ELSEVIER

The Journal of Systems and Software 67 (2003) 19–30

The Journal of Systems and Software

www.elsevier.com/locate/jss

OS Portal: an economic approach for making an embedded kernel extensible

Da-Wei Chang, Ruei-Chuan Chang *

Department of Computer and Information Science, National Chiao Tung University, 1001 Ta Hsueh Road, 30050 Hsinchu, Taiwan, ROC

Received 28 April 2001; received in revised form 6 August 2001; accepted 22 January 2002

Abstract

With the rapid development of embedded system techniques and Internet technologies, network-enabled embedded devices have grown in their popularity. One critical design trend of such devices is that they are shifting from static and fixed-function systems to more dynamic and extensible ones, which are capable of running various kinds of applications. To support the diversity of the applications, kernels on these devices must be extensible. However, making embedded kernels extensible is challenging due to the shortage of resources on these devices.

In this paper, we propose the operating system portal framework, which makes embedded kernels become extensible while keeping the added overheads minimal. By storing kernel modules on a resource-rich server and loading them on demand, the need for equipping a local storage on the device is eliminated. In addition, we propose mechanisms for reducing the memory requirements and performing on-line module replacement on the embedded devices.

According to the performance evaluation, our approach requires only 1% of the resource requirements, compared to the traditional approaches. This allows our framework to be applied on a wide range of embedded devices.

Keywords: Extensible kernels; Embedded kernels; Network-enabled embedded devices

1. Introduction

In recent years, Internet technologies together with the rapid development of embedded system techniques have made many network-enabled embedded devices prominent. One critical design trend of these devices is that consumers are driving the shift of embedded devices from static and fixed-function systems to more dynamic and extensible ones (Sun, 2001). At the hardware level, extensible devices such as Visor handhelds (Handspring, 2002) contain expansion slots so that they can become wireless Internet devices, mobile phones, MP3 players, and etc. However, many kinds of functionality require not only hardware components but also software system modules. For example, a Bluetooth-enabled device requires a hardware Bluetooth module as well as the Bluetooth protocol stack. For another example, to access data on a disk, not only the disk drive is required but also the driver and the file system are needed. At the software level, users may wish to download various kinds of applications from remote sites and execute them on their devices. Many of the downloaded applications require extra support from the operating system. For instance, a multimedia application may specify a different scheduling policy to satisfy its own need. For another instance, a high priority network application may ask the system to use a priority based packet scheduling policy instead of the default one (e.g., FCFS) to increase its throughput. Therefore, the kernels of these devices must be extensible so as to support the diversity of these runtime-added functionality and downloaded applications. However, supporting extensible kernels requires more resources and therefore is not affordable for many resource-constrained embedded devices. Owing to the low cost requirement, these devices are usually equipped with only small-sized ROMs or RAMs, and they often have no local disks.

Due to the reason mentioned above, many embedded kernels are still not dynamically extensible. For

* Corresponding author.
E-mail address: rc@cc.nctu.edu.tw (R.-C. Chang).

0164-1212/03/$ - see front matter © 2002 Elsevier Science Inc. All rights reserved.
doi:10.1016/S0164-1212(02)00084-5

example, the VRTX (Mentor, 2000) kernel is not extensible; the Nucleus (Accelerated, 2001) and eCOS (Red, 2001) kernels are only *statically extensible*, which means that they are allowed to be configured at compile time only. Such extensibility is not enough. Imagine that a user plugs a disk to his embedded device via a USB link and downloads a Java applet to read information from the disk. To allow the applet to access the disk, a file system should be presented in the embedded kernel. If the kernel is not configured to have a file system, the user will have to stop all the applications, re-configure the kernel, install and reboot the kernel again. For another example, an application may have several threads and desire to specify a different scheduling policy to satisfy its own need. If the kernel is not configured to provide the policy, the above re-configuration process should be performed again. Obviously, these situations do result in inconvenience to the users. Therefore, embedded kernels should be extensible (i.e., *dynamically extensible*) even though the devices are resource-limited.

Many desktop operating systems such as Linux are already extensible. By using the technique of loadable kernel modules (LKMs) (Pomerantz, 1999), these kernels can be extended at run time. In addition to the LKM, many research efforts such as *micro-kernels*, *extensible kernels*, and *Java operating systems* also focus on the extensibility of kernels. However, none of them address the resource-limited problem of embedded devices. Therefore, our goal is to make an embedded kernel extensible while keeping the added overheads minimal.

We propose a framework, named operating system portal (OSP) framework, to achieve this goal. In this framework, we assume that embedded devices are continuously connected to a resource-rich server (i.e., the OS Portal). The OS Portal, as the name indicates, acts as a portal site of kernel modules. It provides all the possible kernel modules that an embedded device may need. An embedded kernel is just equipped with a base set of modules on its initialization. During the execution of the applications, the embedded kernel may download other modules and perform module replacement on demand (e.g., replacing the current thread-scheduling module, say first-in-first-out (FIFO), with a new one, say Round-Robin (RR)).

Our work is unique in that we move the job of module linking, which is traditionally performed on the embedded kernel, to the OS Portal. This makes an embedded kernel extensible while keeping the increase on the memory footprint minimal. According to the performance measurement, the overhead of our technique is only about 1%, compared to those of the traditional approaches. As a consequence, our technique can be applied to a much wider range of embedded systems. In addition, we use a mechanism named cooperation based module replacement to perform *on-line* replacement of kernel modules. On-line module replacement is necessary since it is not feasible to exit all the running programs before replacing kernel modules. However, on-line replacement is more complex than the off-line one. This is because kernel modules always encapsulate some run-time information, and replacing a kernel module with a new one usually involves the transfer of this information. In this mechanism, such an information-transfer is accomplished through the cooperation between the involved modules. The benefit of this mechanism is that it hides the details of the module replacement from the irrelevant part of the kernel, making the kernel easier to maintain and upgrade.

The rest of the paper is organized as follows. Section 2 presents the alternative approaches that can be used to make embedded kernels extensible. We describe the design and implementation of the OSP framework in Section 3. The performance results are given in Section 4. Section 5 shows the research efforts that are related to ours. We describe the limitations of the OSP framework in Section 6. Finally, conclusions and future works are given in Section 7.

2. Alternative approaches

In this section we describe two alternative approaches that can be used to make an embedded kernel extensible: network file system (NFS) based approach and socket based approach. Both are based on the LKM model.

2.1. Network file system based approach

Many desktop operating systems (e.g., Linux) are already extensible since they provide a mechanism to load kernel modules at run time. In these systems, modules are usually accessed via the local file systems. To address the problem that embedded devices usually have no local storages, we may adapt the system by performing the following tasks. First, we place the kernel modules on a remote site to save the storage need of the embedded devices. Second, we replace the local file system with a network one so that the embedded kernel can load modules from the remote site. Fig. 1 shows the architecture of the NFS based approach. From the figure we can see that, a kernel-level dynamic loader is responsible for loading modules via the NFS. Note that the module loaded to the client host (i.e., embedded device) is still left un-linked. The dynamic linker has to link it with the client-side kernel via the kernel symbol table.

The advantage of this approach is that it does not require any kernel modifications, and hence is easy to implement. However, the resource consumption of this approach might prevent it from being applied to resource-limited embedded devices. To make an embed-

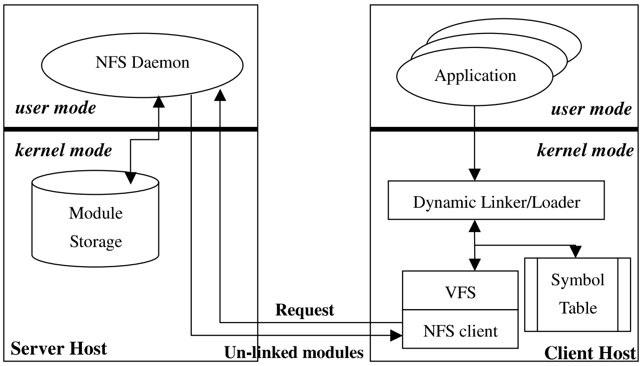

Fig. 1. The NFS based architecture.

ded kernel extensible by using this approach, the following overheads are indispensable. First, the kernel must be equipped with a dynamic linker/loader. Second, the code and data of the VFS and NFS client should be included. Third, a kernel symbol table is necessary for the dynamic linking of modules. And finally, to support multiple object file formats, an object file reader has to be included in the client for each object file format. According to our experiments, the total size of these overheads is about hundreds of Kbytes, which is unaffordable for resource-constrained embedded devices.

2.2. Socket based approach

From Fig. 1 we can see that, the VFS layer and the NFS client are not a must for dynamic loading of modules. They only provide a file system interface for the dynamic loader. Therefore, they can be excluded from the client kernel by replacing the interface with a socket-like one. Fig. 2 shows the architecture of this socket based approach. A client-side dynamic loader issues requests to the server host. On the server side, a user-level server process is responsible for processing these requests and sending the requested modules to the client. Similar to the NFS based approach, the modules sent to the client are left un-linked. They are linked together with the client kernel by the client-side dynamic linker.

This approach differs from the NFS based one in the following two aspects. First, it requires some kernel modifications. A traditional dynamic loader loads modules via a file system interface. Instead, this approach requires the dynamic loader to use a socket interface. Second, this approach eliminates the overheads of the VFS and the NFS client, making itself more feasible for resource-limited devices. However, the other overheads (i.e., the dynamic loader/linker, the symbol table, and the possible object file readers) are still a heavy pressure for such devices.

3. Design and implementation

In this section, we shall describe the design and implementation of the OSP framework. Section 3.1 gives an overview of the OSP architecture. We describe two main techniques used in the OSP: server-side module linking and cooperation based module replacement in Sections 3.2 and 3.3, respectively. Finally, the overall control flow of extending an embedded kernel is presented in Section 3.4.

3.1. Architecture overview

Fig. 3 shows the architecture of the OSP framework. It follows the client-server model. All the dynamically loadable modules are located on the server host. A user-level process (i.e., the OS Portal process) is responsible for loading, linking and transmitting these modules to the clients. A kernel-level module manager is installed on the client to make the client kernel extensible. During the client startup, the module manager registers the client to the OS Portal. After the registration, the module manager is allowed sending requests to load modules from the OS Portal. Generally speaking, generation of

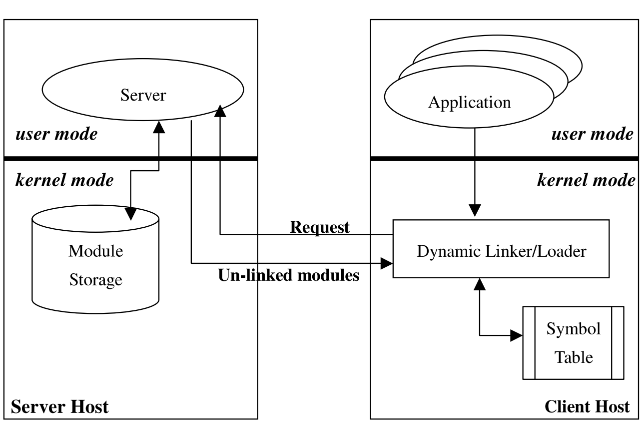

Fig. 2. The socket based architecture.

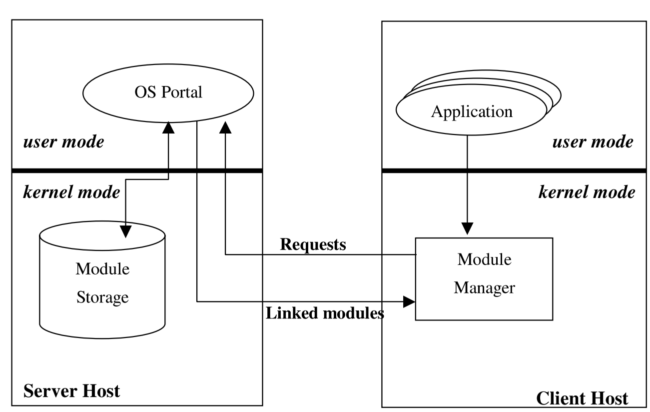

Fig. 3. The OS Portal architecture.

requests is related to the execution of client-side applications. For example, if a client application wants to use another CPU scheduling policy which is not supported by the current kernel, the module manager will send a request to the OS Portal, download that scheduling policy, and replace the current policy with the new one.

3.2. Server-side module linking

The goal of the server-side module linking is to make a resource-limited embedded kernel extensible, while keeping the imposed overheads minimal. As mentioned above, the OS Portal process on the server machine receives requests from client kernels and performs tasks according to those requests. There are two types of requests: REGISTRATION and LOAD_MODULE. The former is used for a client to register itself to the OS Portal, while the latter is used to load a specific module from the OS Portal.

The OS Portal performs the following tasks when it receives a REGISTRATION request. First, it authenticates the client (for security or billing purpose). In our current implementation, the authentication is performed by using the user name and password provided by the client. After the client is authenticated, the OS Portal creates an in-memory symbol table for the client. The symbol table is created based on the *symbol file*, which is uploaded to the OS Portal prior to the execution of the client. The symbol file is obtained by *nm*, a GNU binary utility (Free, 1998) that prints the symbols and addresses of an object file, while building the client kernel. Note that, although clients can be dynamically extended, they share a common *base kernel configuration*. Therefore, they can use the same copy of the symbol file. After the symbol table is created, it will be updated as modules are dynamically loaded to the client.

When the OS Portal receives a LOAD_MODULE request, it locates the requested module first. If the module is not found, it returns back a "module not found" error and let the client kernel process that error. The client kernel can simply discard this error message or request another module. If the module is found, the OS Portal will try to load and link it according to the symbol table of the client. In order to perform the module linking, the OS Portal has to know where the module will reside in the client-side memory (i.e., the starting address of the module). This address is client-specific and therefore should be provided by the client. It seems straightforward that a LOAD_MODULE request can carry a *module_address* parameter to specify the starting address of the module. However, sometimes it is impossible for a kernel to determine the module address without knowing the information (e.g., size) of the module. This is because that the kernel memory allocator may choose different memory allocation mechanisms for different size requirements. For example, in LyraOS (Cheng et al., 2000; Yang et al., 1999), the embedded kernel that is used in our work, memory blocks that are smaller than half of a page size are allocated via the power-of-two allocator. Other memory blocks are allocated via the multiple-of-page allocator. Therefore, if the size is unknown, the client kernel will not be able to determine which memory allocator to invoke. As a consequence, it cannot determine the starting address of the module.

Message Size	Init() Address	Cleanup() Address	Linked Image

Fig. 4. The message buffer.

We solve this problem by allowing the OS Portal to send the module size to the client when the client requests the module. After the client determines the starting address, it sends this address back to the OS Portal for module linking. Obviously, this requires another round trip of messages and therefore may degrade the performance of module loading. However, since module loading does not happen frequently, the extra round trip of messages will not have large impacts on the overall performance. In addition, client kernels can cache the sizes of frequently used modules and hence avoid the extra round trip of messages.

After the module address is determined, the OS Portal is able to link the module with the client kernel. This is achieved by resolving symbols in the module via the symbol table of the client kernel. We modify the code of the Linux kernel module utilities 2.2.2 (Ekwall, 2001) to perform the symbol resolution. [1]

After the module is linked, the OS Portal marshals the image and other information to a message buffer, and then sends the buffer to the client. Fig. 4 shows the format of the message buffer. Note that each module has two pre-defined management routines. The *init*( ) routine is used to initialize the module, and replace the *current* module with this one. The *cleanup*( ) routine is invoked when the module is to be replaced. The addresses of these two routines are contained in the message buffer to allow the client kernel to setup the module information quickly.

When the client kernel receives the message buffer, it can simply setup the module information according to its need, and then invoke the init( ) routine to replace the current running module with the new one (i.e., this module).

[1] In the OSP framework, the OS Portal is able to support clients running on different processors. The client processor information can be sent during client registration and the requested module for the target processor can be linked and sent to the client. In current implementation stage, we only support x86 clients. Clients running on other processors will be supported in the future.

At the end of this section, we describe the space overheads that are eliminated by the server-side module linking. First, the dynamic linker can be removed from the client kernel. This includes not only the code and data segments of the dynamic linker but also the dynamic memory that it allocates. Second, the embedded kernel does not have to keep the symbol table in the memory. This can save large memory space if there are lots of symbols in the table. And third, it is not necessary to provide an object file reader for each object file format. It is the responsibility of the OS Portal to support multiple object file formats. All module images transmitted to the client are already linked (i.e., resolved). Therefore, the client has no idea about the format of the linked images. This enables us to add new object file readers without modifying the client kernels, making the whole system easier to upgrade.

3.3. Cooperation based module replacement

In this section, we show how modules are replaced at run time. One important requirement of the module replacement is that the details of the replacement should be hidden from the rest of the kernel. Only the modules involved have the idea of how the replacement performs. This makes the separation between the client kernel and the modules more clear, and therefore allows the development and management of the kernel easier. To achieve this, we use a technique, namely cooperation based module replacement, to replace modules at run time. Fig. 5(a) shows the module replacement interface (MRI). This interface contains only one routine: init( ), which is used to trigger the module replacement. Fig. 5(b) and (c) show two example module interfaces based on the MRI, one is a scheduler interface and the other is a memory allocator interface. Both the module interfaces contain a *mri* field for module replacement. In addition, each interface contains a cleanup( ) routine, which is used for the handoff of run-time module information. For example, the *scheduler.cleanup*( ) routine returns a list of runnable threads that are managed by the scheduler module, and the *memory_allocator.cleanup*( ) routine returns a list of memory blocks managed by the memory allocator module. Note that the cleanup( ) routine of the old module should be invoked in the init( ) routine (specifically, the *mri.init*( ) routine) of the new one during the module replacement. In other words, when the client kernel downloads a new module from the OS Portal, it triggers the module replacement by invoking the init( ) routine of the new module. The init( ) routine will in turn invoke the cleanup( ) routine of the old (i.e., current) module. As a result, the run-time module information is handed over to the new module.

In addition to these routines, each module interface contains a set of module-specific routines. For example, the scheduler module interface shown in Fig. 5(b)

```
typedef struct {
            void (*init)(void);       // Trigger the module replacement and
                                      // initialize the module
} MRI_t ;
```

(a)

```
typedef struct
{
    MRI_t     mri;                           // Module replacement interface
    int (*checkout)( struct thread ** );     // Remove a thread from the run queue
    int (*checkin)( struct thread * );       // Insert the thread into the run queue
    ThreadList_t (*cleanup)(void);           // Clean up the module
} scheduler_t;
```

(b)

```
typedef struct
{
    MRI_t     mri;                           // Module replacement interface
    void (*malloc)(unsigned int size) ;      // Allocate memory
    void  (*free)(void *ptr) ;               // Free memory
    MemList_t (*cleanup)(void);              // Clean up the module
} memory_allocator_t;
```

(c)

Fig. 5. (a) Module replacement interface, (b) an example scheduler interface, and (c) an example memory allocator interface.

contains the *checkin*( ) and *checkout*( ) routines that are used to add/remove a thread to/from the run queue of the scheduler. Similarly, the memory allocator interface in Fig. 5(c) contains the *malloc*( ) and *free*( ) routines to allocate/free memory blocks.

In the following, we shall describe how cooperation based module replacement works by presenting the design of a CPU scheduling system that supports on-line replacement of scheduling policies. Other kernel systems can also be designed in the same way.

We take the interface shown in Fig. 5(b) as the scheduler interface. The scheduling policy is encapsulated in the checkin( ) and checkout( ) routines. Other part of the kernel does not have to know the implementation details of the policy. To replace the current scheduler S1 with a new one S2, the client kernel performs the following tasks. First, it contacts the OS Portal and downloads S2. Second, it invokes the init( ) routine of S2. In this routine, S2 in turn calls the cleanup( ) routine of S1 so as to make S1 check out all its runnable threads, and inserts these threads into a thread list, which is returned to S2. And then, S2 checks in all the threads in the list according to its policy. Finally, it frees the memory used by S1 and sets the pointer to the current scheduler (i.e., cur_scheduler) to itself.

3.4. Overall control flow

The control flow of the client registration and module loading is shown in Fig. 6. The steps are as follows.

1. After initialization, the client kernel sends a REGISTRATION message (together with its user name and password) to the OS Portal.
2. The OS Portal authenticates the client, and then constructs the symbol table for the client.
3. The OS Portal sends a message back to the client to tell whether the client is authenticated.
4. If a client needs a module, it sends a LOAD_MODULE message to the OS Portal. The parameters include the module name, the starting address of the module, and a v bit that indicates whether the starting address is valid. If the starting address is not valid, messages 6 and 7 are needed to assist the client kernel to determine the starting address. Otherwise, these two messages are not necessary.
5. The OS Portal loads the requested module and the related information to its memory.
6. To help the client kernel determine the module address, the OS Portal sends the size of the module to the client.
7. After receiving the size information, the client kernel determines the module address and sends it to the OS Portal.
8. The OS Portal uses the starting address provided in message 4 or 7 to link the module. Symbols are resolved via the symbol table constructed in step 2.
9. The linked image and the related information are marshalled in a message buffer, which is sent to the client.
10. The client kernel invokes the init( ) routine of the module to trigger the module replacement.

4. Performance measurement

In this section, we compare the performance of the OSP framework with the other two approaches: the NFS based and the socket based approaches. In the comparison, we provide detailed breakdowns of their overheads. In addition, we also show the performance of the OSP framework by presenting the request processing time and the server throughput.

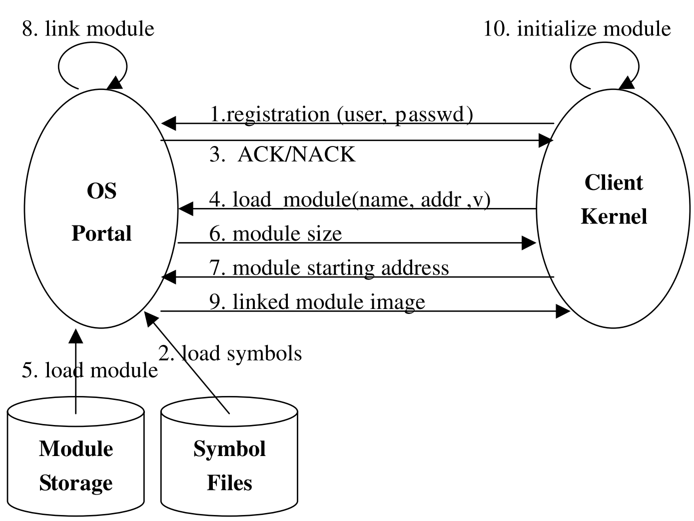

Fig. 6. The control flow of registration and module loading.

Table 1
Modules and total size of LyraOS

Categories	Modules
Core	Initialization, Memory, Thread, Scheduler
Drivers	Keyboard, Mouse, IDE HD, Ethernet
Networking	TCP, UDP, IP
Total size	200,400 bytes

4.1. Experimental environment

The experimental environment consists of a client and a server host that are connected via a 10 Mbits/s Ethernet. The server host is a Pentium II 233 MHz machine with 128 MB RAM, running Linux 2.0.36. The OS Portal is a user-level process on the server host. The client host is a Pentium 133 MHz machine with 32 MB RAM, running LyraOS 1.8. Table 1 shows the modules and total size of the LyraOS.

We implement three scheduler modules in LyraOS: RR, FIFO, and fixed-priority (PRIO) scheduling. The sizes of these modules are 2888, 2824, and 3328 bytes, respectively.

In the next two sections, we compare the three approaches in terms of space overheads, which can be divided into the environment overheads and the module overheads.

4.2. Environment overhead

4.2.1. NFS based approach

Table 2 shows the environment overheads of the client kernel under the NFS based approach. Because most of the items are not currently available on LyraOS, we obtain these values by measuring the sizes of their versions on Linux 2.0.36. The *dynamic loader/linker* item shows the static size of the *insmod* program (Ekwall, 2001), which allows privileged users to load modules into a running Linux kernel. The sizes of the following 3 items are obtained by building the Linux kernel and seeing their corresponding object file sizes. Note that the Sun RPC is included since the NFS client is based on it (Kohler et al., 2000). [2] It should be noted that symbols in the object files are already stripped before we report the sizes. In addition to the above overheads, NFS based approach requires the kernel symbol table to be stored on the client. Since symbol tables are maintained in the OS Portal process in our system, we can measure this overhead by recording the high watermark of the memory that are used by the symbol table and the related data structures. The *dynamic memory* item in Table 2 shows the size of this overhead after a client has registered to the OS Portal. There are two points worth noting. First, the size of the symbol table is proportional to the number of symbols. The current configuration of LyraOS has 707 symbols, which result in an overhead of about 50 Kbytes. Another kernel configuration that is equipped with a window management system has 1410 symbols, which lead to an overhead of about 110 Kbytes (out of the 331 Kbytes total kernel size). Second, this size will increase at run time as the kernel loads more modules during its execution. For example, the size will increase to 67 Kbytes if all the three scheduler modules are loaded into the kernel.

Table 2
Environment overheads of the NFS based approach

Items	Size (bytes)
Dynamic loader/linker	25,612
VFS	62,636
NFS client	25,272
SUN RPC	51,064
Dynamic memory	50,988
Total	215,582

From the table we can see that an overhead of 210 Kbytes is required to make an embedded kernel extensible. This overhead is larger than the size of the original kernel, 200 Kbytes. Therefore, this approach may not become feasible unless the system vendors equip more memory in their devices.

4.2.2. Socket based approach

Table 3 shows the environment overheads of the socket based approach. Since it does not rely on the file system interface, the overheads of VFS, NFS client, and Sun RPC can be eliminated. The other two overheads are the same as those in the NFS based approach. From the table we can see that the total overhead is about 76 Kbytes, which still results in a notable increase in the kernel size. In addition, the same as the NFS based approach, the dynamic memory item will increase as module-loading events occur.

4.2.3. OS Portal approach

The only overhead of the OS Portal approach is the module manager, which is only 2600 bytes according to our measurement. Note that since we do not keep the

Table 3
Environment overheads of the socket based approach

Items	Size (bytes)
Dynamic loader/linker	25,612
Dynamic memory	50,988
Total	76,600

[2] Originally, the Sun RPC is based on UDP, whose static size is 5980 bytes after all the symbols for linking are stripped. If the original kernel does not contain UDP, this size should be included in the environment overheads. Otherwise, the Sun RPC implementation should be changed to use an alternative transport-layer mechanism that the kernel supports.

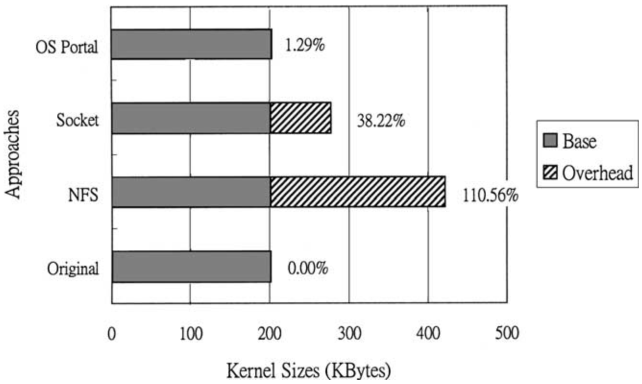

Fig. 7. Kernel sizes of different approaches.

symbol table on the client, the overhead will not increase during the loading of modules. In addition, adding support to new object file formats requires no further overheads on the client. Obviously, our approach does achieve the goal of making an embedded kernel extensible while keeping the added overhead minimal.

Fig. 7 summaries up the environment overheads of these approaches. The numbers in percentages show the proportions of the sizes of overheads to the original kernel size (i.e., the *base size*). Except for the OS Portal approach, the others result in noticeable increases in the kernel sizes.

4.3. Module overhead

In addition to the environment overheads, the OS Portal approach also differs from the others in the transmission sizes of modules (i.e., module overheads). Table 4 presents the module overheads of the three scheduler modules we implemented. In this table, the numbers in parentheses are the ratios of module overheads under the OS Portal approach to those under the other two approaches. From the table we can see that, the transmission sizes under the OS Portal approach are only about half of the sizes under the other approaches. This is because the former transmits modules in the form of linked images, while the latter transmits modules in the form of object files. Generally speaking, object files occupy more space than linked images since the former contain more overheads, such as symbols for dynamic linking and other data structures. One exception is the declaration of large static data areas in the module (e.g., a large static, un-initialized array). This will usually make the linked image larger than the object file since the data areas are not contained in the latter. However, we can avoid this problem by allocating large data areas dynamically. As a result, the transmission sizes of modules can usually be smaller under the OS Portal approach.

Table 4
Transmission sizes of modules

Modules	Approaches		
	NFS	Socket	OS Portal
FIFO	2824	2824	1240 (44%)
RR	2888	2888	1244 (43%)
PRIO	3328	3328	1700 (51%)

4.4. Request processing time

Table 5 shows the required time for processing requests. The *registration* item presents the time for the OS Portal to perform a REGISTRATION request. A major part of the time is spent on constructing the symbol table for the client. As we described earlier, the symbol table has 707 entries. The *module processing* item shows the time for the OS Portal to process a LOAD_MODULE request to the FIFO scheduler module. It includes the time to locate and load the module to the memory, link it with the client kernel, marshall the response to a message buffer, and then send the buffer down to the TCP/IP stack. The last item (i.e., *module downloading*) presents the total elapsed time for the client to download the FIFO scheduler. Since the module-loading operation does not happen frequently, we expect this 20-ms-delay as an acceptable value and will not cause noticeable performance degradation on the client.

4.5. OS Portal server throughput

In order to see the throughput of the OS Portal server, we wrote a micro-benchmark to measure the number of requests the server can service in a second. In the benchmark, we fork several client processes. After registering to the OS Portal server, each client process makes as many module requests as possible in a predefined period of time (currently, 30 s). After the time period, each client process reports its number of completed module requests (i.e., requests that the corresponding modules are received by the client successfully). Therefore, we can get the number of requests the server handles in a second by summing up the numbers and dividing the result by the time period.

To get a clearer view of the OS Portal throughput, we run the benchmark under four configurations, which are shown in Table 6. In this table, the *OS Portal host* column shows the configurations of the host on which the OS Portal process runs. Similarly, the *client host* column

Table 5
Request processing times

Task	Time (ms)
Registration	4.14
Module processing	3.47
Module downloading	20.60

Table 6
Configurations for testing the OS Portal throughput

Configuration name	OS Portal host	Client host	Connection type
PII_REMOTE	Pentium II 233 MHz, 128 MB RAM	Pentium 300 MHz, 128 MB RAM	TCP/IP over 10 Mbits/s Ethernet
PII_LOCAL	Pentium II 233 MHz, 128 MB RAM	Pentium II 233 MHz, 128 MB RAM	UNIX domain socket
K7_REMOTE	K7 600 MHz, 256 MB RAM	Pentium II 233 MHz, 128 MB RAM	TCP/IP over 100 Mbits/s fast Ethernet
K7_LOCAL	K7 600 MHz, 256 MB RAM	K7 600 MHz, 256 MB RAM	UNIX domain socket

shows the configurations of the host on which the client processes run. In this experiment, we run all client processes on the same host. The *connection type* column presents the connection mechanisms between the OS Portal and the client processes. Under the remote configurations (i.e., PII_REMOTE and K7_REMOTE), clients communicate with the OS Portal via TCP/IP sockets over 10 or 100 Mbits/s Ethernet. Under the local configurations (i.e., PII_LOCAL and K7_LOCAL), they use UNIX domain sockets for communication.

Fig. 8 shows the server throughput under these four configurations. From the figure we can see that, the OS Portal performs better under the local configurations. This is straightforward since UNIX domain socket does not incur the complex TCP/IP stack and the network delay. In addition, if the client processes are relatively fewer (i.e., fewer than 4), an increase in the client number will usually lead to an increase in the server throughput. This happens because the client processes have not saturated the server yet. However, once the server is saturated, further increase in the client number will just result in more overheads and resource contentions, and hence cause degradation in the server throughput.

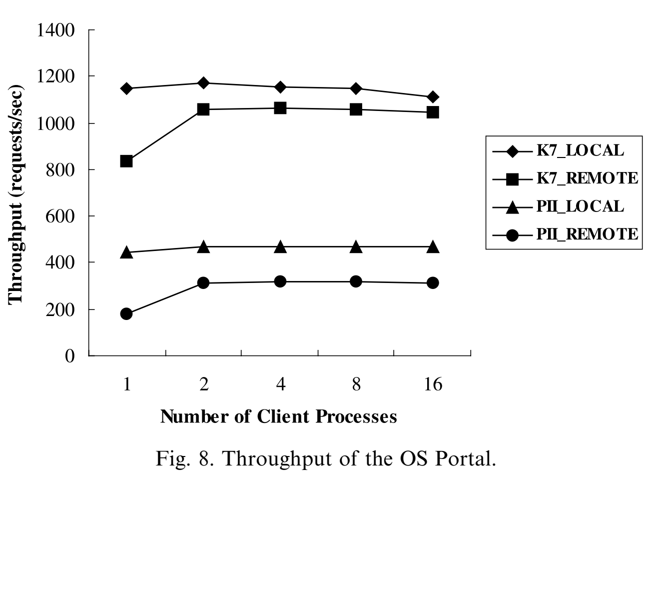

Fig. 8. Throughput of the OS Portal.

It should be noted that, the x-axis (i.e., number of client processes) of Fig. 8 does not reflect the actual client numbers that the OS Portal can support. As we mentioned above, each client in this experiment spends most of its CPU time performing module requests. This is not the normal case because an embedded kernel will request to the OS Portal only when it *occasionally* needs a kernel module that it does not have. This figure does reveal that the OS Portal server can support more than 1000 requests per second under the K7 configurations, and about 400 requests per second under the PII configurations. This is fairly enough for a server located in a home or a small/medium scaled enterprise.

5. Related work

For the last two decades, many research efforts on extensible systems have been proposed in the literature. However, such systems usually require much more resources. None of them addressed the problem of achieving the extensibility on resource-constrained embedded devices. Moreover, some of them result in the degradation of the system performance. In the following, we describe these research efforts. To make the description more clear, we divide some of the research efforts into several categories: micro-kernel, extensible kernel, Java operating system, and LKM. Other efforts are described following the description of the efforts in the above categories.

5.1. Micro-kernel

Research on micro-kernels (Rashid et al., 1989a,b; Zuberi and Shin, 1996) moves most of the traditional operating system functionality to user-level server processes. It simplifies the extension of the operating systems since the extension can be achieved by adding or replacing the server processes. However, it suffers from the performance problem. Since most operating system services are implemented in user level, a service request requires two more context switches and another pair of protection-domain crossings. Moreover, micro-kernels and traditional embedded kernels are totally different in the kernel structure. Therefore, extending embedded kernels with this approach involves kernel re-design. In contrast with the micro-kernel approach, we focus on making an existing embedded kernel become extensible without re-designing the kernel. In addition, we use kernel-level modules so that our system will not suffer from the performance problem. Finally, we make use of the server-side module linking to reduce the resource requirements (specifically, the memory footprints) of the client kernels.

5.2. Extensible kernel

To address the performance problem of micro-kernels, extensible kernels allow user applications to inject codes into the kernels so as to extend them. Because the injected codes run in the kernel mode, extensible kernels usually result in better performance than that of micro-kernels. However, the injected codes usually cannot run in full speed because of the security problems. Extensible kernels often perform run-time checks to ensure that the injected codes will not damage the kernels. For example, SPIN (Bershad et al., 1995) uses software fault isolation (Wahbe et al., 1993) to restrict the memory area that the injected codes can access. Systems such as packet filters (Mogul et al., 1987), and HiPEC (Lee et al., 1994) allow interpretative codes to be injected and use kernel mode interpreters to enforce the system security. Such run-time checks do degrade the system performance.

The main reason of the performance problem is that the kernels do not trust the extension codes. In contrast with their approaches, we assume that kernel extensions (i.e., kernel modules) are trusted. This assumption is used in the previous research (Auslander et al., 1997). It also holds for most of the UNIX implementations. In our approach, kernel modules are developed by trusted system programmers or third parties, and are verified prior to execution. Therefore, they can run in full speed. In addition, our approach requires no re-design of the operating systems and puts efforts on reducing the resource requirements of the kernels.

5.3. Java operating system

Java operating systems (JOS, 1997; Saulpaugh et al., 1999) allow the programmers to write system modules such as TCP/IP, and file systems in Java (Ritchie, 1997). The system modules are compiled as Java class files and are loaded by the JVM at run time. Liao et al. (1996) take another approach. They insert a JVM into a micro-kernel so as to allow users to write Java programs to extend the kernel. Similar to the previous techniques, these approaches also suffer from performance problems. The major problem is that Java codes are interpreted, not compiled. Although the performance of interpretation improves since the birth of Java, there is still a performance gap between the interpreted and the compiled codes. Some Java platforms take advantage of just-in-time (JIT) compilation to improve the performance. This technique, however, is hard to be applied on resource-limited devices since it consumes too much memory to perform the compilation. In addition, the codes generated by the JIT compilation are less optimized since there is not much time to perform code optimizations.

In contrast with the Java operating systems, we use compiled codes. Since codes are compiled off-line, there is much time to perform optimizations. In addition, we focus on making a C-language based embedded kernel extensible instead of re-writing a whole kernel in Java. And finally, we put efforts on reducing the resource requirements of the kernels.

5.4. Loadable kernel module

As described earlier, many desktop operating systems such as Linux, provide LKM to extend their kernels at run time. After being installed, kernel modules can run in full speed without any further run-time checks. The major problem of the LKM is its space overheads. As shown in Section 4, it requires much more overheads than our approach.

Oikawa et al. (1996) take an approach similar to the LKM. They make RT-Mach (Tokuda et al., 1990) extensible by introducing dynamic kernel modules (DKMs). The same as LKMs, DKMs are stored as files and are able to be loaded/unloaded at run-time. However, DKMs are managed by a user-level DKM server, instead of the kernel. The server is responsible for loading a DKM when it is needed, and unloading a DKM when the kernel is short of memory. In contrast to this approach, we manage the kernel modules at remote site, reducing much local memory requirements. In addition, we address the issue of run-time module replacement, which is not mentioned in their work.

In addition to the research described above, there are still many efforts on system extensibility or customizability. At the end of this section, we give brief descriptions on them.

Kohler et al. (2000) propose a configurable IP router architecture named *Click*. Under this architecture, router functions are implemented in *elements*, which can finally be composed into a single Linux kernel module. By composing different elements, the developers can customize the router according to their needs. However, this customization is too coarse-grained even that it does support fine-grained components (i.e., elements). This happens because all elements are finally linked into a single module, and therefore the customization requires the whole router to be unloaded before the new router can be loaded. Router Plugins (Decasper et al., 2000) is a software architecture for fine-grained customization of routers. It uses the LKM mechanism supported by the kernels to load different plugins to extend the functionality of the router. Since routers are usually equipped with rich resources, the researchers did not address the resource-limited problems that we encountered.

The OSKit (Ford et al., 1997) allows users to build their customized kernels by composing different system components. It also has a linking toolkit, named Knit (Reid et al., 2000), to assist users when composing these components. Different from our work, the researchers put most of their efforts on the link-time techniques, instead of providing a customizable and economic run-time environment.

Helander and Forin (1998) propose a modular system architecture. They also describe a dynamic module-updating mechanism that is similar to our module replacement technique. However, in their architecture, the extension modules are loaded via the file system. Therefore, it is similar to the LKM approach that we mentioned above. As we described in Section 4, our approach requires much less overheads than theirs.

DEIMOS (Clarke and Coulson, 1998), Kea (Veitch and Hutchinson, 1996), and Pebble (Gabber et al., 1999) are all extensible systems. However, they maintain symbol tables on the local site for dynamic extension. In addition, they do not address the problem of run-time module replacement.

Finally, some other research (Gheith et al., 1994; Messer and Wilkinson, 1996) allows the internal implementation, or the invocation methods of a service to be varied at run time, according to the invocation attributes. However, since all of the possible implementations of a service have to be installed in prior, this approach does not consider the resource consumption of the kernels.

6. Limitations

Although the OSP framework provides a way to extend embedded kernels effectively. It does have some limitations. In this section, we will describe these limitations.

Scalability: In our prototype implementation, we use single server machine as the OS Portal. Moreover, the OS Portal maintains a persistent connection for each on-line client. Therefore, the average request latency for a client will become longer as the number of on-line client increases. According to our experiment, the request latency is proportional to the number of on-line clients. When there are 100 clients, the request latency is 0.124 s while it becomes 1.26 s as the client number reaches to 1000. The experiment reveals that the request latency will become unacceptable if there are large number of clients, say 10,000. We intend to use the following approaches to increase the scalability of the framework. First, we will replace the single server machine with a cluster of servers. Second, we intend to use connectionless protocols for client-server communication to reduce the server overheads for each client. In the future we will implement the approaches and evaluate the result performance.

Static module interfaces: Although the client kernels can *dynamically* download kernel modules to extend their functionality, module interfaces are *static*. They must be defined and implemented in the client kernels in advance. For example, if a client kernel does not implement a scheduler interface, it cannot perform module replacement on scheduler modules. To enable the replacement, a scheduler interface must be incorporated into the kernel and therefore kernel modification is needed.

7. Conclusions and future works

In this paper, we propose the OSP framework to make an embedded kernel extensible while keeping the added overheads minimal. In this framework, embedded kernels are extended via kernel modules that are stored in the OS Portal server. We propose server-side module linking to reduce the space overheads of the embedded kernels. Moreover, we use the cooperation based module replacement technique to perform on-line module replacement.

To prove the concept of the OSP framework, we implement a dynamically replaceable scheduling system based on the framework. According to the performance measurement, the space overhead of the OSP framework is only about 1%, when compared to those of the other approaches. Besides, its performance is acceptable for current network technology.

In addition to the scheduling subsystem, we will implement other subsystems in the future for the completeness of our work. Moreover, current implementation of the OSP framework uses TCP/IP as the transport-layer protocol. However, owing to the rapid development of wireless technologies such as Wireless LAN and Bluetooth, it should be useful to adapt our system to wireless network. However, we should consider the problem of mobility management if we use wireless technologies as the transport-layer mechanisms. For example, if a client roams from an area controlled by the current OS Portal to another, the kernel symbol table for the client should be transferred to the new OS Portal. In the future, we shall try to implement the OSP framework on wireless links, measure their performance, and perform research on the mobility management in this framework.

Acknowledgement

This work is supported by National Science Council under project NSC90-2218-E-009-010.

References

Accelerated Technology, 2001. The Nucleus PLUS embedded real-time kernel. Available from <http://www.acceleratedtechnology.com/new_content/main_pgs/nuc_plus.html>.

Auslander, M., Franke, H., Gamsa, B., Krieger, O., Stumm, M., 1997. Customization lite. In: Proceedings of the 6th Workshop on Hot Topics in Operating Systems, pp. 43–48.

Bershad, B., Savage, S., Pardyak, P., Sirer, E.G., Becker, D., Fiuczynski, M., Chambers, C., Eggers, S., 1995. Extensibility, safety and performance in the SPIN operating system. In: Proceedings of the 15th ACM Symposium on Operating System Principles, pp. 267–284.

Cheng, Z.Y., Chiang, M.L., Chang, R.C., 2000. A component based operating system for resource limited embedded device. In: Proceedings of the IEEE International Symposium on Consumer Electronics, pp. 27–31.

Clarke, M., Coulson, G., 1998. An architecture for dynamically extensible operating systems. In: Proceedings of the 4th International Conference on Configurable Distributed Systems, pp. 145–155.

Decasper, D., Dittia, Z., Parulkar, G., Plattner, B., 2000. Router plugins: a software architecture for next generation routers. IEEE/ACM Transactions on Networking 8 (1), 2–15.

Ekwall, B., 2001. The Linux Module Utilities, Version 2.2.2-pre6. Available from <http://www.kernel.org/pub/linux/utils/kernel/modutils/v2.2/>.

Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., Shivers, O., 1997. The flux OSKit: a substrate for OS and language research. In: Proceedings of the 16th ACM Symposium on Operating Systems Principles, pp. 38–51.

Free Software Foundation, 1998. The GNU binary utilities. Available from <http://www.gnu.org/manual/binutils/>.

Gabber, E., Small, C., Bruno, J., Brustoloni, J., Silberschatz, A., 1999. The pebble component-based operating system. In: Proceedings of the 1999 USENIX Annual Technical Conference, pp. 267–281.

Gheith, A., Mukherjee, B., Silva, D., Schwan, K., 1994. KTK: kernel support for configurable objects and invocations. In: Proceedings of 2nd International Workshop on Configurable Distributed Systems, pp. 92–103.

Handspring Inc., 2002. Visor handhelds. Available from <http://www.handspring.com/products/visorfamily/index.jhtml?prod_cat_name = Family>.

Helander, J., Forin, A., 1998. MMLite: a highly componentized system architecture. In: Proceedings of the 8th ACM SIGOPS European Workshop, pp. 96–103.

JOS Project, 1997. JOS: an open, portable, and extensible Java object operating system. Available from <http://www.metamech.com/wiki/view/Main/ AboutJOS>.

Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M.F., 2000. The click modular router. ACM Transactions on Computer Systems 18 (4), 263–297.

Lee, C.H., Chen, M.C., Chang, R.C., 1994. HiPEC: high performance external virtual memory caching. In: Proceedings of the 1st Symposium on Operating Systems Design and Implementation, pp. 153–164.

Liao, W.S., See-Mong, T., Campbell, R.H., 1996. Fine-grained, dynamic user customization of operating systems. In: Proceedings of the 5th International Workshop on Object Orientation in Operating Systems, pp. 62–66.

Mentor Graphics Corp, 2000. VRTXoc Real-Time Executive: a powerhouse for system-on-chip applications. Available from <http://www.mentor.com/embedded/vrtxos/VRTXoc_Realfinal.pdf>.

Messer, A., Wilkinson, T., 1996. Components for operating system design. In: Proceedings of the 5th International Workshop on Object Orientation in Operation Systems, pp. 128–132.

Mogul, J., Rashid, R., Accetta, M., 1987. The Packer Filter: an efficient mechanism for user-level network code. In: Proceedings of the 11th ACM Symposium on Operating Systems Principles, pp. 39–51.

Oikawa, S., Sugiura, K., Tokuda, H., 1996. Adaptive object management for a reconfigurable microkernel. In: Proceedings of the 5th International Workshop on Object Orientation in Operation Systems, pp. 67–71.

Pomerantz, O., 1999. Linux kernel module programming guide. Available from <http://www.linuxdoc.org/LDP/lkmpg/mpg.html>.

Rashid, R., Julin, D., Orr, D., Sanzi, R., Baron, R., Forin, A., Golub, D., Jones, M., 1989a. Mach: a system software kernel. In: Proceedings of the 34th Computer Society International Conference, pp. 176–178.

Rashid, R., Baron, R., Forin, A., Golub, D., Jones, M., Julin, D., Orr, D., Sanzi, R., 1989b. Mach: a foundation for open systems. In: Proceedings of the 2nd Workshop on Workstation Operating Systems, pp. 109–113.

Red Hat Inc., 2001. The eCOS documentation. Available from <http://sources.redhat.com/ecos/docs-latest/>.

Reid, A., Flatt, M., Stoller, L., Lepreau, J., Eide, E., 2000. Knit: component composition for systems software. In: Proceedings of the 4th Symposium on Operating Systems Design and Implementation, pp. 347–360.

Ritchie, S., 1997. Systems programming in Java. IEEE Micro 17 (3), 30–35.

Saulpaugh, T., Clements, T., Mirho, C.A., 1999. The Inside JavaOS Operating System. Addison-Wesley.

Sun Microsystems Inc., 2001. Java embedded server: a white paper. Available from <http://www.sun.com/software/embeddedserver/whitepapers/index.html>.

Tokuda, H., Nakajima, T., Rao, P., 1990. Real-Time mach: towards a predictable real-time system. In: Proceedings of the USENIX Mach Workshop, pp. 73–82.

Veitch, C., Hutchinson, N.C., 1996. Kea—a dynamically extensible and configurable operating system kernel. In: Proceedings of the 3rd International Conference on Configurable Distributed Systems, pp. 236–242.

Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L., 1993. Efficient software-based fault isolation. In: Proceedings of the 14th ACM symposium on Operating Systems Principles, pp. 203–216.

Yang, C.W., Lee, C.H., Chang, R.C., 1999. Lyra: a system framework in supporting multimedia applications. In: Proceedings of the 1999 IEEE International Conference on Multimedia Computing and Systems, pp. 204–208.

Zuberi, K.M., Shin, K.G., 1996. EMERALDS: a microkernel for embedded real-time systems. In: Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium, pp. 241–249.

Da-Wei Chang was born on July 24, 1973. He received his B.S., M.S., and Ph. D. degrees in Computer and Information Science from National Chiao Tung University, Hsinchu, Taiwan, R.O.C., in 1995, 1997 and 2001 respectively. He is currently a postdoc in Computer and Information Science at National Chiao Tung University. His research interests include operating systems, embedded system design, and Java.

Ruei-Chuan Chang was born on January 30, 1958. He received his B.S. degree (1979), his M.S. degree (1981), and his Ph.D. degree (1984), all in Computer Engineering from National Chiao Tung University. He is currently a professor in Computer and Information Science at National Chiao Tung University, Taiwan, R.O.C. He is also an Associate Research Fellow at the Institute of Information Science, Academia Sinica, Taipei. His research interests include operating systems, wireless communication technologies, and embedded systems.