Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps

Jim Keniston
IBM
jkenisto@us.ibm.com

Ananth Mavinakayanahalli
IBM
ananth@in.ibm.com

Prasanna Panchamukhi
IBM
prasanna@in.ibm.com

Vara Prasad
IBM
varap@us.ibm.com

Abstract

The ptrace system-call API, though useful for many tools such as gdb and strace, generally proves unsatisfactory when tracing multithreaded or multi-process applications, especially in timing-dependent debugging scenarios. With the utrace kernel API, a kernel-side instrumentation module can track interesting events in traced processes. The uprobes kernel API exploits and extends utrace to provide kprobes-like, breakpoint-based probing of user applications.

We describe how utrace, uprobes, and kprobes together provide an instrumentation facility that overcomes some limitations of ptrace. For attendees, familiarity with a tracing API such as ptrace or kprobes will be helpful but not essential.

1 Introduction

For a long time now, debugging user-space applications has been dependent on the ptrace system call. Though ptrace has been very useful and will almost certainly continue to prove its worth, some of the requirements it imposes on its clients are considered limiting. One important limitation is performance, which is influenced by the high context-switch overheads inherent in the ptrace approach.

The utrace patchset [3] mitigates this to a large extent. Utrace provides in-kernel callbacks for the same sorts of events reported by ptrace. The utrace patchset reimplements ptrace as a client of utrace.

Uprobes is another utrace client. Analogous to kprobes for the Linux® kernel, uprobes provides a simple, easy-to-use API to dynamically instrument user applications. Details of the design and implementation of uprobes form the major portion of this paper.

We start by discussing the current situation in the user-space tracing world. Sections 2 and 3 discuss the various instrumentation approaches possible and/or available. Section 4 goes on to discuss the goals that led to the current uprobes design, while Section 5 details the implementation. In the later sections, we put forth some of the challenges, especially with regard to modifying text and handling of multithreaded applications. Further, there is a brief discussion on how and where we envision this infrastructure can be put to use. We finally conclude with a discussion on where this work is headed.

2 Ptrace-based Application Tracing

Like many other flavors of UNIX, Linux provides the ptrace system-call interface for tracing a running process. This interface was designed mainly for debugging, but it has been used for tracing purposes as well. This section surveys some of the ptrace-based tracing tools and presents limitations of the ptrace approach for low-impact tracing.

Ptrace supports the following types of requests:

- Attach to, or detach from, the process being traced (the “tracee”).
- Read or write the process’s memory, saved registers, or user area.
- Continue execution of the process, possibly until a particular type of event occurs (e.g., a system call is called or returns).

Events in the tracee turn into `SIGCHLD` signals that are delivered to the tracing process. The associated `siginfo_t` specifies the type of event.

2.1 Gdb

gdb is the most widely used application debugger in Linux, and it runs on other flavors of UNIX as well. *gdb* controls the program to be debugged using ptrace requests. *gdb* is used mostly as an interactive debugger, but also provides a batch option through which a series of *gdb* commands can be executed, without user intervention, each time a breakpoint is hit. This method of tracing has significant performance overhead. *gdb*'s approach to tracing multithreaded applications is to stop all threads whenever any thread hits a breakpoint.

2.2 Strace

The *strace* command provides the ability to trace calls and returns from all the system calls executed by the traced process. *strace* exploits ptrace's `PTRACE_SYSCALL` request, which directs ptrace to continue execution of the tracee until the next entry or exit from a system call. *strace* handles multithreaded applications well, and it has significantly less performance overhead than the *gdb* scripting method; but performance is still the number-one complaint about *strace*. Using *strace* to trace itself shows that each system call in the tracee yields several system calls in *strace*.

2.3 Ltrace

The *ltrace* command is similar to *strace*, but it traces calls and returns from dynamic library functions. It can also trace system calls, and extern functions in the traced program itself. *ltrace* uses ptrace to place breakpoints at the entry point and return address of each probed function. *ltrace* is a useful tool, but it suffers from the performance limitations inherent in ptrace-based tools. It also appears not to work for multithreaded programs.

2.4 Ptrace Limitations

If *gdb*, *strace*, and *ltrace* don't give you the type of information you're looking for, you might consider writing your own ptrace-based tracing tool. But consider the following ptrace limitations first:

- Ptrace is not a POSIX system call. Its behavior varies from operating system to operating system, and has even varied from version to version in Linux. Vital operational details ("Why do I get two SIGCHLDs here? Am I supposed to pass the process a SIGCONT or no signal at all here?") are not documented, and are not easily gleaned from the kernel source.
- The amount of perseverence and/or luck you need to get a working program goes up as you try to monitor more than one process or more than one thread.
- Overheads associated with accessing the tracee's memory and registers are enormous—on the order of 10x to 100x or more, compared with equivalent in-kernel access. Ptrace's PEEK-and-POKE interface provides very low bandwidth and incurs numerous context switches.
- In order to trace a process, the tracer must become the tracee's parent. To attach to an already running process, then, the tracer must muck with the tracee's lineage. Also, if you decide you want to apply more instrumentation to the same process, you have to detach the tracer already in place.

3 Kernel-based Tracing

In the early days of Linux, the kernel code base was manageable and most people working on the kernel knew their core areas intimately. There was a definite pushback from the kernel community towards including any tracing and/or debugging features in the mainline kernel.

Over time, Linux became more popular and the number of kernel contributors increased. A need for a flexible tracing infrastructure was recognized. To that end, a number of projects sprung up and have achieved varied degrees of success.

We will look at a few of these projects in this section. Most of them are based on the kernel-module approach.

3.1 Kernel-module approach

The common thread among the following approaches is that the instrumentation code needs to run *in kernel*

mode. Since we wouldn't want to burden the kernel at times when the instrumentation isn't in use, such code is introduced only when needed, in the form of a kernel module.

3.1.1 Kprobes

Kprobes [2] is perhaps the most widely accepted of all dynamic instrumentation mechanisms currently available for the Linux kernel. Kprobes traces its roots back to DProbes (discussed later). In fact, the first version of kprobes was a patch created by just taking the minimal, kernel-only portions of the DProbes framework.

Kprobes allows a user to dynamically insert "probes" into specific locations in the Linux kernel. The user specifies "handlers" (instrumentation functions) that run before and/or after the probed instruction is executed. When a probepoint (which typically is an architecture-specific breakpoint instruction) is hit, control is passed to the kprobes infrastructure, which takes care of executing the user-specified handlers. [2] provides an in-depth treatment of the kprobes infrastructure (which, incidentally, includes jprobes and function-return probes).

The kernel-module approach was a natural choice for kprobes: after all, the goal is to access and instrument the Linux kernel. Given the privilege and safety requirements necessary to access kernel data structures, the kernel-module approach works very well.

3.1.2 Utrace

A relatively new entrant to this instrumentation space is utrace. This infrastructure is intended to serve as an abstraction layer to write the next generation of user-space tracing and debugging applications.

One of the primary grouses kernel hackers have had about ptrace is the lack of separation/layering of code between architecture-specific and -agnostic parts. Utrace aims to mitigate this situation. Ptrace is now but one of the clients of utrace.

Utrace, at a very basic level, is an infrastructure to monitor individual Linux "threads"—each represented by a `task_struct` in the kernel. An "engine" is utrace's basic control unit. Typically, each utrace client establishes an engine for each thread of interest. Utrace provides three basic facilities on a per-engine basis:

- *Event reporting:* Utrace clients register callbacks to be run when the thread encounters specific events of interest. These include system call entry/exit, signals, exec, clone, exit, etc.
- *Thread control:* Utrace clients can inject signals, request that a thread be stopped from running in user-space, single-step, block-step, etc.
- *Thread machine state access:* While in a callback, a client can inspect and/or modify the thread's core state, including the registers and u-area.

Utrace works by placing tracepoints at strategic points in kernel code. For traced threads, these tracepoints yield calls into the registered utrace clients. These callbacks, though happening in the context of a user process, happen when the process is executing in kernel mode. In other words, utrace clients run in the kernel. Ptrace is one utrace client that lives in the kernel. Other clients—especially those used for *ad hoc* instrumentation—may be implemented as kernel modules.

3.1.3 Uprobes

Uprobes is another client of utrace. As such, uprobes can be seen as a flexible user-space probing mechanism that comes with all the power, but not all the constraints, of ptrace. Just as kprobes creates and manages probepoints in kernel code, uprobes creates and manages probepoints in user applications. The uprobes infrastructure, like ptrace, lives in the kernel.

A uprobes user writes a kernel module, specifying for each desired probepoint the process and virtual address to be probed and the handler to run when the probepoint is hit.

A uprobes-based module can also use the utrace and/or kprobes APIs. Thus a single instrumentation module can collect and correlate information from the user application(s), shared libraries, the kernel/user interfaces, and the kernel itself.

3.1.4 SystemTap

SystemTap [4] provides a mechanism to use the kernel (and later, user) instrumentation tools, through a simple

awk-like event/action scripting language. A SystemTap script specifies code points to probe, and for each, the instrumentation code to run when the probepoint is hit. The scripting language provides facilities to aggregate, collate, filter, present, and analyze trace data in a manner that is intuitive to the user.

From a user's script, the *stap* command produces and loads a kernel module containing calls to the kprobes API. Trace data from this module is passed to user space using efficient mechanisms (such as relay channels), and the SystemTap post-processing infrastructure takes care of deciphering and presenting the gathered information in the format requested by the user, on the *stap* command's standard output.

SystemTap can be viewed as a wrapper that enables easy use of kernel instrumentation mechanisms, including, but not limited to, kprobes. Plans are afoot to exploit utrace, uprobes, the "markers" static tracing infrastructure, and a performance-monitoring-hardware infrastructure (perfmon), as they become part of the mainline kernel.

3.2 Interpreter-based approaches

Two more instrumentation tools, DProbes and DTrace, bear mention. Both tools endeavor to provide integrated tracing support for user applications and kernel code.

DProbes [1] traces its roots to IBM's OS/2® operating system, but has never found a foothold in Linux, except as the forebear of kprobes. DProbes instrumentation is written in the RPN programming language, the assembly language for a virtual machine whose interpreter resides in the kernel. For the i386 architecture, the DProbes C Compiler (dpcc) translates instrumentation in a C-like language into RPN. Like kprobes and uprobes, DProbes allows the insertion of probepoints anywhere in user or kernel code.

DTrace is in use on Solaris and FreeBSD. Instrumentation is written in the D programming language, which provides aggregation and presentation facilities similar to those of SystemTap. As with DProbes, DTrace instrumentation runs on an interpreter in the kernel. DTrace kernel probepoints are limited to a large but predefined set. A DTrace probe handler is limited to a single basic block (no ifs or loops).

Tool	Event Counted	Overhead (usec) per Event
ltrace -c	function calls	22
gdb -batch	function calls	265
strace -c	system calls	25
kprobes	function calls	0.25
uprobes	function calls	3.4
utrace	system calls	0.16

Table 1: Performance of ptrace-based tools (top) vs. *ad hoc* kernel modules (bottom)

3.3 Advantages of kernel-based application tracing

- Kernel-based instrumentation is inherently fast. Table 1 shows comparative performance numbers of ptrace-based tools vs. kernel modules using kprobes, uprobes, and utrace. These measurements were taken on a Pentium® M (1495 MHz) running the utrace implementation of ptrace.
- Most systemic problems faced by field engineers involve numerous components. Problems can percolate from a user-space application all the way up to a core kernel component, such as the block layer or the networking stack. Providing a unified view of the flow of control and/or data across user space and kernel is possible only via kernel-based tracing.
- Kernel code runs at the highest privilege and as such, can access all kernel data structures. In addition, the kernel has complete freedom and access to the process address space of all user-space processes. By using safe mechanisms provided in kernel to access process address spaces, information related to the application's data can be gleaned easily.
- Ptrace, long the standard method for user-space instrumentation, has a number of shortcomings, as discussed in Section 2.4.

3.4 Drawbacks of kernel-based application tracing

- The kernel runs at a higher privilege level and in a more restricted environment than applications. Assumptions and programming constructs that are valid for user space don't necessarily hold for the kernel.

- Not everybody is a kernel developer. It's not prudent to expect someone to always write "correct" kernel code. And, while dealing with the kernel, one mistake may be too many. Most application developers and system administrators, understandably, are hesitant to write kernel modules.
- The kernel has access to user-space data, but can't easily get at all the information (e.g., symbol table, debug information) to decode it. This information must be provided in or communicated to the kernel module, or the kernel must rely on post-processing in user space.

SystemTap goes a long way in mitigating these drawbacks. For kernel tracing, SystemTap uses the running kernel's debug information to determine source file, line number, location of variables, and the like. For applications that adhere to the DWARF format, it wouldn't be hard for SystemTap to provide address/symbol resolution.

4 Uprobes Design Goals

The idea of user-space probes has been around for years, and in fact there have been a variety of proposed implementations. For the current, utrace-based implementation, the design goals were as follows:

- *Support the kernel-module approach.* Uprobes follows the kprobes model of dynamic instrumentation: the uprobes user creates an *ad hoc* kernel module that defines the processes to be probed, the probepoints to establish, and the kernel-mode instrumentation handlers to run when probepoints are hit. The module's `init` function establishes the probes, and its cleanup function removes them.
- *Interoperate with kprobes and utrace.* An instrumentation module can establish probepoints in the kernel (via kprobes) as well as in user-mode programs. A uprobe or utrace handler can dynamically establish or remove kprobes, uprobes, or utrace-event callbacks.
- *Minimize limitations* on the types of applications that can be probed and the way in which they can be probed. In particular, a uprobes-based instrumentation module can probe any number of processes of any type (related or unrelated), and can probe multithreaded applications of all sorts. Probes can be established or removed at any stage in a process's lifetime. Multiple instrumentation modules can probe the same processes (and even the same probepoints) simultaneously.
- *Probe processes, not executables.* In earlier versions of uprobes, a probepoint referred to a particular instruction in a specified executable or shared library. Thus a probepoint affected all processes (current and future) running that executable. This made it relatively easy to probe a process right from exec time, but the implications of this implementation (e.g., inconsistency between the in-memory and on-disk images of a probed page) were unacceptable. Like ptrace, uprobes now probes per-process, and uses `access_process_vm()` to ensure that a probed process gets its own copy of the probed page when a probepoint is inserted.
- *Handlers can sleep.* As discussed later in Section 5, uprobes learns of each breakpoint hit via utrace's signal-callback mechanism. As a result, the user-specified handler runs in a context where it can safely sleep. Thus, the handler can access any part of the probed process's address space, resident or not, and can undertake other operations (such as registering and unregistering probes) that a kprobe handler cannot. Compared with earlier implementations of uprobes, this provides more flexibility at the cost of some additional per-hit overhead.
- *Performance.* Since a uprobe handler runs in the context of the probed process, ptrace's context switches between the probed process and the instrumentation parent on every event are eliminated. The result is significantly less overhead per probe hit than in ptrace, though significantly more than in kprobes.
- *Safe from user-mode interference.* Both uprobes and the instrumentation author must account for the possibility of unexpected tinkering with the probed process's memory—e.g., by the probed process itself or by a malicious ptrace-based program—and ensure that while the sabotaged process may crash, the kernel's operation is not affected.
- *Minimize intrusion on existing Linux code.* For hooks into a process's events of interest, uprobes

uses those provided by utrace. Uprobes creates per-task and per-process data structures, but maintains them independently of the corresponding data structures in the core kernel and in utrace. As a result, at this writing, the base uprobes patch, including i386 support, includes only a few lines of patches against the mainline kernel source.

- *Portable to multiple architectures.* At the time of this writing, uprobes runs on the i386 architecture. Ports are underway to PowerPC, x86_64, and zLinux (s390x). Except for the code for single-stepping out of line, which you need (adapting from kprobes, typically) if you don't want to miss probepoints in multithreaded applications, a typical uprobes port is on the order of 50 lines.

5 Uprobes Implementation Overview

Uprobes can be thought of as containing the following overlapping pieces:

- data structures;
- the register/unregister API;
- utrace callbacks; and
- architecture-specific code.

In this section, we'll describe each of these pieces briefly.

5.1 Data structures

Uprobes creates the following internal data structures:

- `uprobe_process` – one for each probed process;
- `uprobe_task` – one for each thread (task) in a probed process; and
- `uprobe_kimg` (Kernel IMaGe) – one for each probepoint in a probed process. (Multiple uprobes at the same address map to the same `uprobe_kimg`.)

Each `uprobe_task` and `uprobe_kimg` is owned by its `uprobe_process`. Data structures associated with return probes (uretprobes) and single-stepping out of line are described later.

5.2 The register/unregister API

The fundamental API functions are `register_uprobe()` and `unregister_uprobe()`, each of which takes a pointer to a uprobe object defined in the instrumentation module. A uprobe object specifies the pid and virtual address of the probepoint, and the handler function to be run when the probepoint is hit.

The `register_uprobe()` function finds the `uprobe_process` specified by the pid, or creates the `uprobe_process` and `uprobe_task(s)` if they don't already exist. `register_uprobe()` then creates a `uprobe_kimg` for the probepoint, queues it for insertion, requests (via utrace) that the probed process "quiesce," sleeps until the insertion takes place, and then returns. (If there's already a probepoint at the specified address, `register_uprobe()` just adds the uprobe to that `uprobe_kimg` and returns.)

Note that since all threads in a process share the same text, there's no way to register a uprobe for a particular thread in a multithreaded process.

Once all threads in the probed process have quiesced, the last thread to quiesce inserts a breakpoint instruction at the specified probepoint, rouses the quiesced threads, and wakes up `register_uprobe()`.

`unregister_uprobe()` works in reverse: Queue the `uprobe_kimg` for removal, quiesce the probed process, sleep until the probepoint has been removed, and delete the `uprobe_kimg`. If this was the last `uprobe_kimg` for the process, `unregister_uprobe()` tears down the entire `uprobe_process`, along with its `uprobe_tasks`.

5.3 Utrace callbacks

Aside from registration and unregistration, everything in uprobes happens as the result of a utrace callback. When a `uprobe_task` is created, uprobes calls `utrace_attach()` to create an engine for that thread, and listens for the following events in that task:

- *breakpoint signal* (SIGTRAP on most architectures): Utrace notifies uprobes when the probed task hits a breakpoint. Uprobes determines which probepoint (if any) was hit, runs the associated probe handler(s), and directs utrace to single-step the probed instruction.

- *single-step signal* (SIGTRAP on most architectures): Utrace notifies uprobes after the probed instruction has been single-stepped. Uprobes does any necessary post-single-step work (discussed in later sections), and directs utrace to continue execution of the probed task.
- *fork/clone:* Utrace notifies uprobes when a probed task forks a new process or clones a new thread. When a new thread is cloned for the probed process, uprobes adds a new `uprobe_task` to the `uprobe_process`, complete with an appropriately programmed utrace engine. In the case of `fork()`, uprobes doesn't attempt to preserve probepoints in the child process, since each uprobe object refers to only one process. Rather, uprobes iterates through all the probe addresses in the parent and removes the breakpoint instructions in the child.
- *exit:* Utrace notifies uprobes when a probed task exits. Uprobes deletes the corresponding `uprobe_task`. If this was the last `uprobe_task` for the process, uprobes tears down the entire `uprobe_process`. (Since the process is exiting, there's no need to remove breakpoints.)
- *exec:* Utrace notifies uprobes when a probed task execs a new program. (Utrace reports exit events for any other threads in the process.) Since probepoints in the old program are irrelevant in the new program, uprobes tears down the entire `uprobe_process`. (Again, there's no need to remove breakpoints.)
- *quiesce:* Uprobes listens for the above-listed events all the time. By contrast, uprobes listens for quiesce events only while it's waiting for the probed process to quiesce, in preparation for insertion or removal of a breakpoint instruction. (See Section 6.2).

5.4 Architecture-specific code

Most components of the architecture-specific code for uprobes are simple macros and inline functions. Support for return probes (uretprobes) typically adds 10–40 lines. By far the majority of the architecture-specific code relates to "fix-ups" necessitated by the fact that we single-step a copy of the probed instruction, rather than single-stepping it in place. See "Tracing multithreaded applications: SSOL" in the next section.

6 Uprobes Implementation Notes

6.1 Tracing multithreaded applications: SSOL

Like some other tracing and debugging tools, uprobes implements a probepoint by replacing the first byte(s) of the instruction at the probepoint with a breakpoint instruction, after first saving a copy of the original instruction. After the breakpoint is hit and the handler has been run, uprobes needs to execute the original instruction in the context of the probed process. There are two commonly accepted ways to do this:

- *Single-stepping inline (SSIL):* Temporarily replace the breakpoint instruction with the original instruction; single-step the instruction; restore the breakpoint instruction; and allow the task to continue. This method is typically used by interactive debuggers such as gdb.
- *Single-stepping out of line (SSOL):* Place a copy of the original instruction somewhere in the probed process's address space; single-step the copy; "fix up" the task's state as necessary; and allow the task to continue. If the effect of the instruction depends on its address (e.g., a relative branch), the task's registers and/or stack must be "fixed up" after the instruction is executed (e.g., to make the program counter relative to the address of the original instruction, rather than the instruction copy). This method is used by kprobes for kernel tracing.

The SSIL approach doesn't work acceptably for multithreaded programs. In particular, while the breakpoint instruction is temporarily removed during single-stepping, another thread can sail past the probepoint. We considered the approach of quiescing, or otherwise blocking, all threads every time one hits a probepoint, so that we could be sure of no probe misses during SSIL, but we considered the performance implications unacceptable.

In terms of implementation, SSOL has two important implications:

- Each uprobes port must provide code to do architecture-specific post-single-step fix-ups. Much of this code can be filched from kprobes,

but there are additional implications for uprobes. For example, uprobes must be prepared to handle any instruction in the architecture's instruction set, not just those used in the kernel; and for some architectures, uprobes must be able to handle both 32-bit and 64-bit user applications.

- The instruction copy to be single-stepped must reside somewhere in the probed process's address space. Since uprobes can't know what other threads may be doing while a thread is single-stepping, the instruction copy can't reside in any location legitimately used by the program. After considering various approaches, we decided to allocate a new VM area in the probed process to hold the instruction copies. We call this the *SSOL area*.

To minimize the impact on the system, uprobes allocates the SSOL area only for processes that are actually probed, and the area is small (typically one page) and of fixed size. "Instruction slots" in this area are allocated on a per-probepoint basis, so that multiple threads can single-step in the same slot simultaneously. Slots are allocated only to probepoints that are actually hit. If uprobes runs out of free slots, slots are recycled on a least-recently-used basis.

6.2 Quiescing

Uprobes takes a fairly conservative approach when inserting and removing breakpoints: all threads in the probed process must be "quiescent" before the breakpoint is inserted/removed.

Our approach to quiescing the threads started out fairly simple: For each task in the probed process, the `[un]register_uprobe()` function sets the `UTRACE_ACTION_QUIESCE` flag in the `uprobe_task`'s engine, with the result that utrace soon brings the task to a stopped state and calls uprobes's quiesce callback for that task. After setting all tasks on the road to quiescence, `[un]register_uprobe()` goes to sleep. For the last thread to quiesce, the quiesce callback inserts or removes the requested breakpoint, wakes up `[un]register_uprobe()`, and rouses all the quiesced threads.

It turned out to be more complicated than that. For example:

- If the targeted thread is already quiesced, setting the `UTRACE_ACTION_QUIESCE` flag causes the quiesce callback to run immediately in the context of the `[un]register_uprobe()` task. This breaks the "sleep until the last quiescent thread wakes me" paradigm.

- If a thread hits a breakpoint on the road to quiescence, its quiesce callback is called before the signal callback. This turns out to be a bad place to stop for breakpoint insertion or removal. For example, if we happen to remove the `uprobe_kimg` for the probepoint we just hit, the subsequent signal callback won't know what to do with the SIGTRAP.

- If `[un]register_uprobe()` is called from a uprobe handler, it runs in the context of the probed task. Again, this breaks the "sleep until the last quiescent thread wakes me" paradigm.

It turned out to be expedient to establish an "alternate quiesce point" in uprobes, in addition to the quiesce callback. When it finishes handling a probepoint, the uprobes signal callback checks to see whether the process is supposed to be quiescing. If so, it does essentially what the quiesce callback does: if it's the last thread to "quiesce," it processes the pending probe insertion or removal and rouses the other threads; otherwise, it sleeps in a pseudo-quiesced state until the "last" thread rouses it. Consequently, if `[un]register_uprobe()` sees that a thread is currently processing a probepoint, it doesn't try to quiesce it, knowing that it will soon hit the alternate quiesce point.

6.3 User-space return probes

Uprobes supports a second type of probe: a return probe fires when a specified function returns. User-space return probes (uretprobes) are modeled after return probes in kprobes (kretprobes).

When you register a uretprobe, you specify the process, the function (i.e., the address of the first instruction), and a handler to be run when the function returns.

Uprobes sets a probepoint at the entry to the function. When the function is called, uprobes saves a copy of the return address (which may be on the stack on in a register, depending on the architecture) and replaces the return address with the address of the "uretprobe trampoline," which is simply a breakpoint instruction.

When the function returns, control passes to the trampoline, the breakpoint is hit, and uprobes gains control. Uprobes runs the user-specified handler, then restores the original return address and allows the probed function to return.

In uprobes, the return-probes implementation differs from kprobes in several ways:

- The user doesn't need to specify how many "return-probe instance" objects to preallocate. Since uprobes runs in a context where it can use `kmalloc()` freely, no preallocation is necessary.
- Each probed process needs a trampoline in its address space. We use one of the slots in the SSOL area for this purpose.
- As in kprobes, it's permissible to unregister the return probe while the probed function is running. Even after all probes have been removed, uprobes keeps the `uprobe_process` and its `uprobe_tasks` (and utrace engines) around as long as necessary to catch and process the last hit on the uretprobe trampoline.

6.4 Registering/unregistering probes in probe handlers

A uprobe or uretprobe handler can call any of the functions in the uprobes API. A handler can even unregister its own probe. However, when invoked from a handler, the actual [un]register operations do not take place immediately. Rather, they are queued up and executed after all handlers for that probepoint have been run and the probed instruction has been single-stepped. (Specifically, queued [un]registrations are run right after the previously described "alternate quiesce point.") If the `registration_callback` field is set in the uprobe object to be acted on, uprobes calls that callback when the [un]register operation completes.

An instrumentation module that employs such dynamic [un]registrations needs to keep track of them: since a module's uprobe objects typically disappear along with the module, the module's cleanup function should not exit while any such operations are outstanding.

7 Applying Uprobes

We envision uprobes being used in the following situations, for debugging and/or for performance monitoring:

- Tracing timing-sensitive applications.
- Tracing multithreaded applications.
- Tracing very large and/or complex applications.
- Diagnosis of systemic performance problems involving multiple layers of software (in kernel and user space).
- Tracing applications in creative ways – e.g., collecting different types of information at different probepoints, or dynamically adjusting which code points are probed.

8 Future Work

What's next for utrace, uprobes, and kprobes? Here are some possibilities:

- *Utrace += SSOL.* As discussed in Section 6.1, single-stepping out of line is crucial for the support of probepoints in multithreaded processes. This technology may be migrated from uprobes to utrace, so that other utrace clients can exploit it. One such client might be an enhanced ptrace.
- *SystemTap += utrace + uprobes.* Some SystemTap users want support for probing in user space. Some potential utrace and uprobes users might be more enthusiastic given the safety and ease of use provided by SystemTap.
- *Registering probes from kprobe handlers.* Utrace and uprobe handlers can register and unregister utrace, uprobes, and kprobes handlers. It would be nice if kprobes handlers could do the same. Perhaps the effect of sleep-tolerant kprobe handlers could be approximated using a kernel thread that runs deferred handlers. This possibility is under investigation.
- *Tracing Java.* Uprobes takes us closer to dynamic tracing of the Java Virtual Machine and Java applications.

- *Task-independent exec hook.* Currently, uprobes can trace an application if it's already running, or if it is known which process will spawn it. Allowing tracing of applications that are yet be started and are of unknown lineage will help to solve problems that creep in during application startup.

9 Acknowledgements

The authors wish to thank Roland McGrath for coming up with the excellent utrace infrastructure, and also for providing valuable suggestions and feedback on uprobes. Thanks are due to David Wilder and Srikar Dronamraju for helping out with testing the uprobes framework. Thanks also to Dave Hansen and Andrew Morton for their valuable suggestions on the VM-related portions of the uprobes infrastructure.

10 Legal Statements

References

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

[2] Mavinakayanahalli et al. Probing the Guts of Kprobes. In *Proceedings of the Ottawa Linux Symposium*, 2006.

[3] Roland McGrath. Utrace home page. `http://people.redhat.com/roland/utrace/`.

[4] SystemTap project team. SystemTap. `http://sourceware.org/systemtap/`.