1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

REUSE LINUX DEVICE DRIVERS IN EMBEDDED SYSTEMS[1]

Chi-Wei Yang[+], *Paul C. H. Lee*[£], *and Ruei-Chuan Chang*[+£]

Department of Computer and Information Science[+]
National Chiao-Tung University, Hsinchu, Taiwan, R.O.C.
Email: chiwei@os.nctu.edu.tw, rc@cc.nctu.edu.tw

Institute of Information Science[£]
Academia Sinica, Nankang, Taiwan, R.O.C.
Email: paul@iis.sinica.edu.tw

ABSTRACT

Device driver is one component that is usually ignored by research community in operating systems. Their design mechanisms, accessing semantics, operating behaviors and runtime performance are crucial to operating system architecture and total system performance. In this paper, an I/O package to reuse Linux device drivers in embedded system is introduced. Via this package, the whole Linux device-driver source tree can be reused without any modifications. The motivations why to do this work and the detailed design and implementation issues are addressed. This I/O package was also quantitatively evaluated. The empirical results show that the incorporated drivers' performance is comparable with those operated under Linux, and those native drivers under *Vega* kernel.

1. INTRODUCTION

For those who work with operating systems, they know that their biggest troubles mostly come from the fact that there always exist devices waiting for new drivers. And so do the existing drivers waiting for maintenance. Only the commercial systems can get supports from device venders in supplying and maintaining device drivers. For academic systems, driver writing is always a major burden in system researches, because academic community neither has enough human resources in developing device drivers nor various proprietary device specifications for driver writing [Lee 95]. For running any system projects and experiments, it is important to find a way to escape from the curse of driver writing.

Another phenomenon about device drivers is the long-term ignorance in system researches [Rawson 97]. Since the primary mission of operating system is to manage the hardware resources, the device drivers should get more attentions in system researches. Their software architecture, accessing semantics, operating behaviors and runtime performance will decide the total system performance and operating patterns. To do such kind of researches, however, needs firstly to build a system testbed for further experiments. This kind of environment setup needs a lot of human resources, which is usually beyond the ability of academic research community. It costs too much to develop device drivers just for trying some research ideas, and that is why the device drivers receive little attentions in system researches. This motivates us to build a testbed with bundles of device drivers, which should be separated from complex operating system internal semantics

The solution we suggest here is to reuse shareware device driver codes. In our work, the Linux operating system [Beck 96] is selected as the device driver source pool because it is the biggest shareware pool in the world and all its sources are free and opened to those who are interested. A software package, which is named as the *wrapper-socket* in this paper, is implemented to emulate the original Linux kernel semantics. So that users can reuse the Linux device drivers by accessing the exported interface of that package. We implemented this package in the GNU programming environment and integrated this package into our *Vega* kernel [Lee 98.b] for evaluations and experiments.

The design goals of this package are briefly described as follows. First, this package can incorporate the whole Linux device driver source trees without any modifications. New Linux device drivers and future upgraded drivers can also be easily incorporated into the package without evident porting efforts. Second, this package should perform as well as those drivers operated under Linux, and the native drivers running under *Vega* kernel. Third, this package should isolate the Linux device driver semantics that users can easily use this package and port to their specific kernels without needing to know the Linux kernel internals first.

[1] This work is a part of the *RAMOS* project in IIS, which aims at developing system software for real-time and multimedia applications. *RAMOS* is sponsored by NSC in part under the granted number NSC 88-2213-E-001-016.

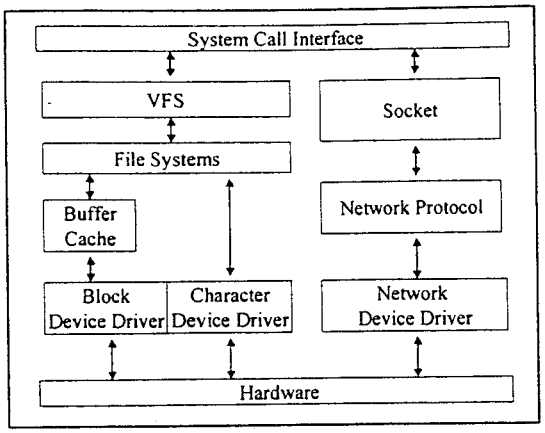

Figure 1: An overview of Linux I/O system.

This paper is organized as follows. Section 2 shows the design and implementation of the wrapper-socket for incorporating Linux device drivers. System evaluations are given in Section 3. Section 4 is about the related work of the world. This paper is concluded in Section 5.

2. DESIGN AND IMPLEMENTATION

In this section, we first briefly describe the Linux device driver architecture, the *Vega* kernel and the overall system overview. Followed are the design approaches and the encountered problems and experiences.

2.1 Linux Device Driver Architecture

The basic Linux I/O system architecture is illustrated in Figure 1. The block, character and network are three main device driver types classified in Linux. Character devices are those which handle data in serialized byte streams. Data handled by character devices does not need to be cached by buffer cache and usually cannot be randomly accessed. Block devices, on the contrary, access data in units of block. Since direct writing or reading to the block devices is costly in just modifying a small portion of block, buffer cache is applied in I/O subsystems to reduce latency time. Usually block devices permit randomly accessing data. Network devices are mainly composed of network cards. Ethernet cards and FDDI adapters are examples.

Character and block devices are abstracted as special files in Linux file systems. For each device, a pair of major and minor number is associated with this device. Major numbers are assigned to different types of devices, and minor numbers are used to distinguish devices of the same type. Character and block device drivers both export the *file_operations* interface. Figure 2 lists this interface. Linux uses two arrays, one for character devices and the other for block devices, to record the address of each device's *file_operations* structure. Major numbers are used as indexes into the array. The interface between buffer cache and block device driver is a request function, which the buffer cache uses to access block devices.

Unlike block and character devices which have pre-

```
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);
```

Figure 2: Linux *file_operations* interface.

assigned major numbers, names of network devices represented the type of device that they are. These names are dynamically allocated. Unlike block and character devices, network devices typically do not appear in the file systems, since applications use Berkley sockets to send/receive data.

The network device driver's interface routines are shown in Figure 3. The *hard_start_xmit* function is used to transmit packets. User applications send data through Berkley socket interface. These data are passed down layer by layer.

```
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 de-
vice *dev, unsigned char * haddr)
int change_mtu(struct device *dev, int new_mtu)
struct iw_statistics* get_wireless_stats (struct device *dev)
```

Figure 3: The network device driver interface.

When network protocols had added all the required headers, the *sk_buff* structure, which is used for packet processing, is passed to network device drivers. Then the drivers send packets to the network. Most network cards are interrupt-driven with an interrupt handler associated with each device driver. The interrupt handler is used to process events such as transmitting done, receiving a packet and handling error conditions.

2.2 Vega Kernel

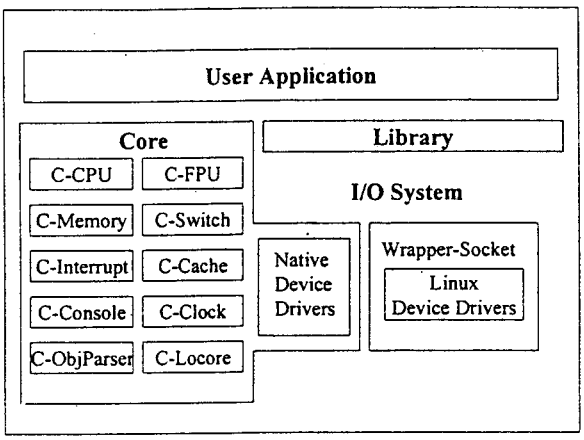

Figure 4: System building blocks.

Vega kernel is the core component of *Lyra* operating system built in IIS, which aims at supplying execution environment for real-time and multimedia applications. *Vega* is a set of well-designed, well-documented and clear-interface kernel-level software components. It is designed to abstract the hardware resources of general computer systems. For example, the clock core component is software that handles the Intel 8259 chip. The advantage of this design is to isolate low-level machine dependent codes from higher-level system semantics.

The functionality of *Vega*, however, is very simple. This means the complex and specific system services are left to the operating system personality, the *Lyra*. Because *Vega* has simple functionality, *Vega* creates little interference to the wrapper-socket package. Figure 4 shows the building blocks of the system.

2.3 Design and Implementation of the Wrapper-Socket

There are two main considerations in designing the wrapper-socket to incorporate unmodified Linux device drivers. One is about compiling and the other is about semantic. To compile unmodified Linux device driver sources needs all the referenced variables and functions present. For example, the block device driver needs a function named *block_read*, which is the buffer cache read function. This function is not necessary to all systems but it is referenced in the device driver sources. In such case, a dummy function should be created for passing the compiling checks.

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 5: The wrapper-socket exported interface.

For semantic considerations, we need to provide the same executing environment as that in Linux in order to emulate and separate the Linux kernel semantics.

2.3.1 Wrapper-Socket Exported Interface

The exported wrapper-socket interface is listed in Figure 5. It exports raw read/write interface. The *init()* function is called during system initialization. It is used to set up driver specific data structures, probe for devices, register interrupt handling routines and request for 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 access controls in the *open()* function. When an I/O service finishes, the *close()* function is invoked. The *ioctl()* function is device-specific. It is used to pass special commands to devices or to do configurations.

2.3.2 Device Name Space Emulation

In Linux, character and block devices appear as special files in the file systems. A special file contains a major and minor number for identifying the devices. In order not to modify the source codes, we use the same assigned major/minor numbers in our system to identify devices. Network devices, however, does not appear in file systems in Linux, hence, no major and minor numbers are assigned for networking devices. Since we prefer a uniform interface for each class of devices, we assign major numbers to network devices according to network types. Thus, they are accessed in the same way as other classes of devices.

2.3.3 Synchronization Semantics Emulation

Different kernel execution semantics are used in Linux and *Vega*. When Linux kernel service is executed, it is not preempted unless it voluntarily relinquishes CPU. In embedded systems, there is no difference for kernel services and applications. This will cause synchronization problems, since Linux device drivers are part of Linux kernel, they assume they have full control over the executing sequence. This means that when executing device driver codes, except the device driver explicitly calls scheduling function, the device driver will run without preemption. We solve this problem by letting the wrapper-socket set a flag before entering Linux device driver codes, either in interrupt handling or in ordinary service execution. The scheduling component is aware of the existence of this flag. If this flag is set, no preemption should be taken. Otherwise, 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.

2.3.4 Emulation for Time, Timer, IRQ and I/O Management

Most time and timer related functions are defined as inline functions or macros in Linux header files. Since we incorporate these header files without modifications, we don't need to pay attentions to these functions. However, there are variables and mechanisms these functions rely on, which our system must provide. 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 per ten-milliseconds. This variable is used to provide a global view of the system time. The wrapper-socket uses the *Vega* time management component to maintain this variable. Another time-related variable, the *loops_per_second*, records how many *decrement* instructions this machine can do in a second. We port the codes from Linux, and this value is calculated by the wrapper-socket before Linux device drivers are initialized in our system.

The emulation of timer functions consists mainly of manipulation a linear list, which is composed of timeout values in ascending order and their corresponding timeout functions to be called. On each timer interrupt, this list is checked to see if any timeout occurs. If there is one, the registered function is invoked. Functions for adding/deleting timer functions are also provided by the wrapper-socket. While maintaining the linear list, interrupts are disabled, such that the modifications of the list entries are atomic.

The wrapper-socket provides functions for Linux interrupt-driven device drivers to register/de-register interrupt handling routines, which are based on the *Vega* interrupt management component. Since Linux interrupt-driven device drivers are classified into fast and slow interrupts [Lee 98.a], the wrapper-socket must provide mechanisms to support these two kinds of 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 for the timer interrupt. The wrapper-socket also provides routines for automatic IRQ detection, to automatically detect the interrupt numbers of devices. The design is based on the allocated IRQ value, which is returned by the *Vega* interrupt core component.

I/O ports are used for communicating 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. We port I/O management routines from Linux to our wrapper-socket.

2.3.5 Emulation for Addressing and Privilege Checking

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. In *Vega*, the kernel segment and user segment is the same, but in Linux the addresses for these two segments are not the same. Hence, these address translation and data movement routines are modified. The memory read/write privilege and user permissions checking are removed since we didn't use this kind of information in our system.

2.3.6 Wrapper-Socket for Linux Block Device Drivers

The *init()* function for block device drivers consists mainly three steps. First, an array to record device-driver request-functions and a linked list of all general disk data structures are initialized. Second, the initialization routine of each Linux device driver is invoked. Finally, the wrapper-socket records some block device information, such as device sizes.

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

I/O reads/writes are synchronous in our design, which is similar to Linux. The wrapper-socket implements these *read()/write()* functions as raw device I/O. The main work done in *read()/write()* functions is to fill adequate values in *request/buffer_head* structures and then calls the request functions of block device drivers if no current request is processing. The *buffer_head* structure represents one block size data, and the *request* structure contains a linked list of *buffer_head* structures. 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 blocks there. When actual data transfer is done, the one that requests for I/O is waked up.

2.3.7 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. Third, the network bottom half handler is initialized. No access control mechanisms are implemented in the *open()/close()* functions because 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. 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

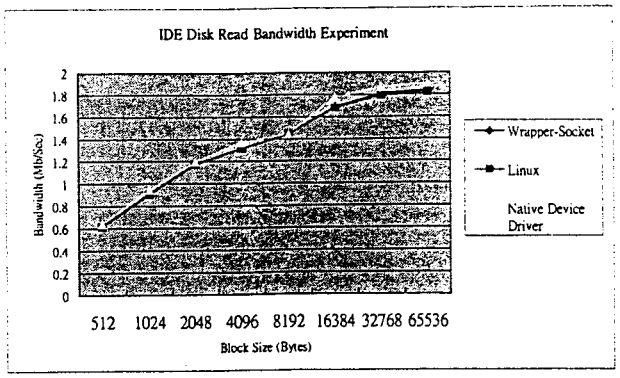

Figure 6: IDE disk read bandwidth experiment.

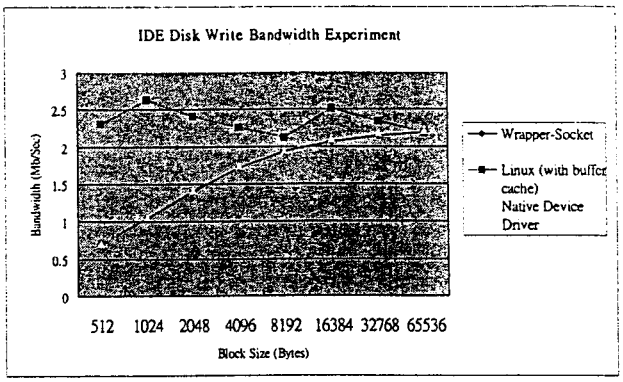

Figure 7: IDE disk write bandwidth experiment 1.

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.

3. SYSTEM EVLULATIONS

We evaluate the performance of the wrapper-socket plus Linux device drivers. The bandwidth and latency time of Linux device drivers is of interest. Experimental environment includes an Intel Pentium 100MHz CPU, 16MB RAM, a 540MB hard disk and a 3COM 3c509 Ethernet card.

3.1 Device Driver Bandwidth

In this experiment, we compare the bandwidth of devices operated in Linux and in *Vega*, which are further divided into wrapper-socket plus Linux device drivers and custom designed native device drivers. For block device drivers, we take IDE (Integrated Device Electronics) device driver for example. 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 may or may not be transmitted immediately. The implementation of the wrapper-socket also suffers from this problem.

For the wrapper-socket plus Linux IDE driver and the native driver, we use a thread that reads physically continuous 20MB data sequentially. The read/write block size is ranged from 512 bytes to 65536 bytes. In Linux, we opened */dev/hdb* file for access. The read bandwidth results are shown in Figure 6. In our test platform, the disk bandwidth increases with 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 7. In our test platform, the bandwidth increases with block size. In Linux, due to effects of the 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 8. We noticed that our test platform performs better than Linux in the last few block sizes. The reason is that, 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. In these experiments, we found that the native device driver performs better than other combinations. However, the difference between the wrapper-socket plus Linux IDE driver and the native driver is small.

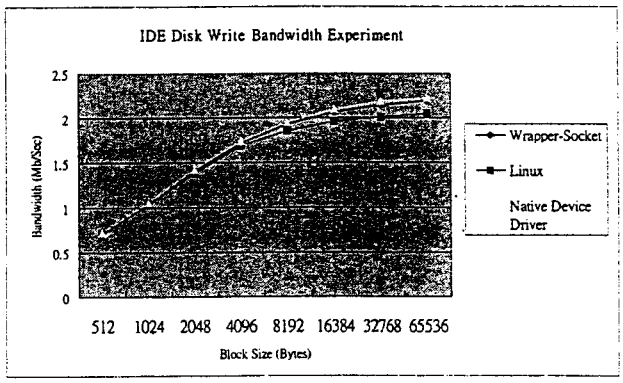

Figure 8: IDE disk write bandwidth experiment 2.

3.2 Device Driver Latency

In order to use unmodified Linux device drivers, the wrapper-socket must fill data structures that are used in Linux device drivers. We don't define new common data structures and do translations in between. On the contrary, we use data structures already defined in Linux for designing the wrapper-socket. The advantage is the minimal overhead being induced since the wrapper-socket does all the necessary parts. We do several experiments to measure the wrapper-socket as well as Linux device drive processing time in this section.

Table 1: The read/write latency time of IDE device driver (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

Table 2: The latency time for sending/receiving network packets (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

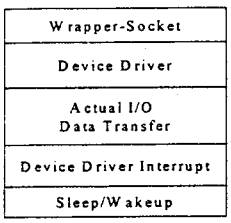

Figure 9: Split time for block device driver accessing.

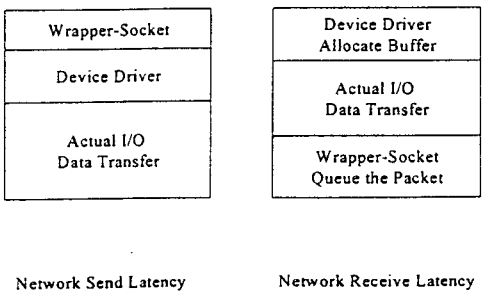

Figure 10: Network send/receive latency time.

The split latency time of IDE device driver and the network device driver 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 transfer, and sleep/wakeup time as shown in Figure 9. The wrapper-socket latency time mainly comes from constituting of data *request* and *buffer_head* structures, which is needed in the device drivers. We use a block size of 512 bytes in the 512-byte case and 1024 bytes block size in other case.

As illustrated in Table 1, the processing time generally increases with read/write size because 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 10 and the measured latency time in Table 2. In the network sending case, the latency time for wrapper-socket comes mainly from allocating a *sk_buff* data structure and time for copying 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 because 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 put this packet in queues. 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.

4. RELATED WORK

A work to incorporate unmodified Linux device drivers into Mach 4.0 is most related to ours [Goel 96]. Several aspects of Mach are modified. For example, the in-kernel device independent layer of Mach is modified to recognize the possibility of Linux device driver emulation and the address mapping method is also modified.

The work to reuse existing shareware operating system codes is done [Ford 97]. A toolkit named OSKit is built to be a substrate for kernel and language research. 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. For solving the problem of different presentations for one type of data in different operating systems, e.g. Linux uses skbuffs while FreeBSD uses mbufs to represent network packets, the OSKit defines an internal common data structure and does the translation in each component "glues." Thus, the function of these glues is to emulate the original operating system environment, which is similar to our wrapper-socket.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have presented our work for reusing Linux device drivers in embedded systems. A software package is designed and implemented to incorporate Linux device driver sources. Embedded systems can use this package without suffering implementing a lot of device drivers when their systems are in developing stages. After the developing stages, the system designers can change the drivers to their specific ones on their wish. So far, this work is verified on *Vega* kernel and works well for embedded systems. We plan to use this package in larger systems and believe that it can perform as well as in embedded kernels.

The contributions of this paper are clear. First, this work supports a good research platform for device driver researches, because it contains a bundle of device drivers and the operating system interference is isolated. Second, few human resources are further needed for driver implementation and maintenance. There exist a lot of hackers in the cyber space in maintaining the Linux operating system, their efforts plus our work can contribute to any system researches if driver stuff is needed in the projects. Third, our design permits to incorporate new device drivers and upgraded device drivers. What needed to do is to move the new version driver source tree into the building environment and build it. So far, we found what we need to modify is only to change a Linux version number in the source tree. Linux device drivers will check this version number that is emulated by the *wrapper-socket*. Except this number, all driver upgrade is easy and does not need any modification to the source tree.

Except the X86 version, Linux also supports different processors. We plan to enhance the *wrapper-socket* to incorporate device drivers for different platforms. Beside, we are interested at topics about QoS (quality of service) supports in operating systems. We plan to use this package to explore resource reservation schemes for guaranteed system services. The work presented in this paper is a good experimental platform for this kind of researches.

REFERENCES

[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.

[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.

[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.

[Lee 95] Paul C. H. Lee, Mei-Ling Chiang, Shang-Te Shu, Ta-Chuan Liu, Wu-Yang Chung and Ruei-Chuan Chang, 'Experiences in Porting μ-Kernel Operating System to the CONVEX Supercomputer', *Journal of Information Science and Engineering.* 12, 1995, p.p. 167-192.

[Lee 98.a] Paul C. H. Lee, Chi-Wei Yang and Ruei-Chuan Chang, 'An Integrated Core-Work for Fast Information-Appliance Buildup', Technical Report TR-IIS-98-006, Institute of Information Science, Acamedia Sinica, Taiwan, R.O.C., 1998.

[Lee 98.b] Paul C. H. Lee, 'From Micro Kenrels to Micro Cores', Technical Report, Institute of Information Science, Acamedia Sinica, Taiwan, R.O.C., 1998.

[Rawson 97] Freeman Rawson, 'Experiences with the Development of a Microkernel-based, Multiserver Operating System', In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI)*, Cape Cod, Massachusetts, USA, May 1997.