Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Probing the Guts of Kprobes

Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston
IBM Linux Technology Center
ananth@in.ibm.com, prasanna@in.ibm.com, jkenisto@us.ibm.com
Anil Keshavamurthy
Intel Open Source Technology Center
anil.s.keshavamurthy@intel.com
Masami Hiramatsu
Hitachi Systems Development Laboratory
hiramatu@sdl.hitachi.co.jp

Abstract

Kernel Probes (kprobes) can insert probes into a running kernel for purposes of debugging, tracing, performance evaluation, fault injection, etc. A user-defined handler is run when a probepoint is hit. From the barebones implementation in Linux 2.6.9, kprobes has undergone a number of improvements—support for colocated probes, function-return probes, reentrant probes, and the like. Handlers are now executed without any locks held, leading to lower overhead compared to the earlier "single spinlock serialization" method. Other enhancements are on the anvil—the kprobe "booster" series, userspace probes and watchpoint probes, to name a few. This paper will trace the developments in kprobes and also touch upon the current state of the aforementioned enhancements.

1 Introduction

Kernel probes (kprobes) is a simple, lightweight kernel instrumentation mechanism, which provides a facility to execute a user-defined handler when a probepoint[1] is hit. Since making its first appearance in the Linux™ kernel in linux-2.6.9-rc2, kprobes has proved to be an invaluable tool for kernel hackers, providing a facility to dynamically insert `printk()`s or counters into a running kernel, thus reducing the burden of having to statically compile a new kernel, just for instrumentation purposes. Additionally, kprobes has been extensively used for kernel tracing [9], performance evaluation, fault-injection, etc. Several tools (e.g., SystemTap [4]) now use the kprobes infrastructure as a base.

Kprobes has evolved from its first appearance. A number of new features have been added: support for colocated probes, function-return probes, lockless handler execution, and lately, the kprobe-boosters.

[1] The probepoint is the point of instrumentation—the text address where the kprobe is registered.

This paper gives a brief history of kprobes, then delves into the basics. It then goes on to cover functional and performance enhancements that have been done, concluding with a brief report on the works in progress.

2 A brief history

Kprobes finds its beginnings in IBM®'s DProbes [1, 5, 8]. DProbes included, in addition to the basic probing mechansim, a Reverse Polish Notation (RPN) interpreter, a probe manager, a Dynamic Probe Event Handler (DPEH), and a DProbes C Compiler (DPCC). On Rusty Russell's suggestion, the essential portions of the kernel probing mechanism and interfaces were abstracted out from DProbes, so probe handlers could be implemented as simple C functions that would run in the context of the kernel, when compiled in as a kernel module (or even compiled into the kernel). This minimal infrastructure could then live in the mainline kernel and other facilities could be built around it.

Thus, kprobes was born.

3 Kprobes basics

Kprobes works by modifying the program text by replacing the instruction at the probepoint with a breakpoint instruction. The instruction originally at the probepoint is copied to a separate scratch area, where it is suitable to be single-stepped out-of-line.[2] While the instruction size is known on RISC architectures, on CISC, the size of program text copied out to the scratch area is heuristically set to the maximum instruction size for that architecture.

The kprobes infrastructure in kernel is divided into architecture-agnostic and architecture-specific files. This design lends itself to easy porting to other architectures.

Kprobes makes use of the notifier mechanism in the kernel to hook the kernel exception handlers. For example, on the i386 architecture, kprobes is notified of int3 (breakpoint) traps, debug (single-step) traps and page faults. Since these are typically the exceptions that are of interest to a kernel debugger, the notifier chain put in place by kprobes can be used by the debuggers, too. However, the kprobes notifier is registered with the highest possible priority so as to ensure that it is the first to be invoked upon an exception hit. This is necessary since kprobes run transparent to the user, in contrast to a kernel debugger, which typically needs user intervention.

From a user's point of view, the important fields in `struct kprobe` are:

- `addr`: Text address where the kprobe is to be registered. The user *must* supply this field.
- `pre_handler`: User-defined routine that runs just *before* the instruction at the probepoint executes.
- `post_handler`: User-defined routine that runs just *after* the instruction at the probepoint executes.
- `fault_handler`: User-defined routine that runs in case of a fault during:
  - Execution of the instruction at the probepoint.
  - Execution of the user-defined handlers at the time of a kprobe hit.

[2] Single-stepping out-of-line allows us to leave the breakpoint instruction in place, so that the probepoint is never missed, even on an SMP system.

It is important that the handlers that are not being used in the context of a particular probe, be set to NULL.

3.1 Kprobe registration

A call to `register_kprobe()` triggers the kprobe registration process. The caller typically supplies the probepoint and the handlers to be run on the probepoint hit. Insertion of a kprobe is not allowed on sections of kernel code that are part of the kprobe infrastructure, nor on other kernel code used by kprobes (the exception handlers, for instance). A request to register a kprobe in these text areas fails with `-EINVAL`.

Control is then passed to the architecture specific helpers. These helpers run the required sanity checks to ensure that architectural restrictions are adhered to. (For example, certain instructions are unsafe to probe.) In addition, the original instruction at the probepoint is copied to a known scratch area, which is suitable to be single-stepped out of line.

The text at the probepoint is then replaced with the breakpoint opcode for that architecture. The icaches are then flushed so the change in program text is seen consistently on the other processors.

Some points of note:

- Except for PowerPC® and for IA64, kprobes makes no attempt to verify that the probepoint is at an instruction boundary.
- Certain architectures (such as x86_64, i386, and PowerPC) have incorporated NO_EXECUTE support in the kernel. The kprobe object, where the instruction would normally be stored, is typically not on an executable page. So for these architectures, the kprobes infrastructure allocates (using `module_alloc()`) and tracks scratch pages that have execute support, which are used to store a copy of the instruction at each probepoint.

3.2 Handler execution

Kprobes is notified first upon a breakpoint hit. Looking up the hash list of registered kprobes confirms whether the breakpoint hit was a result of a registered kprobe. It is possible that the exception was not a result of a kprobe hit (some other debugger could have inserted the breakpoint), in which case kprobes returns control to the notifier infrastructure, so other registered notifiers can be invoked. In the absence of any subsequent notifiers or if the other notifiers don't recognize the breakpoint as one of theirs, the default kernel exception handler takes over.

The kprobes status flags are set appropriately and the user-defined pre_handler is called. The pre_handler is where the kprobe user can gather the desired information, *before* the probed instruction is executed. Depending on the return value from the pre_handler, the instruction pointer (in `struct pt_regs`) is set to the copy of the original instruction at the probepoint location, and appropriate flags are set to single-step out-of-line.

Control returns to kprobes after the instruction copy is single-stepped. The post_handler is called if the kprobe has one associated with it. Here the kprobe user can gather information just *after* the probed instruction is executed.

Certain instructions change the execution flow (e.g., relative calls, returns, and branches). Since the instruction at the probepoint is executed out-of-line, instructions that depend on the instruction pointer at the time of execution,

need fixing up. (For x86_64 instructions that use rip-relative addressing, the instruction copy itself must be modified.) Such fixups are done transparent to the user, the flags are restored to their original states and the instruction pointer is set to the instruction immediately following the probepoint.

It is essential that the user-defined pre_ (and post_) handlers are error-free: that the handlers don't cause another exception (page_fault or otherwise). In case a fault does happen during the kprobe handler execution, the `kprobe_fault_handler()` is invoked. If the kprobe user has a fault_handler defined, it is given a chance to rectify the fault—especially if it is a fault deliberately induced by the user, for purposes of fault injection and the like. In case the user fault_handler isn't able to handle the fault, kprobes tries to fix it up on a best-effort basis.[3] If the referenced page is not memory resident, the function will return `-EFAULT`. In cases where the fixup isn't sufficient, the system fault handler kicks in, resulting, possibly, in a system crash.

Work is currently in progress to make the kprobe fault handling more robust—in particular, to protect the system from faults caused by erroneous handlers.

Preemption is disabled for the whole duration of kprobe processing, from the time kprobes is notified of the probepoint hit until the post_handler executes and any fault handling is complete.

3.3 Kprobe unregistration

Kprobe unregistration (triggered by calling `unregister_kprobe()`) entails putting the original instruction back at the probepoint location, flushing all icaches and removing the kprobe entry in the hash list. In case a separate scratch area is used for out-of-line single-stepping, it is returned to the free pool, so it can be reused.

In order to facilitate portability of kprobe modules, certain opaque datatypes are defined. These are aliased to appropriate architecture specific datatypes. Here is an example:

```
i386:
typedef u8 kprobe_opcode_t;

PowerPC:
typedef unsigned int kprobe_opcode_t;

ia64:
typedef struct kprobe_opcode {
        bundle_t bundle;
} kprobe_opcode_t;
```

4 Functional enhancements

The initial prototype had a few restrictions:

- At most one kprobe at an address
- Global spinlock to serialize execution of all kprobe handlers
- No handling of reentrant probes
- No support for function-return probes

These restrictions have now been remedied as is described in the following sections.

4.1 Jumper probes

In many a debug activity, there is a need to record or inspect arguments passed to a function. Jumper probes (jprobes, in short) satisfy

[3] As of writing this paper, kprobe excetption recovery is limited to a call to fixup_exception(). This enables a handler to safely call a fixup-enabled function, such as `__copy_from_user_inatomic()`.

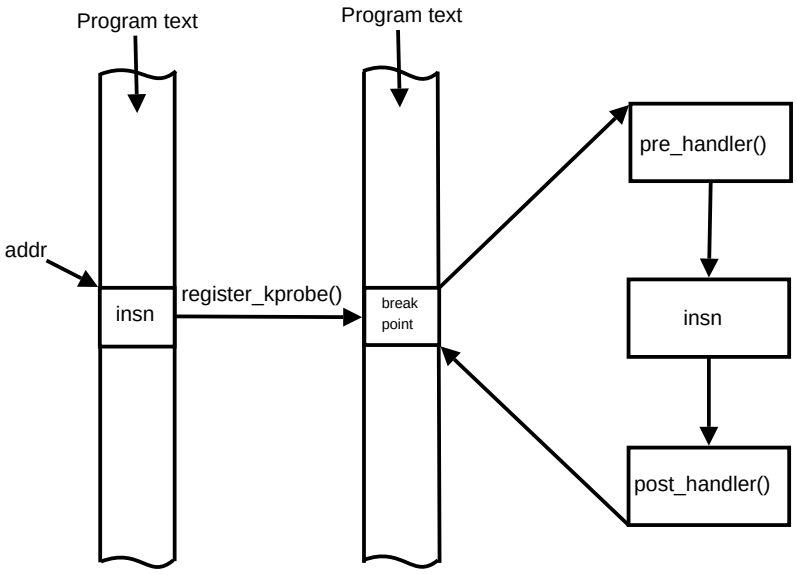

Figure 1: Kprobe flow of control

this need. To access the arguments of function `foo()`, jprobes requires the user to implement `foo'()`, a mirror function[4] of `foo()`. Using the underlying kprobes infrastructure, jprobes ensures that `foo'()` is given control before `foo()`, so the user can inspect or gather function arguments in runtime. Control is then returned to `foo()`, where normal execution continues.

4.1.1 The guts of jprobes

Jprobes is built on the kprobe infrastructure (In fact, `struct jprobe` has a `struct kprobe` embedded in it.) `register_jprobe()` triggers registration of a jprobe. The user supplies the entry point of the function to be probed as well as the mirror prototype that will be run before the function executes.

The in-kernel jprobes infrastructure provides two architecture-specific helpers that are aliased as the kprobe's pre and break_handlers. Upon the breakpoint hit, the `setjmp_pre_handler()` first saves the function argument space before transferring control to the mirrored function. This is necessary, since, by ABI definition, the callee owns the function argument space and could overwrite it as a result of tail-call optimization. The `pt_regs` are also saved and the instruction pointer is modified to point to the user-supplied mirror function. By returning a non-zero value, the `setjmp_pre_handler()` tells kprobes to just return from the exception without any further processing (setting up single-step, for instance). Thus, the mirror function executes upon return from the breakpoint exception.

The mirror function must call `jprobe_return()` once the user is done recording or inspecting the function arguments. `jprobe_return()` is a placeholder for the architecture-specific breakpoint instruction on most architectures—ia64 is a notable exception—which again drives us into the kprobes exception handler.

[4] Both `foo()` and `foo'()` have the same function prototype.

Though this exception entry isn't due to a kprobe hit, the kprobe state variables indicate that a kprobe is in process, indicating the possibility of this being a return from a jprobe. The clincher is the presence of a break_handler associated with the kprobe in process. So, the `break_handler()` is called.

The `longjmp_break_handler()` now gets control and does basic sanity checks and then restores the saved argument space and the saved `pt_regs`. Upon successful return from the `longjmp_break_handler()`, execution continues as it would following a normal kprobe hit.

`unregister_jprobe()` does nothing more than unregistering the associated kprobe.

4.1.2 Overhead as compared to kprobes

Jprobes cause two breakpoint exceptions and a single-step exception, in addition to copying the `pt_regs` and the argument space. The overhead of a jprobe is therefore about 1.5 times that of a normal kprobe. Additionally, any kprobe optimization will benefit jprobes too.

4.2 Colocated probes

A fundamental restriction with the legacy kprobes code was that one could have at most one kprobe or one jprobe at any given probepoint. Features such as function-return probes require a probe at the entry to a function. This would mean that no other probe could be inserted at that function entry and this restriction had to be remedied. Another requirement was that the overloading of kprobes at the same location had to happen transparently to the user. This implied that no new interfaces could be introduced.

The concept of an "aggregate kprobe" (ap) was invented. An ap is a kprobe with special predefined handlers. When a second kprobe is registered at a particular probepoint (so that we have p1 and p2 probing the same address), kprobes creates an ap and puts p1 and p2 on the ap's list. The ap then replaces p1 in the hash list.

When a breakpoint associated with multiple kprobes is hit, the `aggregate_pre_handler()` is invoked. This walks the list of registered kprobes at the location, invoking the individual pre_handlers in turn. Similarly, the `aggregate_post_handler()` takes care of invoking the individual post_handlers. Kprobes keeps track of which kprobe's handler is currently being run, so that only its fault_handler is invoked if its associated `pre_/post_handler` generates a fault.

It is thus possible to have any number of kprobes at a given probepoint, along with at most one jprobe (due to the way jprobes work).

4.3 Function-return probes

A function-return probe (hereafter referred to simply as a "return probe") fires when a specified function returns. Such probes can be useful for function-boundary tracing, function timing, or tracking a function's return values. Return probes are currently implemented for the i386, x86_64, ia64, and PowerPC architectures.

The `register_kretprobe()` function takes as its sole argument a pointer to `struct kretprobe`. This object specifies the entry address of the function to be probed, the handler to be executed, and a value called *maxactive*, which is discussed later in this section.

A return probe is implemented as follows:

- When `register_kretprobe()` is called, kprobes establishes a probepoint at the entry point of the probed function.
- When the probed function is called, this entry probepoint is hit, and a special handler, `pre_handler_kretprobe()`, is run. Each architecture's ABI defines where the return address can be found upon entry to a function. For example, for i386 and x86_64, it's atop the stack; for ia64 and PowerPC, it's in a particular register. `pre_handler_kretprobe()` saves a copy of the return address and replaces it with the address of a special piece of code called the `kretprobe_trampoline()`.
- When the probed function executes a return instruction, control passes to kprobes via the `kretprobe_trampoline()`. Kprobes runs the user-specified handler associated with the return probe, then continues execution at the "real" (saved) return address.[5]

4.3.1 Return-probe instances

There may be multiple instances of the same function running ("active") at the same time:

- On an SMP system, several CPUs may be executing the same function simultaneously.
- A function may be recursive.
- A function may yield the CPU via preemption, by taking a mutex or semaphore, or by calling `schedule()` explicitly. Another task may subsequently enter the same function.

Kprobes needs to keep track of the "real" return address of every active instance of every return-probed function. The object used to track this information is `struct kretprobe_instance` (rpi for short). `pre_handler_kretprobe()`, which saves the return address, runs in an environment where it cannot sleep, so it cannot allocate rpis as they are needed. Therefore, `register_kretprobe()` pre-allocates all the rpis that are to be used for that particular return probe. Since kprobes cannot determine how many instances of a function might become active, we rely on the user's knowledge of the function.

Before calling `register_kretprobe()`, the caller sets the `maxactive` member of `struct kretprobe` accordingly. Kprobes documentation [2] in the Linux source tree provides guidelines for setting maxactive. It's not a disaster if maxactive is set too low; some probes will simply be missed. The `nmissed` field in `struct kretprobe` accumulates a count of such misses.

Support for a pool of "spare" rpis, which may be shared by all return probes in an instrumentation module, is being contemplated—the aim being to keep `nmissed` low without over-allocating rpis, even in cases where `maxactive` cannot be accurately estimated.

4.3.2 Implications of return-address replacement

The above-described implementation has several implications:

- An rpi must hang around until its function returns, even if the corresponding return probe has been unregistered.

[5] When the handler runs, the return value of the function is available to the user-specified handler in one of the CPU registers—for example, `regs->eax` for i386 or `regs->gpr[3]` for PowerPC.

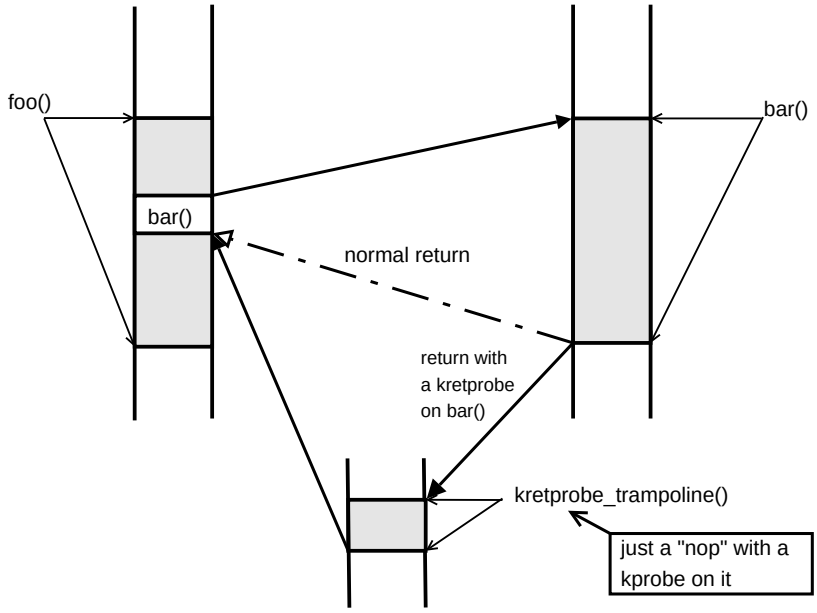

Figure 2: Return-probe flow of control

- Probing functions such as `schedule()` or `do_exit()` that return in strange ways (or not at all) will yield results that are valid, but perhaps unexpected to users unfamilar with how return probes work.
- When a task exits, kprobes must recycle any rpi objects associated with functions in that task that won't return. To streamline this operation, rpis are hashed by task pointer.
- Since data structures associated with return probes are constantly changing (as functions are called and return), locking cannot be solely by task, or solely by probe. Currently, all operations on return-probe data structures are guarded by a single global lock.
- When a task has one or more return-probed functions active, stack traces will typically report `kretprobe_trampoline()` rather than the actual return address for the probed functions.

4.3.3 Overhead of return probes

The overhead of a return probe is approximately the same as that of a jprobe. Registering an entry kprobe and matching return probe yields about the same overhead as the return probe alone.

4.4 Robust handling of reentrant probes

A general usage scenario for kprobes is one where a user writes a simple handler to gather the required data. This could sometimes involve a call to another kernel function.

Imagine a use-case where we have a kprobe on `foo()`. In the handler of this kprobe, if the user calls `bar()` and if `bar()` has a kprobe on it, we have a case of reentry. Another potential cause is a kprobe on an asynchronous routine (such as an IRQ handler), which can potentially be triggered during the processing on

a kprobe.[6]

Ideally, no other kprobe must be hit during a kprobe processing. Since this cannot realistically be enforced, there should be a graceful recovery mechanism. This required a few changes:

- Adding a kprobe state to indicate reentry
- Adding a counter to struct kprobe to track the number of reentries
- Adding an auxiliary structure to store state variables and flags of the kprobe that was being processed at the time of reentry.

The variable `current_kprobe` tells if we are in the midst of processing a kprobe. If it is not `NULL`, we have reentered due to another kprobe hit. In that case, the `kprobe_status` is set to indicate reentry and a counter `nmissed` in `structkprobe` is incremented to indicate it to the user. The state variables of the kprobe previously under process is saved in the auxiliary structure (called `struct prev_kprobe`) and the pre_handler is bypassed so that recursive reentries are avoided. Similar checks in the `kprobe_post_handler()` ensure that the post_handler for the reentered probe is bypassed. After the reentered kprobe's instruction is single-stepped, kprobes uses data in the auxiliary structure to continue processing of the original kprobe.

5 Performance enhancements

Though DProbes used per-CPU tracking and pre-probe locking, to keep matters simple, the legacy kprobes code used a single spinlock to serialize kprobe execution. Also, a single set of variables were used to track the kprobe being processed, its state, the processor flags at the time of exception, etc. This obviously did not scale well.

Among the alternatives that were considered were read-write locks and (better still) RCU [3, 7]. All the alternatives required independent tracking of the kprobe and its state on a per-CPU basis. This resulted in the creation of a kprobe control block (`struct kprobe_ctlblk`).

5.1 Tracking kprobes on a per-CPU basis

A `kprobe_ctlblk` is a per-CPU structure, which is used to track the status of the kprobe in process, the processor flags at the time of exception, a copy of the `pt_regs` at the time of jprobe invocation and, for the reentry case, some housekeeping information about the probe that was being processed at the time of reentry. Depending on the architecture, `struct kprobe_ctlblk` can contain additional elements. The i386 variant is shown below:

```
/* per-CPU kprobe control block */
struct kprobe_ctlblk {
 unsigned long kprobe_status;
 unsigned long kprobe_old_eflags;
 unsigned long kprobe_saved_eflags;
 long *jprobe_saved_esp;
 struct pt_regs  jprobe_saved_regs;
 kprobe_opcode_t jprobes_stack[MAX_STACK_SIZE];
 struct prev_kprobe prev_kprobe;
};

struct prev_kprobe {
 struct kprobe *kp;
 unsigned long status;
 unsigned long old_eflags;
 unsigned long saved_eflags;
};
```

Additionally, `current_kprobe` was made per-CPU and tells what kprobe is currently being processed on the CPU. It is is explicitly

[6] Most architectures run kprobes with interrupts enabled. An exception is i386.

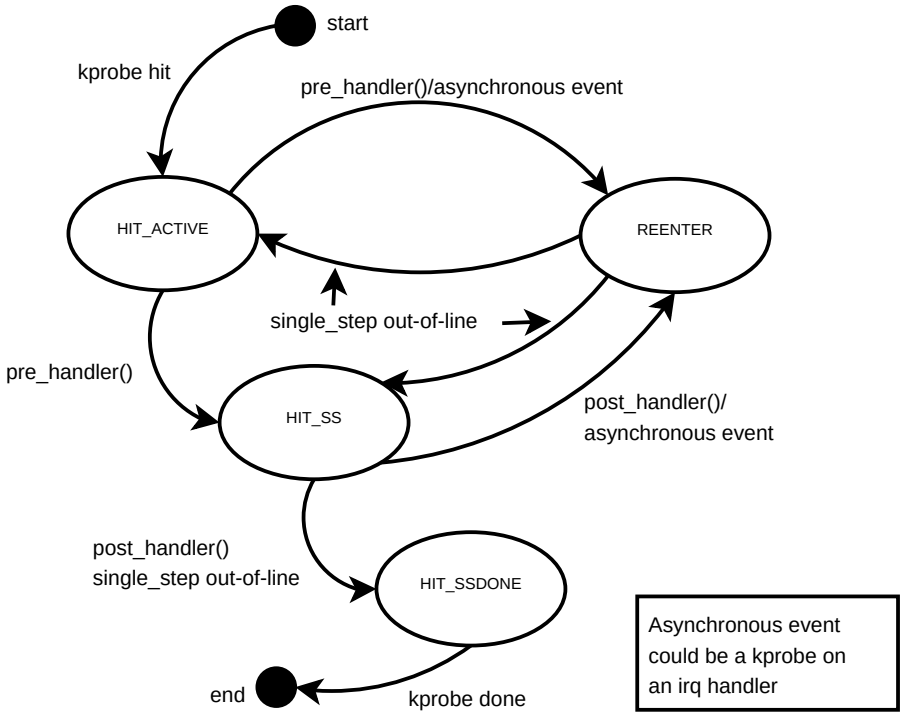

Figure 3: Kprobes state machine

set to `NULL` if no kprobe is currently active on the given CPU. `current_kprobe` is especially useful as it provides a quick and easy way to test if an exception induced entry into kprobe infrastructure is indeed due to a legitimate kprobe activity. It also provides an easy way to handle reentrancy.

5.2 Locking changes to use RCU

With per-CPU tracking out of the way, a locking scheme had to be worked out that would take advantage of it. A straightforward change would be to modify the serializing spinlock to a rwlock. The write lock could then be held during kprobe registration and unregistration while handlers could run with the read lock held. This approach was prototyped [6] and later discarded as there was a better approach—use RCU.

RCU requires that the write side be atomic while the read side can execute in a lock-free manner. Depending on the usage model, the RCU consumer has to use appropriate locking to ensure write-side atomicity.

Kprobes imposes the following restrictions:

- Handlers cannot block.
- Handlers run with preemption disabled.

`synchronize_sched()` is therefore a tailor-made solution for the update side, as it guarantees that all non-preemptive sections have completed. In addition, a mutex ensures serialization during hash-list updates.

With RCU, the hash lookup, which is a read-only operation, can be lock-free. This, however, brings a restriction that handlers have to be reentrant.

With these changes, multiple kprobes (same or different) can run in parallel, leading to a great improvement in scalability when compared to the earlier method.

6 The kprobe booster series

Kprobe and kretprobe (return-probe) boosters improve the performance of kprobes by eliminating exceptions, where possible. They can significantly reduce probepoint overhead, which can be important when probing time-sensitive or frequently executed code paths.

6.1 Kprobe-booster

As described in Section 3, in classic kprobes, a probepoint hit involves two exceptions, a breakpoint and a single-step. The former is essential, in order to break into kernel execution, but the latter may not be. Recall the steps that occur when an instruction is single-stepped out-of-line:

1. Kprobes single-steps a copy of the instruction, and the resulting trap returns control to kprobes.
2. The kprobe's post_handler, if any, is run.
3. After Step 1, the instruction pointer, return address, or other value may be wrong because of the difference in address between the instruction copy and the original instruction. Kprobes fixes things up as necessary.
4. Kprobes returns from the trap, and execution continues at the instruction following the probed instruction.

Step 2 can be eliminated if the kprobe doesn't have a post_handler. For many instruction types, no fixup is necessary and Step 3 can be eliminated. Steps 1 and 4 can then be replaced by a single jump. This jump instruction is simply appended to the buffer that contains the copy of the probed instruction.

6.2 Kretprobe-booster

The classic kretprobe uses two kprobes for each probe: one entry kprobe that saves the original return address, and the other on the trampoline. The latter can be replaced with assembly code which stores all registers, calls kretprobe's `trampoline_handler()`, restores registers, and finally returns to the original return address saved by the entry kprobe.

The boosters don't change existing kprobes API. These features are currently prototyped for i386 and merged into 2.6.16-rc1-mm5.

6.3 Implementation of the boosters

Kprobe booster involved the following steps:

- Addition of a tristate *boostable* flag:
  - -1 means that the probe can't be boosted.
  - 0 means that the probe can be boosted, but isn't ready to be boosted.
  - 1 means that the probe is ready to be boosted (i.e., the appropriate branch instruction has been appended).
- Identifying *boostable* instructions:
  - Any instruction that refer to the execution address (relative jump, call, software interrupt, etc.), cannot be boosted.

  - A machine-dependant corrolory: Any instruction that has a hardware side-effect (such as `cpuid`, `wrmsr`, etc.), cannot be boosted, since they may depend on the instruction pointer.

  The kprobe-booster classifies instructions accordingly, setting *boostable* to 0 if the instruction is boostable and to -1 if the instruction isn't boostable.

- Preparing to boost: If the probed instruction is boostable, the kprobe-booster must adjust the execution address register transparently as if the instruction has executed in-line. For this adjustment, a "jump" instruction is inserted after the copied instruction.

  The kprobe-booster uses information on the first kprobe hit to determine the exact location to insert the jump instruction. After the first single-step is performed, the execution address points the head of the next instruction on the instruction buffer. In the other words, this is the jump insertion point. Thus the kprobe-booster inserts a jump which jumps to the original address, and sets the *boostable* flag of the kprobe to 1.

- Boosting the kprobe: On subsequent hits, the kprobe is boosted if:

  - Its *boostable* flag is 1.
  - It does not have a post_handler.
  - The kernel preemption is disabled.

The kretprobe-booster works by emulating the breakpoint on the `kretprobe_trampoline()`. This is accomplished by saving the registers on the stack and calling the `trampoline_handler()`, which takes care of calling the user-defined handler and returning the rpi to the free list. Upon return from the `trampoline_handler()` the kretprobe-booster restores the saved registers, puts the original return address back on stack,[7] and returns to the normal execution flow.

7 Performance gains

7.1 Gains from kprobe locking changes

Locking changes—allowing handlers to run lockless and in parallel—significantly improved kprobes performance on SMP systems. Figure 4 illustrates performance gains of using RCU and per-CPU tracking for kprobes as compared to the legacy single-spinlock synchronization method. (The test basically is the result of a microbenchmark that drives CPUs to a rendezvous point through an IPI and the CPUs are made to spin in a loop calling a routine with a kprobe registered on it).

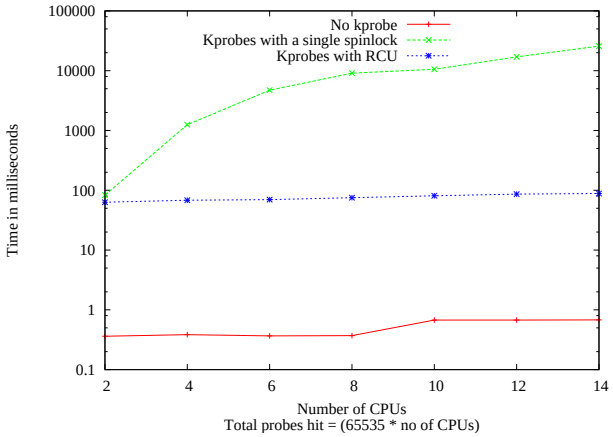

Figure 4: Kprobes performance comparison

[7] Some trickery involved here—at this point, the location where the return address has to be stored is occupied by EFLAGS. So EFLAGS is copied into the CS slot to make room for the return address. CS doesn't need to be recovered since we are always in the kernel context.

Kernel	kp	jp	rp	kp + rp	jp +rp
2.6.16 (no booster)	0.57	1.00	0.92	0.99	1.40
2.6.16-mm2 (with booster)	0.26	1.05	0.45	0.86	1.30

Table 1: Booster gains on an Intel® Pentium™ M, 1495MHz system

7.2 Gains from the kprobe-booster patchset

Table 1 illustrates the gains from the kprobe-booster patchset. All times in microseconds.

8 Work in progress

8.1 Userspace probes

Userspace probes (uprobes) provide a facility to dynamically instrument userspace applications. Key design issues include the following:

- Should the uprobe infrastructure be in kernel or userspace? What are the advantages/disadvantages of both?
- Should the probes be visible system-wide?
- Should the probes be inserted on a per-process or per-executable basis?
- If per-process, should it be inherited across a `fork()`?

An approach that uses some mm/vfs tricks to insert probes on a system-wide basis was prototyped [13]. This prototype provides a facility to insert probes even on applications that are yet to begin execution. Handlers run in the kernel context. There has been some pushback from the community to reconsider the approach.

At the time of this writing, a discussion was ensuing on LKML [12] with regard to what the most appropriate approach would be. A few options have been thrown up, one of which is to provide a system call interface (similar to ptrace) for this purpose.

8.2 Watchpoint probes

Watchpoint probes provide a simple interface for setting kernel-space watchpoints. Watchpoint probe mechanism uses the CPU's hardware debug registers to monitor data. At the time of this writing, an early prototype for the kernel watchpoint probe interface was avaliable for i386 [10, 11].

Many user-space debuggers, such as gdb, use the ptrace interface to set the watchpoint probes for local use. Typically, a user-space watchpoint is per-process, and so is set on a single CPU at a time. A kernel watchpoint, on the other hand, is set on all CPUs. Thus, there is a need to provide a nonintrusive, flexible, low-level facility for allocating debug registers. This facility would provide a way to relinquish global allocations when a more demanding user comes along and needs more debug registers for local use (or for a different kind of global use). To be nonintrusive, a global user of debug registers has to give them up when they are used by ptrace, for example.

Work is in progress to provide a common low-level mechanism that can be useful for ptrace and other users of debug registers.

9 Conclusions

Kprobes provides a simple, flexible, lightweight, easy-to-use[8] mechanism for creating ad hoc kernel instrumentation. As the kprobes user community has grown, there has been a demand for more ways to probe (jprobes, return probes, userspace probes, watchpoint probes), more flexibility for kprobes-based instrumentation (colocated probes, reentrant probes), and less overhead in very probe-intensive situations (locking changes, kprobe and kretprobe boosters). Most of these features are now in the Linux kernel; others are in the prototype stage.

10 Acknowledgements

The authors would like to acknowledge the work of the DProbes team, including Richard J. Moore, Suparna Bhattacharya, Vamsikrishna S, Bharata Rao, Michael Grundy, Thomas Zanussi, and others.

Special thanks to Maneesh Soni for his unrelenting support.

The authors wish to thank Roland McGrath, Rusty Lynch, Hien Nguyen, Will Cohen, and Andi Kleen for helping out at various stages during the kprobes development; and to acknowledge David Miller for his sparc64 kprobes port.

Thanks are due to all others in the Linux Kernel community who have helped improve the kprobes infrastructure through reviews, patches, bug reports, and suggestions.

[8] Refer to [2] for usage examples.

11 Legal statements

References

[1] DProbes. `http://sourceware.org/systemtap/kprobes/index.html`.

[2] `Documentation/kprobes.txt`. In the Linux Kernel sources.

[3] `Documentation/RCU/*`. In the Linux Kernel sources.

[4] SystemTap. http://sourceware.org/systemtap/.

[5] Suparna Bhattacharya. Dynamic Probes—Debugging by Stealth. In *Proceedings of Linux.Conf.Au*, 2003.

[6] Ananth N. Mavinakayanahalli. Kprobes: Remove global kprobe_lock, July 2005. http://sources.redhat.com/ml/systemtap/2005-q3/msg00182.html.

[7] Paul McKenney and Dipankar Sarma *et al.* Read Copy Update. In *Proceedings of the Ottawa Linux Symposium*, 2002.

[8] Richard J. Moore. A universal dynamic trace for Linux and other operating systems. In *FREENIX*, 2001.

[9] Prasanna S. Panchamukhi. Kernel debugging with Kprobes: Insert printk's into Linux kernel on the fly, August 2004. http://www-106.ibm.com/developerworks/library/l-kprobes.html?ca=dgr-lnx%w07Kprobe.

[10] Prasanna S. Panchamukhi. Hardware debug register allocation mechanism, August 2005. http://marc.theaimsgroup.com/?l=linux-kernel&m=112539056208001&w=2.

[11] Prasanna S. Panchamukhi. Lightweight interface for kernel-space watchpoint probes, August 2005. http://marc.theaimsgroup.com/?l=linux-kernel&m=112539056207779&w=2.

[12] Prasanna S. Panchamukhi. [RFC] Approaches to user-space probes, March 2006. http://marc.theaimsgroup.com/?l=linux-kernel&m=114344261621050&w=2.

[13] Prasanna S. Panchamukhi. User space probes support, March 2006. http://marc.theaimsgroup.com/?l=linux-kernel&m=114283503327535&w=2.