Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

LTTng: Tracing across execution layers, from the Hypervisor to user-space

Mathieu Desnoyers
École Polytechnique de Montréal
mathieu.desnoyers@polymtl.ca

Michel Dagenais
École Polytechnique de Montréal
michel.dagenais@polymtl.ca

Abstract

This presentation discusses the upcoming changes to be proposed to the kernel tracing field by the LTTng community. It will start by explaining what has been mainlined (per-cpu atomic operations, Linux Kernel Markers.) Then, the focus will turn to the patch set currently developed and for which the mainlining process is in progress. An important part of this presentation will talk about the efficient system-wide user-space tracing infrastructure being designed. Work done for tracing across execution layers, including the Hypervisors, will also be shown.

The mainlining status of kernel tracing will be a key element of this talk. Considering the increasing amount of news articles written on this subject, many attendees, from the kernel hacker to the system administrator, should find interest in this presentation.

1 Introduction

Since last year's symposium, where the need the industry has for a tracer in the Linux kernel has been demonstrated [1], the expectations from the Linux community for tracing tools matching DTrace [2] seem to have grown [4]. A lot has happened since then in the various tracing projects, with results still waiting to find their way into the kernel mainline.

This paper presents the current state of the work performed in the LTTng project which have been integrated or is planned to be integrated in the Linux kernel. It details the "Immediate Values," improvements for the "Linux Kernel Markers" and discusses the kernel instrumentation patch set, based on the markers, submitted to the Linux community.

2 Related Work

Other projects with similar goals have already tackled areas of the tracing problem. Credit must be given to the K42 team [9] at IBM Research for developing a highly scalable operating system implements a lock-free, mostly atomic trace buffering mechanism (except for subbuffer switch.) The Kprobes developers at IBM, Intel, and Hitachi and the Djprobes [7] team at Hitachi have also pioneered the area of dynamic kernel code modification on the x86 architecture, providing the ability to insert custom instrumentation based on breakpoints or jumps based on dynamic code modification. The shortcoming of these two methods seems to be the performance impact of the breakpoint and the fact that none of these can guarantee access to the local variables in the middle of a function, since they can be optimized away by the compiler.

The SystemTAP [6] project is built on top of Kprobes and the Linux Kernel Markers to provide a scriptable language to create probes, which can be connected on any of those two information sources to extract information from the running kernel. LTTng learned from the lessons brought by the first generation of tracer, LTT [10]. It also reused the instrumentation found in LTT.

More recent work includes the "Driver Tracing Infrastructure" (DTI) [8] and the "Generic Trace Setup and Control" (GTSC), which aim at providing a standard driver tracing infrastructure for drivers.

3 Mainlining Status

In the past years, the LTTng project has proved its usefulness and yet, the ground work required in the Linux kernel before a kernel tracer can really become usable is not over. The next section will present the pieces of infrastructure required by LTTng which have been integrated in the mainline kernel.

3.1 Linux Kernel Markers

LTTng depends on the Linux Kernel Markers [3] to provide the instrumentation of the core kernel. It uses the Linux Kernel Markers as primary information source, but could connect to other sources of information if needed. The markers provide an interface to source code instrumentation that simplifies adapting to code source changes, separating the concept of "high level trace event" from the actual code source. The markers can be dynamically activated, and can provide information to probes registering on specific markers from either the code kernel or GPL modules. Other projects, such as SystemTAP [6], also support hooking on markers.

3.2 Per-CPU Atomic Operations

The LTTng kernel tracer does not only need to be fast, but it also needs to be reentrant with respect to other execution contexts, when it reserves space in its memory buffer. Using per-CPU data structures and buffers helps eliminating false sharing and eliminates concurrency coming from other processors. However, local interrupts, both maskable and non-maskable (NMI), will try to write events to the same trace buffers concurrently.

The algorithm for lock-less NMI-safe buffer management [5] is based on extensive use of the compare-and-swap atomic operation. It is known, however, to be slower than interrupt disable on SMP systems. The Per-CPU atomic operations, also known as "local ops" do best of both: they offer reentrancy with respect to NMI contexts and are faster than interrupt disable on many architectures. The reason for such performance is that these operations don't need neither LOCK prefix nor memory barriers, since they update memory local to a given CPU.

In addition to the LTTng tracer, the Per-CPU Atomic Operations are currently being used in an experimental patch for the SLUB allocator, which uses the local compare-and-swap primitive in the allocate and free fast paths. Initial performance improvements range from two to threefold compared to the version using interrupt disable.

4 Forthcoming Kernel Changes

4.1 Immediate Values Optimization

The immediate values are a derivative work of the Linux Kernel Markers. They provide an infrastructure to encode, in the instruction stream of the Linux kernel, static and global variables which are read-often, but updated rarely. The read-side does not have to read any information from data cache, since it's already encoded in the form of an immediate value at each variable reference site; therefore, all the information needed is present within the instruction stream.

Updates are done dynamically by updating the immediate values in the load immediate instructions on a live running kernel, upon each variable modification.

The original goal of immediate values was to provide a very efficient activation mechanism for the Linux Kernel Markers. With their current version, in kernel 2.6.25, each encountered marker adds a memory read to check if the marker is enabled. The impact on data cache therefore grows as more markers are added to kernel cache-hot instruction paths.

Using the immediate values, to encode the branch condition in the instruction stream, helps solving this problem. Instead of polluting the data cache, markers based on immediate values encode the branch condition directly in the instruction stream. Assembly example of immediate values use by markers on x86_32 and x86_64 goes as follow. The first example focuses on the added code to `schedule()` cache-hot code. It adds 2 bytes for the immediate value load, a 2-byte test and a 6-byte conditional near jump, for a total of 10 bytes.

```
356:  b0 00              mov   $0x0,%al
358:  84 c0              test  %al,%al
35a:  0f 85 1e 03 00 00  jne   67e <schedule+0x449>
```

For smaller functions such as `wake_up_new_task()`, the conditional jump only takes 2 bytes since the offset can be expressed as a short jump, for a total of 6 bytes.

```
848e: b0 00     mov   $0x0,%al
8490: 84 c0     test  %al,%al
8492: 75 7f     jne   8513 <wake_up_new_task+0x97>
```

This infrastructure can be used simply by replacing every reference to a static or global variable "var" by a `imv_read(var)` and by changing each update to the variable by an `imv_set(var)`, the latter being a preemptable function. Variables with size of 1, 2, 4, or 8 bytes can be referred to. If the architecture does not support updating one of these type size on a live system, a normal variable read is used. This is the case for a 8 bytes variable on a 32 bits x86, which cannot be encoded as an immediate value of a single instruction, and for variables larger than 2 bytes on PowerPC, because instructions are limited to 4 bytes in size and take only 2 bytes operands. If no immediate value optimization is implemented for a given architecture, the generic fallback is used: a standard memory read.

Some work is currently being done to improve even further immediate values used as boolean condition for a branch. The goal is to minimize the impact of disabled markers on a running system, replacing the mov, test and branch instructions by a sequence of 2-byte nops and either a 2-byte short jump or a 1-byte nop and 5-byte jump. Since the compiler might reorder instructions between the mov, test, and jne/je instructions, this optimization is only done when the pattern is detected as unmodified by the compiler. Initial results show that the 97% of the 120 trace points added to the Linux kernel in the LTTng instrumentation do not suffer from such compiler modifications on x86_32 and that the success rate stays at 90% on x86_64. Knowing for sure where the test and branch instructions are would require some work on the compiler.

The performance impact of a loop instrumented with different techniques is shown in tables 1 and 2. This loop executes some ALU work in the baseline. It is then compared with the performance impact of the same loop with an added inactive marker using a sequence of mov, test, and branch instructions, and with a normal marker reading a memory variable.

It is then compared with the "ftrace" approach, using a function call replaced by nops. The latter method is also used in DTrace. The second column shows the same results with a baseline which flushes the data cache containing the information accessed to show how each method behaves when the data is cache cold. We can see that the cache cold impact is much higher for the disabled function call when it references information not present in the cache or in the registers. This is required to perform the stack setup, even if the function call is disabled with nops. The non-optimized markers have a similar data cache impact.

The difference of impact between the cached runs could be considered as non-significant and amortized by the pipeline, but the real difference comes from the uncached memory accesses, where the runtime cost ranges from 41.8 to 154.7 cycles.

It must be noted that, on the code size aspect, the markers also add about 50 bytes in an unlikely branch. With `gcc -O2` or `-freorder-blocks`, this branch is placed away from cache-hot instructions and therefore does not stress the instruction cache. The data added by each marker is placed in a special section, only needed when the markers are activated.

4.2 Instrumentation

Once the marker infrastructure is in place to support instrumentation, the following step to have a useful tracer is to start integrating a core instrumentation set in the kernel. The instrumentation proposed in the LTTng project is divided into architecture independent and dependant patch sets.

Architecture independent instrumentation is by far the largest, yet the simplest, instrumentation with 86 markers inserted in the filesystem, inter-process communication, kernel, memory management, networking, and library code. Its simplicity comes from the fact that it only instruments C code in a straightforward way. It's therefore easy to benefit from the small performance overhead of the markers.

The architecture dependant instrumentation currently supports the following architectures, from the most complete to the less: x86_32, x86_64, PowerPC, ARM, MIPS, SuperH, Sparc, and S/390. Instrumentation at the assembly level requires some extra mechanisms to efficiently extract information from system calls. Those are implemented in the form of a new `TIF_KERNEL_TRACE` thread flag added to every architecture. It can be enabled or disabled at runtime to control system call tracing activation for all the system threads. This new thread flag is tested in assembly to check if the `do_syscall_trace()` functions, which contains the system call entry and exit markers, must be called.

In addition to the instrumentation of the kernel code, dumping the kernel structures requires the addition of

x86 Pentium 4, 3.0GHz, Linux 2.6.25-rc7	Added cycles (cached)	Added cycles (uncached)
Optimized marker	0.002	0.07
Normal marker	0.004	154.7
Stack setup + (1+4 bytes) NOPs (6 local var.)	0.04	0.6
Stack setup + (1+4 bytes) NOPs (1 pointer read, 5 local var.)	0.03	222.8

Table 1: Comparison of markers and disabled function impact on x86_32

AMD64, 2.0GHz, Linux 2.6.25-rc7	Added cycles (cached)	Added cycles (uncached)
Optimized marker	-1.2	0.2
Normal marker	-0.3	41.8
Stack setup + (1+4 bytes) NOPs (6 local var.)	-0.5	0.01
Stack setup + (1+4 bytes) NOPs (1 pointer read, 5 local var.)	2.7	51.8

Table 2: Comparison of markers and disabled function impact on x86_64

new in-kernel accessors. This information extraction is typically done at trace start to have a complete picture of the operating system state. When a trace is examined in a viewer, this recorded initial state can be updated using the information in the trace, and the system state is thus available for viewing and analysis purposes for the whole trace duration. Functionality must be added to dump the important kernel structures in the trace buffers, in a way that permits to identify when the data structures are changing concurrently. Typically, the /proc file system expects the kernel structures to stay unchanged between two consecutive reads. If they change, it will result in the loss of information that can't be linked with the element being added or removed from the structures. The output text will be truncated at the offset of the currently requested read operation.

The LTTng state dump module dumps the kernel structures to the trace in multiple iterations, releasing the locks after a fixed number of elements, to make sure operations such as dumping all the memory maps of all the processes in the system won't generate high latencies. Detection of concurrent data structure modification is done by the rest of the kernel instrumentation; since every manipulation to these data structures is traced, the trace analyzer can re-create the data structure at any given point of the trace after the end of state dump.

5 User Space Tracing

Work performed in the user-space tracing area involved porting the Linux Kernel Markers to user-space so that they can be used in libraries. The linker scripts are modified to add a new section which contains the markers placed in each object. A library init function is linked with each object to allow registration of the markers to the kernel through an additionnal system call.

Activation of markers can then be done system-wide. It would allow to easily turn on instrumentation of the NPTL pthread mutexes at the user-space level, or to instrument glibc memory allocation primitives and link this information with the kernel memory requests.

As a first step, the extraction of information could be done through a string, passed as an argument to a trace system call. The reason for using system calls rather than other mechanisms is that this technique does not depend on other libraries to open files and help instrumenting user-space programs executed at boot time.

Eventually, extracting the information without going through a system call would help to minimize tracing performance impact. It would, however, imply that shared buffers should be made accessible for writing to each traced process. Because of security concerns, these buffers cannot be shared between the various processes, as done in the K42 research operating system. There is therefore still work to do in this area.

6 Hypervisor Tracing

The Xen hypervisor has already its own tracer, xentrace. It exports fixed-size data to userspace through a buffer shared with a process running on domain0. The process communicates with the hypervisor to activate tracing through hypercalls.

An experimental port of LTTng to the Xen hypervisor has been realized. The lttd-xen daemon has been created by modifying the lttd daemon to use new hypercalls rather than debugfs. The same has been done to lttctl and liblttctl: they have been ported to use hypercalls rather than a netlink socket. Because we use variable-sized events, which represent the data in its most compact form, we were able to generate traces twice as small as xentrace.

The main interest in having a tracer extracting information in the same format as the operating system and userland is to help analyze concurrency, race and other timing problems across execution domains.

7 Conclusion

With Kprobes and Linux Kernel Markers already in the mainline kernel, the road seems to be opening for integration of more parts required to have a solid tracing infrastructure in the kernel, namely the immediate values, a kernel instrumentation, and eventually, support for userspace tracing.

Once the kernel goals are reached, the focus will be easier to turn on the other aspects of tracing, which includes the choice of userland markers and standardization of hypervisor tracing.

References

[1] Martin Bligh, Rebecca Schultz, and Mathieu Desnoyers. Linux kernel debugging on google-sized clusters. In *Proceedings of the Ottawa Linux Symposium 2007*, 2007.

[2] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *USENIX '04*, 2004.

[3] Jonathan Corbet. Kernel markers. August 2007.

[4] Jonathan Corbet. On dtrace envy. August 2007.

[5] Mathieu Desnoyers and Michel Dagenais. The lttng tracer : A low impact performance and behavior monitor for gnu/linux. In *Proceedings of the Ottawa Linux Symposium 2006*, 2006.

[6] Frank Ch. Eigler. Problem solving with systemtap. In *Proceedings of the Ottawa Linux Symposium 2006*, 2006.

[7] Masami Hiramatsu and Satoshi Oshima. Djprobes - kernel probing with the smallest overhead. In *Proceedings of the Ottawa Linux Symposium 2006*, 2006.

[8] David Wilder. Unified driver tracing infrastructure. In *Proceedings of the Ottawa Linux Symposium 2007*, 2007.

[9] Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Supercomputing, 2003 ACM/IEEE Conference*, 2003.

[10] Karim Yaghmour and Michel R. Dagenais. The linux trace toolkit. *Linux Journal*, May 2000.