Problem Solving With Systemtap

Frank Ch. Eigler

Red Hat

`fche@redhat.com`

Abstract

Systemtap is becoming a useful tool to help solve low-level OS problems. Most features described in the future tense at last year's OLS are now complete. We review the status and recent developments of the system. In passing, we present solutions to some complex low-level problems that bedevil kernel and application developers.

Systemtap recently gained support for static probing markers that are compiled into the kernel, to complement the dynamic `kprobes` system. It is a simple and fast mechanism, and we invite kernel developers and other trace-like tools to adopt it.

1 Project status

At OLS 2005, we presented[4] systemtap, the open source tool being developed for tracing/probing of a live unmodified linux system. It accepts commands in a simple scripting language, and hooks them up to probes inserted at requested code locations within the kernel. When the kernel trips across the probes, routines in a compiled form of the script are quickly run, then the kernel resumes. Over the last year, with the combined efforts of a dozen developers supported by four companies, much of this theory has turned into practice.

1.1 Scripting language

The systemtap script is a small domain-specific language resembling `awk` and C. It has only a few data types (integers and strings, plus associative arrays of these), full control structures (blocks, conditionals, loops, functions). It is light on punctuation (semicolons optional) and on declarations (types are inferred and checked automatically). Its core concept, the "probe," consists of a probe point (its trigger event) and its handler (the associated statements).

Probe points name the kernel events at which the statements should be executed. One may name nearly any function, or a source file and line number where the breakpoint is to be set (just like in a symbolic debugger), or request an asynchronous event like a periodic timer. Systemtap defines a hierarchical probe point namespace, a little like DNS.

Probe handlers have few constraints. They can print data right away, to provide a sort of on-the-fly `printk`. Or, they can save a timestamp in a variable and compare it with a later probe hit, to derive timing profiles. Or, they can follow kernel data structures, and speak up if something is amiss.

The scripting language is implemented by a translator that creates C code, which is in turn compiled into a binary kernel module. Probe

points are mapped to virtual addresses by reference to the kernel's DWARF debugging information left over from its build. The same data is used to resolve references to kernel "target-side" variables. Their compiled nature allows even elaborate probe scripts to run fast.

Safety is an essential element of the design. All the language constructs are subjected to translation- and run-time checks, which aim to prevent accidental damage to the system. This includes prevention of infinite loops, memory use, and recursion, pointer faults, and several others. Many checks may be inspected within the translator-generated C code.

Some safety mechanisms are incomplete at present. Systemtap contains a blacklist of kernel areas that are deemed unsafe to probe, since they might trigger infinite probing recursion, locking reentrancy, or other nasty phenomena. This blacklist is just getting started, so probing using broad wildcards is a recipe for panics. Similarly, we haven't sufficiently analyzed the script-callable utility functions like our `gettimeofday` wrapper to ensure that it is safe to call from any probe handler. Work in these directions is ongoing.

1.2   Recent developments

The most basic development since last summer is that the system *works*, whereas last year we relied on several mock-ups. You can download it[1], build it, and use it on your already installed kernels today. It is not perfect nor complete, but nor is it vapourware.

kprobes has received a heart transplant. The most significant of these was truly concurrent probing on multiprocessor machines, made possible by a switch to RCU data structures.

Implementation details are discussed in a separate paper [3] during this conference. In order to exploit the parallelism enabled by this improvement, systemtap supports variables to track global *statistics aggregates* like averages or counts using contention-free data structures.

For folks who like the exhilaration of full control, or have a distaste for the scripting language, Systemtap supports bypassing the cushion. In "guru mode," systemtap allows intermingling of literal C code with script, to go beyond the limitations of pure script code. One can query or manipulate otherwise inaccessible kernel state directly, but bears responsibility for doing so *safely*.

Systemtap documentation is slowly growing, as is our collection of sample scripts. There is a fifteen-page language tutorial, and a few dozen worked out examples on our web site. More and more first-time users are popping up on the mailing list, so we are adapting to supporting new users, not just fellow project developers.

1.3   Usage scenarios

While it's still early, systemtap has suggested several uses. First is simple exploration and profiling. A probe on "timer.profile" and collecting stack backtrace samples gets one a coarse profile. A probe at a troubled function, with a similar a stack backtrace, tells one who is the troublemaker. Probes on system call handling functions (or more conveniently named aliases defined in a library) give one an instant system-wide `strace`, with as much filtering and summarizing as one may wish. As a taste, figure 1 demonstrates probing function nesting within a compilation unit.

Daniel Berranger [1] arranged to run systemtap throughout a Linux boot sequence (`/etc/init.d` scripts) to profile the I/O

---

[1] http://sourceware.org/systemtap/

and forking characteristics of the many startup scripts and daemons. Some wasteful behavior showed up right away in the reports. On a similar topic, Dave Jones [2] is presenting a paper at this conference.

Another problem may be familiar: an overactive `kswapd`. In an old Red Hat Enterprise Linux kernel, it was found that some inner page-scanning loop ran several orders of magnitude more iterations than anticipated, due to some error in queue management code. Does this kind of thing not happen regularly? Systemtap was not available for diagnosing this bug, but it would have been easy to probe loops in the suspect functions, say by source file and line number, to count and graph relative execution counts.

2 Static probing markers

Systemtap recently added support for *static probing markers* or "markers" for short. This is a way of letting developers designate points in their functions as being candidates for systemtap-style probing. The developer inserts a macro call at the points of interest, giving the marker a name and some optional parameters, and grudgingly recompiles the kernel. (The name can be any alphanumeric symbol, and should be reasonably unique across the kernel or module. Parameters may be string or numeric expressions.)

In exchange for this effort, systemtap marker-based probes are faster and more precise than kprobes. The better precision comes from not having to covet the compiler's favours. Such fickle favours include retaining clean boundaries in the instruction stream between interesting statements, and precisely describing positions of variables in the stack frame. Since markers don't rely on debugging information,

neither favour is required, and the compiler can channel its charms into unabated optimization. The speed advantage comes from using direct call instructions rather than `int 3` breakpoints to dispatch to the systemtap handlers. We will see below just how big a difference this makes.

```
STAP_MARK (name);
STAP_MARK_NS (name,num,string);
```

Just putting a marker into the code does nothing except waste a few cycles. A marker can be "activated" by writing a systemtap probe associated with the marker name. All markers with the same name are identified, and are made to call the probe handler routine. Like any other systemtap probe, the handler can trace, collect, filter, and aggregate data before returning.

```
probe kernel.mark("name") { }
probe module("drv").mark("name") { }
```

2.1 Implementation

As hinted above, the probe marker is a macro[2] that consists of a conditional indirect function call. Argument expressions are evaluated in the conditional function call. Similarly to C++, an explicit argument-type signature is appended to the macro and the static variable name.

```
#define STAP_MARK(n) do { \
  static void (*__mark_##n##_)(); \
  if (unlikely (__mark_##n##_)) \
    (void) (__mark_##n##_()); \
} while (0)
```

In x86 assembly language, this translates to a load from a direct address, test, and a conditional branch over a call sequence. The

---

[2]Systemtap includes a header file that defines a scores of type/arity permutations.

load/zero-test is easily optimized by "hoisting" it up (earlier), since it is operating on private data. With GCC's `-freorder-blocks` optimization flag, the instructions for the function call sequence tend to be pushed well away from (beyond) the hot path, and get jumped to using a conditional forward branch. That is ideal from the perspective of hardware static branch prediction.

A new static variable is created for each macro. If the macro is instantiated within an inline function, all inlined instances within a program will share that same variable. Systemtap can search for the variables in the symbol table by matching names against the stylized naming scheme. Further, systemtap deduces argument types from the signature suffix, so it can write a type-safe function to accept the parameters and dispatch to a compiled probe handler.

During probe initialization, the static variable containing the marker's function pointer is simply overwritten to point at the handler, and it is cleared again at shutdown.[3]

This design implies that only a single handler can be associated with any single marker: other systemtap sessions are locked out temporarily. Should this become a problem for particularly popular markers, we can add support for "multi-marker" macros that use some small number of synonymous static variables instead of one. This would trade utility for speed.

2.2   Performance

Several performance metrics are interesting: code bloat, slowdown due to a dormant marker, dispatch cost of an active marker. These quantities may be compared to the classic kprobes

_____
[3]These operations atomically synchronize using `cmpxchg`.

alternative. On all these metrics, markers seem to perform well.

For demonstration purposes, we inserted marker macros in just two spots in a 2.6.16-based kernel: the scheduler context-switch routine, just before `switch_to` (passing the "from" and "to" `task->pid` numbers), and the system call handler `sys_getuid` (passing `current->uid`). All tests were run on a Fedora Core 5 machine with a 3 GHz Pentium 4 HT.

Code bloat is the number of bytes of instruction code needed to support the marker, which impacts the instruction cache. With kprobes, there is no code inserted, so those numbers are zero. We measured it for static markers by disassembling otherwise identical kernel binaries, compiled with and without markers.

function	test	call
getuid	10	19
context_switch	19	34

Slowdown due to a dormant marker is the time penalty for having a potential but unused probe point. This quantity is also zero for kprobes. For our static markers, it is the time taken to test whether the static variable is set, and it being clear, to bypass the probe function call. It may incur a data cache miss (for loading the static variable), but the actual test and properly predicted branch can be nearly "free."

Indeed, a microbenchmark that calls an marker-instrumented `getuid` system call in a tight loop a million times has minimum and average times that match one that calls an uninstrumented system call (`getgid`). A different microbenchmark that runs the same marker macro but in user space, surrounded by `rdtscll` calls, indicates a cost of a handful of cycles each: 4–20.

Since the slowdown due to a dormant marker is so small, we plan to measure a heavily instrumented kernel macroscopically. However, adding markers strategically into the kernel is challenging, if they are to represent plausible extra load.

Finally, let's discuss the dispatch speed of an active marker. This is important because it relates inversely to the maximum number of probes that can trigger per unit time. The overhead for a reasonable frequency of probe hits should not overwhelm the system. For our static markers, the dispatch overhead consists of the indirect function call. On the test platform, this additional cost is just 50–60 cycles.

For kprobes, an active probe includes an elaborate process involving triggering a breakpoint fault (`int 3` on x86), entering the fault handler, identifying which handler belongs to that particular breakpoint address, calling the handler, single-stepping the original instruction under the breakpoint, and probably some other steps we left out.

A realistic systemtap-based microbenchmark measured the time required for one round trip of the same functions used above: the marker-instrumented `sys_getuid` and uninstrumented `sys_getgid`. Each probe handler is identical, and increments a script-level global counter variable for each visit. The following matrix summarizes the typical number of nanoseconds per system call (lower is better) with the listed instrumentation active.

function	marker	kprobe	both	neither
getuid	820	2100	2250	620
getgid		2100		620

Note that the complete marker-based probes run in 200 ns, and kprobes-based probes run in 1480 ns. Some arithmetic lets us work backward, to estimate just the dispatching times and exclude the systemtap probes. The cost of the 50–60 cycles of function call dispatch for the markers (measured earlier) takes about 20 ns on the test host. That implies that the systemtap probe handler took about 180 ns. Since identical probe handlers were run for both kprobes and markers, we can subtract that, leaving 1300 ns as the kprobes dispatch overhead.

While the above analysis only pretends to be quantitative, it gives some evidence that markers have attractive performance: cheap to sit around dormant, and fast when activated.

3 Next steps

3.1 User-space probes

Systemtap still lacks support for probing userspace programs: we can go no higher than the system call interface. A kprobes extension is under development to allow the same sorts of breakpoints to be inserted into shared libraries and executables at runtime that it now manages in the kernel. When this part is finished and accepted, systemtap will exploit it shortly. Probes in user space would use a similar syntax to refer to sources or symbols as already available for kernel probe points.

Probing in user space may seem like a task for a different sort of tool, perhaps a plain debugger like gdb, or a fancier one like frysk[4], or another supervisor process based on `ptrace`. However, we believe that the handler routine of even a user-space probe should run in kernel space, because:

1. The microsecond level speed of a kprobes "round trip" is still an order of magnitude faster than the equivalent process state query / manipulation using the ptrace API.

---

[4]`http://sources.redhat.com/frysk`

2. Some problems require correlation of activities in the kernel with those in user-space. Such correlations are naturally expressed by a single script that shares variables amongst kernel- and user-space probes.

Once we pass that hurdle, joint application of user-space kprobes and static probing markers will make it possible for user-space programs and libraries to contain probing markers too. This would let libraries or programs designate their own salient probe points, while enjoying a low dormant probe cost. Language interpreters like Perl and PHP can insert markers into their evaluation loops to mark events like script function entries/exits and garbage collection. Complex applications can instrument multithreading events like synchronization and lock contention.

3.2   Debugging aid

Systemtap is becoming stable enough that kernel developers should feel comfortable with using it as a first-ditch debugging aid. When you run into a problem where a little bit of tracing, profiling, event counting might help, we are eager to help you write the necessary scripts.

3.3   Sysadmin aid

We would like to develop a suite of systemtap scripts that supplant tools like `netstat`, `vmstat`, `strace`. For inspiration, it may be desirable to port the OpenSolaris DTrace-Toolkit[5], which is a suite of dtrace scripts to provide an overview of the entire system's activity. Systemtap will make it possible to save

and reuse *compiled* scripts, so that deployment and execution of such a suite could be easier and faster.

3.4   Grand unified tracing

There are many linux kernel tracing projects around. Every few months, someone reinvents LTT and auditing. While the author does not understand all the reasons for which these tools tend not to be integrated into the mainstream kernel, perhaps one of them is performance.

To the extent that is true, we propose that these groups consider using a shared pool of static markers as the basic kernel-side instrumentation mechanism. If they prove to have as low dormant cost and as high active performance as initial experience suggests, perhaps this could motivate the various tracing efforts and kernel subsystem developers to finally join forces. Let's designate standard trace/probe points once and for all. Tracing backends can attach to these markers the same way systemtap would. There would be no need for them to maintain kernel patches any more. Let's think about it.

References

[1] Daniel Berranger.
    `http://people.redhat.com/`
    `berrange/systemtap/bootprobe/`,
    January 2006.

[2] Dave Jones. Why Userspace Sucks. In
    *Proceedings of the 2006 Ottawa Linux
    Symposium*, July 2006.

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

---

[5]`http://www.opensolaris.org/os/`
`community/dtrace/dtracetoolkit/`

[4] Vara Prasad et al. Dynamic Instrumentation of Production Systems. In *Proceedings of the 2005 Ottawa Linux Symposium*, volume 2, pages 49–64, July 2005.

```
# cat socket-trace.stp
probe kernel.function("*@net/socket.c") {
  printf ("%s -> %s\n", thread_indent(1), probefunc())
}
probe kernel.function("*@net/socket.c").return {
  printf ("%s <- %s\n", thread_indent(-1), probefunc())
}

# stap socket-trace.stp
     0 hald(2632): -> sock_poll
    28 hald(2632): <- sock_poll
[...]
     0 ftp(7223): -> sys_socketcall
  1159 ftp(7223):  -> sys_socket
  2173 ftp(7223):   -> __sock_create
  2286 ftp(7223):    -> sock_alloc_inode
  2737 ftp(7223):    <- sock_alloc_inode
  3349 ftp(7223):    -> sock_alloc
  3389 ftp(7223):    <- sock_alloc
  3417 ftp(7223):   <- __sock_create
  4117 ftp(7223):   -> sock_create
  4160 ftp(7223):   <- sock_create
  4301 ftp(7223):   -> sock_map_fd
  4644 ftp(7223):    -> sock_map_file
  4699 ftp(7223):    <- sock_map_file
  4715 ftp(7223):   <- sock_map_fd
  4732 ftp(7223):  <- sys_socket
  4775 ftp(7223): <- sys_socketcall
[...]
```

Figure 1: Tracing and timing functions in `net/sockets.c`.

Proceedings of the
Linux Symposium

Volume One

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