been measured in megabits per second, those of the next decade are likely to have data transfer rates on the order of gigabits per second. High-speed networks, which can provide data at this rate, and multimedia applications, which can similarly consume data, are two obvious forcing functions pushing on software architectures to support high-performance I/O.

The in-kernel drivers in previous versions of Mach acted not only as device controllers, but also as data buffers. This meant that data would be ferried across the user/kernel boundary as it passed between the application and the device. For low-performance I/O devices, such as SCSI disks, the additional data transfer time was not important. But, for high-bandwidth applications, it was necessary to reduce the length of the I/O path.

User-level drivers can provide for increased I/O performance because it is possible to avoid expensive data copies (either physical or virtual) between user and kernel space. Data can instead flow directly between the application and the device. This can be done by *mapping* the device directly into an application's address space, just as is done with display devices for graphics-based workstations. Where architecturally possible, the Mach 3.0 kernel allows an application to map a hardware device into its address space.

Location Transparency

While being able to access devices from user-level solves one set of problems, it has the potential to introduce another. It is important to us that we be able to control a device from any machine, not just from the one to which the device is attached. For this reason, we have designed an IPC-based device interface, rather than one based on kernel traps, allowing us to implement remote device driver management. As is customary in message-passing kernels, the device is viewed as a server to which client programs make remote procedure calls (RPC). This ability is critical on "NORMA" (No Remote Memory Access) multiprocessors, in which each processor runs its own instance of the Mach kernel [Barrera 92]. For example, on the Hypercube, we can run the UNIX server on a fast i860 processor and have it drive a SCSI controller attached to a slower i386 processor.

1.2. Some Problems We Weren't Trying To Solve

While designing the I/O interface for Mach 3.0 another thing was quite clear: the user of such interface was not going to be a final user application program, but rather an operating system server such as a filesystem or protocol server. Therefore we have made no attempt to masquerade I/O devices as files or any other "uniform" programming abstraction on the basis that i.) such a uniformity, if desired, should be provided at a higher level, and ii.) providing it at the device level creates unnecessary problems for the many different types of servers that would use the interface. In particular, the Mach kernel is intended to support a variety of different operating system environments, such as BSD UNIX [Golub et al. 90], MS-DOS [Rashid et al. 91], and MacOS, each one exporting its own device abstraction. Our approach allows servers for these operating systems to implement their abstractions at the lowest possible level.

The rest of this paper is structured as follows. In Section 2 we describe the structure of our device independent device drivers. In Section 3 we discuss user-level device management. In Section 4 we briefly describe the IPC-based I/O interface. In Section 5 we discuss some crucial aspects of the performance of the new I/O system. In Section 7 we briefly discuss some related work. Finally, we discuss the system's current status.

2. Device Independent Device Drivers

Early versions of Mach have used pre-existing BSD UNIX device drivers with minor modifications which adapt them to the Mach's virtual memory and thread management systems. These drivers typically come from the machine's vendor, and are therefore different across different vendors. Nevertheless, there is much in common both across hardware devices and across the software that drives them.

The basic observation that led us in the design of a new implementation paradigm is that hardware devices, especially in the case of workstations, are built with off-the-shelf components, such as video RAMDAC chips, serial line UARTs, SCSI controllers, Ethernet controllers and so on. Although each chip behaves differently, within a class of devices all chips basically perform the same set of functions. For example, every video controller chip has the ability to move the cursor, and every serial line controller chip can be instructed to set speed and parity. Identifying those common functions and encapsulating them at the bottom of a class-like hierarchy produces a system where the code for one chip can be easily replaced by the code for a different one. Not only does this enhance portability, but it allows us to easily integrate new and better versions of chips as they become available over time.

By creating classifications for devices, and then identifying the chip-dependent interface for each class, we are able to write device drivers that are largely independent of the actual make and model of the piece of hardware that they are driving. Instead of character and block devices we now have a set of functionally grouped device drivers including the screen, console, disk, tape, serial line, and Ethernet. Each driver has one or more layers of chip-independent code, which provides both the external interface and implements the logic behind the workings of the device driver itself. Only simple, core functions at the bottom level deal with the hardware directly. Portability and code sharing are greatly increased by this structure. There is no need, for example, to rewrite the code which drives a previously handled SCSI when porting to a new machine that uses a new processor from a new vendor.

Our approach to device independent device management is similar to the one used in Mach's machine-independent VM interface (pmap module) to the various Memory Management Units (MMU) [Rashid et al. 87]. The pmap layer encapsulates MMU dependencies beneath the bulk of the VM system. This greatly increases portability, because only the pmap layer needs to be changed for a new architecture, and code sharing, because much of the pmap layer is constant across similar MMUs.

In the rest of this section, we describe the structure of each of the major device classes supported in Mach 3.0.

2.1. The Serial Line Driver

Serial lines and their drivers are among the oldest components of UNIX's I/O system. The chips in use today, for example, are essentially the same as those that were available ten years ago. Nevertheless, on the software side, each system vendor has supplied its own version of the serial line driver, but they all derive from the original code written at AT&T [Ritchie and Thompson 78]. This situation creates unnecessary code duplication. In addition, we have seen several cases where the same UNIX ioctl is encoded differently because the ioctl interface has semantics that can be specific to device drivers. Since this interface is visible to applications, the proliferation of "similar but not quite the same" device drivers has created binary compatibility problems for us.

The Mach 3.0 serial line driver is split into a device independent component and a small device (chip) dependent part. The device independent component deals with character buffering, and all console-related code including:

- open/close/read/write functions,
- start/stop operations,
- modem controls,

- interrupt handlers for the simple case of devices that work on a one-interrupt-per-character basis.
- send/receive of characters in polling mode to the system console for debugging and error messages,
- switch code for the bitmap driver (mouse and console callouts),
- finding the appropriate console line in a generic kernel,

The chip-dependent layer implements only those operations that manipulate the device registers directly. These include probing for existence, setting of speed, parity and modem control, and moving characters on and off the chip.

2.2. The Screen Driver

The portions of code that are shared across all screen devices are the terminal emulator, fonts, screen saver, interface routines such as open/close/read/write, event handling logic for both motion and keypress events, and other status control operations such as controlling the screen saver and the cursor position.

Code that is specific to each screen device controls probing, notification of open/close operations, character painting at a given location, scrolling, cursor motion, video on/off and blanking, enabling/disabling of vertical-retrace interrupts, and returning the physical address of registers for user space mapping. Even at the lower level, which is chip-specific, we have been able to share some code across devices. For example, all displays based on framebuffers share the same code for painting characters and scrolling text.

Separate from the screen module, but logically part of the same driver are the drivers for the keyboard and mouse. These are structured as devices in their own right, but are only invoked from the serial line or screen drivers and not by general user applications. The keyboard driver remaps the keyboard's keycodes into ascii characters for the terminal emulation task. The mouse driver repacks bytes from the mouse into coordinates and mouse/tablet button keypresses. Device specific components handle the format of the mouse reports and the keycode translation tables.

2.3. The SCSI Driver

Most current workstations provide a single SCSI interface for accessing mass-storage devices such as disks and tapes through a common transport layer.

Our new SCSI driver has three layers. The upper one is specific to each of the major devices defined in the SCSI-2 standard. The code at this layer handles the queueing of requests, tape read errors, bad blocks, disk labels and so on. This layer is implemented as a common source file for open/close/read/write functions and a switch into device-type specific functions for extra open/close/start/restart operations. Common open-time operations include, for instance, dynamically probing a yet-unseen target, and bringing the target online and locking it if it contains removable media. Specific operations for a disk include setting the logical block size, and reading the size and geometry of the disk. Specialized functions, such as disk formatting and bad block scanning, are also exported at this layer.

The second layer defines the encoding of commands into SCSI messages, but also includes other utilities such as a watchdog to recognize a hung SCSI bus, data structure allocation and initialization code, and the definition of the per-target status record.

The bottommost layer handles the hardware proper and only has two interface functions: one to probe, and one to start a SCSI command. There is only one single upcall from this layer, to notify completion of a SCSI command and start the next one for the same device.

A Methodology for Handling SCSI Chips

SCSI chips typically require several interrupts per transaction, therefore it is important to dismiss the interrupts quickly. Some of the older SCSI chips, for example, require between 5 and 21 interrupts per disk read and write operation. We have structured our SCSI chip module as a set of "scripts," which are a list of condition-action pairs. One script might cover all SCSI commands that needs to receive data from the device, another one for transfers in the opposite direction. The condition encodes a possible value from the status registers, and the action is a function pointer. At interrupt time, the status registers are compared against the condition. If they match, the action routine is invoked. Otherwise, control transfers to an error handler associated with the script. At each interrupt, the anticipated condition-action pair is advanced to the next entry in the script until the command completes. For example, disconnections are handled as errors in the processing of a regular, non-disconnecting script. The script pointer is simply saved in the target device's status record and restored later when the target reconnects.

Our use of scripts, which draws on the design of the NCR 53C700, simplifies the writing of the chip-specific code by allowing us to use a single generic control module. Only scripts, action functions, and error handlers need to be written for each new SCSI chip. The more sophisticated SCSI boards, which include a processor, memory and other logic, do not require scripts because they are capable of handling most of the protocol details on their own.

Pushing Harder on SCSI

SCSI is a flexible model for device management — nearly any device can be interfaced via SCSI. In Mach 3.0, we need only write a small amount of machine-independent code to make a new device accessible across all machines. For example, we have connected two machines via a SCSI cable, much like we do with Ethernets, inventing a "host" device that can be used just like an Ethernet. This required only 46 lines of new machine-independent code, and the sharing of another 130 lines with the tape driver. We were able to use the existing structure to handle all of the SCSI nuances. Handling of a CD-ROM only required adding two lines of C code to the existing disk driver to prevent the issuing of write requests. We have dual-mounted the same disk on a DECstation 3100 and a IBM PC with only one additional line of code in the existing DS3100 adapter module.

2.4. The Ethernet Driver

The Mach 3.0 I/O system includes support for only one Ethernet driver based on the Lance chip controller. As most machines use this chip, we have not had much incentive to factor out code common to other Ethernet controller chips. Nevertheless, in importing Ethernet drivers from vendors, we have observed a "cloning" syndrome similar to that for other device drivers (which drive different chips). Our Lance driver is used on four different workstations. The driver copes with a variety of minor system dependencies through the use of callouts to machine-dependent functions that handle the movement of data in and out of the Lance's memory, and for translating a host address into a physical address usable by the Lance chip. Most of the system dependencies are due to the different ways in which the Lance handles DMA across different platforms.

The bulk of the machine independent code in the network driver deals with more general issues such as allocating and deallocating IPC buffers, delivering messages to users, and using the packet filter [Mogul et al. 87]. This code is common to all Ethernet drivers and can be generalized to any network interface.

3. User-Level Device Management

Devices can be managed from user-level by vectoring all device interrupts out to an application's thread. The kernel maps to user space the device's registers, a shared page containing some control information, and some memory for handling DMA to/from the device. When an interrupt comes, a small interrupt routine[1] saves any volatile register state in the shared page for later use by the user code, and then dismisses the interrupt, typically by disabling the interrupt enable bit in the device, or by reading an "interrupt-acknowledge" register. When the user thread runs it just invokes the driver's interrupt routine as if it were handling the interrupt in kernel-mode. After all necessary processing, the thread then re-enables interrupts in the device.

Our approach to user-level device management allows us to reuse existing kernel-mode drivers to a large extent, even though they run in user-mode. We have generally been able to run a kernel-mode driver in user-mode by providing some simple "scaffolding" for facilities that are normally present in the kernel, such as priority emulation and memory allocation, but not normally present in a user-level application. The only synchronization required for user-level device management is between the kernel's interrupt handler and the application thread. Presently, we use Mach's general `thread_suspend` (from user-mode) and `thread_resume` (from kernel-mode) primitives.

The small interrupt routine that vectors hardware interrupts to threads can be loaded in the kernel either dynamically or statically. Presently, we do it statically at link time, although we could provide a server that does dynamic linking and downloading in kernel space using the system's VM primitives, as is done on the NeXT. The interrupt routine only needs to invoke one kernel function to wake up the interrupt thread.

We should note that our user-level strategy scheme only requires one dedicated thread per device. It does not actually dictate whether this thread runs in user or kernel mode. Indeed on certain architectures, where mapping device registers is not possible, it might be mandatory that the thread runs in privileged mode.

Presently, we are running with user-level drivers for the Ethernet and the SCSI disk. The Ethernet driver was the first user-level driver we wrote, and is in fact the same custom driver (see Section 2) that first ran in the kernel. Our main motivation for moving the driver out of the kernel was our dissatisfaction with the performance of the in-kernel driver. By mapping the driver directly into the UNIX server's address space where the network protocols are implemented, we avoid one extra copy of the data, almost doubling the speed relative to earlier versions of Mach 3.0. In fact, current network performance for throughput intensive applications, such as FTP, is about the same as that for Mach 2.5, which implements UNIX in kernel space.

For the SCSI driver, we initially used the vendor's code directly on a DECstation 5000, and only later moved on to our own device independent machinery described earlier. We did this in order to assess the impact, in terms of performance and programmability, of moving existing, mature drivers out of the kernel would have. Performance is discussed in Section 5. In terms of programming, we were pleased to discover that the effect was minimal, and was all concentrated on the interface between the driver and the scaffolding code, not between the driver and the device. In fact, during our initial port, we didn't try to understand much of the code in the vendor's original driver — it wasn't necessary.

[1] For example, on the MIPS architecture, the routine is 128 bytes.

4. I/O Interface

The I/O interface is defined in a language-independent MiG definition file and consists of the following remote procedure calls:

device_open(*master_device_port*, *mode*, *name*, *device* **)** Open procedure, returns a *device* port

device_close(*device* **)** Close procedure.

device_write(*device*, *mode*, *recnum*, *data*, *num_bytes*, *bytes_written* **)** Write procedure, returns the number of bytes actually written.

device_read(*device*, *mode*, *recnum*, *bytes_wanted*, *data*, *bytes_read* **)** Read procedure, returns the data and the amount of bytes read. Reply can be asynchronous.

device_map(*device*, *protection*, *offset*, *size*, *pager*, *unmap* **)** Map procedure, returns a port *pager* for mapping to user space, usable with **vm_map**().

device_set_status(*device*, *flavor*, *status* **)** Change the device status, device-specific.

device_get_status(*device*, *flavor*, *status* **)** Inquiry the device status, device-specific.

Device names are strings, and are system-specific. Our convention is to use an alphabetic string followed by an optional decimal number which identifies different instances of similar devices. Record numbers are interpreted in a device-specific manner: a disk uses this unsigned index to point to a physical block, while a serial line just ignores it. Read and write operations can either return data inline or out-of-line. For devices that return data asynchronously, like the Ethernet, for example, a read call can be split in the request and reply sides, possibly with a different thread dequeueing replies. Operations on the status of a device, such as modem control operations on a serial line for instance, are very much device-specific.

Note that any entity that abides by this interface qualifies as a Mach device, whether it is implemented inside or outside of the kernel. The same interface is exported by the kernel for the devices it handles itself, therefore a user application will see no difference whether the driver is implemented in the kernel or in a user process. It is conceivable that a user-space driver could export some other interface, perhaps shared memory based, to other tasks on the same machine. Indeed, the current prototype, in which the SCSI driver is in the same task as default pager [Golub & Draves 91], exports the disk to the UNIX server via the RPC interface and to the default pager via local function calls.

It's important to note that Mach's support for distributed shared memory [Forin et al. 89] does not enable remote mappings of the chip's registers because devices do not access their registers through the memory management unit (MMU).

For devices that are implemented inside the kernel we provide a layer of code that handles VM and scheduling. At this level, we wire pages that are used to move data between user and kernel space. We also use the page-list technique, described in [Barrera 91], to speed up the paths through the VM code.

Scheduling issues are also handled here. Each device-specific function returns a code indicating whether the operation requested was able to complete, or was queued for later processing. If queued, the address of a completion function is noted in the request record. When the request has been handled, the driver's interrupt routine causes the completion function to be executed within the context of a kernel-mode thread. For example, in the case of a device write, the completion function deallocates memory and sends back a simple reply message to the writer with the a completion code indicating success or failure.

5. Performance

We consider two performance measurements for the new I/O system. The first is in terms of the reduction in size of device driver code. This is primarily a function of the new device independent drivers. The second measurement is in terms of performance; that is, how fast can data be pushed through the I/O system.

5.1. Size Considerations

We have observed, on average, a factor of two reduction in the size of device drivers relative to those provided by vendors. Moreover, our new drivers often include additional functionality. The screen driver, for instance, is one fourth the size (MIPS object code) of that shipped by the vendor, and now includes a terminal emulator. The extra code needed to support a hi-resolution screen required only 2KB of object code, compared to over 60KB if the new driver were cloned from an existing one (as is the standard practice). The chip-dependent code in the serial line driver takes about 4KB for each of the two chips we currently support. The support code for the NCR 53C94 SCSI chip is about 10KB, half the size of the vendor's chip-specific code. The most complicated SCSI chip so far needs about 17KB of MIPS object code. The simplest one is about 5KB of Intel 386 object code.

The machine-independent code is also compact. For example, the machine-independent code to support all of the SCSI tapes is about 4KB. The total size of the Lance driver for four machines is less than 8KB. The size of the machine-independent code for the entire I/O system in a generic DECstation configuration is 154KB.

The strictly machine-dependent device code is less than 6KB, and all of that is for handling DMA. Moreover, all of code is written in C. The remaining machine-dependent code in the system is 92KB, including 20KB of debugger support code and 13KB of floating point emulation code. Table 1 summarizes these numbers and shows that the new I/O system is significantly "less" machine-dependent (and therefore more portable) than other components in the system.

DECstation MK64 Generic Kernel

Component	Size (KB)	%
MI I/O	154	96.6
MD I/O	6	3.4
MI other code	364	79.8
MD other code	92	20.2
Total MI	518	84.1
Total MD	98	15.9

Table 1: Maximum Kernel Object Code Sizes.

5.2. Speed Considerations

Our new drivers perform no worse than those that they replace. In some cases, performance is even improved because of the mapped devices and the generic script facilities which allow us to rapidly dismiss anticipated interrupts.

The Screen and Serial Drivers

For the screen and serial drivers, there are no observable performance differences between our new drivers and the vendor's. In the case of the screen driver, this is because the vendor's driver was already mapped into user space, and because the kernel resident code has little impact on performance. In the case of the serial driver, it's because measuring performance differences at the slow speeds of 9600 or 19200 baud (typically the maximum rate for serial lines) is difficult.

The Ethernet Driver

For the Ethernet driver, we measured substantial performance improvements over the vendor's original driver. For example, an FTP using the same 4.3 BSD network code (pre Van Jacobsen) between two DECstation 3100s went from 120KB/sec to 230KB/sec.

The SCSI Driver

Initially, we measured the performance of an out-of-kernel SCSI disk driver which was identical to the vendor's original in-kernel driver. That is, we did not measure the impact that "device independence" had on the performance of the SCSI driver. We discovered that the in-kernel and out-of-kernel drivers performed similarly. The additional cost of having to dispatch a device interrupt out to user-level was insignificant compared to the long seek and rotational delays associated with disks (the average delay we saw was about 10 ms across a number of SCSI disks).

We next measured the impact that our device independent approach had on performance by replacing the vendor's driver (at user-level) with our own. On a DECstation 3100, we saw the maximum disk throughput improve from 700KB/sec to 850KB/sec with our new driver.[2] The throughput here was limited by a slow disk. We then replaced the disk with a faster one, and measured throughput of 1.52MB/sec, which is the maximum rate at which data can be moved between the SCSI buffer and the processor's main memory.

The SCSI driver is a particularly challenging case because of the large number of interrupts required to perform common device functions. We were clearly adding some overhead to the interrupt path. Because many SCSI devices tend to generate many interrupts per hardware operation, we were concerned that extensive coding changes would be required to get good performance. As the performance numbers demonstrate, this turned out not to be the case.

6. Some Observations about I/O Systems

This investigation of the I/O subsystem was originally just motivated by the need of doing a clean, free, reference port of Mach 3.0 to one of the many possible workstations on the market. The findings of the process, and past experiences in porting Mach to the many machines we ported it to are intriguing enough to prompt some more general reflections.

[2] In order to measure maximum throughput, we wrote a carefully tuned file-reading program that does reads out of order to maximize "hits" on the sector's location.

6.1. Horror Stories

Cutting the Wrong Corners

Economy is the foremost rule that has driven the design of current workstations. The results are oftentimes detrimental to performance. Most workstation manufacturers choose to include only a a cheap, dumb SCSI chip rather than a smart, more expensive SCSI board. This means anywhere between 5 and 21 interrupts to the CPU per (disconnecting) disk read or write operation. As an extreme case, we have seen an early SCSI disk disconnect on each and every sector transferred. This required 3*16+5=53 interrupts to read an 8KB disk block. We changed our SCSI driver to optionally disable disconnections for selected targets, but such work-arounds should not be necessary.

As another example, we have ported Mach to a multiprocessor which was built without any DMA support for disk I/O. The idea was that a multiprocessor machine can probably waste one processor in dealing exclusively with I/O. The CPU in this case must pick each individual byte out of the SCSI chip, just like a serial line. Unfortunately, the particular SCSI chip chosen would run 5 times faster in synchronous mode — a mode that necessitates a true DMA path to memory.

Balancing Costs

Many customers are willing to pay extra money for faster and color displays. This has generated a variety of solutions and offerings, often concealing important economic and performance considerations. Many users find it hard to understand why a high performance color machine should be slower at scrolling screen text than a monochrome one. At least some of the efforts in designing graphic accelerators, for example, should go into including higher speed screen memory. Moreover, it makes little sense to attach a slow graphics I/O processor to a fast CPU.

Delivering Promised Function

Another area where we hit many obstacles is the one of DMA. Any DMA device that cannot be used to access each and every byte at any physical address creates a software problem which can only be solved by data copies that slow down the machine. This is even more of a problem considering that with today's CPUs, memory is often the bottleneck. We have seen machines that can only DMA two good bytes every other two bytes, some that can only use a good byte every four (and not byte zero), and some that get a good 16 bytes in a row, but only every other 16 bytes. In other instances, the DMA is "normal," but the mapping between physical address and address to be used by the DMA chip is incredibly complicated. DMA chips which can address as much as the CPU can are rare.

Caches

A big cache helps with the performance of user applications, but is less helpful for the operating system [Ousterhout 90]. As for the I/O system, a machine with a DMA chip is essentially a multiprocessor with cache coherency problems which should not be overlooked. If the cache does not snoop the bus, it is necessary to factor into each I/O operation the cost of flushing the cache, which on many machines is not a trivial one, not even for a relatively small address range such as a page size. The cost of flushing can be as high as 25% of the entire page fault cost. In Mach, we can somehow help by avoiding the instruction cache flush for pages that are not mapped (by the user) with execute permission, as we do on the MIPS architecture for instance. This only mitigates the problem, and only in the case of separated instruction and data caches.

6.2. Suggestions

An I/O system that performs in the gigabyte throughput range will require radical departures from today's practices. High bandwidth will only be possible with large grain data transfers, effective buffering and memory mapping techniques. This is only possible if hardware and software cooperate.

It is important to handle more than one transaction per interrupt because interrupts have a bad effect on cache and CPU (pipeline) performance. High-performance I/O systems will have to reduce the number of interrupts required to handle data transfer. Otherwise, tomorrow's faster CPUs will spend all their time handling one network packet at a time, just as they do today with serial lines. Large data transfers are possible because main memories are large enough to hold data in anticipation of it being used, and modern virtual memory systems are able to effectively cache the data.

Given our experiences with with user mode drivers, we can consider other uses of memory mapping techniques that improve performance. Consider, for instance, a machine where each device is accessible as a separate memory bank on the main memory bus. This large piece of dual-ported memory is where the operating system server allocates buffers used for I/O. This structure gains two advantages. First, the bus is used only for CPU transactions since devices DMA to their local memory. Second, the copy of data in and out of application space is made by using mapped file techniques [Golub et al. 90] only once and in large chunks. This copy eliminates the need for data cache flushes, because the source data can be marked as non-cacheable.

An I/O interface different from that of UNIX would avoid even this one copy. Many other operating systems have successfully used such a buffer "reserve-fill-release" strategy.

An alternative setup is one where the interface between the main CPU and an I/O device is in terms of an external pager interface itself. The device itself is the external pager and interacts with the main CPU in terms of pagein/pageout requests in a fault-driven fashion. This is a generalization of our work on shared memory servers [Forin et al. 89]. The difference now is that instead of only dealing with "communication" issues we also deal with "permanent storage" and data retrieval issues. If the memory mapping is between two hosts, then we have a distributed shared memory semantics. If the mapping is between a host and a peripheral device (disk, tape, printer, scanner), then we will either retrieve data (read fault) from the device or write it to the device (write fault). By mapping, the host communicates to the device the data it wants to address (e.g. what disk blocks). By faulting, the host signifies that the transfer should take place. In this way, we can use lazy evaluation to drive I/O devices. The device now has the advantage of being able to make decisions of its own as to what stays in the main memory and what doesn't. It can, for instance, remove access to a page just because it is convenient to write it out at that particular point in time, or it can prefetch data and supply it to the kernel in anticipation of an upcoming need.

7. Related Work

Other systems use some of the same techniques for I/O management that we have used in Mach 3.0. Jim Gettys [Gettys 91] has recently rewritten the screen driver for Ultrix by factoring the code into chip-specific modules and generic code. In Sprite [Ousterhout et al. 88], device drivers are structured like ours — functionally specialized into much the same set, although the implementation does not stress chip-specificity as much as Mach's. An experimental version of UNIX based on a micro-kernel done at DEC ran with device drivers in user-space [Palmer & Palmer 89].

8. Current Status

The work described here has been a developing part of the Mach 3.0 kernel since the middle of 1989. We invite the interested reader to obtain a copy of Mach 3.0 by way of anonymous FTP to CS.CMU.EDU.

The device independent serial driver has been ported to two chips (DEC DZ7085 and Zilog 5380) on three machines. The screen driver currently handles two monochrome and five color display types, and is used on two workstation types (VAX and MIPS based). Other ports are under way.

The SCSI driver has been ported to four different workstation types (VAX, MIPS, I386, M88k) and handles five different SCSI controllers ranging from the first-generation NCR 5380 to the second-generation NCR 53C94 to the user-friendly Adaptec 1540. Others, outside of CMU, are using these drivers for Mach ports to other systems.

The user-level Ethernet driver has been in use now for almost two years on three versions of the DECstation (2100, 3100 and 5000/200) with a fourth one just completed (5000/120) and a fifth one underway (Omron Luna 88k). It is distributed as part of the single-server UNIX emulator from CMU.

References

[Barrera 91] Barrera, J. S. *A Fast Mach Network IPC Implementation.* In *Proceedings of the Second USENIX Mach Symposium*, This issue, November 1991.

[Barrera 92] Barrera, J. S. *Operating System Support for Multicomputers.* PhD dissertation, School of Computer Science, Carnegie Mellon University, To be completed in 1992.

[Forin et al. 89] Forin, A., Barrera, J., Young, M., and Rashid, R. Design, Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach. In *1988 Winter Usenix*, January 1989.

[Gettys 91] Gettys, J. E–mail communication posted on the *mach3* mailing list, July 1991.

[Golub & Draves 91] Golub, D. and Draves, R. Moving the Default Memory Manager Out of the Mach Kernel. In *Proceedings of the Second USENIX Mach Symposium*, This issue, November 1991.

[Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, June 1990.

[Mogul et al. 87] Mogul, J., Rashid, R., Accetta, M. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 39–51, 1987.

[Ousterhout et al. 88] Ousterhout, J., Cherenson, A., Douglis, F. The Sprite Network Operating System In *IEEE Computer, Vol 21-2*, pages 23–26, February 1988.

[Ousterhout 90] Ousterhout, J. Why Aren't Operating Systems Getting Faster As Fast As Hardware? In *Proceedings of the Summer 1990 USENIX Conference*, June 1990.

[Palmer & Palmer 89] Palmer, R., Palmer, L. Informal Communication at the *First OSF Kernel Developers Meeting*, Cambridge, September 1989.

[Rashid et al. 89] Rashid, R., Baron, R., Forin, A., Golub, D., Jones, M., Julin, D., Orr, D., Sanzi, R. Mach: A Foundation for Open Systems. In *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, page 109–113, September 1989.

[Rashid et al. 87] Rashid, R., Tevanian, Jr., A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1987.

[Rashid et al. 91] Rashid, R., Malan, G., Golub, D., and Baron, R. DOS as a Mach 3.0 Application. In *Proceedings of the Second USENIX Mach Symposium*, This issue, November 1991.

[Ritchie and Thompson 78] Ritchie, D., Thompson, K. The UNIX time-sharing system. In *Bell System Technical Journal*, July 1978.

[Tokuda & Nakajima91] Tokuda, H., Nakajima, T. Evaluation of Real-Time Synchronization in Real-Time Mach. In *Proceedings of the Second USENIX Mach Symposium*, This issue, November 1991.