JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 26, 39-55 (2010)

A Server-side Pre-linking Mechanism for Updating Embedded Operating System Dynamically*

BOR-YEH SHEN AND MEI-LING CHIANG+
Department of Computer Science
National Chiao Tung University
Hsinchu, 300 Taiwan
E-mail: byshen@cs.nctu.edu.tw
+*Department of Information Management*
National Chi Nan University
Puli, Nantou, 545 Taiwan
E-mail: joanna@ncnu.edu.tw

To allow embedded operating systems to update their components on-the-fly, dynamic update mechanism is required for operating systems to be patched or added extra functionalities in without the need of rebooting the machines. However, embedded environments are usually resource-limited in terms of memory size, processing power, power consumption, and network bandwidth. Thus, dynamic update for embedded operating systems should be designed to make the best use of limited resources. In this paper, a server-side pre-linking mechanism is proposed to make dynamic updates of embedded operating system efficiently. Applying this mechanism can reduce not only memory usage and CPU processing time for dynamic update, but also data transmission size for updated components. Power consumption can be reduced as well. We have implemented this mechanism in LyraOS which is a component-based embedded operating system. Performance evaluation shows that the size of updated components applying the proposed dynamic update mechanism can be 65-86% smaller than applying the approach of Linux loadable kernel modules. Especially, the overheads in embedded clients are minimal since the component linking time in embedded clients is eliminated.

Keywords: embedded system, operating system, dynamic update, modules, memory protection

1. INTRODUCTION

Component-oriented development methodology has become very popular in embedded operating system design since embedded applications and hardware devices are more and more versatile and complex. Through the support of component libraries, embedded operating systems can be configured to meet the requirements of versatile hardware devices and different application needs. In this way, operating system developers or researchers can focus on their interested components without understanding the whole operating systems. The development time of an embedded system can thus be decreased.

Recently, providing dynamic component update is a critical design trend in component-based operating systems. Dynamic component update allows operating systems to update their components on-the-fly without rebooting the whole systems or stopping any

Received September 30, 2008; accepted January 7, 2009.
Communicated by Sung Y. Shin, Jiman Hong and Tei-Wei Kuo.
* This work was supported in part by the National Science Council of Taiwan, under grants No. NSC 95-2221-E-260-017 and NSC 96-2628-E-260-009.

system services. This opens up a wide range of opportunities: fixing bugs, upgrading services, improving algorithms, adding extra functionalities, runtime optimization, *etc.* Although many operating systems have already supported different kinds of mechanisms to extend their kernels, they usually do not aim at resource-limited environments. For instance, Linux uses a technique called loadable kernel modules (LKMs) [1]. By using this technique, Linux can load modules, such as device drivers, file systems, or system call to extend the kernel at run time. However, LKMs may take lots of overheads in embedded environments.

Since embedded systems are usually resource limited, in order to keep the added overheads minimal while providing dynamic update in an embedded operating system, we propose the server-side pre-linking mechanism which is a client-server model. Under this mechanism, components are pre-linked and stored in the server hosts. Embedded clients that request components do not need to perform dynamic component linking when new components are added into kernel, which saves the linking overhead in embedded clients. Component linking is actually performed on the server before the components are requested by clients, which saves the processing time on server hosts when the components are requested by clients. Applying this server-side pre-linking mechanism, the size of the components on server hosts can be reduced since they are pre-linked and do not need symbol table information for relocation and linking. Therefore, memory usage and data transmission size for updated components can be also reduced. Besides, CPU processing time for dynamic update and power consumption can be decreased as well.

To demonstrate the feasibility of the proposed dynamic component update and component protection mechanisms, we have designed and implemented them in LyraOS [2-4] operating system. LyraOS is a research operating system designed for embedded systems, which uses component-oriented design in the system development. However, just like many embedded operating systems such as eCos [5] and MicroC/OS-II [6], originally, LyraOS can only be statically configured at source-code level and the system cannot be updated or extended on-the-fly. Performance evaluations show that the sizes of the components applying the proposed dynamic update mechanism can be 65-86% smaller than the ones applying the approach of Linux loadable kernel modules. The size of the loader responsible for loading downloaded components into embedded kernel in LyraOS is only about 1% and 7% as compared with that of the loader for Linux loadable kernel module of Linux 2.4 and Linux 2.6. The component loading time also takes only a few milliseconds since embedded clients do not need to perform dynamic component linking. The component invocation time also adds only a few overheads caused by providing dynamic component exported interface and memory protection for un-trusted components.

Although the proposed dynamic component update and component protection mechanisms are implemented in LyraOS operating system, these experiences can serve as the reference for other component-based embedded operating systems that require an efficient and safe mechanism to dynamically update their components.

The rest of this paper is organized as follows. Section 2 introduces the LyraOS operating system. Section 3 details the design and implementation of the proposed dynamic update mechanism. Section 4 shows performance evaluation results. Section 5 introduces the related work and section 6 concludes this paper.

2. LYRAOS

LyraOS [2-4] is a component-based operating system which aims at serving as a research vehicle for operating systems and providing a set of well-designed and clear-interface system software components that are ready for Internet PC, hand-held PC, embedded systems, *etc*. It was implemented mostly in C++ and few assembly codes. It is designed to abstract the hardware resources of computer systems such that low-level machine dependent layer is clearly cut from higher-level system semantics. Thus, it can be easily ported to different hardware architectures [4].

Fig. 1 shows system architecture of LyraOS. Each system component is completely separate, self-contained, and highly modular. Components in LyraOS can be statically configured at source-code level. In addition to being light-weight system software, it is a time-sharing multi-threaded microkernel. Threads can be dynamically created and deleted, and thread priorities can be dynamically adjusted. Besides, it provides a preemptive prioritized scheduling and supports various mechanisms for passing signals, semaphores, and messages between threads. However, just like many embedded operating systems such as eCos [5] and MicroC/OS-II [6], it is a single-address-space operating system and runs only in kernel mode without memory protection.

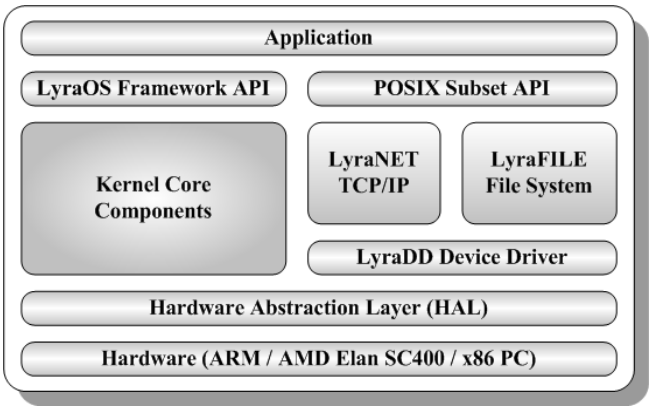

Fig. 1. LyraOS system architecture.

On top of the microkernel, a micro window component with Windows OS look and feel is provided [2]. Besides, the LyraFILE component [7], a light-weight VFAT-based file system, supports both RAM-based and disk-based storages. Especially, LyraOS provides the Linux device driver emulation environment [8, 9] to make use of Linux device drivers. Under this emulation environment, Linux device driver codes can be integrated into LyraOS without modification. Furthermore, the LyraNET [10] component, a TCP/IP protocol stack derived from Linux TCP/IP codes [11], is provided. For adapting into embedded systems, LyraNET is implemented with the zero-copy mechanism for reducing protocol processing overhead and memory usage.

3. DESIGN AND IMPLEMENTATION

This section presents the proposed dynamic component update mechanism in LyraOS in details. It includes the implementation of server-side pre-linking and dynamic update in embedded clients.

3.1 Architecture Overview

According to the implementation of the component-based LyraOS operating system, an updatable unit may be a set of functions and global variables or an encapsulation of data members and methods. In both cases, software developers usually need to define a clear interface to the unit or make the unit inherit the interface from a virtual base class. Originally, other components should invoke the unit only through the static interface.

In this research, the proposed dynamic component update mechanism is implemented in LyraOS. Components are executable and linkable format (ELF) [12] files, and they can be a set of functions, global variables, or C++ classes. They do not have to use static interface. The only thing that updatable components need to do is to register their exported methods to the component manager. Then, the external components will invoke these methods through the component manager.

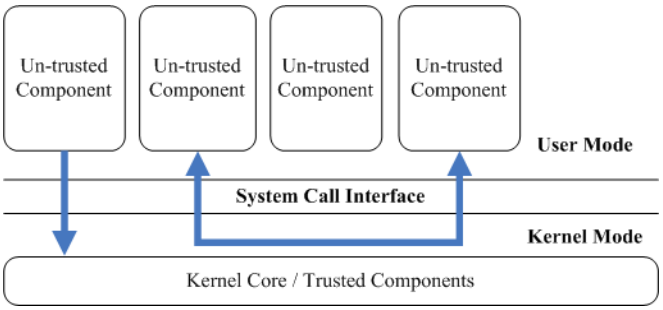

Fig. 2. Trusted and un-trusted components.

Additionally, to save system overhead while making LyraOS system more flexible and safe, all of the updatable components are separated into trusted components and un-trusted components as well. In order to avoid downloading un-trusted components to cause system crash, the original LyraOS is divided from single mode into user and kernel modes like Fig. 2. Trusted components are located in kernel mode and can invoke system services directly. Un-trusted ones are located in user mode and run in different protection domains enforced by hardware memory protection. Components permit system services invocation and communicate with other ones only through the system call invocation when they are un-trusted.

In LyraOS system using the proposed server-side pre-linking mechanism, all the dynamically updatable components are located on the server host and are pre-linked. A component server running on the server-side is responsible for responding to clients' requests and then loading and transmitting these pre-linked components to the embedded

clients. A dynamic loader called LyraLD within the operating system kernel on the embedded client is responsible for downloading and installing pre-linked components. A component manager manages all of the components on the client-side and provides an interface for client-side applications to add, remove, or invoke them. For example, if an embedded client wants to add a new functionality, it will send a request through the component manager interface to the LyraLD. LyraLD will then send a request to a remote component server to download a new component. The component server will respond with a pre-linked component which provides the functionality requested. Finally, the LyraLD will download and install this component directly without the need of linking or relocation.

3.2 Server-side Pre-linking

Since embedded environments are usually resource-limited, the server-side component pre-linking mechanism is proposed and implemented to keep the imposed overheads minimal while providing dynamic component update in an embedded operating system.

As mentioned in section 3.1, components in our design and implementation have been linked on the server host before they are requested by embedded clients. These components are linked according to their types (*i.e.*, trusted or un-trusted) and symbol tables of embedded clients. The trusted component will be linked with the kernel symbol of the embedded client while the un-trusted one will be linked with the user library symbol table of the client. Especially, we do not need to know where the component will reside in the embedded client's memory (*i.e.*, the starting address of the component). All of the updatable components will be linked at the same starting virtual address through the linker script we have defined. Then the components will be relocated by embedded client's relocation hardware that will be described later. Because the updatable components can be linked in a prior time instead of on demand, the component processing time on the server host can be saved when they are requested by clients.

Fig. 3 shows the proposed server-side pre-linking architecture. The component server located on the server host receives requests from the embedded client kernels and performs tasks as follows. If a pre-linked component is found in the pre-linked component storage, the component server will send the pre-linked component to the embedded clients immediately. Otherwise, the component server will link the components on demand.

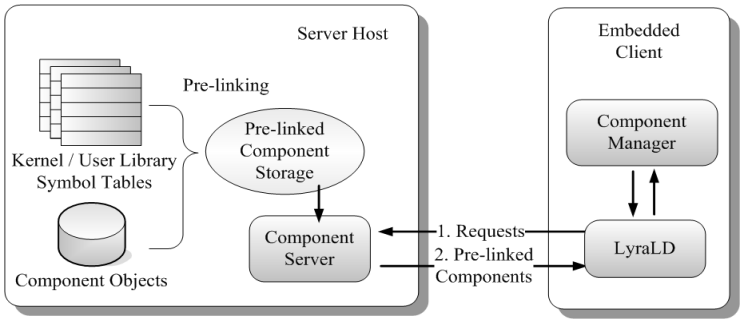

Fig. 3. Server-side pre-linking architecture.

The merits of the proposed approach can be summarized as follows. This server-side component pre-linking mechanism can save not only the memory and the disk storage on embedded clients but also the component transmitting time. This is because the sizes of updatable components are reduced since components are pre-linked and do not need symbol table information for relocation and linking. Besides, it eliminates the need for clients to perform dynamic linking, which can reduce processing overhead for embedded clients. The power consumption of embedded devices can thus be decreased.

3.3 Client-side Loading

A dynamic component loader called LyraLD and a component manager are developed in LyraOS to perform dynamic component loading and component management. Both the LyraLD and the component manager reside in the kernel level. Currently, the LyraLD uses the trivial file transfer protocol (TFTP) [13] to download pre-linked components from the component server.

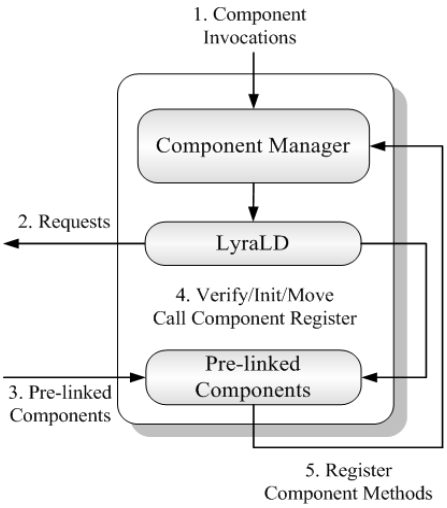

Fig. 4. Client-side loading.

Fig. 4 shows the steps of client-side component loading and installing. First, the component manager receives an invocation request to load a new component. Second, the component manager checks whether the component exists or not. If the component is not found in the client side, the component manager will call LyraLD to send a request to the remote component server to download this component. Third, the LyraLD downloads a pre-linked component image returned from the remote component server to the client-side memory. Fourth, after the LyraLD reads the pre-linked component image's header from the memory address where the image is located, the LyraLD will verify the pre-linked component image, initialize component environments, and move each section of the image to the virtual address that the ELF header specified. Finally, the LyraLD will jump to the entry address of the component image to execute the component's ini-

tialization function that registers the component exported methods to the component manager.

Table 1 shows our component manager API. Components can be added, removed, updated, and invoked through these APIs. In order to provide dynamic component exported interface, the register method can register the component exported methods to the component method vector table when a component is loaded. As a component is downloaded and loaded into memory, the LyraLD will get the entry point address from the header of the component and then jump to this address to perform the registration of component's methods.

Table 1. Component manager API.

Methods	Descriptions
CM::Add(name, ver)	The CM::Add() method adds a new component name with version ver from a remote component server and returns component's ID.
CM::GetCID(name, ver)	The CM::GetCID() method returns component ID of component name (version ver).
CM::Invoke(cid, mid, arg)	The CM::Invoke() method invokes a method mid of a component cid and passes arguments arg through the component manager.
CM::Register (mid, fptr)	The CM::Register() method registers method's ID mid and its method address fptr to the component manager.
CM::Remove(cid)	The CM::Remove() method removes component whose ID is cid.
CM::Update(old, new)	The CM::Update() method updates a component from component ID old to component ID new.

Fig. 5 shows our component interface. This function would be implemented by developers and will be linked as the entry point of updatable components during server-side pre-linking. Every updatable component has to implement this interface to register its methods and transfer its states. As the component jumps to the entry point, the component will invoke the register method to register its exported methods to the component manager. Therefore, other components can invoke these methods through the component manager without using static component interface. When a component is to be removed, all of the component's information including current states and function pointers of the component exported methods should be removed, too.

```
function entry(Opt, Addr)

switch(Opt)
begin

case REGISTER:
CM::Register(1, functionA);
CM::Register(2, functionB);
// ......
break;
```

```
case IMPORT:
      // convert and import
      // component states from Addr
      break;

case EXPORT:
// export states of this component
// return address of export states
break;

end

end function
```

Fig. 5. Component interface.

Methods in Table 1 provide the component communication interface. Components must communicate with each other through the component manager. This is because we provide dynamic component exported interface and these interfaces of components are managed by the component manager. When invoking other components' methods, it is necessary to pass a component ID, a method ID, and arguments to the component communication interface.

3.4 Component Relocation

The component relocation in our system implementation takes advantage of the ARM fast context switch extension (FCSE) mechanism [14, 15]. The FCSE is an extension in the ARM MMU. It modifies the behavior of an ARM memory translation. By taking advantage of this feature, we make each component have its own address space and relocate in the first 32MB of memory. As shown in Fig. 6, there is only one page table in our system. The 4GB virtual address space is divided into 128 blocks, each of size 32MB. Each block can contain a component which has been compiled to use the address ranging from 0x00000000 to 0x01FFFFFF. Each block is identified with a PID (Process ID) register. PID is a 7-bit number that identifies which block the current component is loaded into. Through the FCSE mechanism, we can switch between components' address spaces by changing the PID register and do not have to flush caches and TLBs. The same functionality can be achieved by other architectures which provide paging and an address space identifier (ASID) found on many RISC processors such as Alpha, MIPS, PA-RISC, and SPARC.

However, there is a critical problem about communication among components. Since every component has its own address space, a component cannot pass a pointer-type argument that is pointed to another address space. Due to this reason, a shared memory mechanism is used to solve this problem. A memory region which is greater than 32MB is reserved to store data that the argument points to. This is due to the fact that if an address is greater than 32MB, it will not be modified by FCSE. This means that the address space of components from 32MB to 4GB is shared. This also allows components to directly access the kernel core or user libraries which are out of the first 32MB without changing PID or page tables.

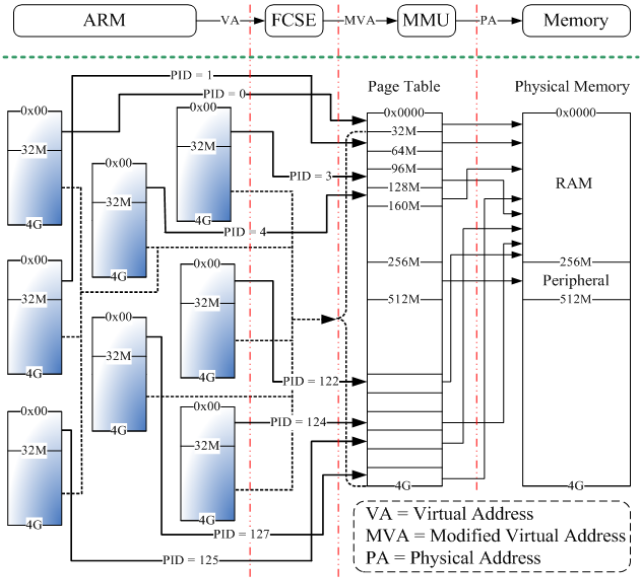

Fig. 6. Relocation by FCSE mechanism.

3.5 Component Protection

The ARM architecture provides a domain mechanism [14, 15] to make different protection domains running with the same page table. We use this mechanism to make each un-trusted component have its own protection domain. A domain access control register (DACR) can be used to control the access permissions of components.

Currently, each un-trusted component's first descriptors of the page table are associated with one of the sixteen domains and its own DACR status. The DACR describes the status of the current component with respect to each domain. Since trusted components are the components that have been verified, they can use the same protection domain as kernel core and run in the kernel mode. However, although un-trusted components run in the user mode, they may also have vicious codes to affect other un-trusted components. Therefore, they should be located in different protection domains and use the client access types. Thus, the current un-trusted components will not be affected as a new un-trusted component is loaded into the system.

Although ARM only supports 16 domains which may be less than the number of un-trusted components concurrently in our system, other approaches such as domain recycling [16, 17] can be applied to resolve this problem.

4. PERFORMANCE

This section presents the performance evaluation of the proposed dynamic component update mechanism implemented in LyraOS operating system. The performance and space overheads of the proposed server-pre-linking mechanism are mainly evaluated.

Without server-side pre-linking mechanism, mechanisms like server-side linking [18] or client-side linking [1] must be applied for providing dynamic updates in embedded operating systems. In client-side linking, component linking should be done on client side when the component is requested. In contrast, in server-side linking, component linking can be done on server side when the component is requested. Differently, in the proposed server-side pre-linking, component linking can be done on server side in a prior time before the component is requested. Because embedded Linux is used popularly in embedded devices, thus, our experiments use Linux loadable kernel modules as the representative of client-side linking mechanism. The comparison of using client-side linking, server-side linking, and server-side pre-linking mechanisms is briefly listed in Table 2.

Table 2. Comparison of component loading and linking under various mechanisms.

	Client-side Linking Mechanism	Server-side Linking Mechanism	Server-side Pre-linking Mechanism
Component linking	Done on client on demand	Done on server host on demand	Done on server host in a prior time
Component loading	Done on client	Done on client	Done on client

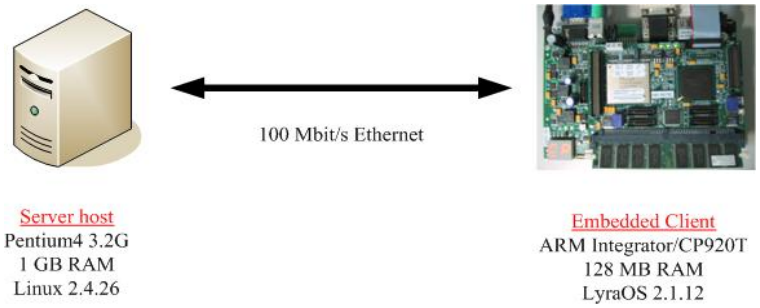

Fig. 7. Experimental environment.

4.1 Experimental Environment

Fig. 7 shows our experimental environments. The experimental hardware consists of a client and a server host that are connected via a 100 Mbits/sec Ethernet. The server host is a Pentium 4 3.2GHz PC with 1GB RAM, running Linux 2.4.26. The client host is an ARM Integrator/CP920T development board with 128 MB RAM, running LyraOS 2.1.12.

4.2 Comparison of Space Overheads

Table 3 shows the loader sizes of the client kernel. The size of LyraLD is compared with the sizes of Linux LKMs linker/loader under kernel version both 2.4 and 2.6. The fundamental difference between Linux 2.4 and Linux 2.6 is the relocation and linking of kernel modules are done in the user level or kernel level. Loadable kernel modules in Linux are ELF object files which can be loaded by a user program called **insmod**. In

Linux 2.4, the **insmod** does all the work of linking Linux kernel module to the running kernel. While the linking is done, it generates a binary image and then passes it to the kernel. In Linux 2.6, the **insmod** is a trivial program that only passes ELF objects directly to the kernel, and then the kernel does the linking and relocation.

In Table 3, the Linux 2.4 module linker/loader shows the static and dynamic size of the **insmod** program on Linux 2.4.26. The Linux 2.6 module linker and module loader show the object sizes of module linker and module loader which were measured from the object files of **kernel/module.c** and **kernel/kmod.c** in the Linux 2.6.19 source tree. All symbols in these programs and object files have already been stripped. The result shows that the size of LyraLD is less than 1% of the module linker/loader under Linux 2.4 and is about 7% of the module linker/loader under Linux 2.6.

Table 3. Sizes of loaders.

Loader	Object Code Size	
Linux 2.4 module linker/loader	618,712 bytes	(static linked)
	133,140 bytes	(dynamic linked)
Linux 2.6 module linker	14,088 bytes	(kernel/module.o)
Linux 2.6 module loader	2,060 bytes	(kernel/kmod.o)
LyraLD (LyraOS loader)	1,140 bytes	

Table 4. Kernel and symbol sizes.

Items	Size
Linux 2.6.19 kernel image (vmlinux)	1,219,296 bytes
Linux 2.6.19 kernel image (zImage)	1,181,932 bytes
Linux 2.6.19 symbol table	505,487 bytes
LyraOS kernel image	35,752 bytes
LyraOS kernel symbol table	24,850 bytes

In addition, to perform the dynamic linking, Linux also requires the kernel symbol table to be stored on the client host. The size of the symbol table is dependent on the client-side kernel. Table 4 shows that the kernel symbol table of LyraOS is about 24 Kbytes. It occupies almost 70% of the LyraOS kernel size. The kernel symbol table of Linux 2.6.19 is about 494 Kbytes. It occupies about 40% of the Linux kernel size.

Table 5 shows the component space overheads of the task scheduler, the interrupt handler, the timer driver, the serial driver, the signal, and the semaphore components in LyraOS and Linux. In this table, the column of Linux shows the sizes of ELF object files of these components under the Linux LKMs approach. The column of LyraOS shows the size of pre-linked images of these components under the LyraOS server-side pre-linking approach. The numbers in parentheses are the ratios of component overheads under LyraOS to those under the Linux LKMs.

The result shows that the sizes of components under the LyraOS approach are only about 14-35% of the sizes under the Linux LKMs approach. This is because the LKMs mechanism contains more overheads for dynamic linking, such as symbol tables, string tables, relocation data, and other data structures. In contrast, components under the LyraOS server-side pre-linking approach are pre-linked and thus do not need symbol table information for component relocation and linking.

4.3 Component Loading/Pre-linking Time

Table 6 shows the component client-side loading and server-side pre-linking time of those components described above. The component loading takes only a few milliseconds. Tables 5 and 6 show that the component loading time is not related to the sizes of the components. This is because the loader has to initialize some of the ELF sections. For example, BSS (Block Started by Symbol) is a memory section where uninitialized C/C++ variables are stored. If there is a BSS section in a component, it needs to clear the memory section to zero while the component is loaded into memory.

Table 5. Component overheads.

Components	Under Linux LKMs approach (A)	Under LyraOS server-side pre-linking approach (B)	Ratio (B/A)
Task scheduler	4280 bytes	604 bytes	(14%)
Interrupt handler	7544 bytes	1612 bytes	(21%)
Timer driver	4424 bytes	992 bytes	(22%)
Serial driver	5640 bytes	1324 bytes	(23%)
Signal	7768 bytes	2736 bytes	(35%)
Semaphore	4116 bytes	632 bytes	(15%)

Table 6. Component loading and pre-linking time.

Components	Client-side Component Loading Time	Server-side Component Pre-linking Time
Task scheduler	20.31ms	26ms
Interrupt handler	31.17ms	25ms
Timer driver	39.86ms	35ms
Serial driver	30.44ms	32ms
Signal	22.04ms	29ms
Semaphore	20.32ms	28ms

From the server-side pre-linking time we can also see that embedded clients can save lots of time when new components are loaded since the linking has been done previously on the server. In particular, the server host performing the server-side pre-linking runs on a Pentium4 3.2GHz machine whereas the frequency of embedded client's ARM920T processor is only about 200MHz. If the component linking is performed on the embedded clients, it could cause large overheads for processing, storage, and power consumption.

Table 7 compares the client-side component loading and processing time when components are requested by clients under server-side pre-linking mechanism and server-side linking mechanism. The component processing time under server-side pre-linking mechanism is only about 42-55% of the time under the server-side linking mechanism since components are linked in a prior time instead of on demand. This shows that component pre-linking can greatly reduce lots of time when components are loaded and processed on embedded clients. Without server-side pre-linking, embedded clients need to spend 80-132% more time to wait for server-side component linking. If without server-

side pre-linking or server-side linking, embedded clients need to perform component loading and component linking all by themselves. This client-side linking would incur embedded clients lots of overheads in CPU processing and power consumption, space overhead for symbol tables in component linking is also needed.

Table 7. Component loading and processing time.

Components	Using Server-side Prelinking Mechanism (A)	Using Server-side Linking Mechanism (B)	Ratio (A/B)
Task scheduler	20.31ms	> 46.31ms	< 44%
Interrupt handler	31.17ms	> 56.17ms	< 55%
Timer driver	39.86ms	> 74.86ms	< 53%
Serial driver	30.44ms	> 62.44ms	< 49%
Signal	22.04ms	> 51.04ms	< 43%
Semaphore	20.32ms	> 48.32ms	< 42%

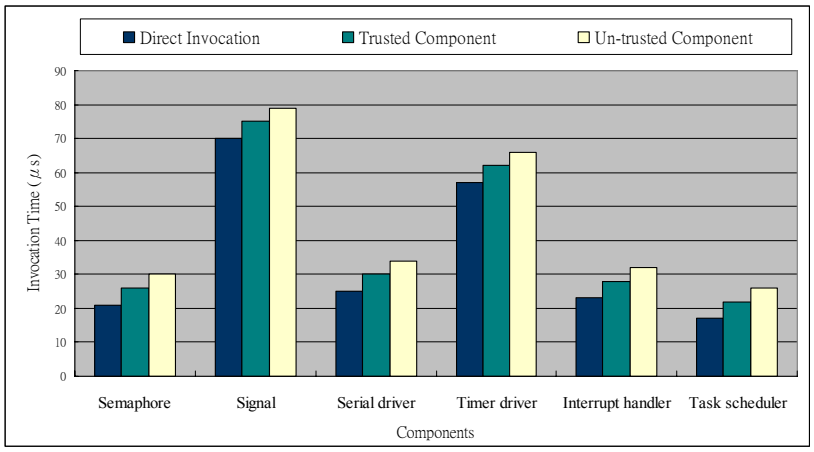

Fig. 8. Component invocation time.

4.4 Component Invocation Time

Fig. 8 shows the component invocation time. We invoke a method of each component described above. In Fig. 8, "Direct Invocation" measures the invocation time of the direct component invocation. That is, direct component invocation invokes methods directly without calling the component manager and system calls. "Trusted Component" measures the invocation time of the trusted component invocation through the component manager. "Un-trusted Component" measures the invocation time of the un-trusted component invocation through the system call and the component manager. The result shows that it adds only a few overheads by providing dynamic component exported interface and memory protection for un-trusted components. Besides, relocation by hardware (*i.e.*, FCSE) also keeps the overhead of switching between components' address space minimal.

5. RELATED WORK

There are many researches of dynamic update provided in operating systems. Linux Loadable Kernel Modules (LKMs) [1] are object files that contain codes to extend the running kernel. They are typically used to add support for new hardware, file systems, or for adding system calls. When the functionality provided by an LKM is no longer required, it can be unloaded. Linux uses this technology to extend its kernel at run time. However, Linux modules can be removed only when they are inactive. Another problem of LKMs is space overheads. It needs additional kernel symbol table in client site and additional symbol table in loadable modules due to dynamic symbol linking. Dynamic symbol linking also takes lots of time during module loading. In contrast, the proposed server-side pre-linking approach can eliminate these overheads and component sizes are smaller since symbol tables are not needed in client site and in loadable components since components have been per-linked. Besides, the LKMs require privilege permission to perform kernel modules loading. All of these modules are located in the kernel level and have the same permission as kernel. If a vicious module is loaded in the kernel, operating system may crash.

The operating system portal (OSP) [18] is a framework which adopts a client-server model to make an embedded kernel extensible while keeping the added overheads minimal. In OSP framework, all the dynamically loadable modules are located on the server host. A user-level 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. The proposed server-side pre-linking mechanism is similar to the server-side linking mechanism proposed in OSP framework. Unlike the OSP framework, the proposed server-side pre-linking mechanism does not have to negotiate between client and server to know the starting address of components on each client host because components will be relocated by client' relocation hardware. Component linking can be performed on the server side previously before components are requested by clients. Thus, the components processing time on server hosts can also be saved since it does not need to link components for each request of clients.

K42 [19-22] is an open-source research operating system developed for cache-coherent 64-bit multiprocessor systems. It is implemented in C++ language and uses a modular object-oriented design. By using the hot-swapping mechanism, K42 allows an object instance to be transparently switched to another implementation while system is running. Dynamic update is an extension of hot-swapping, and supports changing every object of a specific class. However, the exported interfaces of updatable C++ classes are fixed. Besides, it does not aim for embedded environments and requires much more resources. Although K42 provides a powerful dynamic update mechanism to switch C++ object instances transparently and change every object of a specific class on-the-fly, it may add much more overhead and result in the degradation of the system performance in embedded environments.

Dynamic C++ class [23] is a class whose implementation can be dynamically changed during program execution. Operating systems can apply this mechanism to make C++ classes updatable. However, each implementation must have a static interface. It is not flexible enough for developers to implement components. Furthermore, the old versions of components and the new versions of components are coexistent. When an

update occurs, it only loads components into systems. It is impossible to be applied in some of the critical operating system components since some of them (*e.g.*, task scheduler) can not coexist in operating systems.

SOS [24] is a dynamic operating system for mote-class sensor nodes. SOS uses dynamically loadable software modules to create a system supporting dynamic addition, modification, and removal of network services. The SOS kernel provides a set of system services that are accessible to the modules through a jump table in the program memory. Furthermore, modules can also invoke functions in another module. The SOS kernel provides a dynamic function registration service for modules. Modules can register functions that they provide with the SOS kernel. The kernel stores information regarding the dynamic functions in a function control block (FCB) data structure. Processes can use a system call to subscribe a function. Although flexible enough for system developers, the SOS kernel has to support complex symbol linking at run time. The dynamic symbol linking may cause lots of overheads. Besides, SOS only supports the module insertion and removal. It can not transfer state between old modules and new modules.

6. CONCLUSION

In this paper, a server-side pre-linking mechanism is proposed to make an embedded operating system more extensible while keeping the added overheads minimal. The embedded operating system can be updated dynamically without the need of dynamic linker and symbol table. Besides, the dynamic component exported interface can let developers change component exported interfaces easily. Furthermore, to save system overhead while making LyraOS more flexible and safe, components are separated into trusted and un-trusted ones, which run in different protection domains enforced by hardware memory protection.

After applying the proposed mechanisms in our target embedded operating system, LyraOS, the performance evaluations show that the loader size under LyraOS is only about 1% and 7% as compared with the Linux loadable kernel module of the Linux 2.4 and the Linux 2.6. The component overhead under LyraOS is only about 14-35% of the Linux loadable kernel module. Besides, the component loading time takes only a few milliseconds. The component invocation time also adds a few overhead caused by providing dynamic component exported interface and memory protection for un-trusted components.

In the future, we will address the component dependency problem and provide a demand loading mechanism. Such that, we need not download all dependent components, and these dependent components will be downloaded to the client host as they are needed.

REFERENCES

1. B. Henderson, “Linux loadable kernel module HOWTO,” http://www.tldp.org/HOWTO/Module-HOWTO/.
2. LyraOS homepage, http://163.22.34.199/joannaResearch/LyraOS/index.htm.

3. C. W. Yang, C. H. Lee, and R. C. Chang, "Lyra: A system framework in supporting multimedia applications," in *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, Vol. 2, 1999, pp. 204-208.
4. Z. Y. Cheng, M. L. Chiang, and R. C. Chang, "A component based operating system for resource limited embedded devices," in *Proceedings of IEEE International Symposium on Consumer Electronics*, 2000, pp. 27-31.
5. eCos, http://sources.redhat.com/ecos/.
6. MicroC/OS-II, http://www.ucos-ii.com/.
7. M. L. Chiang and C. R. Lo, "LyraFILE: A component-based VFAT file system for embedded systems," *International Journal of Embedded Systems*, Vol. 2, 2006, pp. 248-259.
8. C. H. Chen, "LyraDD: Design and implementation of the device driver model for embedded systems," Master Thesis, Department of Information Management, National Chi Nan University, 2004.
9. C. W. Yang, P. C. H. Lee, and R. C. Chang, "Reuse Linux device drivers in embedded systems," in *Proceedings of the 1998 International Computer Symposium*, 1998, pp. 260-267.
10. M. L. Chiang and Y. C. Lee, "LyraNET: A zero-copy TCP/IP protocol stack for embedded systems," *Real-Time Systems*, Vol. 34, 2006, pp. 5-18.
11. D. A. Rusling, "The Linux kernel," http://www.tldp.org/LDP/tlk/tlk.html, 2002.
12. Tool Interface Standard (TIS), Executable and Linking Format (ELF) Specification, Version 1.2, 1995, http://www.x86.org/ftp/manuals/tools/elf.pdf.
13. K. Sollins, "The TFTP protocol (Revision 2)," http://www.ietf.org/rfc/rfc1350.txt, 1992.
14. S. Furber, *ARM System-on-Chip Architecture*, 2nd ed., Addison-Wesley, Great Britain, 2000.
15. D. Seal, *ARM Architecture Reference Manual*, 2nd ed., Addison-Wesley, Great Britain, 2001.
16. A. Wiggins and G. Heiser, "Fast address-space switching on the strongARM SA-1100 processor," in *Proceedings of the 5th Australasian Computer Architecture Conference*, 2000, pp. 97-104.
17. A. Wiggins, H. Tuch, V. Uhlig, and G. Heiser, "Implementation of fast address-space switching and TLB sharing on the strongARM processor," in *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, 2003, pp. 352-364.
18. D. W. Chang and R. C. Chang, "OS portal: An economic approach for making an embedded kernel extensible," *Journal of Systems and Software*, Vol. 67, 2003, pp. 19-30.
19. C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis, "System support for online reconfiguration," in *Proceedings of USENIX Annual Technical Conference*, 2003, pp. 141-154.
20. A. Baumann, J. Appavoo, D. D. Silva, O. Krieger, and R. W. Wisniewski, "Improving operating system availability with dynamic update," in *Proceedings of Workshop on Operating System and Architectural Support for the On-Demand IT InfraStructure*, 2004, pp. 21-27.
21. A. Baumann, J. Kerr, J. Appavoo, D. D. Silva, O. Krieger, and R. W. Wisniewski,

"Module hot-swapping for dynamic update and reconfiguration in K42," in *Proceedings of Linux.conf.au* (*LCA*), 2005.
22. A. Baumann, J. Appavoo, D. D. Silva, J. Kerr, O. Krieger, and R. W. Wisniewski, "Providing dynamic update in an operating system," in *Proceedings of the USENIX Annual Technical Conference*, 2005, pp. 279-291.
23. G. Hjálmtýsson and R. Gray, "Dynamic C++ classes – A lightweight mechanism to update code in a running program," in *Proceedings of the USENIX Annual Technical Conference*, 1998, pp. 65-76.
24. C. C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *Proceedings of the 3rd International Conference on Mobile Systems*, *Applications and*, *Services*, 2005, pp. 163-176.

Bor-Yeh Shen (沈柏曄) received his M.S. degree in Information Management from National Chi Nan University, Puli, Taiwan, in 2006. He is currently working toward his Ph.D. degree at National Chiao Tung University, Hsinchu, Taiwan. His research interests include compilation for embedded systems, embedded operating systems, computer architectures, and virtual machines.

Mei-Ling Chiang (姜美玲) received her B.S. degree in Management Information Science from National Cheng Chi University, Taipei, Taiwan, in 1989. She received the M.S. degree in 1993 and her Ph.D. degree in 1999 in Computer and Information Science from National Chiao Tung University, Hsinchu, Taiwan. Now she is an Associate Professor in the Department of Information Management at National Chi Nan University, Puli, Taiwan. Her current research interests include operating systems, embedded systems, and clustered systems.