Djprobe—Kernel probing with the smallest overhead

Masami Hiramatsu
Hitachi, Ltd., Systems Development Lab.
masami.hiramatsu.pt@hitachi.com

Satoshi Oshima
Hitachi, Ltd., Systems Development Lab.
satoshi.oshima.fk@hitachi.com

Abstract

Direct Jump Probe (djprobe) is an enhancement to kprobe, the existing facility that uses breakpoints to create probes anywhere in the kernel. Djprobe inserts jump instructions instead of breakpoints, thus reducing the overhead of probing. Even though the kprobe "booster" speeds up probing, there still is too much overhead due to probing to allow for the tracing of tens of thousands of events per second without affecting performance.

This presentation will show how the djprobe is designed to insert a jump, discuss the safety of insertion, and describe how the cross self-modification (and so on) is checked. This presentation also provides details on how to use djprobe to speed up probing and shows the performance improvement of djprobe compared to kprobe and kprobe-booster.

1 Introduction

1.1 Background

For the use of non-stop servers, we have to support a probing feature, because it is sometimes the only method to analyze problematic situations.

Since version 2.6.9, Linux has kprobes as a very unique probing mechanism [7]. In kprobe ((a) in Figure 1), an original instruction at probe point is copied to an *out-of-line buffer* and a break-point instruction is put at the probe point. The break-point instruction triggers a break-point exception and invokes `pre_handler()` of the kprobe from the break-point exception handler. After that, it executes the out-of-line buffer in single-step mode. Then, it triggers a single-step exception and invokes `post_handler()`. Finally, it returns to the instruction following the probe point.

This probing mechanism is useful. For example, system administrators may like to know why their system's performance is not very good under heavy load. Moreover, system-support venders may like to know why their system crashed by salvaging traced data from the dumped memory image. In both cases, if the overhead due to probing is high, it will affect the result of the measurment and reduce the performance of applications. Therefore, it is preferable that the overhead of probing becomes as small as possible.

From our previous measurement [10] two years ago, the processing time of kprobe was about 1.0 usec whereas Linux Kernel State Tracer (LKST) [2] was less than 0.1 usec. From our previous study of LKST [9], about 3% of overhead for tracing events was recorded. Therefore, we decided our target for probing overhead should be less than 0.1 usec.

1.2 Previous Works

Figure 1 illustrates how the probing behaviors are different among kprobe, kprobe-booster and djprobe when a process hits a probe point.

As above stated, our goal of the processing time is less than 0.1 usec. Thus we searched for improvements to reduce the probing overhead so as it was as small as possible. We focused on the probing process of kprobe, which causes exceptions twice when each probe hit. We predicted that most of the overhead came from the exceptions, and we could reduce it by using jumps instead of the exceptions.

We developed the kprobe-booster as shown in Figure 1(b). In this improvement, we attempted to replace the single-step exception with a jump instruction, because it was easier than replacing a break-point. Thus, the first-half of processing of kprobe-booster is same as the kprobe, but it does not use a single-step exception in the latter-half. This improvement has already been merged into upstream kernel since 2.6.17.

Last year, Ananth's paper [7] unveiled efforts for improving kprobes, so the probing overheads of kprobe

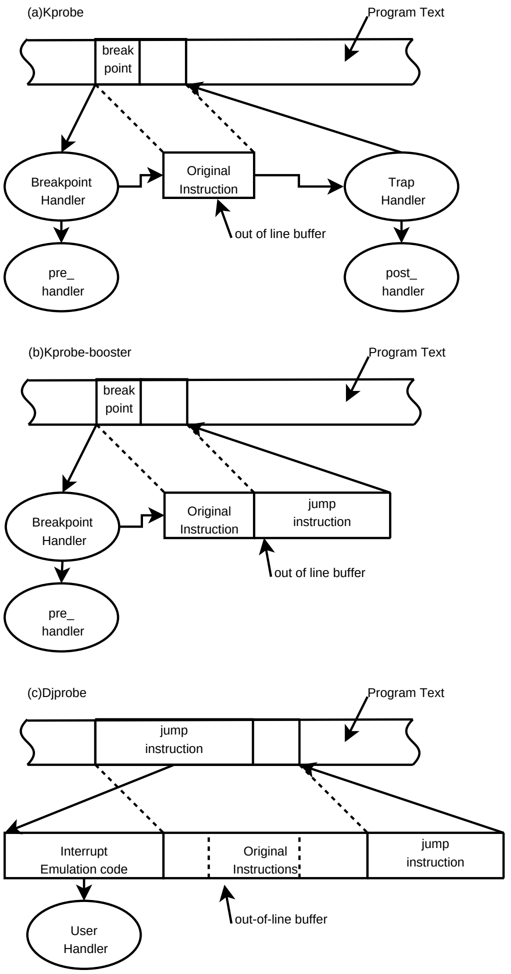


Figure 1: Kprobe, kprobe-booster and djprobe

and kprobe-booster became about 0.5 usec and about 0.3 usec respectively. Thus the kprobe-booster succeeded to reduce the probing overhead by almost half. However, its performance is not enough for our target.

Thus, we started developing djprobe: *Direct Jump Probe*.

1.3 Concept and Issues

The basic idea of djprobe is simply to use a jump instruction instead of a break-point exception. In djprobe ((c) in Figure 1), a process which hits a probe point jumps to the out-of-line buffer, calls probing handler, executes the "original instructions" on the out-of-line buffer directly, and jumps back to the instruction following the place where the original instructions existed. We will see the result of this improvement in Section 4.

There are several difficulties to implement this concept. A jump instruction must occupy 5 bytes on i386, replacement with a jump instruction changes the instruction boundary, original instructions are executed on another place, and these are done on the running kernel. So…

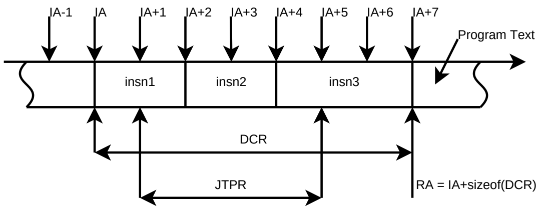
- Replacement of original instructions with a jump instruction must not block other threads.
- Replacement of original instructions which are targeted by jumps must not cause unexpectable crashes.
- Some instructions such as an instruction with relative addressing mode can not be executed at out-of-line buffer.
- There must be at least one instruction following the replaced original instructions to allow for the returning from the probe.
- Cross code modification in SMP environment may cause General Protection Fault by Intel Erratum.
- Djprobe (and also kprobe-booster) does not support the `post_handler`.

Obviously, some tricks are required to make this concept real. This paper discribes how djprobe solve these issues.

1.4 Terminology

Before discussing details of djprobe, we would like to introduce some useful terms. Figure 2 illustrates an example of execution code in CISC architecture. The first instruction is a 2 byte instruction, second is also 2 bytes, and third is 3 bytes.

In this paper, IA means *Insertion Address*, which specifies the address of a probe point. DCR means *Detoured Code Region*, which is a region from insertion address to the end of a detoured code. The detoured code consists of the instructions which are partially or fully covered by a jump instruction of djprobe. JTPR means *Jump Target Prohibition Region*, which is a 4 bytes (on i386) length region, starts from the next address of IA. And, RA means *Return Address*, which points the instruction next to the DCR.



ins1: 1st Instruction

ins2: 2nd Instruction

ins3: 3rd Instruction

IA: Insertion Address

RA: Return Address

JTPR: Jump Target Prohibition Region

DCR: Detoured Code Region

Figure 2: Terminology

2 Solutions of the Issues

In this section, we discuss how we solve the issues of djprobe. The issues that are mentioned above can be categorized as follows.

- Static-Analysis Issues
- Dynamic-Safety Issue
- Cross Self-modifying Issue
- Functionality-Performance Tradeoff Issue

The following section deeply discuss how to solve the issues of djprobe.

2.1 Static Analysis Issues

First, we will discuss a safety check before the probe is inserted. Djprobe is an enhancement of kprobes and it based on implementation of Kprobes. Therefore, it includes every limitation of kprobes, which means djprobe cannot probe where kprobes cannot. As figure 3 shows, the DCR may include several instructions because the size of jump instruction is more than one byte (relative jump instruction size is 5 bytes in i386 architecture). In addition, there are only a few choices of remedies at execution time because "out-of-line execution" is done directly (which means single step mode is not used). This means there are 4 issues that should be checked statically. See below. Static safety check must be done before registering the probe and it is enough that it be done just once. Djprobe requires that a user must not insert a probe, if the probe point doesn't pass safety checks. They must use kprobe instead of djprobe at the point.

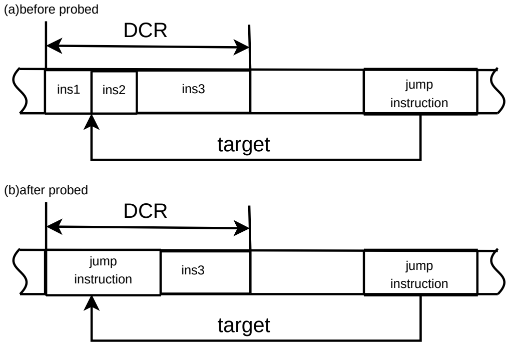


Figure 3: Corruption by jump into JTPR

2.1.1 Jumping in JTPR Issue

Replacement with a jump instruction involves changing instruction boundaries. Therefore, we have to ensure that no jump or call instructions in the kernel or kernel modules target JTPR. For this safety check, we assume that other functions never jump into the code other than the entry point of the function. This assumption is basically true in gcc. An exception is setjmp()/longjmp(). Therefore, djprobe cannot put a probe where setjmp() is used. Based on this assumption, we can check whether JTPR is targeted or not by looking through within the function. This code analysis must be changed if the assumption is not met. Moreover, there is no effective way to check for assembler code currently.

2.1.2 IP Relative Addressing Mode Issue

If the original instructions in DCR include the IP (Instruction Pointer, `EIP` in i386) relative addressing mode instruction, it causes the problem because the original instruction is copied to out-of-line buffer and is executed directly. The effective address of IP relative addressing mode is determined by where the instruction is placed. Therefore, such instructions will require a correction of a relative address. The problem is that almost all relative jump instructions are "near jumps" which means destination must be within –128 to 127 bytes. However, out-of-line buffer is always farther than 128 bytes. Thus, the safety check disassembles the probe point and checks whether IP relative instruction is included. If the IP relative address is found, the djprobe can not be used.

2.1.3 Prohibit Instructions in JTPR Issue

There are some instructions that cannot be probed by djprobe. For example, a call instruction is prohibited. When a thread calls a function in JTPR, the address in the JTPR is pushed on the stack. Before the thread returns, if a probe is inserted at the point, `ret` instruction triggers a corruption because instruction boundary has been changed. The safety check also disassembles the probe point and check whether prohibited instructions are included.

2.1.4 Function Boundary Issue

Djprobe requires at least one instruction must follow DCR. If DCR is beyond the end of the function, there is no space left in the out-of-line buffer to jump back from. This safety check can easily be done because what we have to do is only to compare DCR bottom address and function bottom address.

2.2 Dynamic-Safety Issue

Next, we discuss the safety of modifying multiple instructions when the kernel is running. The dynamic-safety issue is a kind of atomicity issue. We have to take care of interrupts, other threads, and other processors, because we can not modify multiple instructions atomically. This issue becomes more serious on the preemptive kernel.

2.2.1 Simultaneous Execution Issue

Djprobe has to overwrite several instructions by a jump instruction, since i386 instruction set is CISC. Even if we write this jump atomically, there might be other threads running on the middle of the instructions which will be overwritten by the jump. Thus, "atomic write" can not help us in this situation. In contrast, we can write the break-point instruction atomically because its size is one byte. In other words, the break-point instruction modifies only one instruction. Therefore, we decided to use the "bypass method" for embedding a jump instruction.

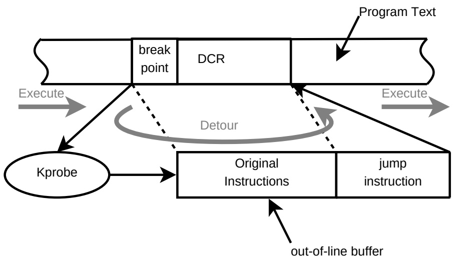


Figure 4: Bypass method

Figure 4 illustrates how this bypass method works.

This method is similar to the highway construction. The highway department makes a bypass route which detours around the construction area, because the traffic can not be stopped. Similarly, since the entire system also can not be stopped, djprobe generates an out-of-line code as a bypass from the original code, and uses a break-point of the kprobe to switch the execution address from the original address to the out-of-line code. In addition, djprobe adds a jump instruction in the end of the out-of-line code to go back to RA. In this way, other threads detour the region which is overwritten by a jump instruction while djprobe do it.

What we have to think of next is when the other threads execute from within the detoured region. In the case of non-preemptive kernel, these threads might be interrupted within the DCR. The same issue occurs when we release the out-of-line buffers. Since some threads might be running or be interrupted on the out-of-line code, we have to wait until those return from there. As you know, in the case of the non-preemptive kernel, interrupted kernel threads never call scheduler. Thus, to solve this issue, we decided to use the scheduler synchronization. Since the `synchronize_sched()` function waits until the scheduler is invoked on all processors, we can ensure all interrupts, which were occurred before calling this function, finished.

2.2.2 Simultaneous Execution Issue on Preemptive Kernel

This wait-on-synchronize_sched method is premised on the fact that the kernel is never preempted. In the preemptive kernel, we must use another function to wait the all threads sleep on the known places, because some threads may be preempted on the DCR. We discussed this issue deeply and decided to use the freeze_processes() recommended by Ingo Molnar [6]. This function tries to freeze all active processes including all preempted threads. So, preempted threads wake up and run after they call the try_to_freeze() or the refrigerator(). Therefore, if the freeze_processe() succeeds, all threads are sleeping on the refrigerator() function, which is a known place.

2.3 Cross Self-modifying Issue

The last issue is related to a processor specific erratum. The Intel® processor has an erratum about unsynchronized cross-modifying code [4]. On SMP machine, if one processor modifies the code while another processor pre-fetches unmodified version of the code, unpredictable General Protection Faults will occur. We supposed this might occur as a result of hitting a cache-line boundary. On the i386 architecture, the instructions which are bigger than 2 bytes may be across the cache-line boundary. These instructions will be pre-fetched twice from 2nd cache. Since a break-point instruction will change just a one byte, it is pre-fetched at once. Other bigger instructions, like a long jump, will be across the cache-alignment and will cause an unexpected fault. In this erratum, if the other processors issue a serialization such as CPUID, the cache is serialized and the cross-modifying is safely done.

Therefore, after writing the break-point, we do not write the whole of the jump code at once. Instead of that, we write only the jump address next to the break-point. And then we issue the CPUID on each processor by using IPI (Inter Processor Interrupt). At this point, the cache of each processor is serialized. After that, we overwrite the break-point by a jump op-code whose size is just one byte.

2.4 Functionality-Performance Tradeoff Issue

From Figure 1, djprobe (and kprobe-booster) does not call post_handler(). We thought that is a trade-off between speed and the post_handler. Fortunately, the SystemTap [3], which we were assuming as the main use of kprobe and djprobe, did not use post_handler. Thus, we decided to choose speed rather than the post_handler support.

3 Design and Implementation

Djprobe was originally designed as a wrapper routine of kprobes. Recently, it was re-designed as a jump optimization functionality of kprobes.[1] This section explains the latest design of djprobe on i386 architecture.

3.1 Data Structures

To integrate djprobe into kprobes, we introduce the length field in the kprobe data structure to specify the size of the DCR in bytes. We also introduce djprobe_instance data structure, which has three fields: kp, list, and stub. The kp field is a kprobe that is embedded in the djprobe_instance data structure. The list is a list_head for registration and unregistration. The stub is an arch_djprobe_stub data structure to hold a out-of-line buffer.

From the viewpoint of users, a djprobe_instance looks like a special aggregator probe, which aggregates several probes on the same probe point. This means that a user does not specify the djprobe_instance data structure directly. Instead, the user sets a valid value to the length field of a kprobe, and registers that. Then, that kprobe is treated as an aggregated probe on a djprobe_instance. This allows you to use djprobe transparently as a kprobe. Figure 5 illustrates these data structures.

3.2 Static Code Analysis

Djprobe requires the safety checks, that were discussed in Section 2.1, before calling register_kprobe(). Static code analysis tools, djprobe_static_code_analyzer, is available from djprobe development site [5]. This tool also provides the length of DCR. Static code analysis is done as follows.

[1] For this reason, djprobe is also known as *jump optimized kprobe*.

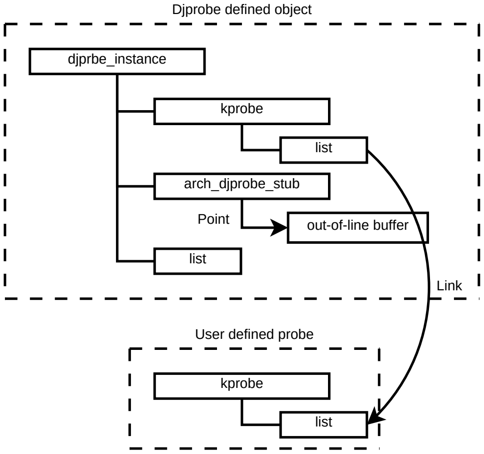


Figure 5: The instance of djprobe

```
[37af1b] subprogram
         sibling             [37af47]
         external            yes
         name                "register_kprobe"
         decl_file           1
         decl_line           874
         prototyped          yes
         type                [3726a0]
         low_pc              0xc0312c3e
         high_pc             0xc0312c46
         frame_base          2 byte block
          [   0] breg4 4
```

Figure 6: Example of debuginfo output by eu-readelf

3.2.1 Function Bottom Check

djprobe_static_code_analyzer requires a debuginfo file for the probe target kernel or module. It is provided by the kernel compilation option if you use vanilla kernel. Or it is provided as debuginfo package in the distro.

Figure 6 shows what debuginfo looks like.

First of all, this tool uses the debuginfo file to find the top (low_pc) and bottom (high_pc) addresses of the function where the probe point is included, and makes a list of these addresses. By using this list, it can check whether the DCR bottom exceeds function bottom. If it finds this to be true, it returns 0 as "can't probe at this point."

There are two exceptions to the function bottom check. If the DCR includes an absolute jump instruction or a function return instruction, and the last byte of these instructions equals the bottom of the function, the point can be probed by djprobe, because direct execution of those instructions sets IP to valid place in the kernel and there is no need to jump back.

3.2.2 Jump in JTPR Check

Next, djprobe_static_code_analyzer disassembles the probed function of the kernel (or the module) by using objdump tool. The problem is the current version of objdump cannot correctly disassemble if the BUG() macro is included in the function. In that case, it simply discards the output following the BUG() macro and retries to disassemble from right after the BUG(). This disassembly provides not only the boundaries information in DCR but also the assembler code in the function.

Then, it checks that all of jump or call instructions in the function do not target JTPR. It returns 0, if it find an instruction target JTPR.

If the probe instruction is 5 bytes or more, it simply returns the length of probed instruction, because there is no boundary change in JTPR.

3.2.3 Prohibited Relative Addressing Mode and Instruction Check

djprobe_static_code_analyzer checks that DCR does not include a relative jump instruction or prohibited instructions.

3.2.4 Length of DCR

Djprobe requires the length of DCR as an argument of register_kprobe() because djprobe does not have a disassembler in the current implementation. djprobe_static_code_analyzer acquires it and returns the length in case that the probe point passes all checks above.

3.3 Registration Procedure

This is done by calling the register_kprobe() function. Before that, a user must set the address of a probe point and the length[2] of DCR.

[2] If the length field of a kprobe is cleared, it is not treated as a djprobe but a kprobe.

3.3.1 Checking Conflict with Other Probes

First, `register_kprobe()` checks whether other probes are already inserted on the DCR of the specified probe point or not. These conflicts can be classified in following three cases.

1. Some other probes are already inserted in the same probe point. In this case, `register_kprobe()` treats the specified probe as one of the collocated probes. Currently, if the probe which previously inserted is not djprobe, the jump optimization is not executed. This behavior should be improved to do jump optimization when feasible.
2. The DCR of another djprobe covers the specified probe point. In this case, currently, this function just returns `-EEXIST`. However, ideally, the djprobe inserted previously should be unoptimized for making room for the specified probe.
3. There are some other probes in the DCR of the specified djprobe. In this case, the specified djprobe becomes a normal kprobe. This means the `length` field of the `kprobe` is cleared.

3.3.2 Creating New djprobe_instance Object

Next, `register_kprobe()` calls the `register_djprobe()` function. It allocates a `djprobe_instance` object. This function copies the values of `addr` field and `length` field from the original kprobe to the `kp` field of the `djprobe_instance`. Then, it also sets the address of the `djprobe_pre_handler()` to the `pre_handler` field of the `kp` field in the `djprobe_instance`. Then, it invokes the `arch_prepare_djprobe_instance()` function to prepare an out-of-line buffer in the `stub` field.

3.3.3 Preparing the Out-of-line Buffer

Figure 7 illustrates how an out-of-line buffer is composed.

The `arch_prepare_djprobe_instance()` allocates a piece of executable memory for the out-of-line buffer by using `__get_insn_slot()` and setup its contents. Since the original `__get_insn_slot()` function can handle only single size of memory slots, we modified it to handle various length memory slots. After allocating the buffer, it copies the template code of the buffer from the `djprobe_template_holder()` and embeds the address of the `djprobe_instance` object and the `djprobe_callback()` function into the template. It also copies the original code in the DCR of the specified probe to the next to the template code. Finally, it adds the jump code which returns to the next address of the DCR and calls `flush_icache_range()` to synchronize i-cache.

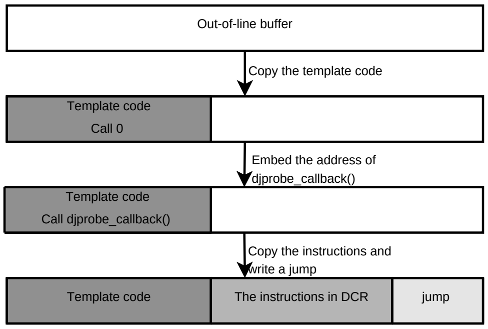


Figure 7: Preparing an out-of-line buffer

3.3.4 Register the djprobe_instance Object

After calling `arch_prepare_djprobe_instance()`, `register_djprobe()` registers the `kp` field of the `djprobe_instance` by using `__register_kprobe_core()`, and adds the `list` field to the `registering_list` global list. Finally, it adds the user-defined kprobe to the `djprobe_instance` by using the `register_aggr_kprobe()` and returns.

3.4 Committing Procedure

This is done by calling the `commit_djprobes()` function, which is called from `commit_kprobes()`.

3.4.1 Check Dynamic Safety

The `commit_djprobes()` calls the `check_safety()` function to check safety of dynamic-self modifying. In other words, it ensures that

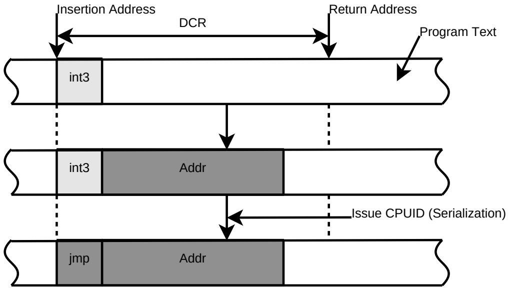


Figure 8: Optimization Procedure

no thread is running on the DCR nor is it preempted. For this purpose, check_safety() call synchronize_sched() if the kernel is non-preemptive, and freeze_processes() and thaw_processes() if the kernel is preemptive. These functions may take a long time to return, so we call check_safety() only once.

3.4.2 Jump Optimization

Jump optimization is done by calling the arch_preoptimize_djprobe_instance() and the arch_optimize_djprobe_instance(). The commit_djprobes() invokes the former function to write the destination address (in other words, the address of the out-of-line buffer) into the JTPR of the djprobe, and issues CPUID on every online CPU. After that, it invokes the latter function to change the break-point instruction of the kprobe to the jump instruction. Figure 8 illustrates how the instructions around the insertion address are modified.

3.4.3 Cleanup Probes

After optimizing registered djprobes, the commit_djprobe() releases the instances of the djprobe in the unregistering_list list. These instances are linked by calling unregister_kprobe() as described Section 3.6. Since the other threads might be running on the out-of-line buffer as described in the Section 2.2, we can not release it in the unregister_kprobe(). However, the commit_djprobe() already ensured safety by using the check_safety(). Thus we can release the instances and the out-of-line buffers safely.

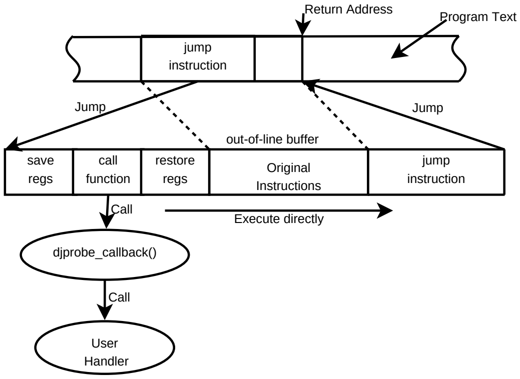


Figure 9: Probing Procedure

3.5 Probing Procedure

Figure 9 illustrates what happens when a process hits a probe point.

When a process hits a probe point, it jumps to the out-of-line buffer of a djprobe. And it emulates the break-point on the first-half of the buffer. This is accomplished by saving the registers on the stack and calling the djprobe_callback() to call the user-defined handlers related to this probe point. After that, djprobe restores the saved registers, directly executes continuing several instructions copied from the DCR, and jumps back to the RA which is the next address of the DCR.

3.6 Unregistration Procedure

This is done by calling unregister_kprobe(). Unlike the registration procedure, un-optimization is done in the unregistration procedure.

3.6.1 Checking Whether the Probe Is Djprobe

First, the unregister_kprobe() checks whether the specified kprobe is one of collocated kprobes. If it is the last kprobe of the collocated kprobes which are aggregated on a aggregator probe, it also tries to remove the aggregator. As described above, the djprobe_instance is a kind of the aggregator probe. Therefore, the function also checks whether the aggregator is djprobe (this is done by comparing the pre_handler field and the address of djprobe_pre_handler()). If so it calls unoptimize_djprobe() to remove the jump instruction written by the djprobe.

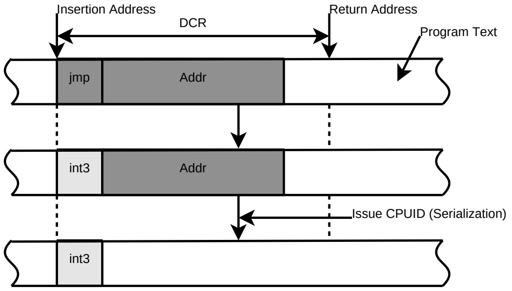


Figure 10: Un-optimization Procedure

3.6.2 Probe Un-optimization

Figure 10 illustrates how a probe point is un-optimized.

The `unoptimized_djprobe()` invokes `arch_unoptimize_djprobe_instance()` to restore the original instructions to the DCR. First, it inserts a break-point to IA for protect the DCR from other threads, and issues `CPUID` on every online CPUs by using `IPI` for cache serialization. After that, it copies the bytes of original instructions to the JTPR. At this point, the djprobe becomes just a kprobe, this means it is un-optimized and uses a break-point instead of a jump.

3.6.3 Removing the Break-Point

After calling `unoptimize_djprobe()`, the `unregister_kprobe()` calls `arch_disarm_kprobe()` to remove the break-point of the kprobe, and waits on `synchronize_sched()` for cpu serialization. After that, it tries to release the aggregator if it is not a djprobe. If the aggregator is a djprobe, it just calls `unregister_djprobe()` to add the `list` field of the djprobe to the `unregistering_list` global list.

4 Performance Gains

We measured and compared the performance of djprobe and kprobes. Table 1 and Table 2 show the processing time of one probing of kprobe, kretprobe, its boosters, and djprobes. The unit of measure is nano-seconds. We measured it on Intel® Pentium® M 1600MHz with UP kernel, and on Intel® Core™ Duo 1667MHz with SMP kernel by using `linux-2.6.21-rc4-mm1`.

method	orignal	booster	djprobe
kprobe	563	248	49
kretprobe	718	405	211

Table 1: Probing Time on Pentium® M in nsec

method	orignal	booster	djprobe
kprobe	739	302	61
kretprobe	989	558	312

Table 2: Probing Time on Core™ Duo in nsec

We can see djprobe could reduce the probing overhead to less than 0.1 usec (100 nsec) on each processor. Thus, it achived our target performance. Moreover, kretprobe can also be accelerated by djprobe, and the djprobe-based kretprobe is as fast as kprobe-booster.

5 Example of Djprobe

Here is an example of djprobe. The differences between kprobe and djprobe can be seen at two points: setting the `length` field of a `kprobe` object before registration, and calling `commit_kprobes()` after registration and unregistration.

```
/* djprobe_ex.c -- Direct Jump Probe Example */
#include <linux/version.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kprobes.h>

static long addr=0;
module_param(addr, long, 0444);
static long size=0;
module_param(size, long, 0444);

static int probe_func(struct kprobe *kp,
                      struct pt_regs *regs) {
   printk("probe point:%p\n", (void*)kp->addr);
   return 0;
}

static struct kprobe kp;

static int install_probe(void) {
   if (addr == 0) return -EINVAL;

   memset(&kp, sizeof(struct kprobe), 0);
   kp.pre_handler = probe_func;
   kp.addr = (void *)addr;

   kp.length = size;

   if (register_kprobe(&kp) != 0) return -1;

   commit_kprobes();
   return 0;
}

static void uninstall_probe(void) {
   unregister_kprobe(&kp);

   commit_kprobes();
}

module_init(install_probe);
module_exit(uninstall_probe);
MODULE_LICENSE("GPL");
```

6 Conclusion

In this paper, we proposed djprobe as a faster probing method, discussed what the issues are, and how djprobe can solve them. After that, we described the design and the implementation of djprobe to prove that our proposal can be implemented. Finally, we showed the performance improvement, and that it could reduce the probing overhead dramatically. You can download the latest patch set of djprobe from djprobe development site [5]. Any comments and contributions are welcome.

7 Future Works

We have some plans about future djprobe development.

7.1 SystemTap Enhancement

We have a plan to integrate the static analysis tool into the SystemTap for accelerating kernel probing by using djprobe.

7.2 Dynamic Code Modifier

Currently, djprobe just copies original instructions from DCR. This is the main reason why the djprobe cannot probe the place where the DCR is including execution-address-sensitive code.

If djprobe analyzes these sensitive codes and replaces its parameter to execute it on the out-of-line buffer, the djprobe can treat those codes. This idea is basically done by kerninst [1, 11] and GILK [8].

7.3 Porting to Other Architectures

Current version of djprobe supports only i386 architecture. Development for x86_64 is being considered. Several difficulties are already found, such as RIP relative instructions. In x86_64 architecuture, RIP relative addressing mode is expanded and we must assume it might be used. Related to dynamic code modifier, djprobe must modify the effective address of RIP relative addressing instructions.

To realize this, djprobe requires instruction boundary information in DCR to recognize every instruction. This should be provided by djprobe_static_code_analyser or djprobe must have essential version of disassembler in it.

7.4 Evaluating on the Xen Kernel

In the Xen kernel, djprobe has bigger advantage than on normal kernel, because it does not cause any interrupts. In the Xen hypervisor, break-point interruption switches a VM to the hypervisor and the hypervisor upcalls the break-point handler of the VM. This procedure is so heavy that the probing time becomes almost double.

In contrast, djprobe does not switch the VM. Thus, we are expecting the probing overhead of djprobe might be much smaller than kprobes.

8 Acknowledgments

We would like to thank Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston, Anil Keshavamurthy, Maneesh Soni, Frank Ch. Eigler, Ingo Molnar, and other developers of kprobes and SystemTap for helping us develop the djprobe.

We also would like to thank Hideo Aoki, Yumiko Sugita and our colleagues for reviewing this paper.

9 Legal Statements

Linux is a registered trademark of Linus Torvalds.

References

[1] kerninst. `http://www.paradyn.org/html/kerninst.html`.

[2] Lkst. `http://lkst.sourceforge.net/`.

[3] SystemTap. `http://sourceware.org/systemtap/`.

[4] Unsynchronized cross-modifying code operation can cause unexpected instruction execution results. Intel Pentium II Processor Specification Update.

[5] Djprobe development site, 2005. `http://lkst.sourceforge.net/djprobe.html`.

[6] Re: djprobes status, 2006. `http://sourceware.org/ml/systemtap/2006-q3/msg00518.html`.

[7] Ananth N. Mavimalayanahalli et al. Probing the Guts of Kprobes. In *Proceedings of Ottawa Linux Symposium*, 2006.

[8] David J. Pearce et al. Gilk: A dynamic instrumentation tool for the linux kernel. In *Computer Performance Evaluation/TOOLS*, 2002.

[9] Toshiaki Arai et al. Linux Kernel Status Tracer for Kernel Debugging. In *Proceedings of Software Engineering and Applications*, 2005.

[10] Masami Hiramatsu. Overhead evaluation about kprobes and djprobe, 2005. `http://lkst.sourceforge.net/docs/probe-eval-report.pdf`.

[11] Ariel Tamches and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *OSDI*, February 1999.

Proceedings of the
Linux Symposium

Volume One

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Review Committee

Proceedings Formatting Team