An Integrated Core-Work for Fast Information-Appliance Buildup[1]

Paul C. H. Lee, Chi-Wei Yang, Ruei-Chuan Chang

Institute of Information Science,
Academia Sinica, Nankang, Taipei, Taiwan
{paul, chiwei, joanna, rc}@iis.sinica.edu.tw

Abstract

The advances in digital technologies have given birth to many applications, which were hard to image in the past decade but are real in current time. For the versatile but specialized applications, general system software in tradition does not perform well to serve their specific demanding requests, because the general approach is designed to concern with most of the general cases. Besides, there exists complex dependency among software components in a traditional system, such that to modify or to reuse software components is not easy for application designers. In this report, the specialized applications that can be benefited from configurable and customized system components are termed as *information-appliances*. A framework for building up information-appliances efficiently is the body of this report. All the low-level core components that relate to hardware specifications are designed from scratch. The current version is dedicated for Intel 386 or newer processors.

The contributions of this report can be illustrated as follows. First, our work supplies a research and development (R&D) platform. Through our work, researchers and designers can build their information-appliance prototypes efficiently that they do not need to implement so many low-level components first just for trying one tiny idea. Second, all the core components are designed modularly and well documented in exported and imported interface. Information-appliance performance can be benefited from this design by dropping any components that are not used by them. In addition, our work supply higher chances for specific management since each component is isolated and independent to others. Application designers can have more controls in behavior of each component. Third, our work includes a wrapper-socket for

[1] This work is a part of the *Ramos* project in IIS for building the basic system software components touching hardware. The driver layer is to make use of the shareware – the Linux device drivers.

incorporating Linux X86 device drivers. This specific core component, the wrapper-socket, enables our work to make reuse of the biggest device driver source pool in the world directly. Any new patches or versions about the Linux device driver sources can patch to our framework directly, just with very minor modifications to the wrapper-socket implementation. Finally, the whole core-work is evaluated via empirical measurements. From these experimental results, the experiences about the system buildup, discussions about the device driver architecture impacts to real-time applications are also presented.

Keywords: kernel, Component, Linux, operating system, device driver

1. Introduction

So far, building software as independent components is one non-stopped trend in application designs and implementations. Its practical contributions bring application programmers with higher throughputs and make application designs easier and faster. Via making standards in defining each software component, software-reusability can be achieved that applications can just be built by integrating these components. Previous proposed CORBA, OLE, VBX and OpenDoc all are standards used to define the cross-platform component architecture. By just looking at their industry impacts, it is not hard to know that component-ware architecture is how important to the software design and buildup. Unfortunately, so far we see nothing special the system software benefited from the promise component-ware technologies. Most reasons are due to the fact that system software is the primary interface to hardware architecture and devices. Efficiency is the most important criteria for system software buildup. Efficient deign usually means that software is built by ad hoc skills, which cause software hard to be maintained and built. It is expected one day that system software can also be built by as simple as application designs. And this is also the primary goal of our work.

1.1 Motivation

On the research in finding a way for efficient system software components, a fact is found to answer the questions. It is found that traditional systems can’t meet all the demanding needs for all applications because they are originally designed for servicing all the applications [Bershad 95]. System performance usually degrades because of inducing this generality. If the system service can be specialized to meet application’s needs, system performance will behave better and beyond what it is

that we cannot image before [Lee 94, Lee 96a, Lee 96b]. Table 1 and Figure 1 illustrate an example about building such a specialized Network File System (NFS).

In the example illustrated in Table 1 and Figure 1, the F540 is a specialized system designed for NFS service only. It employs the Data ONTAP 3.14 real-time kernel as its system software. On the contrary, the Lifeline Series 1000 uses the Digital Ultrix 4.0A as its operating system even though the Lifeline Series 1000 is built for NFS service only. The Lifeline Series 1000 usually has better hardware configuration than those of F540 do. The performance results evaluated by the SPECnfs_A93 benchmarks showed great performance gap between these servers [SPEC 96, SPEC 97].

Taking the standing points of consumers, it means consumers will spend more money in buying the Lifeline Series 1000 and cost more in maintaining the general and bigger Ultrix operating system, but they will get poor performance than those buying the FS540. This motivated us that we need a system, which can be configured, specialized and composed easily. Applications built on top of such a system can be constructed by combining the services this system provides. Aiming on this goal, the first step is to build the basic core components from scratch. In this report, a framework is designed and implemented. The primary purpose is that with modular and easy-to-customize software components, designers can build versatile systems via this framework more efficiently. Of course, there remain other practical problems that need to be solved. For example, how to write a bundle of device drivers is a very serious problem to any system development in academic environments. Thus a middle-ware to incorporate existing shareware device drivers is another purpose of this report.

1.2 Information-Appliance

Of course, system built via our framework is not so powerful than a general system since generality conflicts with our design goals. In this report, a new term, the *Information-Appliance*, is used to refer to those systems and applications that are not designed for general use. It will not be so large in code size and has *exact* function supported from the system software. Information-appliances are different from embedded systems in that embedded systems typically have a finite set of functions supported and most functions are supported directly from the hardware, which are not easily expandable such as automatic teller machines and CD players. Information-appliances are composed mainly of software. Their functions come mostly from software and could be easily reconfigured and expanded.

Table 1: Comparison of NFS server hardware.

	Lifeline Series 1000	**Network Appliance F540**
CPU	Alpha 366MHz	Alpha 275MHz
Primary Cache	16KBI+16KBD	16KBI+16KBD
Secondary Cache	2MB(I+D)	2MB(I+D)
Other Cache	32MB NVRAM per RAID Controller	8MB NVRAM
Memory	512MB	256MB
OS Version	Digital UNIX 4.0A	Data ONTAP™3.1.4Beta
Num Disk Controllers	3	1
Num Disks	37	14
Num File Systems	12	1
Num Net Controllers	1	1
Type Network	FDDI	FDDI

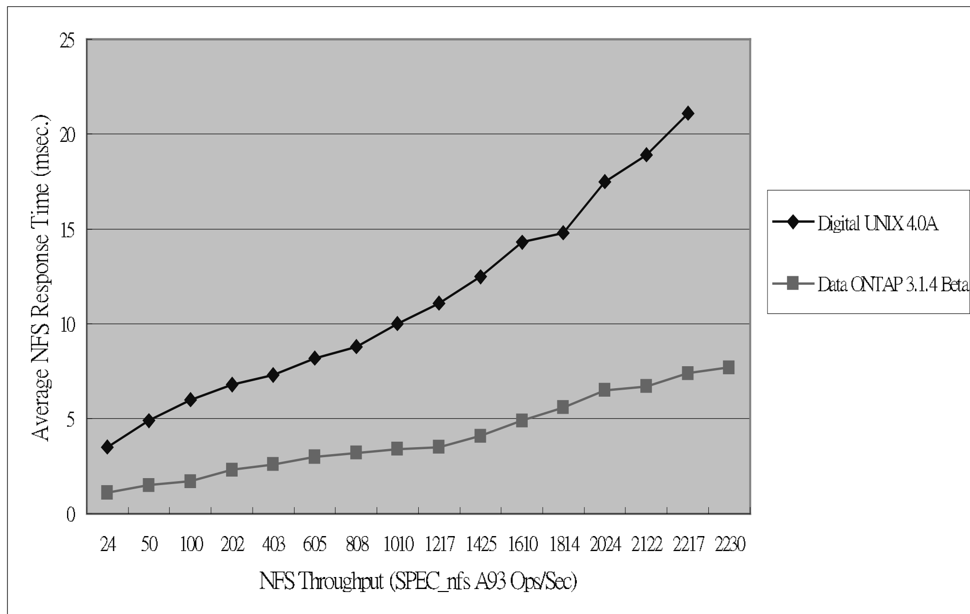

Figure 1: Performance comparison for NFS servers.

Under formal definition, *information-appliance is a system that supports communication, information storage and user interactions, which is usually special-purpose and does not need general and complete operating system functions supported. It can be reconfigured from general software components. Due to its specialization, the functions supported can more meet the special application's needs, which usually causes better performance and saves unnecessary cost. Information-appliance should be built on an easy to customize platform but does not loss any functionality that it needs.* It is evident that the information-appliance requires a system framework to support modularized software components, which have clear interface and are independent to other components.

1.3 Contributions

The primary contribution of this report is the design and implementation of a framework for fast information-appliance buildup. The contributions can be characterized as follows. First, the framework provides low-level machine dependent codes. There are many low-level machine dependent codes, e.g. bootstrapping codes, that are basically uninterested for most researchers but they are needed for a system to work. Providing these codes could save a lot of time and more time could be spent on interested parts. Researchers and designers can build their information-appliance prototypes efficiently since they do not need to write many low-level machine dependent codes before testing one tiny idea. Second, the framework provides core components modularly and as more independent to others as possible. The import and export interfaces are also well documented. With independent and modular design, the framework is easier to customize. Designers could pick up only needed software components. This saves unnecessary cost as compared to full-supported functions a traditional system provides. In addition, designers could have more control in the behavior of each component since they are independent and modular. For those components of interest, designers could develop new software components according to the required import/export interfaces to replace existing software components or add new ones. Third, the framework can make use of Linux device drivers. Linux is a freely distributed operating system and it has many device drivers supported. A wrapper-socket is designed to incorporate almost unmodified Linux device driver sources. With the wrapper-socket, a lot of device drivers could be used in our framework. Incorporating almost unmodified Linux device driver sources means next time when new version Linux is released, new version device driver sources can be incorporated by applying a minor enhancement to the wrapper-socket.

1.4 Related Work

Goel and Duchamp design an emulation layer to incorporate unmodified Linux block, network and SCSI (Small Computer System Interface) device drivers into Mach 4.0 [Goel 96]. The in-kernel device independent layer of Mach is modified to recognize the possibility of Linux device driver emulation. Other parts of Mach are also modified for emulation. For example, addresses used in Linux device drivers are identical to physical addresses while Mach doesn't. Hence, they modify Mach to use the same assumption. In our design, the core component provides minimal functions supported and we

write emulation codes using these supported functions.

A toolkit named OSKit is built to be a substrate for kernel and language research [Ford 97]. Via the toolkit, a customized system can be built using software components provided by the OSKit. The OSKit consists of Linux and FreeBSD device drivers, NetBSD and FreeBSD networking protocol stacks, NetBSD file systems and other parts such as bootstrapping codes and standard C libraries. The OSKit is mainly composed of general-purpose software components from other general-purpose operating systems. These operating systems may have different presentation for one data presentation. For example, Linux uses skbuffs and FreeBSD uses mbufs to represent network packets. To solve this problem, the OSKit defines an internal common data structure and does the translation in each component "glues." This causes performance penalty if frequent translations are needed.

The problem for system customization had been addressed [Auslander 97, Krieger 97]. The main idea of customization is via providing building-block components and letting applications specify needed building blocks. These building blocks specified by applications constitute virtual resources, e.g. files or memory regions. Since applications provide information for these building blocks, this allows different virtual resources to have different characteristics that are best suitable for each application.

Linux is a general-use operating system. Barabanov modified Linux to support specialized applications, the real-time tasks [Barabanov 97]. The Linux kernel can't be preempted when executing in kernel mode. This is not suitable for real-time applications since they need predictable behavior of the executing codes and short response time for asynchronous events. In his design, real-time tasks are implemented as loadable kernel modules and execute in the same address space as Linux kernel. The original Linux kernel is also a real-time task, which has the lowest priority. Major modifications to the Linux source codes are on the interrupt handling part. Software interrupts are used for the original Linux kernel such that it can't block higher priority real-time tasks. The drawback of this approach is the lack of support for real-time tasks, e.g. real-time tasks can not easily make use of Linux device drivers, networking, etc.

There are many people working on extensible operating systems. Mach [Accetta 86] is a micro-kernel architecture operating system. Many traditional operating system services are moved to user space as user servers. The advantage of this architecture is the flexibility of adding or replacing new user servers. A potential drawback is the performance since many user-kernel context switches would occur when user applications request for system services. Exokernel [Eagler 95] tries to reduce the number of context switches between user and kernel space by abstracting underlying hardware and

providing them as libraries. In SPIN, applications could dynamically link their specialized codes into kernel to add new services, replace old ones or simply move the codes from user space to kernel space [Bershad 95].

Real-time and multimedia applications are getting more attentions in recent years. Real-time applications generally have the property that requested data is meaningful only if data is arrived on time. Rajkumar et al. proposed a resource-centric approach for designing operating system kernels [Rajkumar 98]. The kernel is termed as *resource kernel*. The main idea is that applications specify their needs and the kernel is responsible for satisfying these demands once the kernel granted the requests. The interface for specifying these requests are designed for applications that is periodical in nature such as real-time applications.

2. System Architecture

Our work can be divided into two parts, one is the low-level system core components used for building a real-time kernel prototype. The other is a middle-ware for incorporating and making use of Linux device drivers. The latter is needed because in academic research environment, we do not have so many human resources in keeping the device drivers as new as hardware devices are created. And we also do not have so many human resources to write down so many drivers. To make use of existing ones seems a good way for us to try at the system building stage.

2.1 Overview of the System Architecture

The overall system architecture is shown in Figure 2. As can see from the figure, main components are core, I/O system and library. The core component has functions that is mostly machine dependent and has necessary functions to set up the executing environment. The I/O system exports functions for doing raw device I/O and the library provides functions that are usually supported by standard C libraries. The user program together with these system components constitutes the special purpose application.

Through the low-level core components, the kernel body is built on top of them. For example, the kernel semaphore services should be built on top of *lock* core component, *interrupt* core component and *switch* core component. The kernel time and timer services should be built on top of *clock* core

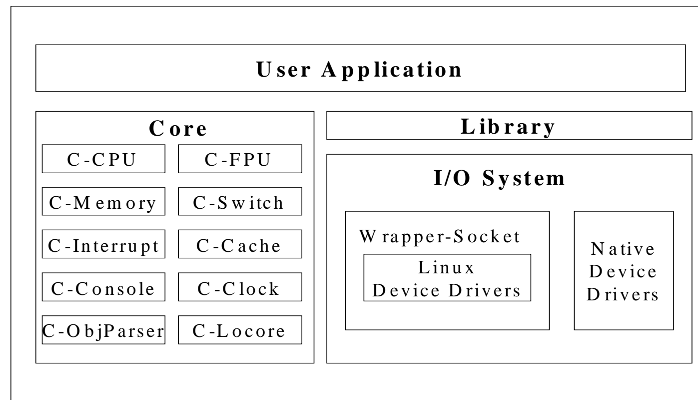

Figure 2: Overview of system architecture.

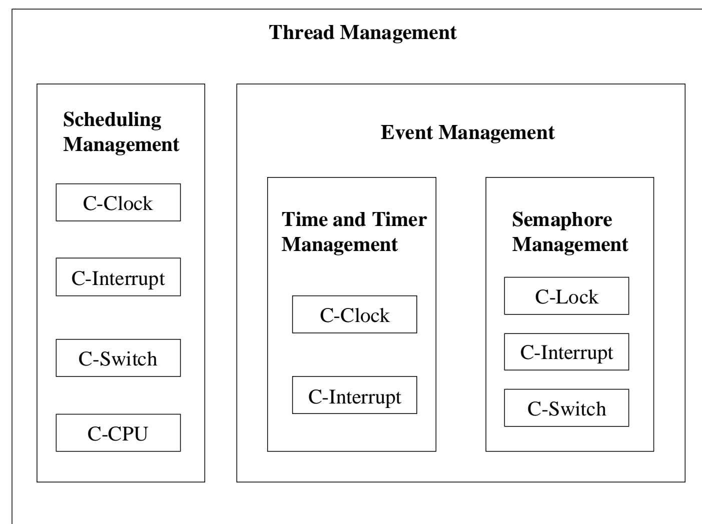

Figure 3: An example for using core components.

component and *interrupt* core component. And the thread management is built on top of all of the core components and other kernel services. For example, the event management, the semaphore management, the scheduling management, etc. Figure 3 is a simplified graph that shows this kind of dependency.

2.2 The Core Component

Here we usually mean that the system core components are the software, which are strongly dependent on hardware configurations. For example, the clock core component is the software that handles the Intel 8259 chip and the CPU (Central Processing Unit) core component handles the Intel Pentium processor, the functions such as to indicate the CPU states is busy or idle and the priority for this CPU's owner thread, etc. The current implemented core components are illustrated as follows.

```
struct cpu {
        struct thread *pc_thread;        /* Thread CPU's running */
        uint pc_locks;                   /* # locks held by CPU */
        uint pc_pri;                     /* Priority running on CPU */
        uchar pc_flags;                  /* See below */
        uchar pc_num;                    /* Sequential CPU ID */
        uchar pc_preempt;                /* Flag that preemption needed */
        uchar pc_nopreempt;              /* > 0, preempt held off */
        ulong pc_time[2];                /* HZ and seconds counting */
        ulong pc_ticks;                  /* Ticks queued for clock */
        struct percpu *pc_next;          /* Next in list--circular */
};
```

Figure 4: The *CPU* data structure.

C-CPU

C-CPU is the software to abstract the CPU. This means that any software built on top of C-CPU only knows this is a CPU here and which functions exported by this CPU. Unlike previous kernel implementation, the software does not know whether this is a MIPS processor or an Intel Pentium processor. The data structure to abstract the CPU is illustrated in Figure 4. And due to the page size limitation of this report, the member functions for C-CPU is omitted.

C-Memory

The C-Memory is the core component that handles the physical memory management. Information such as the physical page is valid or invalid, free or occupied and which logical address is mapped to this page if mapping is specified in this system. The structure *pm-core* is the data structure keeping per physical page information and the structure *pm-set* is used for keeping information about the physical memory pool. How to maintain the page table to do physical and virtual addresses mapping is another mission of this component. Based on this C-Memory core component, the memory allocation scheme is built via the buddy algorithm [Challab 87]. Each allocated memory size is a power of two. We wish other powerful memory functions be added on top of this component in the future. For example, to allocate contiguous physical memory for accessing by I/O system in DMA (Direct Memory Access) is one important function. To add more functions in supporting physical to virtual addresses mapping is needed in servicing any inter thread synchronization. Following illustrated the *pm-core* data structure.

```
struct core {
        uchar c_flags;              /* Flags */
        uchar c_dummy1;
        ushort c_psidx;             /* Index into pset */
        union {
                struct pset         /* Pset page is used under */
                        *_c_pset;
                …
                struct core         /* Free list link when free */
                        *_c_free;
        } _c_u;
#define c_pset _c_u._c_pset
#define c_word _c_u._c_word
#define c_free _c_u._c_free
};
```

Figure 5: The *core* data structure.

C-FPU

This is a simple core component for abstracting the floating-point processor, if any real floating point processor exists in the system. On the other hand, if there is no such co-processor exists, the functions can be emulated by software skills. On that case, the C-FPU cannot emulate the total functions by itself only. It requires the help from C-Interrupt component. The current implementation only includes the case that there is a hardware floating-point processor attached in this system. The member functions so far include the *c_fpu_enable( ), c_fpu_disable( ), c_fpu_init( ), c_fpu_detect( ),* etc. All the functions are implemented in assembly codes.

C-Interrupt

C-Interrupt is the key component in the system and should be the most important component within the total system. The preemptive and multi-threading functions are done by periodical scheduling routines in the highest priority interrupt, e.g. IRQ 0. The entire I/O subsystem is built on the interrupt core component that device drivers register their interrupt service routines and respond to any external

interrupts when C-Interrupt triggers their handling routines. In this component, it includes functions to setup the interrupt table, to register the interrupt service routines and to de-register the routines, to enable and disable interrupts and to setup interrupt masks for filtering any certain interrupts. The whole C-Interrupt component has strong relationship to hardware specification [Crawford 87, Gilluwe 97].

C-Console

C-Console component is originally designed for debugging the system while the system is in developing stage. The primary purpose is to show information about the states of each component and the kernel. Two hardware devices relate to this component. One is the display terminal and the other is the serial port device. In current implementation, the VGA [Ferraro 90] is used as the standard console device and the output message is shown in the target platform. The serial port part is not implemented yet. If done in future, its purpose is to bring message to the remote host for monitoring the target in remote side. Functions such as *c_console_init( )* and *c_console_puts( )* are implemented so far. A simplified ASNI-C like kernel service is implemented on top of this component, which emulate the *puts( )* and *printf( )* calls.

C-ObjParser

C-ObjParser is the only system component that is nothing related to the hardware specification. The function of this component is to parse the object code in any known object code format. The ELF, COFF and AOUT are the most popular standards and should be on this component's wish list. Currently, only the AOUT is implemented.

C-Switch

C-Switch is the component about execution stream context switch. Here, execution stream means the thread that will run on the C-CPU and is built on top of C-Switch component. The most important exported interface for this component is the *setjmp( )* and *longjmp( )* calls. These functions are used to record and reload the execution entries for the running thread. When *setjmp* function is called, a *jmpbuf* buffer is used to store all the context information, which include all the register contents, all the contents of system flags, stack pointer and where is the previous *jmpbuf*. The *longjmp( )* does the reverse operations as *setjmp( )*. Usually, a return value will be handcrafting into the top of operating stack by passing a true or false value to the caller. This skill is used to change the execution flow for

any thread that calls these functions. When compared to traditional embedded kernel implementations [Labrosse 93, Labrosse 95] that disable all the interrupts when preparing the context switch operations, this skill is efficient. Of course, it also causes a little complexity to the design for context switch manager. However, since context switch operations will dominate the system performance, this sacrifice is needed.

C-Cache

This component is designed for any processors that have powerful software TLB (Translation Lookaside Buffer) or cache operating function supports. So far, on Intel 486 or higher processors, only *c_cache_enable( )* and *c_cache_disable( )* are implemented.

C-Locore

C-Locore component includes any functions that are necessary but cannot counted to any other core-components. For example, the most popular *bcopy( )* and *bzero( )* functions are necessary for operating memory copy and initialization. Since so many components will invoke them, it is not a good idea to implement these functions on the C-Memory component. One feature of this component is that all the contents of this component are implemented in assembly codes. No data structures are needed to abstract any physical hardware. They are just the functions for supporting the kernel buildup.

C-Clock

The last component for this report is the clock core component. It includes operations to initialize the hardware chips, to synchronize with C-Interrupt component and to turn on the clock tick interrupt. The other functions include maintaining the system clock, maintaining a timer system for alarming any specific functions and reporting exact time service to the system kernel functions such as *K-ethread_sleep( )* and *K-ethread_wakeup( )* need services from C-Clock. In addition, for systems that require high precision time report, a high-resolution clock using Pentium cycle counter [Mathisen 94] is also implemented in this component.

2.3 The I/O System

In our current implementation, the I/O system consists of native device drivers and a wrapper-socket to incorporate existing Linux device drivers. Device drivers that may not change frequently to support

new products such as mouse drivers are designed as native device drivers. The wrapper-socket incorporates device drivers that may change frequently to support new devices.

Since there are native device drivers and existing Linux device drivers in the I/O system, we need a common interface for these two types of device drivers. Linux device drivers export interfaces that are best suitable for Linux. We need a new interface that is suitable for our framework. The exported interface and the design of wrapper-socket are detailed in Section 3.

2.4 Kernel Personality

Based on the core components and the device driver wrapper-socket, a small multi-threading kernel is built to demonstrate the functionality of the system. Currently, we have several managers composed to the *eRamos* kernel. They are thread managers, memory manager, semaphore manager, debugger manager, console manager, interrupt manager, event manager, timer manager and scheduling manager. Most of the functions and policies of current manager implementations are simple. For example, a FIFO (First In First Out) policy is implemented in the scheduling manager to arrange the execution sequence of each thread. Further experiments for various mechanisms and policies will be touched in future research.

3. The Wrapper-Socket

For those who work with operating systems, they know that the biggest trouble comes from the fact that there always exist infinite devices waiting for drivers. New hardware devices come to this world day by day. The device vendors will only write drivers for new devices on commercial operating systems and well-known kernels. For academia systems, driver writing is usually a major burden in system developments and researches. For running any system projects and experiments, it is important to find a way to escape from the curse of driver writing. The way that we apply here is to try to make reuse of existing shareware device driver codes. And the goal that we expect is that once any efforts paid in this work, only minimal extra efforts are needed in making reuse of newer patches or versions of driver codes.

3.1 Make Use of Linux Operating System

In our work, we select the Linux operating system as the device driver source pool. Linux is an UNIX-like operating system. It is written by Linus Torvalds with assistance from programming hackers all over the world via Internet. The Linux software is freely available and is developed openly that many documents exist in describing the operating system internals [Beck 96, Johnson, LDP].

The reasons we need to make reuse of Linux device driver sources are as follows. First, though there are many device drivers in operating systems such as MS-DOS and Microsoft Windows95, these commercial drivers usually do not release sources to end users. Those binary drivers are very hard to incorporate into any systems that differ from the original ones. Second, as described above, we expect a new framework to be built but not a lot of device drivers to be written. Thus we need to find existing solutions. The Linux is the biggest source pool for system software and is endorsed by the biggest programming hacker pool. Third, device driver is also a system resource within an operating system. We are interested in an issue about relationship between Linux device driver architecture and real-time services. Instead of experimenting under Linux directly, we can incorporate the Linux device drivers into our system first and do experiments and evaluations under our system. The research will be easier than directly working with Linux. In stead of describing the wrapper-socket implementation, the fundamentals about Linux device drivers are introduced in following sections.

3.2 Linux Device Driver Architecture

The Linux kernel is a monolithic kernel that system services are provided via function calls. Device drivers are part of the Linux kernel. In other words, the Linux device drivers are basically privileged shard libraries and consist of low-level device handling routines. User programs call Linux I/O system calls to request any I/O services. The basic I/O system architecture is illustrated in Figure 6.

The block, character and network are three main types of device driver services classified in the Linux. Character devices are those, which handle data that is serialized and usually in byte streams. Data handled by this type of devices does not need to be cached by buffer cache and usually cannot be randomly accessed. Parallel port, mouse, sound card and console-terminal are examples of character devices. It is straight to find out that the character devices are usually slow devices. Block devices, on the contrary, access data in units of block, which is usually a multiple of 512 bytes in Linux. Data operated by this type of devices are usually handled in DMA (Direct Memory Access) technology, which means the data will be transferred from the I/O devices to the memory directly without the

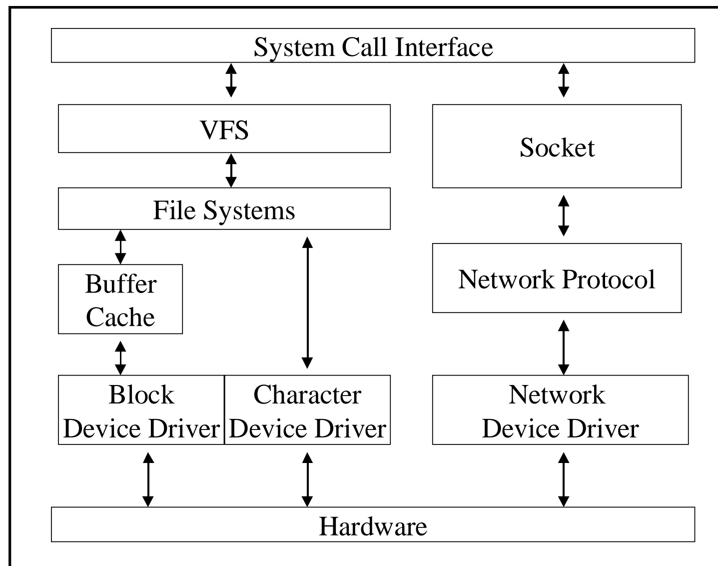

Figure 6: An overview of Linux I/O system.

intervention of CPU. Since direct writing or reading to the block devices are costly in just modifying a small portion of block, the buffer cache is usually applied in I/O subsystems to reduce latency time in reading and writing data. Usually block devices permit randomly accessing data. Hard disk is the most popular block device, for example.

Network devices are mainly composed of network cards. Ethernet cards and FDDI adapters are examples. There are also network devices in pure software, e.g. loop-back device. A loop-back device is used for transmitting network packets to itself.

3.2.1 Character and Block Device Drivers

Character and block devices are abstracted as special files in Linux file systems. For example, the first IDE (Integrated Device Electronics) device is abstracted as /dev/hda. For each device, a pair of major and minor number is associated with each device. Major numbers are assigned to different type of device and minor numbers are used to distinguish devices of the same major number. For example, the major number of primary IDE interface is 3 and disk 0 partition 0 attached to this interface has minor number 0. The major number assignment is unique. Table 2 summarizes some common devices and their major numbers.

Character and block device drivers both export the *file_operations* interface. Table 3 lists the *file_operations* interface. Linux uses two arrays, one for character devices and another for block

Table 2: Some assigned major numbers.

Major number	Character devices	Block devices
1	Memory devices	RAM disk
2	PTY Master	Floppy disk
3	PTY Slave	IDE0
4	TTY	
5	TTY and AUX	
6	Parallel interfaces	
7	Virtual Console	Loopback
8		SCSI disk
9	SCSI tape	RAID disk
10	Bus mice	
11	Raw keyboard	SCSI CD-ROM
12	QIC-02 tape	Mscdex cdrom
13	PC speaker	XT 8bit disk
14	Sound card	BIOS hard disk support
15	Joystick	Sony CDU-31A/CDU-33A CD-ROM
16	Non-SCSI scanners	GoldStar CD-ROM

devices, to record each device's *file_operations*. Device major numbers are used as index into the array.

Character device drivers typically provide their own read/write functions since they are accessed without cache. They generally can't be random accessed so they leave some functions unfilled, e.g. *lseek* function. Block device drivers are accessed through the buffer cache. They need not to write their own read/write functions in the *file_operations* structure. Instead, they use the common read/write functions of buffer cache, which are the *block_read* and *block_write* functions. Since all block device drivers share common read/write functions and block devices may have different characteristics, each block device driver must provide a function to process I/O requests from the buffer cache. The function prototype is the same for all block device drivers but implementation varies. This function is termed as *request function.*

When a process issues I/O requests, the buffer cache handles this request. It checks whether requested blocks reside in buffer cache. In the write case, if requested blocks are in the cache and clean, it does nothing. In the read case, if requested blocks are in the cache and up to date, it returns the requested data. Otherwise it constitutes properly filled I/O request data structures. The request is then

Table 3: Linux *file_operations* interface.

int	lseek (struct inode *, struct file *, off_t, int)
int	read (struct inode *, struct file *, char *, int)
int	write (struct inode *, struct file *, const char *, int)
int	readdir (struct inode *, struct file *, void *, filldir_t)
int	select (struct inode *, struct file *, int, select_table *)
int	ioctl (struct inode *, struct file *, unsigned int, unsigned long)
int	mmap (struct inode *, struct file *, struct vm_area_struct *)
int	open (struct inode *, struct file *)
void	release (struct inode *, struct file *)
int	fsync (struct inode *, struct file *)
int	fasync (struct inode *, struct file *, int)
int	check_media_change (kdev_t dev)
int	revalidate (kdev_t dev)

queued in an array. Device major numbers indexes this array and it is used to record current requests. If there are many requests queued, the elevator algorithm orders these requests. After the request function is invoked, device drivers process these I/O requests.

The read/write data flow for block device drivers is shown in Figure 7. A process calls *read/write* system call. The buffer cache checks whether the requested buffer is up to date. If it is up to date, the buffer cache returns the data buffer or does nothing in *write* system call. If it is not, the block device driver exported request function is called. The process will then wait on the buffer. When actual data transfer is done, an up to date flag is set on the buffer structure. Any time when this process is scheduled, the process checks this flag. If it is not set, it calls the scheduler to run. Otherwise, the transfer of requested data is done and the buffer cache can fulfill the read/write request.

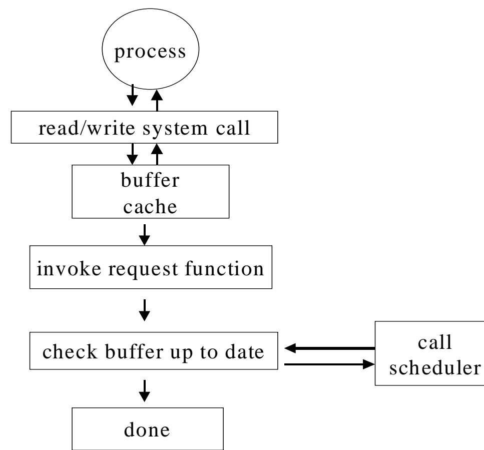

Figure 7: Block device driver read/write flow in Linux.

3.2.2 Network Device Drivers

Unlike block and character devices which have pre-assigned major numbers, network device names represent the type of device that it is. During network device driver initialization, each configured network device driver's initialization routine is called. If a device is found, then it registers with Linux kernel. Thus, the name is dynamically allocated. The name is ordered by found sequence. For example, Ethernet devices are known as eth0, eth1…etc. Unlike block and character devices that have device special files, network devices typically not appear in the file systems since applications use Berkley sockets to send/receive data.

The network device driver's interface service routines are shown in Table 4. The functions may be null functions if the device driver doesn't support that function, but all network device drivers must provide a function to transmit packets, the *hard_start_xmit* function. A more detailed view of the network architecture is shown in
Figure 8. When applications send data through Berkley sockets, these data are passed down layer by layer. Each layer may add its own protocol header. The data structure used in the Linux network packets processing is the *sk_buff* structure, which is designed specially for network packets handling because it is easy to add/remove protocol headers on the structure. When network protocols had added required headers, the *sk_buff* structure is passed to network device drivers and then the driver sends this packet to the network. Almost all network cards are interrupt driven and an interrupt handler is associated with each network device driver. The interrupt handler is used to process events such as transmitting done, receiving a packet and handling error conditions.

3.2.3 Operation Modes

Linux kernel provides three major operation modes for device drivers. They are interrupt, polling and DMA modes. Most of the device drivers are interrupt-driven in Linux and can be further divided into fast interrupt and slow interrupt device drivers. Fast interrupt device driver means when an interrupt occurs, the interrupt handling routine corresponding to that interrupt will run to complete. Other interrupts will be disabled and block there except an explicit interrupt-enable instruction is invoked by the running interrupt handling routine. Slow interrupts mean when executing the interrupt handling routine, this routine may be interrupted by other interrupts having higher priorities. In Linux kernel,

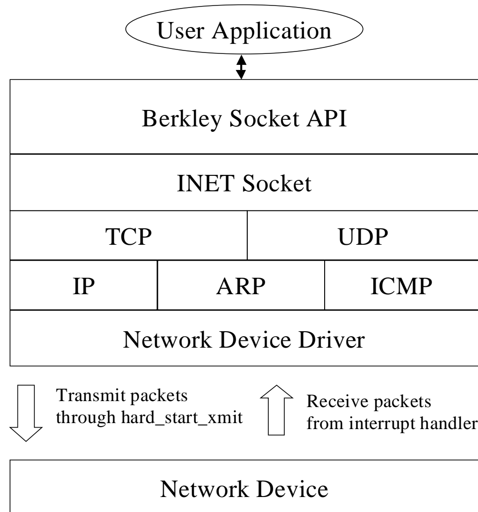

Figure 8: Linux network device driver architecture.

Table 4: Network device driver interface routines.

int	open(struct device *dev)
int	stop(struct device *dev)
int	hard_start_xmit(struct sk_buff *skb, struct device *dev)
int	hard_header (struct sk_buff *skb, struct device *dev, unsigned short type, void *daddr, void *saddr, unsigned len)
int	rebuild_header(void *eth, struct device *dev, unsigned long raddr, struct sk_buff *skb)
void	set_multicast_list(struct device *dev)
int	set_mac_address(struct device *dev, void *addr)
int	do_ioctl(struct device *dev, struct ifreq *ifr, int cmd);
int	set_config(struct device *dev, struct ifmap *map)
void	header_cache_bind(struct hh_cache **hhp, struct device *dev, unsigned short htype, __u32 daddr)
void	header_cache_update(struct hh_cache *hh, struct device *dev, unsigned char * haddr)
int	change_mtu(struct device *dev, int new_mtu)
Struct iw_statistics* get_wireless_stats(struct device *dev)	

disk device drivers are classified as fast interrupt device drivers, while the network device drivers are classified as slow interrupt device drivers.

Fast Interrupt

We take a piece of code from Linux to illustrate how fast interrupts work. The source codes are shown in Figure 9. For simplicity, single CPU version is explained. From line 2 and line 3, register and flag values are saved that may be used in the interrupt handling routine and then a signal is sent to block

further interrupts of this IRQ. In Line 4, a variable named *intr_count* is increased by one. This variable is used to record the depth of the nested interrupt invocations. Because *intr_count* is a shared variable, in order to protect it from concurrent access, the interrupts should be explicitly disabled before accessing its value. In line 5, the interrupt number value of this interrupt handling routine is pushed onto the stack and the actual interrupt handling routine is called in line 6. As this interrupt handling routine is done, interrupts associated with this IRQ could happen again. This is done in line 9 to unblock the interrupt invocation. The variable *intr_count* is then decremented by one. And the saved register and flag values are restored.

Slow Interrupt

The handling codes for slow interrupts are almost the same as those for fast interrupts. A portion of codes is listed in Figure 10. The major difference is in line 6, a *sti* instruction is executed to enable higher priority interrupts to occur before executing the interrupt handling routine. Note that the same interrupt number is still blocked. Another difference is at the last line, after slow interrupts, return-from-system-calls handling codes are executed. This will be explained in bottom-half handling.

Polling mode device drivers are used mostly in slow devices. When commands are sent to devices, the processor will continue to check device's status to see if I/O is complete. Because device drivers are part of the kernel, polling usually slow down the entire system. The processor power is wasted on busy-waiting.

```
2   SAVE_MOST \
3   ACK_##chip(mask,(nr&7)) \
4   "incl "SYMBOL_NAME_STR(intr_count)"\n\t" \
5   "pushl $" #nr "\n\t" \
6   "call "SYMBOL_NAME_STR(do_fast_IRQ)"\n\t" \
7   "addl $4,%esp\n\t" \
8   "cli\n\t" \
9   UNBLK_##chip(mask) \
10  "decl "SYMBOL_NAME_STR(intr_count)"\n\t" \
11  RESTORE_MOST
```

Figure 9: Fast interrupt handling.

```
1  SYMBOL_NAME_STR(IRQ) #nr "_interrupt:\n\t" \
2  "pushl $-"#nr"-2\n\t" \
3  SAVE_ALL \
4  ACK_##chip(mask,(nr&7)) \
5  "incl "SYMBOL_NAME_STR(intr_count)"\n\t"\
6  "sti\n\t" \
7  "movl %esp,%ebx\n\t" \
8  "pushl %ebx\n\t" \
9  "pushl $" #nr "\n\t" \
10 "call "SYMBOL_NAME_STR(do_IRQ)"\n\t" \
11 "addl $8,%esp\n\t" \
12 "cli\n\t" \
13 UNBLK_##chip(mask) \
14 "decl "SYMBOL_NAME_STR(intr_count)"\n\t" \
15 "jmp ret_from_sys_call\n"
```

Figure 10: Slow interrupt handling.

DMA mode device drivers are best suited for multi-tasking operating systems since the CPU doesn’t involve in the actual data transfer, the CPU just sets up the transfer parameters. Most DMA mode device drivers use interrupts to signal the completion of data transfer. This is similar to the interrupt mode device drivers except that a special chip does the actual DMA data transfer. For the reason of PC compatibility, the memory that is DMAable has three constraints. First, the physical address should be below 16MB. Second, the memory used in DMA can’t across the 64KB boundary. Third, the memory should physically continuous. In modern PCI (Peripheral Component Interconnect) bus architecture, however, many constrains are removed.

3.2.4 Bottom Half Handling

Bottom half handling is a mechanism that defers things to be done at a later time. Slow interrupts are often combined with bottom halves to finish the required job together. The main idea is that in the interrupt handling routine, we do as few works as possible and defer other parts, the bottom halves, to a later time. In this way, interrupts are blocked for a minimum time for devices of the same or lower IRQ. A typical use of slow interrupts and bottom-halves is that in the interrupt handler routine, the jobs to be done are queued and the bottom half flag is set. This flag will be check just after the interrupt handling

routine has done. Network packet handling and timer interrupt is suitable for this scheme. When a network packet is received, the packet is queued and a flag is set. The interrupt is enabled while processing the packet. In this way, we may use fewer packets due to blocked interrupts. In the timer interrupt case, we update one global variable and also set a flag. The bottom half then does more things such as time quantum calculation and timer list manipulation.

The implementation of bottom half handling is described. An unsigned integer variable *bh_active* is used to store the requests of bottom halves and there are thirty-two priorities of bottom halves as illustrated in Figure 11 for an example. Each bit in this value indicates whether there are pending bottom halves to be serviced. It is checked before returning from slow interrupts and before returning from system calls.

A piece of code taken from Linux is illustrated in Figure 12. First, *intr_count* value is compared with zero. If it is, this means no interrupt handling routines are running and it is time to invoke bottom-half handling routines if any. Before bottom halves are handled, *intr_count* is increased by one. This makes the invoking of bottom-half handling routines atomic. The bottom halves are handled according to the priorities and are serviced in turn. When all the bottom halves are handled, the variable is checked again to see if any bottom half flag is set. If there are no bottom halves left, it returns.

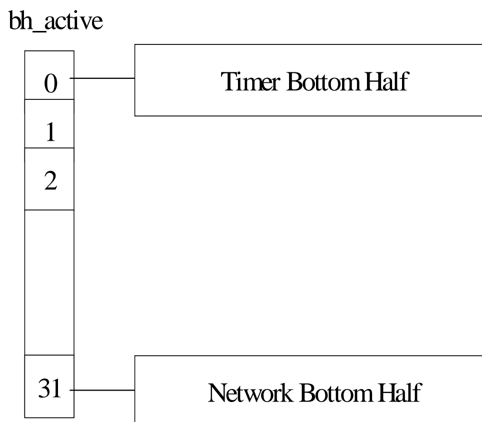

Figure 11: An example for assigned bottom-half priorities.

```
handle_bottom_half:
        incl SYMBOL_NAME(intr_count)
        call SYMBOL_NAME(do_bottom_half)
        decl SYMBOL_NAME(intr_count)
        jmp 9f
        .globl ret_from_sys_call
ret_from_sys_call:
        cmpl $0,SYMBOL_NAME(intr_count)
        jne 2f
9:      movl SYMBOL_NAME(bh_mask),%eax
        andl SYMBOL_NAME(bh_active),%eax
        jne handle_bottom_half
```

Figure 12: Bottom half handling.

3.3 Wrapper-Socket Design and Implementation

Our work has two main goals in designing the middle-ware for incorporating Linux device drivers. First, we need a middle-ware that can make reuse of all kinds of device drivers implemented in Linux. By reusing these device drivers, our work provides as many device drivers as Linux. Second, since new devices come to this world quickly and many people write device drivers for Linux, we need a middle-ware that can make reuse of future version device drivers in Linux. Once new device drivers are released, by applying a quick patch or middle-ware enhancement, new devices can be used in our framework, too. Based on these goals, we decide to incorporate the Linux device driver sources without modifying. The middle-ware is termed as *wrapper-socket* in this report.

There are two main considerations in designing such a wrapper-socket. One is compiling consideration and the other is semantic consideration. Compiling unmodified Linux device driver sources needs all the variables and functions referenced present. These functions and variables may be dummy functions or variables. For example, the block device driver needs a function named *block_read*, which is the buffer cache read function. We don't use this function in our framework but it is referenced in the device driver sources. In such case, we need to create a dummy function just for compiling. For semantic considerations, we need to provide the same executing environment as Linux in order to make use of unmodified Linux device drivers. Linux is a general-purpose operating system and is developed by many people in this world. It has many magic numbers and inter-dependency in

source codes. Hence, there is no clear definition for what functions and variables that can be used by Linux device drivers. The wrapper-socket must provide all these support as Linux kernel.

We first describe the wrapper socket exported interface. Problems and our solutions to incorporate Linux device drives are then discussed. We then illustrate the implementations for incorporating Linux block device drivers and network device drivers.

3.3.1 Wrapper-Socket Exported Interface

The wrapper socket exported interface is listed in Table 5 and a typical use of the interface is shown in Figure 13. The *init()* function is called during system initialization. It is used to set up device-driver specific data structures, probe for devices, register interrupt handling routines and request I/O regions. The *open()* function is called when I/O request for this device is expected. If the *open()* function returns a success value, the I/O services can be operated by *read()/write()* functions. A device driver may implement any access controls in *open()* function if the device driver can handle just one request at a time. This function could always return success if the device driver could queue any number of requests. When an I/O service finishes, the *close()* function is invoked. The *ioctl()* function here is device-specific. It is used to pass special commands to devices.

Table 5: The wrapper-socket exported interface.

void	init (kdev_t dev, int * retval)
void	open (kdev_t dev, int * retval)
void	close (kdev_t dev, int * retval)
void	read (kdev_t dev, void * parms, int * retval)
void	write (kdev_t dev, void * parms, int * retval)
void	ioctl (kdev_t dev, void * parms, int * retval)

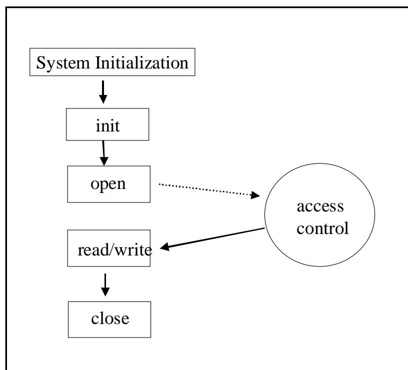

Figure 13: Typical use of the wrapper-socket interface.

As compared to the Linux device driver's interface. The *file_operations* interface is integrated with file systems. The wrapper-socket is designed for device driver layer. The advantage of Linux *file_operations* interface is that the interface is suitable for VFS (Virtual File System Switch) integration. VFS is a mechanism to mount many file systems simultaneously and abstracts everything as files. The advantage of the wrapper-socket interface is that it is designed for device driver layer such that it is more independent to other parts. The interface is uniform for all the types of device drivers. It is easier to develop a native device driver or replace existing ones with custom-designed device drivers.

3.3.2 Emulation for Linux Provided Environment

The wrapper-socket needs to provide Linux compatible mechanisms in order to incorporate unmodified Linux device driver sources. We first describe functions that are needed and supported by other system components and then describe problems that need to be considered and our solutions in designing the wrapper-socket.

Needed Support

An executing unit management component is required. We take a thread as the basic executing unit for example. First, the wrapper-socket needs a function to return the identity of the current executing thread. Second, in the design of incorporating Linux block device drivers, the thread sleeps if an I/O request is not yet complete. Hence, the wrapper-socket needs a function to put a thread to sleep. Third, a function to wake up specified thread is also needed. A memory management component is required. The wrapper-socket needs a function that does memory allocation and a function to free memory. The only constraint for the allocated memory is that the allocated memory must be physically continuous. A console management component is required. The wrapper-socket needs a function that prints characters onto the console. An interrupt management component is required. There are three functions needed by the wrapper-socket. First, the wrapper-socket needs a function to register interrupt handling routines. When an interrupt occurs, the interrupt management component calls the registered routine directly. The wrapper-socket implements mechanisms for handling fast or slow interrupts. Second, a function to de-register interrupt handling routines is needed. Third, the wrapper-socket needs to know currently allocated IRQs of registered interrupt handling routines. This is used in Linux device driver automatic IRQ detection. Finally, the wrapper-socket needs support from time management component. A

variable *jiffies* is maintained by this component. The component also needs to provide timer functions registration/de-registration. A timer function means a function that will be invoked at a specified time. A scheduling management component that can set priority of a thread is also required.

Problems and Our Solutions

A global variable *jiffies* is maintained by the Linux kernel and is extensively used by Linux device drivers. This variable is initialized to zero on system startup and continues to increment by one every ten milliseconds. This variable is used to provide a global view of the current time. Linux device drivers use this variable to implement functions such as timeout detection. We solve the problem by maintaining this shared variable by other system components such as time management components. Another time-related variable *loops_per_second* records how many *decrement* instructions this machine can do in a second. This value is calculated by the wrapper-socket on system startup.

Linux device drivers register functions to be executed at a later time by calling a timer registration functions. The parameters to this timer function are the function address and a timeout value. We provide these functions by designing a timer management component. The implementation of this component is to maintain a linear list of timeout values in ascending order. On each timer interrupt, this list is checked to see if any timeout occurs. If there is one, the function registered is invoked. Functions for deleting registered timer functions must also be provided.

Linux is a multi-tasking and multi-user operating system. User applications run in user mode and device drivers run in kernel mode. In order to move data between user space and kernel space, device drivers call data movement routines. These address translation and data movement routines must be provided. Some I/O control functions in Linux device drivers check user permission and usually needs the highest privilege to invoke the function. If an information-appliance doesn't need this check, the implementation could always return success in such case. Device drivers read and write memory regions from user space. Before the memory region is read or written, device drivers may check the memory region properties. Again, if an information-appliance doesn't need this check, a success is always returned.

The wrapper-socket provides functions for Linux interrupt-driven device drivers to register/de-register interrupt handling routines based on the interrupt management component. Since Linux interrupt-driven device drivers are classified into fast and slow interrupts, the wrapper-socket must provide mechanisms to support these two kinds of interrupts. The wrapper-socket uses the services

provided by C-Interrupt core component and implements these supporting functions for fast and slow interrupts. Interrupts are disabled when calling fast interrupt handling routines and enabled for slow interrupt handling routines. On return of slow interrupts, the wrapper-socket checks whether there are pending bottom halves. The wrapper-socket calls these functions if any flag is set. Since the timer interrupt in Linux is also a slow interrupt, we added bottom halves checking in the timer interrupt. The wrapper-socket also provides routines for automatic IRQ detection for Linux device drivers to automatically detect device's interrupt number.

I/O ports generally are used for communication between CPU and devices. Before using some device-specific I/O regions, Linux device drivers must register the range of I/O regions this device uses. The wrapper-socket provides I/O region management for Linux device drivers.

Linux is a monolithic kernel. When it is executing in the kernel mode, it has full control of the codes to run. This means that when executing in device drivers, except the device driver explicitly call the scheduling function, the device driver will run without preemption. The wrapper-socket must provide a mechanism to ensure the correctness of executing Linux device drier codes. The simplest way is to provide the same semantics as Linux. When executing in Linux device drivers, the wrapper-socket set a flag. The scheduling component must be aware of the existence of this flag. If this flag is set, no preemption should be taken. If the scheduler is not aware of this flag and schedules in the midway of device drivers, error conditions may occur. For example, if commands are sent to devices on the way and other I/O requests are accepted. Another command may be sent to the same device. The behavior of this device is unpredictable since commands may need to be issued in a predefined order. The preempted I/O request in progress doesn't know this since Linux device drivers assume full control over the execution order. Unpredictable behavior may also occur for this preempted I/O request.

3.3.3 Wrapper-Socket for Linux Block Device Drivers

The *init()* function for block device drivers consists mainly three steps. First, some data structures are initialized. These include an array to record device driver request functions and a linked list of all general disk data structures. The general disk data structure is used to record disk information such as disk partition information, disk block size and disk size. Second, the initialization routine of each Linux device driver is invoked. The function of this routine typically consists probing for devices, detecting device's IRQ, requesting I/O regions, adding general disk data structures and setting up some device-

specific data structures. Third, the wrapper-socket records some block device information such as device sizes.

There are no access control mechanisms in *open()/close()* functions since the wrapper-socket queues all requests and services them in turn.

No buffer cache is implemented in the wrapper-socket. Hence, any I/O requests passed to the *read()/write()* functions will cause actual data transfer from/to devices. Two data structures are used in the internal representation of an I/O requests, the *reqeuest* and *buffer_head* structure. A *request* structure may contain many *buffer_head* structures. These *buffer_head* structures are linked together and one *buffer_head* structure contains one block size data. An example is shown in Figure 14. The main work done in *read()/write()* functions is to fill adequate values in *request/buffer_head* structure and then calls the request function of block device drivers if no current request is processing. If there are requests queued already, the wrapper-socket just adds this request to the end of list. The wrapper-socket then waits for I/O completion and may go to sleep. When actual data transfer is done, the one that requests for I/O is waked up. Figure 15 illustrates this for information-appliances that use threads as executing units.

There are currently three I/O control commands defined in the *ioctl()* function for block device driver. One to set block size, one to get block size and one to get number of blocks of a device.

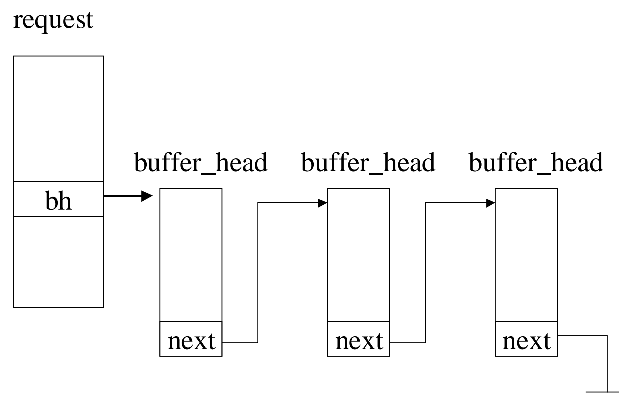

Figure 14: An example for *request* and *buffer_head* structures relationship.

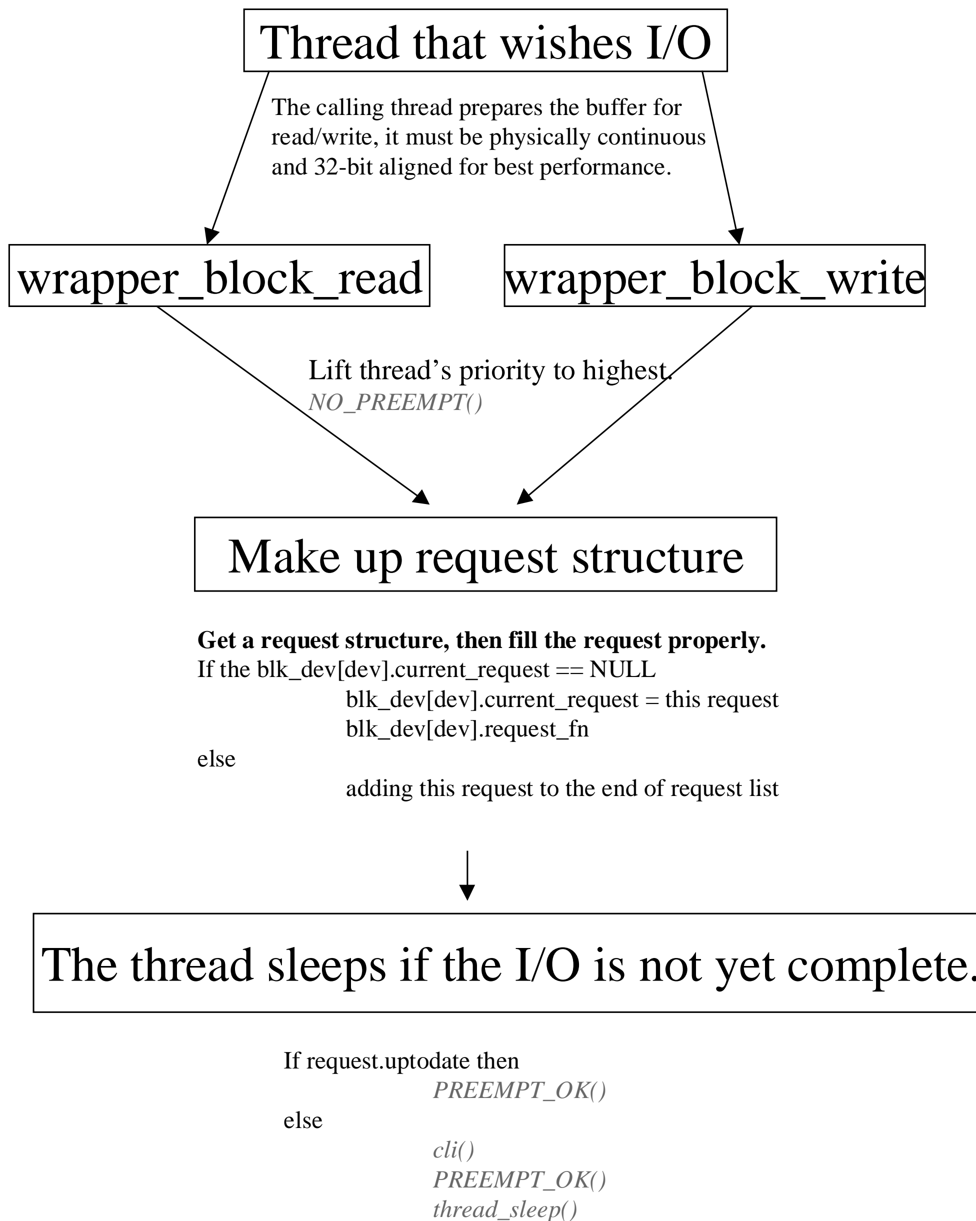

Figure 15: The wrapper-socket read/write design for block device driver.

3.3.4 Wrapper-Socket for Linux Network Device Drivers

The *init()* function has three steps. First, a queue for receiving network packets is initialized. Second, the wrapper-socket calls the initialization and device-open routines of Linux network device drivers. This initialization routine is used to probe for network cards, register I/O regions and register interrupt handling routines. The device-open routine does something needed to bring the network card up. The wrapper-socket then initializes the network packet transmitting queues for each device found. Third, the network bottom half handler is initialized.

No access control mechanisms are implemented in the *open()/close()* functions since the wrapper-socket queues these requests.

read()/write() functions are designed as asynchronous I/O. In the *read()* case, the wrapper-socket maintains a queue for received network packets. The *read()* function will return the first packet in the queue if there exists one or more packets in the queue. The *read()* function returns an error value if no packets are queued. In the *write()* case, the wrapper-socket first allocates a *sk_buff* structure and fills required fields of the structure such as packet length. The wrapper-socket will set the field *arp* of *sk_buff* structure to one in order to prevent invoking of the address resolution protocol. The caller must provide a buffer that contains data to be transmitted, which already contains link layer MAC (Media Access Control) addresses. The wrapper-socket then checks whether the transmitting queue is too long. If the queue exceeds some limit value, the packet is dropped. If there exists room for the packet, the wrapper-socket calls Linux device driver's transmitting routine, the *hard_start_xmit*.

The I/O control function currently has a command that gets the MAC address of the network card. This value is typically filled in the header of transmitting buffers.

4. Performance Evaluations

We have built a small multi-threading kernel and the wrapper-socket component in order to demonstrate the functionality of the system. Several aspects of this framework are evaluated. First, we evaluate the performance of the wrapper-socket plus Linux device drivers. The bandwidth and latency of Linux device drivers are of interest. Second, we perform a test on interrupt latency and an experiment for calculating scheduling cost is performed. Finally, the code size for the wrapper-socket is presented. Table 6 lists the experimental environment.

Table 6: Experimental environment.

CPU	Intel Pentium 100MHz
Memory	16Mb DRAM
Storage	Quantum Maverick 540Mb
Network	3com 3c509
Mainboard	ABIT PX5 with 512k secondary cache

4.1 Device Driver Bandwidth

Currently, the wrapper-socket supports Linux block device drivers and network device drivers. We take IDE device driver for example and evaluate the bandwidth of IDE disk. We don't measure the bandwidth of network cards since the network driver implementation in Linux is asynchronous., and each packet received in the device driver is not guaranteed to be transmitted. The reason is that each network device driver maintains a send queue for each network card and maximum, Ethernet for example, 100 packets can be queued simultaneously. Once the queue is full, the packets sent to the device driver are discarded without warning. The implementation of the wrapper-socket is asynchronous such that it also suffers from this problem, so we don't measure the bandwidth of network cards.

To evaluate the bandwidth of IDE disks, we use a thread that reads physically continuous 20MB data sequentially in our test platform. The read/write block size is ranged from 512 bytes to 65536 bytes. In Linux, we opened /dev/hdb file for access and the read/write unit size is also ranged from 512 bytes to 65536 bytes. The read bandwidth results are shown in Figure 16. In our test platform, the disk bandwidth increases with the block size. The reason is that with a larger block size, the Linux device driver issues fewer commands to the disk drive. Issuing fewer commands to the disk drive saves commands latency and disk heads seek/rotational time. As can be seen from the figure, the bandwidth is competitive with Linux.

The disk write bandwidth is shown in Figure 17. In our test platform, the bandwidth increases with block size. The reason is the same with the case in disk read. In Linux, due to the effects of buffer cache that disk writes may be collected and issued together, the bandwidth does not increase with block sizes.

We opened the /dev/hdb file with *O_SYNC* flag set, which means synchronous write, and do the experiment again. The results are shown in Figure 18. We notice that our test platform performs better

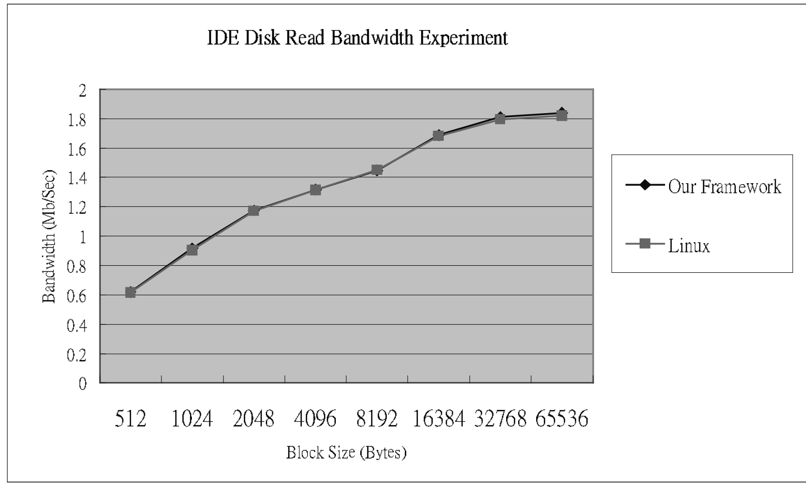

Figure 16: IDE disk read bandwidth experiment.

than Linux in the last few block sizes. The reason is in Linux, user/kernel memory copy for the request data is needed and in our test platform, no memory copy is needed. In Linux, there are system call overheads. In our test platform, since user application resides in the same address space with system components, there is no such overhead.

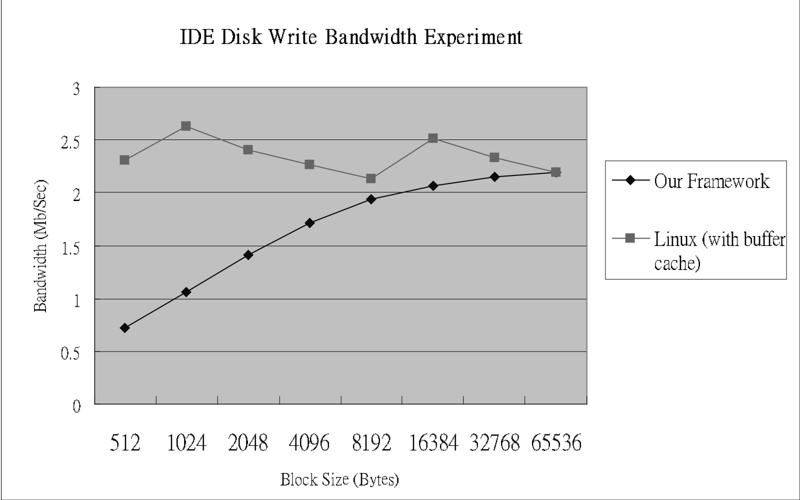

Figure17: IDE disk write bandwidth experiment 1.

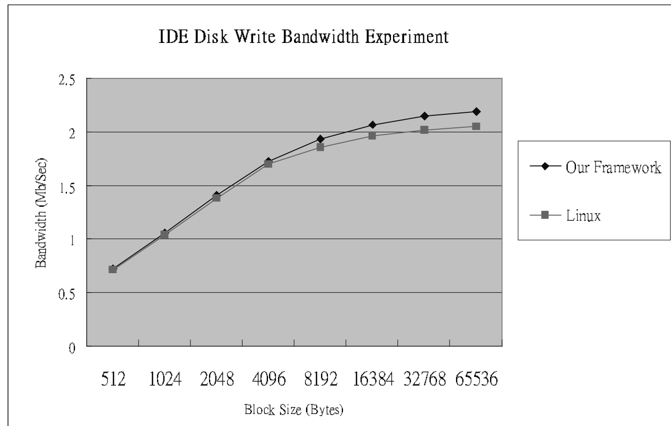

Figure 18: IDE disk write bandwidth experiment 2.

4.2 Device Driver Latency

The latency time of IDE device driver and network device drivers is measured. In the IDE case, the time for serving an I/O request is split into wrapper-socket, device driver, actual I/O data transferred and sleep/wakeup time as shown in Figure 19. The wrapper-socket latency time mainly comes from constituting of data *request* and *buffer_head* structures that are needed in the device drivers. The device driver part is the time to handle these data structures plus processing time in the interrupt handling routine. Actual I/O data transfer is the time between commands are issued and data returned. We don't count actual I/O data transfer time into device driver latency since this depends on the power of physical drive and the location of requested data. We use a block size of 512 bytes in the 512-byte case and 1024 bytes block size in other cases and the measured time unit is microsecond.

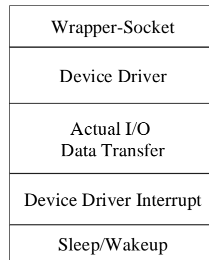

Figure 19: Split time for block device driver accessing.

Table 7: IDE device-driver read/write latency time (in microseconds).

read/write size (bytes)	wrapper-socket time for read	device driver time for read	wrapper-socket time for write	device driver time for write
512	4.98	17.71	5.38	16.77
1024	5.1	21.02	5.47	19.37
2048	6.66	28.82	7.2	25.71
4096	10.13	42.94	10.72	38.95
8192	18.75	71.25	20.89	64.76
16384	40.14	128.11	47.15	116.55
32768	104.57	241.19	123.23	220.27

As illustrated in Table 7, the processing time generally increases with read/write size since with larger requested data size, more data structures must be allocated and processed. The sleep/wakeup time is not shown in the table. Each I/O request needs at most one sleep plus one wakeup time. In our test platform, one sleep plus one wakeup takes about 6.59 microseconds.

The split latency time for network packet sending/receiving can be illustrated in Figure 20 ant the measured latency time in Table 8. In the network sending case, the latency time for wrapper-socket comes mainly from allocating a *sk_buff* data structure and time for coping data from application to the wrapper-socket. The wrapper-socket processing time increases with the size of data. After the wrapper-socket hands this *sk_buff* structure to the network device driver, the device driver outputs this packet and then frees the *sk_buff* memory. The processing time for the device driver is almost the same for all sizes of packet since we don't count the actual I/O transfer time such that the time comes mainly from the freeing of *sk_buff* structure time. In the network receiving case, the time for device driver mainly consists of allocating a *sk_buff* data structure and the wrapper-socket just queues this packet. The processing time for the device driver and wrapper-socket almost has the same value for all packets since actions taken by different sizes of packets are almost the same. Again, actual I/O transfer time is not counted.

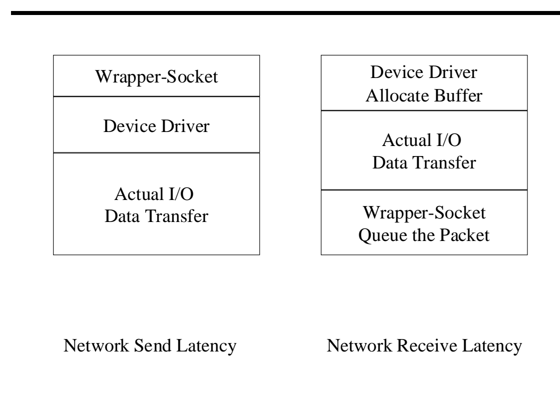

Figure 20: Network send/receive latency time.

4.3 Interrupt Latency and Scheduling Cost

The C-Interrupt component provides routines to register and de-register interrupt handling routines. To get the minimum latency imposed by this component, we measured the null interrupt latency. We register a function that does nothing in the function body with the C-Interrupt component. To measure the latency time, we use software interrupts to invoke the C-Interrupt component and this null function. We measured one hundred times and the average time for null interrupt latency is 1.32 microseconds. For comparison, we also measured null interrupt latency in Linux. We install a kernel module into Linux kernel and this module registers a null function with Linux as the interrupt handling routine. We then write a system call to generate software interrupts such that the interrupt handling routine is invoked. The measured latency time is 4.2 microseconds for a fast interrupt handling routine and 4.75 microseconds for a slow interrupt handling routine.

Scheduling cost is also an important factor in designing an operating system. In our scheduling component, a timer interrupt handling routine is used for priority recalculation. We measured the time between the beginning of timer interrupt and the time when another thread, which is calculated in the timer interrupt to be the next running thread, begins its execution. This value is the scheduling cost. The average time in our test platform is 6.5 microseconds. For comparison, we also measured Linux scheduling cost. The measured scheduling cost is about 14 microseconds.

Table 8: Latency time for network packet sending/receiving (in microseconds).

Packet size (bytes)	wrapper-socket time (send)	device driver time (send)	wrapper-socket time (receive)	device driver time (receive)
100	6.81	2.19	0.79	3.74
200	7.26	2.16	0.79	3.72
300	8.86	2.1	0.79	3.73
400	9.96	2.11	0.79	3.90
500	10.54	2.1	0.79	3.91
600	12.22	2.1	0.79	3.90
700	12.48	2.1	0.79	3.89
800	12.66	2.09	0.79	3.89
900	15.21	2.11	0.79	4.29
1000	15.52	2.1	0.79	4.30
1100	17.72	2.1	0.79	4.29
1200	18.21	2.1	0.79	4.33
1300	18.7	2.1	0.79	4.29
1400	19.26	2.1	0.79	4.29
1500	19.51	2.1	0.79	4.34

4.4 Wrapper-Socket Code Size

The code size of the wrapper-socket is shown in Table 9.

Table 9: The wrapper-socket code size.

Description	Line of Codes	Object Code Size (KB)
IDE Disk Driver	700	9.5
Network Driver	1600	14
IRQ Management	500	12
Initialization	100	2.7
Misc.	420	4

5. Concluding Remarks

5.1 Summary

A special designed system can meet more application's needs. This generally results in better performance and saves unnecessary cost. However, designing such a system requires several system components to be built. In this report, we design and implement a framework to speedup information-appliances buildup for Intel X86 compatible PCs.

The design of the framework includes modularized software components for low-level x86 control software and a wrapper-socket for incorporating existing almost unmodified Linux device driver sources. With the help of this framework, versatile information-appliances can be built easily without paying a lot of efforts in system implementations and customization. Designers of any information-appliance can easily and fast explore their ideas by the help of the wrapper-socket and Linux device driver source codes. This means they do not need to develop bundle device driver codes before testing a small idea.

The exported device driver interface is clear, system designers can design their customized components if the support software cannot meet their requirements or they want further improvements. Even so, the supported components are still optimized enough and efficient for almost applications.

Empirical evaluations show that information-appliances built via this framework perform well. The framework is a practical way to build versatile special purpose systems.

5.2 Future Work

The future work to extend our work is described. First, the wrapper-socket currently supports block and network device drivers. Support for more Linux device drivers in our wrapper-socket is expected in the future, e.g. sound device drivers. Second, we expect an integrated developing environment to be built. Most of all, we need to build traditional source-level debuggers and debuggers for verifying real-time timings in our framework. Third, real-time scheduling algorithms would be added in our framework. With these real-time scheduling algorithms, we can explore the impacts to multimedia applications and quality of service management. Finally, since system software is divided into modular components, a detail system modeling and performance prediction can be achieved. Another goal is to have a total system modeling based on system engineering and theoretical analysis [Katcher 93].

References

[Accetta 86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, 'Mach: A New Kernel Foundation for UNIX Development', *Proceedings of the Summer 1986 USENIX Conference*, Atlanta, GA, July 1986, pp. 93-112.

[Auslander 97] Marc Auslander, Hubertus Franke, Ben Gamsa, Orran Krieger and Michael Stumm, 'Customization Lite', *Proceedings of the 6th Workshop on Hot Topics in Operating Systems(HotOS-VI)*, Cape Cod, MA, USA, May 1997, pp. 43-48.

[Barabanov 97] Michael Barabanov, 'A Linux-based Real-Time Operating System', Master Thesis, New Mexico Institute of Mining and Technology, June 1997.

[Beck 96] Michael Beck, Harald Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus and Dirk Verworner, *Linux Kernel Internals,* Addison-Wesley Publishing Company Inc., September 1996.

[Bershad 95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynsky, D. Becker, C. Chambers and S. Eggers, 'Extensibility, Safety and Performance in the SPIN Operating System', *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995, pp. 267-284.

[Challab 87] D. J. Challab and J. D. Roberts, 'Buddy Algorithms', *The Computer Journal*, 30, (4), 308-315(1987).

[Chorus] Chorus Corporation, 'CHORUS/ClassiX Release 3 Technical Overview', CS/TR-96-119.13, June 1997.

[Crawford 87] John H. Crawford and Patrick P. Gelsinger, *Programming the 80386,* SYBEX Inc., 1987.

[Druschel 97] Peter Druschel, Vivek S. Pai and Willy Zwaenepoel, 'Extensible Systems are Leading OS Research Astray', *Proceedings of the 6th Workshop on Hot Topics in Operating Systems(HotOS-VI)*, Cape Cod, MA, May 1997, pp. 38-42.

[Eagler 95] D. R. Eagler, M. F. Kaashoek and J. O'Toole Jr, 'Exokernel: An Operating Systgem Architecture for Application-Level Resource Management', *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995, pp. 251-266.

[Ferraro 90] Richard F. Ferraro, *Programmer's Guide to the EGA and VGA Cards*, 2nd edition, Addison-Wesley Publishing Company Inc., 1990.

[Ford 97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin and Olin Shivers, 'The Flux OSKit: A Substrate for Kernel and Language Research', *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[Gilluwe 97] Frank van Gilluwe, *The Undocumented PC: a programmer's guide to I/O, CPUs, and fixed memory areas,* 2nd edition, Addison-Wesley Developers Press, 1997.

[Goel 96] Shantanu Goel and Dan Duchamp, 'Linux Device Driver Emulation in Mach', *Proceedings of the Annual USENIX 1996 Technical Conference*, San Diego, CA, USA, January 1996, pp. 65-73.

[Intel] Intel Corporation, 'Intel Architecture Software Developer's Manual,Volume 3: System Programming Guide', http://www.intel.com/design/pentium/manuals/.

[Johnson] Michael K. Johnson, 'Linux Kernel Hackers' Guide', http://www.redhat.com:8080/HyperNews/get/khg.html/, Red Hat Software, 1998.

[Katcher 93] D. Katcher, H. Arakawa and J.K. Strosnider, 'Engineering and Analysis of Fixed Priority Schedulers', *IEEE Transactions on Software Engineering*, 19, (9), 920-934(September 1993).

[Krieger 97] Orran Krieger, 'HFS: A Performance-Oriented Flexible File System Based on Building-Block Compositions', *ACM Transactions on Computer Systems*, 15, (3), 286-321(August 1997).

[Labrosse 93] Jean J. Labrosse, *uC/OS – The Real Time Kernel*, R&D Publications, 1993.

[Labrosse 95] Jean J. Labrosse, *Embedded Systems Building Blocks*, R&D Publications, 1995.

[Lee 94] Paul C. H. Lee, Meng Chang Chen and Ruei-Chuan Chang, 'Hipec: High Performance External Virtual Memory Caching', *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI'94)*, Monterey, California, USA, 1994, pp.153-154.

[Lee 96a] Paul C. H. Lee, Meng Chang Chen and Ruei-Chuan Chang, 'Hipec: A System for Application-Customized Virtual-Memory Caching Management', *Software: Practice and Experiences*, 27, (5), 547-572 (1996).

[Lee 96b] Paul C.H. Lee, Meng Chang Chen and Ruei-Chuan Chang, 'Application-Controlled Virtual Memory Caching Policies', *Proceedings of the 1996 International Conference on Computer Systems Technology for Industrial Applications (CSIA'96)*, Hsinchu, Taiwan, ROC, 1996, pp. 235-242.

[LDP] 'The Linux Documentation Project', http://sunsite.oit.unc.edu/mdw/linux.html/.

[Mathisen 94] Terje Mathisen, 'Pentium Secrets', *Byte*, pp. 191-192, July 1994.

[Mendelsohn 97] Noah Mendelsohn, 'Operating Systems for Component Software Environments', *Proceedings of the 6th Workshop on Hot Topics in Operating Systems(HotOS-VI)*, Cape Cod, MA, May 1997, pp. 49-54

[Messmer 95] Hans-Peter Messmer, *The Indispensable Hardware Book,* Addison-Wesley Publishing Company Inc., 1995.

[Rajkumar 98] Raj Rajkumar, Kanaka Juvva, Anastasio Molano and Shui Oikawa, 'Resource Kernels: A Resource-Centric Approach to Real-Time Systems', *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, San Jose, CA, January 1998.

[SPEC 96] 'SPEC', *SPEC newsletter*, Manassas, VA, Second Quarter, 1996, http://www.specbench.org/osg/sfs93/results.

[SPEC 97] 'SPEC', *SPEC newsletter,* Manassas, VA, Third Quarter, 1997, http://www.specbench.org/osg/sfs93/results.

[Tanenbaum 92] Andrew S. Tanenbaum, *Modern Operating Systems*, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1992.

[Vahalia 96] Uresh Vahalia, *Unix Internals: The New Frontiers*, Prentice Hall Inc., Englewood Cliffs, New Jersey, January 1996.