A Server-side Pre-linking Mechanism for Updating Embedded Clients Dynamically

Bor-Yeh Shen[1] and Mei-Ling Chiang[2]

[1]Department of Computer Science,
National Chiao Tung University, Hsinchu, Taiwan, R.O.C.

[2]Department of Information Management,
National Chi-Nan University, Puli, Taiwan, R.O.C.

byshen@cs.nctu.edu.tw[1], joanna@ncnu.edu.tw[2]

Abstract. 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, we have proposed a server-side pre-linking mechanism 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 update components. Power consumption can be reduced as well. Performance evaluation shows that compared with the approach of Linux loadable kernel modules, the size of update components can be reduced about 14-35% and the overheads in embedded clients are minimal.

Keywords: Embedded System, Operating System, Dynamic Update, Modules, LyraOS.

1 Introduction

Dynamic update allows operating systems to update their components on-the-fly without rebooting the whole systems or stopping any 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 similar to the server-side linking mechanism proposed in the operating system portal (OSP) framework [2]. Unlike the OSP framework, our server-side pre-linking mechanism does not have to negotiate between client and server to know the starting address of components on client hosts. Besides, we can perform component linking on the server-side before components are requested by clients. Thus, we can also save the components processing time on server hosts.

To demonstrate the feasibility of our proposed dynamic component update and component protection mechanisms, we have designed and implemented this mechanism in LyraOS [3] 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 [4] and MicroC/OS-II [5], LyraOS can be only statically configured at source-code level, so system cannot be updated or extended on-the-fly. Performance evaluation shows 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 sizes under LyraOS are only about 14-35% of the Linux loadable kernel module. The component loading time also takes a few milliseconds. 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 our proposed dynamic component update and component protection mechanisms are implemented in LyraOS operating system, we believe that 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 introduces the related work. Section 4 details the design and implementation of our dynamic update mechanism. Section 5 shows our performance evaluation results and Section 6 concludes this paper.

2 LyraOS

LyraOS [3] 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 clear cut from higher-level system semantics. Thus, it can be easily ported to different hardware architectures [6].

Figure 1 shows system architecture of LyraOS. Each system component is complete 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.
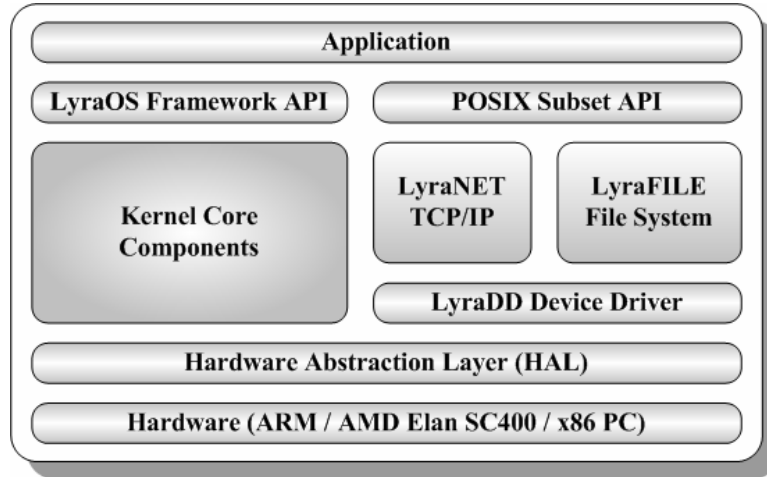
Fig. 1. LyraOS system architecture

3 Related Work

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 its 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. Our approach can eliminate these overheads. 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 kernels. Thus, operating systems may crash because a vicious module is loaded in the kernel.

In operating system portal (OSP) [2], 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 server-side linking mechanism proposed in OSP is similar to our server-side pre-linking mechanism. Unlike the OSP framework, our server-side pre-linking mechanism can perform component linking on the server-side previously before components are requested by clients. We do not have to know the starting address of components on each client host because components will be relocated by client's relocation hardware. Thus, the components processing time on server hosts can also be saved since we do not need to link components for each request of clients.

SOS [7] is a dynamic operating system for mote-class sensor nodes. It 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.

4 Design and Implementation

According to the implementation of our 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, we implement our proposed dynamic component update mechanism in LyraOS. In our system, components are executable and linkable format (ELF) [8] files and components can be a set of functions, global variables, or C++ classes. Components do not have to use static interface. The only one 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.

Additionally, to make our system more flexible and safe, we separate all of the updatable components into two groups, trusted components and un-trusted components. In order to avoid un-trusted components causing our system crash, we divide the original LyraOS from single mode into user and kernel modes. Trusted components are located in kernel mode and can invoke system services directly. Un-trusted components 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 components only through the system call invocation when they are un-trusted.

In our system, 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 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 components. For example, if an embedded client wants to add a new functionality, the embedded client will send a request through the component manager interface to the LyraLD. LyraLD will 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.

4.1 Server-side Pre-linking

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

As mentioned above, components in our design and implementation have been linked on the server-side before components 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 client-side 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 defined. Then the components will be relocated by the client-side relocation hardware that we will describe later. Because the updatable components can be linked in a prior time, we can save the component processing time on the server-side when components are requested.

Figure 2 shows our server-side pre-linking architecture. In our system, there is a component server on the server-side responsible for handling client requests. The component server on the server host receives request 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.

The merits of our approach can be summarized as follows. The server-side component pre-linking can save not only the memory and the disk storage on embedded clients but also the component transmitting time because we downgrade the sizes of updatable components. Besides, it eliminates the need for clients to perform dynamic linking. Furthermore, the power consumption of embedded devices can be also decreased.
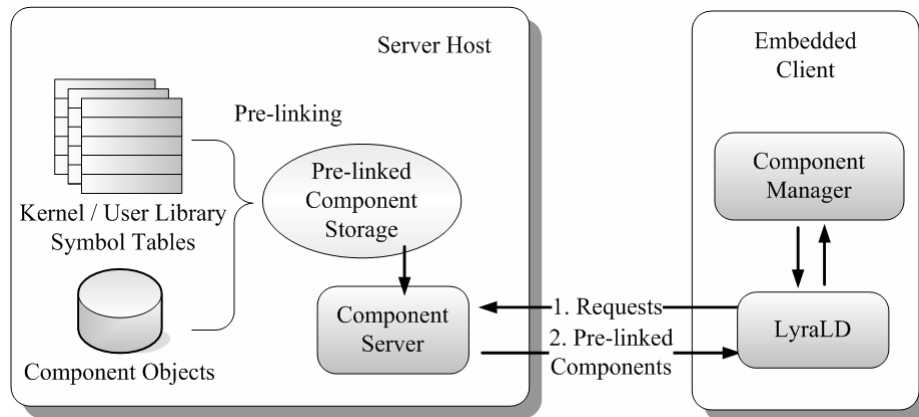

Figu. 2. Server-side pre-linking architecture

4.2 Client-side Loading

We develop a dynamic component loader called LyraLD and a component manager 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 use the trivial file transfer protocol (TFTP) [9] to download pre-linked components from the component server.

Figure 3 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 a 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 initialization 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.
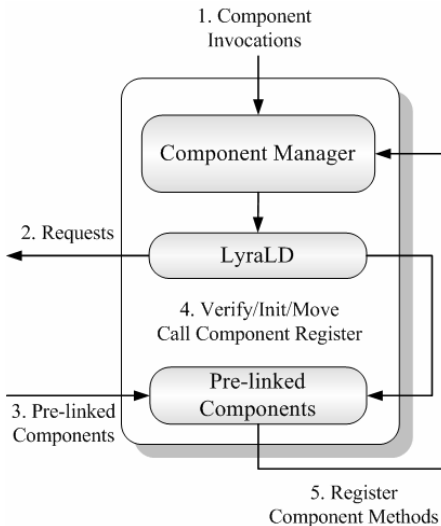

Fig. 3. Client-side loading

```
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. 4. Component interface

Table 1. Component manager API

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

Figure 4 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 we want to remove a component, all of the component information including current states of the component and function pointers of the component exported methods should be removed. Methods in Table 1 provide a component communication interface. In our system, components must communicate with each other through the component manager. This is because we provide dynamic component exported interface in our system and these interfaces of components are managed by the component manager.

4.3 Component Relocation

The component relocation in our system implementation takes advantage of the ARM fast context switch extension (FCSE) mechanism [10]. The FCSE is an extension in the ARM MMU. It modifies the behavior of an ARM memory translation. This modification allows our components to have their own first 32MB address space. Thus, we make each component have its own address space and relocate in the first 32MB of memory. As shown in Figure 5, 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 7-bit PID (Process ID) register. 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.
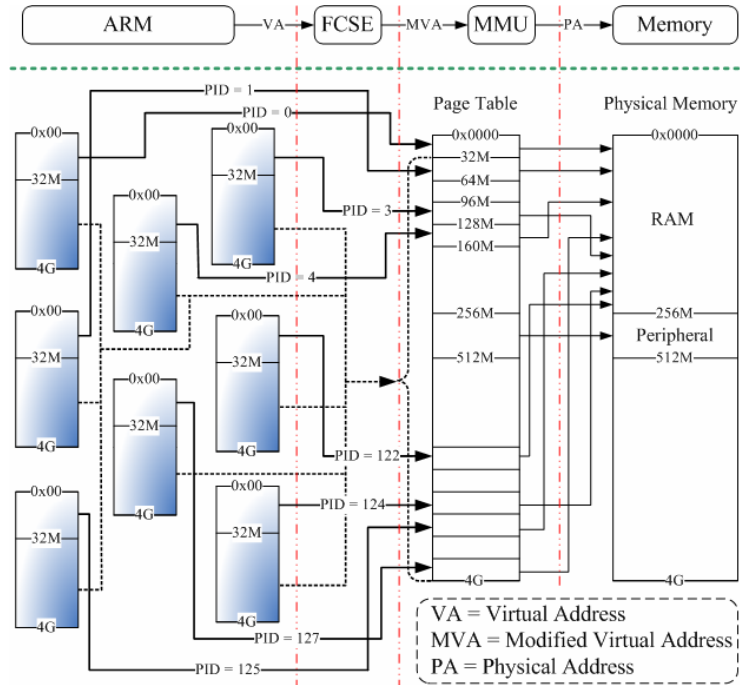
Fig. 5. Relocation by FCSE mechanism

However, there is a critical problem about communication among components. Since every component has an address space itself, we cannot pass component a pointer type argument that is pointed to another address space. Due to this reason, we use a shared memory mechanism to resolve 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 our kernel core or user libraries which are out of the first 32MB without changing PID or page tables.

4.4 Component Protection

The ARM architecture provides a domain mechanism [10] 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 in our system 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 locate in different protection domains and use the client access types. Thus, we can avoid the situation that the current un-trusted components will be affected as we load a new un-trusted component into our system. Although ARM only supports 16 domains which may be less than the number of un-trusted components concurrently in our system, we can apply other approaches such as domain recycling [11,12] to resolve this problem.

5 Performance

This section presents the performance evaluation of the proposed dynamic component update mechanism implemented in LyraOS. We compare the space overheads of our architecture with the Linux loadable kernel modules. The experimental environment 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.

5.1 Comparison of Space Overheads

Table 2 shows the loader sizes of the client kernel. We compare the size of LyraLD to 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, **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 2, 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 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. From the table we can see 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 2. 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 3. Kernel and symbol sizes

Items	Size
LyraOS kernel image	35,752 bytes
LyraOS kernel symbol table	24,850 bytes
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

Table 4. Component overheads

Components	Linux	LyraOS	Ratio
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 5. Component loading and pre-linking time

Components	Client-side Loading	Server-side Pre-linking
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

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. From Table 3 we can see that, the kernel symbol table of LyraOS is about 24 Kbytes in our system. It occupies almost 70% of the LyraOS kernel size. The kernel symbol table of Linux 2.6.19 in our system is about 494 Kbytes. It occupies about 40% of the Linux kernel size.

Table 4 shows the component space overheads of the task scheduler, the interrupt handler, the timer driver, the serial driver, the signal, and the semaphore component 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. From the table we can see 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.

5.2 Component Loading/Pre-linking Time

Table 5 shows the component client-side loading and server-side pre-linking time of those components we described above. The component loading only takes a few milliseconds. From the Table 4 and Table 5, we can see 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 is a memory section where un-initialized C/C++ variables are stored. If there is a BSS section in a component, it needs to clear to zero while the component is loaded into memory. Besides, from the server-side pre-linking time we can see that embedded clients save lots of linking time when new components are loaded since the linking has been done previously on the server. We should know that the server-side pre-linking runs on a Pentium4 3.2GHz machine, and the frequency of ARM920T processors is only about 200MHz. It could cause large overheads if the component linking is performed on the embedded clients.

5.3 Component Invocation Time

In Figure 6, we invoke a method of each component we described above. "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. From the figure we can see that, it only adds a few overheads by providing dynamic component exported interface and memory protection for un-trusted components. Besides, relocation by hardware also keeps the overhead of switching between components' address space minimal.
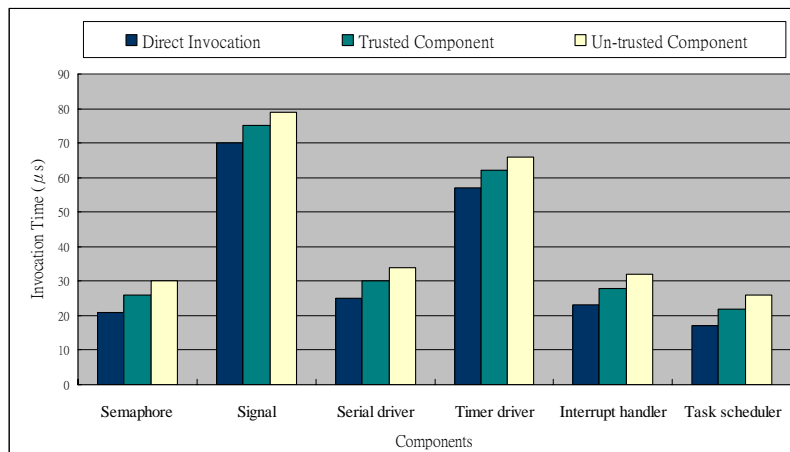

Fig. 6. Component invocation time

6 Conclusion

In this paper, we have proposed a server-side pre-linking mechanism to make an embedded operating system more extensible. The embedded operating system can be updated dynamically without the need of dynamic linker and symbol table. Besides, the dynamic component exported interface can make component developers change component exported interfaces easily. Furthermore, to make the system more flexible, components are separated into trusted components and un-trusted components, 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 evaluation shows 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. The component loading time also 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.

References

1. Linux Loadable Kernel Module HOWTO, http://www.tldp.org/HOWTO/Module-HOWTO/
2. Da-Wei Chang and Ruei-Chuan Chang.: OS Protal: an economic approach for making an embedded kernel extensible. Journal of Systems and Software, Vol. 67, No. 1, pp. 19–30, (2003)
3. LyraOS, http://163.22.34.199/joannaResearch/LyraOS/index.htm
4. eCos, http://sources.redhat.com/ecos/
5. MicroC/OS-II, http://www.ucos-ii.com/
6. Z. Y. Cheng, M. L. Chiang, and R. C. Chang.: A Component Based Operating System for Resource Limited Embedded Devices. IEEE International Symposium on Consumer Electronics, Hong Kong (2000)
7. C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava.: A Dynamic Operating System for Sensor Nodes. Proceedings of the 3rd International Conference on Mobile Systems, Applications and, Services, Seattle, WA, USA (2005)
8. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2, http://www.x86.org/ftp/manuals/tools/elf.pdf
9. The TFTP Protocol (Revision 2), http://www.ietf.org/rfc/rfc1350.txt
10. David Seal.: ARM Architecture Reference Manual Second Edition, Addison-Wesley (2001)
11. Adam Wiggins and Gernot Heiser.: Fast Address-Space Switching on the StrongARM SA-1100 Processor. Proceedings of the 5th Australasian Computer Architecture Conference, Canberra, Australia (2000)
12. Adam Wiggins, Harvey Tuch, Volkmar Uhlig, and Gernot Heiser.: Implementation of Fast Address-Space Switching and TLB Sharing on the StrongARM Processor. Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference, Aizu-Wakamatsu City, Japan (2003)