A Trap-based Mechanism for Runtime Kernel Modification[1]

Young-Pil Kim*, Jin-Hee Choi** and Chuck Yoo*
*Department of Computer Science & Engineering, Korea University, Seoul, Korea
**Samsung Electronics, Suwon, Korea
{ypkim, jhchoi, hxy}@os.korea.ac.kr

Abstract

Runtime modification of kernel code is a difficult problem. However, the need of modifiable kernel is increasing because new requirements and services that are unanticipated at the time of kernel design keep coming in such a system for modern embedded application. Especially, advances of communication technologies prompt the need of flexible kernel because traditional kernel is not suitable to support various services resulting from new communication technologies. For the first step of a dynamic and flexible kernel, in this paper, we introduce a trap-based mechanism that can modify kernel code in runtime. The main advantage of trap-based mechanism is small cost in dynamic reconfiguration in fully configurable kernel. In order to prove it, we compare average cost of our trap-based mechanism with one of previous jmp-based mechanism, and our experimental result shows that average cost reduces by about 80%.

1. Introduction

A key challenge of embedded system is to provide an operating system with a high degree of customizability for required functionality [5]. With high customizability, high adaptability is also required on the embedded systems that support networking. Most studies on network protocols such as TCP have focused on tuning to specific network characteristics, but the advent of heterogeneous mobile network makes such approaches as adapting protocols for specific network difficult.

If our kernel has a flexible mechanism that can manipulate network protocol code dynamically, these problems of protocol adaptation on various network environments can be easily solved [4]. Most of previous research required a wide modification of kernel architecture; therefore, the structural complexity and the performance overhead have increased. To reflect these two issues, we attempt to find a simple and intuitive way, and it is dynamic kernel modification. The purpose of this paper is to propose a flexible kernel mechanism for dynamic kernel modification.

Our work is related to dynamic kernel code modification, and this is not the first work. Previous studies [1], [3] have a common problem in that their work is dependent on a specific hardware platform. In this paper, we try to eliminate such a hardware dependency and analyze previous mechanism in terms of fully configurable kernel.

The rest of the paper is organized as follows. First, the next section discusses related work. In Section 3, we show our motivation with possible and practical application such as dynamic protocol adaptation of network code and define conceptually our target system – fully reconfigurable kernel. Then, Section 4 explains the structural overview and functionality of our trap-based kernel modification mechanism. Section 5 describes the experiments that we conducted to assess the validity of our approach and the results of the performance analysis. Finally, Section 6 concludes.

2. Related Work

Many studies have been done on supporting flexibility and adaptability for operating system kernel [5]. It is difficult for complex systems such as operating systems to optimize the performance and verify the functionality in runtime. To overcome these challenges, extensible operating system structure [5], various profiling tools and debuggers [6], dynamic kernel instrumentation mechanisms [1], [2], [3] have been proposed. Our goal is also to give operating system kernel flexibility and adaptability; however, extensible operating system requires wide modification of conventional kernel code and profiling tools has

[1] This work was supported by grant No.R01-2004-000-10588-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

Proceedings of The Sixth IEEE International Conference on Computer and Information Technology (CIT'06)
0-7695-2687-X/06 $20.00 © 2006 IEEE

limited functionality. We choose dynamic kernel instrumentation as our base mechanism. The advantages of dynamic kernel instrumentation are as follows: 1) fully dynamic changing or modifying of functionality, 2) operating in commodity operating system kernel and 3) operating without a wide area of kernel modification.

The representative work about dynamic kernel instrumentation is that KernInst [1] and DTrace [2] on Sun Ultrasparc platform, and GILK [3] on Intel x86 platform. In several studies, jmp-based branch code is used for code splicing technique that is fundamental mechanism of dynamic kernel instrumentation. However, this technique has some limitation, and we discuss it in Section 4.2. The limitation does not concern in Reduced Instruction Set Computer (RISC) based system, but in Complex Instruction Set Computer (CISC) a special method should be used, and this causes previous instrumentation mechanisms to be dependent on specific hardware architecture [3]. Moreover, previous studies do not concern fully configurable kernel that has significant meaning in customizable operating system research [5].

In this paper, we adopt a trap-based code splicing method to guarantee the hardware architectural independency and to support fully configurable system. Also, we suggest kernel framework and primitives for a fine-grained kernel functional modification.

3. Motivation

In this section, we describe the possible application of our dynamic kernel modification. Especially, we cover the case of a network subsystem because dynamic adaptation of network protocol is good example for runtime kernel code modification. And then, we define conceptually fully reconfigurable kernel - our target system.

3.1 Possible Application: TCP based Network Subsystem

To show possible application of our mechanism, in this section, we demonstrate the protocol adaptation example of network subsystem functionality. Especially, we choose well-known and popular protocol TCP.

3.1.1 Protocol adaptation by TCP MSS adjustment. The fact that the size of TCP segment is relevant to a frame error rate is well analyzed in [4]. To put it briefly, when frame error rate is low, as the larger size of a sender's segment increases, as the higher TCP throughput also increases; however, when frame error rate is high, the tendency changes. That is, the throughput can be reduced rather than increased.

Therefore, we can infer that the unnecessary degradation of TCP throughput can be eliminated if we modify the behavior of managing MSS size. In terms of kernel instrumentation, using this inference, we can define the target of modification and the condition of invocation.

3.1.2 Applying protocol adaptation using dynamic kernel modification. To apply dynamic protocol adaptation in Section 3.1.1 on a kernel using dynamic kernel modification, we analyzed the code for TCP MSS adjusting in Linux network subsystem. As a result, we knew the location of adjusting part. It is the function 'tcp_sync_mss()' in kernel version 2.4.20. Therefore, the entry part of code of 'tcp_sync_mss()' is set to an instrumentation point, and on that point, splicing code is inserted for branching to the new implemented code for our algorithm. In the mobile heterogeneous network, the environment of network can vary dynamically, and when the change of network is detected, we have to adjust network functionality. If we know the exact time of detection, we can request kernel instrumentation to apply modified code on the network code. Therefore, the condition of invocation of kernel instrumentation in protocol adaptation can be explicit request of user or detection of environmental change.

3.2 Conceptual Definition of Fully Reconfigurable Kernel

In our work, we try to give a conventional kernel maximum flexibility and adaptability with minimum effort. In order to achieve our goal, we think that we need a fundamental mechanism to manipulate kernel functionality – their codes or control flows. This manipulation should be done in runtime without kernel shutdown. Therefore, we define our target system as following, and name it *fully reconfigurable kernel*.

Fully reconfigurable kernel is the operating system kernel which has,

1) Dynamic changeability - Kernel functionality can be easily changed in runtime.
2) Full degree of freedom for modification request - The request for modification can occur in anytime and anywhere.
3) Full availability of functional manipulation - The modification can occur in any location of kernel address space.
4) Secure manipulation - All modification should require a privilege level or an authority.

In our definition, the properties of 1) ~ 3) can be achieved by runtime analyzing of kernel code such as building control flow graph (or call graph) and by code splicing (see Section 4.1). In traditional view, kernel is usually trusted area; therefore, kernel should be concrete and secure. In such a view, fully reconfigurable kernel can contaminate trust and security of kernel functionality because this causes frequent changes of kernel functionalities. In order to overcome this issue, we focus on code splicing mechanism and the branch instruction for code splicing is chosen to privileged one.

Full implementation of fully reconfigurable kernel is beyond our scope in this paper. For the first step of achieving it, in our work, we concentrate only manipulation mechanism, and details of it will be described in next section 4.

4. Trap-based dynamic kernel modification mechanism

In this section, we describe our kernel modification framework and its subcomponents. Also, we discuss the details of our code splicing method, trap-based mechanism and its advantages.

4.1 The architecture of dynamic kernel modification mechanism

A key idea of our work is to use trap-based kernel modification mechanism for providing hardware independency and fine-grain manipulation of kernel functionality. The following Fig. 1 describes the overall structure of proposed kernel mechanism.

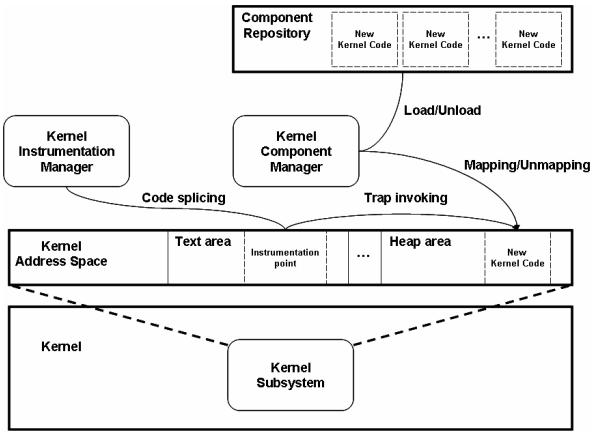

Fig. 1. The architecture of trap-based kernel modification mechanism

Fig. 1 has two main components: *kernel component manager* and *kernel instrumentation manager*. *Kernel component manager* manages a set of codes to be added or modified, and the codes are in the *component repository* that is allocated in kernel buffer. *Kernel component manager* loads new kernel code into *component repository*, and then maps the new code on to kernel address space. *Kernel instrumentation manager* waits a request for kernel modification, and redirects kernel call path to new kernel code in *component repository* when request invocation occurs.

The purpose of this mechanism is to reconfigure an old kernel functionality of running operating system into the new one. Conceptually, we can see kernel functionality as invocation chains of kernel codes in kernel subsystem. The chains have two directions: forward call and backward return. Therefore, those can construct a control flow. The conceptual shape of the control flow appears in Fig.2.

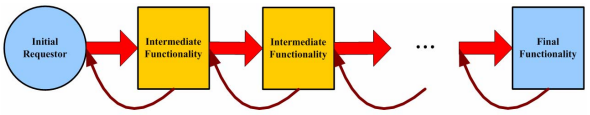

Fig. 2. Chain of control flow

Namely, for extending and reconfiguring the kernel, we need a way to manipulate the chains of control flow. To put it precisely, we need three facilities: *an instrumentation point*, *a set of codes to be added*, and *a way of bypassing of old functionality*.

The reason why we separate the component repository with the instrumentation manager is to guarantee transparency of locations of components. In the component repository, *a new set of codes* with information of management is packaged and deposited, and then when it needed, the codes are loaded into memory and mapped into the kernel address space by the *kernel component manager*.

After loading components which are required, we have to apply them to a running system dynamically. Therefore, we select one element of chain of control flow (*instrumentation point*), and then the additional branch instruction is overwritten on the point. This branch instruction is required for *bypassing the original code to new code*. When the branch instruction is overwritten, the original code on the point can be corrupted; therefore, we have to back up the original code. The original code is placed on the end of new codes. This process is called '*code splicing*' in dynamic instrumentation, and this paper suggests the trap generation instruction as the branch instruction. The trap-based code splicing appears in Fig.3.

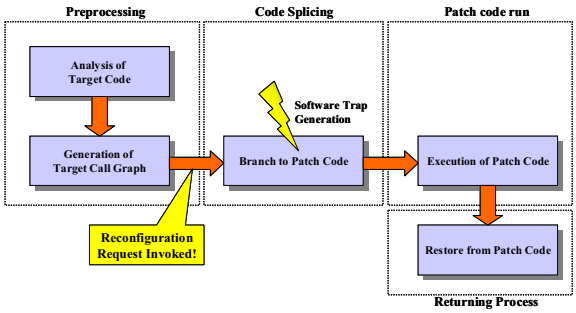

Fig. 3. The process of code splicing

The step *preprocessing process* provides fundamental information for deciding modification point and returning point by analyzing codes of target (parsing the binary and generating call graph). The *preprocessing process* occurs when kernel component manager loads and maps components. After that, in step *code splicing*, a software trap is generated for branching to new patch code, and the new codes are executed. When the original codes need to be recovered, the codes are executed in step *patch code run*, and return to the next location of instrumentation point in step *returning process*.

Now that we have known the mechanism of trap-based kernel modification, and we will discuss the advantages of the trap-based instrumentation in next Section 4.2.

4.2 The advantages of trap-based dynamic instrumentation

This paper suggests the trap-based mechanism as a branch code for code splicing contrary to jmp-based [1], [3] mechanism in previous work. The reason why we chose the trap-based mechanism as code splicing is for the following advantages.

Low branching cost. Previous work mainly pursues the efficiency using the jmp-based branching. Indeed, the cost of one branching of jmp is very cheap; however, it does not on multiple branching. In CISC such as Intel x86, the length of codes is various; therefore, the length of overwritten code is important because the original code and the next code can be corrupted. For preventing this corruption, one of previous work [3] uses multiple branch instructions which use a number of jmp codes which have smallest size and narrow jumping offset. Our insight is that multiple jmps can be heavier than one trap. (A trap instruction has small length; for example, smallest jmp of 8bit offset is 2 bytes; however trap has full offset (32bit), and has same length.)

Hardware Independency. The hardware platform such as RISC has the same length of instructions; therefore, it does not cause a corruption problem by overwritten branching code. However, in CISC, as states above 'low branching cost' section, the length of instructions causes a problem such as corruption of non-target code. To avoid the problem, multiple branch code can be used; however, that solution is dependent on Intel x86 platform which provides various kinds of jmp instructions. This can be another limitation point of previous work. On the contrary, trap generation instruction has same length in RISC or CISC; therefore, we do not need additional operation and can have consistency of mechanism.

Compatibility with conventional system. Trap mechanism is supported by whole commodity operating system, and we can utilize the most code related to trap without severe modification.

Simplicity. Trap-based mechanism requires only one branch code for kernel instrumentation; therefore, this mechanism is simpler than previous way, and requires small amount of implementation code.

5. Experiment and analysis

This section describes the experiment which evaluates the cost of our fundamental mechanism, trap-based kernel instrumentation.

5.1 Evaluation of Trap-based kernel instrumentation mechanism

In this section, we describe the performance evaluation of jmp-based and trap-based instrumentation. With several assumptions, we measured the relative cost of our trap-based mechanism and compared it with jmp-based mechanism, and then we analyzed our results.

Assumptions. In our experiment, three assumptions are needed, and these are follows:

1. We assume that a kernel which kernel instrumentation targets is a fully configurable system.
2. We assume that the kernel code which is inserted additionally does not have a fixed location.
3. We assume that the offset which is used in jmp-based branch instruction is not absolute but relative value.

The most current systems can modify their functionality partially and limitedly. However, we pursue the fully configurable system which can randomly and fine-grainedly modify the system functionality. Therefore, the first assumption will be needed. The second assumption means that we use a dynamic kernel memory allocation for managing a pool which contains additional kernel functionalities in a kernel heap, because the memory address which is

dynamically allocated can vary as the memory allocating policies. The third assumption describes that a code segment which is used in branching not changed, because the source and the destination of branching for kernel instrumentation are in the same address space, namely kernel address space.

Experiments. The experiment that we conducted is measuring the average cost of each kernel instrumentation mechanism: jmp-based kernel instrumentation, trap-based kernel instrumentation. The cost of each basic mechanism is yielded from clock cycles of processor. First of all, we measured the basic cost of jmp instruction and trap instruction using clock cycles. We measured the average clock cycles of each operation for 10000 times, and the results are given in Table 1.

Table 1. Basic cost of jmp and trap mechanism. The specification of platform we measured is Intel Pentium4 1.4GHz processor, Linux 2.4.20 kernel, and gcc 3.2 compiler. The number of code length field indicates bytes

	Clock cycles	Code length	Relative cost
Jmp	84.0012	5, 3, 2	1
Trap	1613.5124	2	19

From our experimental results in Table 1, we define each relative cost between jmp and trap as 1 and 19. The next step is choosing a target set of kernel modification, and collecting the statistical data which contain the frequency ratios of instructions that have different lengths in the target set. We assume that each ratio means the possibility of executing the instruction which has a specific length. The reason of our assumption is that we cannot know exactly which instruction will be processed without real execution, because the instructions that contain conditional branch are affected by the state of registers and the condition of branch; therefore, the results can randomly changed, namely, we cannot know the next executing instruction. As our assumption, in order to yield the cost of kernel instrumentation, it is more reasonable to use a statistical data which contain frequency information of each size of instruction than a specific code case. Therefore, we invoke kernel instrumentation repeatedly for specific times, and on the invocation point, we obtain a possible length of instruction to be overwritten as our statistical data, and then, the each cost can be calculated in jmp-based case and trap-based case. In particular, in jmp-based mechanism, we have to regard the case where the size of instruction code to be overwritten is smaller than that of full-offset jmp instruction, 5 bytes. In that case, small-sized jmp instructions can be used repeatedly, because small one has small-sized branch offset, so it cannot jump to over limitation offset directly. For example, 2-byte jmp has 1-byte offset, so it cannot jump to 1G-byte offset. However, trap instruction branches using a vector number, and the size of instruction is always 2 bytes; therefore, it does not need additional instructions.

Selected codes. For our experiment, we choose several kernel codes according to functionality in Linux kernel 2.4.20, and those are given in following Table 2.

Table 2. Description of target kernel code

Code name	Description	Functionality
eepro100.o	Intel EtherExpress pro100 driver	Network device driver
sched.o	Kernel scheduler primitives	Scheduler
slab.o	Slab allocator	Kernel memory all ocator
page_alloc.o	Kernel free page managing	Kernel memory all ocator

These codes have possibilities that we can choose them for modification of kernel functionalities, such as device drivers, schedulers, and memory managing routines. In order to know the frequency ratios of instructions which have different sizes, we analyzed these selected codes. The following Fig. 4 describes the visual ratio of occurrence of instructions according to their sizes.

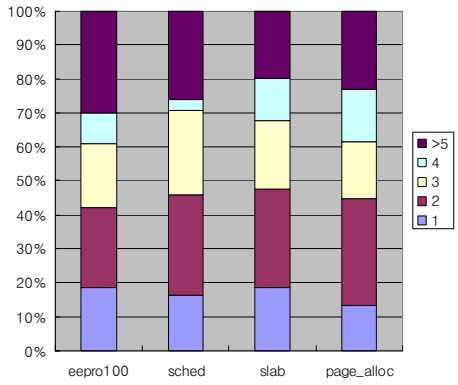

Fig. 4. Distribution ratio of instruction size in selected kernel codes

From the result, we can see that the proportion of instructions which have lengths of over 5 bytes is the maximum 30% in the entire code set. Therefore, we can infer that it is possible to be used unnecessary jmp instructions over the minimum twice for one kernel instrumentation when the instrumentation requests are invoked fairly in the whole code space. In fact, without real execution, we cannot estimate exactly instructions

to be executed, because several instructions need the state of registers or the value of calculated condition. Therefore, for our experiment, we define the possibility of an execution as the frequency ratio of each size of instructions from our result.

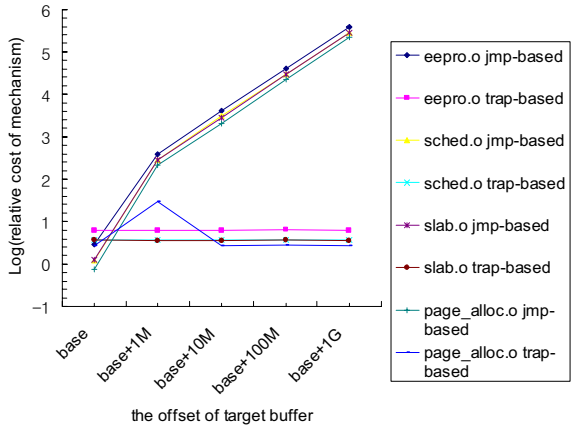

Fig. 5. The result of relative cost of jmp and trap based mechanism in the selected kernel code. The base offset is 1024 byte from the end of the kernel code.

Results and Analysis. The following Fig. 5 is the result of measuring relative cost for jmp-based mechanism and trap-based mechanism when they are used for kernel instrumentation. We define a single cost of jmp as one, and a single cost of trap as 19 according to Table 1. In Fig. 5, X axis means the byte offset of target buffer, and Y axis indicates logarithmical average cost.

In our experiment, we calculated the average cost for 10000 times for selected codes. We do not calculate the cost for code of 1-byte length because the minimum size of branch code is 2 bytes in the both mechanisms, so the average cost can be smaller than a single one. We did not use the real code sequence but possibilities of occurrence of various code sizes, and reflected on the distance of target buffer to be inserted additionally and the size of instruction to overwrite.

From our experimental results, we can infer that it is difficult to apply jmp-based mechanism for kernel instrumentation especially to fully configurable system. It supports our inference that the relative cost of jmp-based mechanism varies according to the offset of target buffer. In Fig. 5, we can see that the relative cost of jmp-based mechanism increases rapidly according to increasing offset of target buffer. On the contrary, from the result, we can see that the relative cost of trap-based mechanism is almost not affected by the offset of target buffer. This result indicates following possibilities. The location of target buffer can vary according to allocating address on a kernel heap, and in a real 32-bit system which has a 4G-byte address space, the distance between instrumentation point and target buffer can be over 1G bytes. There-fore, it is possible that jmp-base mechanism degrades the performance of fully configurable system. On the contrary, our trap-based mechanism shows almost flat and low average cost even if the offset of target buffer increases. As a result, we can infer that the kernel instrumentation based trap mechanism is more suitable for fully configurable system than jmp mechanism.

6. Conclusion

In this paper, we propose a kernel-level mechanism for dynamic kernel modification that can provide an embedded system with high customizability and flexibility. Our proposal is based on a trap mechanism, and using the mechanism, we can eliminate the hardware dependency. We analyze previous jmp-based mechanism, and compare it with our mechanism in our experimental result. We show that our mechanism has more benefits than previous way and our trap-based mechanism can be a fundamental primitive to effectively modify the kernel functionality. Based on our fundamental mechanism, we plan to design and implement fully configurable operating system.

7. References

[1] Ariel Tamches and Barton P.Miller, “Fine-grained dynamic instrumentation of commodity operating system kernels”, In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[2] Bryan M. Cantrill, Michael W. Shapiro and Adam H. Leventhal, “Dynamic Instrumentation of Production Systems”, *USENIX Annual Technical Conference,* 2004.

[3] D. Pearce, P. Kelly, U. Harder, and T. Field, “GILK: A dynamic instrumentation tool for the Linux Kernel”, In *Proceedings of the 12th International Conference on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation* (TOOLS '02), London, United Kingdom, 2002, pp. 220-226.

[4] Jin-Hee Choi, Jin-Ghoo Choi and Chuck Yoo, “Adapting TCP Segment Size in Cellular Networks”, In *Proceedings of International Conference on Networking*, 2005.

[5] G. Denys, F..Pissense and F.Mattijs, “A Survey of Customizable Operating Systems Research”, *ACM Computing Surveys*, Vol. 34, No. 4, 2002, pp. 450-468.

[6] G.H. Kuenning: Kitrace, “Precise Interactive Measurement of Operating Systems Kernels”, *Software—Practice & Experience*, Vol. 25(1), 1995, pp. 1-21.