LTTng, Filling the Gap Between Kernel Instrumentation and a Widely Usable Kernel Tracer

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

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

Abstract

This paper presents an overview of tracing requirements stated by the LTTng user-base. It presents LTTng as a tracer having a wide user-base, with needs different from kernel developers. It presents tracing infrastructure as being made of distinct parts which can be categorized as either common core-kernel infrastructure (instrumentation, tracing time-source) or tracer-specific, driver-like code (trace management, buffering mechanism). This paper builds the case for LTTng mainlining into the Linux kernel by explaining the specific user requirements LTTng fulfills, its degree of maturity and the number of users it has. This case is then supported by showing that most of its code-base does not affect the kernel core.

1 Introduction

With current systems becoming increasingly multi-core and complex, the need for tools to help understanding performance and latency problems is clear[2]. However, a kernel tracing solution usable by the large community of Linux users has not made its way into the mainline Linux kernel yet.

This paper will detail the various tracing solutions currently available in the Linux kernel and explain the distinction between core kernel tracing infrastructure (which must be shared, common and has the largest impact on the kernel code-base) and trace data transport and trace management infrastructures, which can be categorized as driver code.

It will then explain how the LTTng user-base differs from the target user-base of most of the non-core tracing facility currently present in the mainline kernel, therefore building a base for mainline LTTng "driver" code.

2 Tracing Infrastructure in Mainline Kernel

This section will detail the tracing infrastructure integrated in the Linux kernel 2.6.30-rc3.

The first gategory, instrumentation mechanisms, includes Kprobes, Kernel Markers and Tracepoints. Kprobes[9], allow dynamically inserting breakpoints into the Linux kernel on which handlers can be connected. The Linux Kernel Markers[3] allow adding ad-hoc instrumentation along with a format string and a variable argument list. This allows easy addition of instrumentation at the source code level. The Tracepoints[4] are a variant of the Linux Kernel Markers, which provide better manageability of

the instrumentation by requiring an instrumentation declaration to be added into a system-wide header. Tracepoints are meant to allow the kernel instrumentation process to be managed by the subsystem maintainers, along with the overview of the overall community. A third static instrumentation mechanism present in the kernel is the function tracer, which allows instrumentation of function entry and exit by the compiler with an almost non-existing overhead when dynamically disabled.

The second category, kernel tracers, is currently being integrated under the Ftrace[8] umbrella. It includes principally the block I/O tracer[1], the memory I/O tracer, kmemtrace, KVMtrace (tracing Linux KVM), the wakeup tracer and the event tracer. The number of such tracers is increasing from one kernel version to another. The approach taken here is to let tracers attach to Ftrace to provide a data output. One tracer can be selected as the "current" tracer at any given time. Their primary user-base is meant to be kernel developers. The motto Ftrace follows is to include everything needed to use the tracer within the kernel, to make sure kernel developers do not run into userspace package dependency problems.

3 Core Kernel vs Driver Code

As a general guideline coming from the Linux kernel maintainers[1], the kernel community is actively trying to make it easier for contributors to have their kernel drivers merged into the mainline Linux kernel. The *linux-next* tree includes a *staging drivers* section with this precise goal : to integrate new drivers into the kernel tree early in their development.

[1]Referring to Andrew Morton and Greg Kroah-Hartman at LFCS2009[7]

However, this does not apply to core kernel code with very good reasons : modifications to the core kernel code can have broad impacts on the kernel and on many kernel maintainers. Furthermore, they are, by nature, hard to isolate in a specific module.

Therefore, the core kernel code is "jealously kept" from external contributions, and those typically have to go through a very thorough round of review before being accepted.

Looking at the Linux Trace Toolkit Next Generation patchset[2], one might wonder which part of it could be considered as core kernel code and which parts are self-contained drivers.

The core kernel modifications present in the LTTng patchset are limited to the kernel instrumentation infrastructure, the instrumentation per se and the trace clock.

Most of the instrumentation infrastructure : kernel markers and tracepoints, have been merged into the mainline kernel already. The "Immediate Values" patches aim at diminishing the performance impact of dormant instrumentation, which will become increasingly useful as more tracepoints are added into the kernel. The LTTng instrumentation touches various kernel subsystems and is being submitted for integration into the mainline kernel. The time-base LTTng uses (trace clock) aims at providing a reliable and fast time-base suitable for the needs of tracing. Ingo Molnar proposed a trace clock implementation which is currently in mainline but does not have the reliability and performance characteristics provided by the LTTng implementation. Those pieces of infrastructure will therefore have to be submitted for mainlining as "core kernel" modifications.

This is however where stops the LTTng core kernel intrusiveness. LTTng code-base is

[2]`http://www.lttng.org`

made primarily of self-contained kernel modules which aim at using as few pieces of kernel infrastructure as possible, so the instrumentation coverage can be maximized. Therefore, the trace sessions management code, the ring-buffer data extraction mechanism and the buffer layout are all self-contained in kernel modules that do not affect the rest of the Linux kernel code-base.

We can therefore claim that the LTTng trace management mechanism should be considered for mainlining under the same criterions that apply to drivers.

Given that Ftrace provides parts of the features provided by LTTng, one might wonder if LTTng mainlining would in fact duplicate features already provided by Ftrace. The following section will shown how LTTng and Ftrace user-bases and requirements differ.

4 Case for LTTng Mainlining

This section will show how LTTng fits with respect to the various requirements for kernel code to be considered for mainlining, namely : to fulfill specific user requirements, having a large user-base and being actively developed by a group of contributors.

4.1 LTTng Specific User Requirements

LTTng fulfills tracing requirements from developers, system administrators, technical support and users running Linux as their operating system. Those requirements include having the ability to analyze and debug application, library and kernel system-wide performance. Within these users, some of the most demanding need to trace high-performance computing multi-core application workloads (Google servers). At the other end of the spectrum, embedded system developers need to fit within very limited memory and bandwidth resources (Nokia embedded products). It is used in the field on Siemens production systems to gather a continuous flight recorder trace of the system's behavior to circular buffers, providing meaningful bug reports from the end-user site to Siemens technical support teams.

The currently existing tracing solution in the Linux kernel, Ftrace, targets mainly kernel developers. It focuses primarily on providing specialized tracers for kernel behavior to debug kernel-level problems occuring, for example, at the scheduler, driver, memory management, block layer levels. We will see below that this difference of target users makes more difficult the sharing of some common pieces of low-level infrastructure.

Ftrace currently relies on some assumptions about tracer usage specfic to kernel users. Primarily, data is written into physical pages, which limits the maximum event size to a page and requires padding to be added whenever an event would cross a page boundary. The buffer locking structure is specialized for kernel tracing, which makes it unlikely to be reusable for user-space tracing. For instance, assumptions about preemption being disabled at the tracing site are made, which does not hold in user-space. Ftrace uses many function calls which are costly performance-wise on many architectures. For example, on a 64-bits Intel Xeon, adding a function call to the LTTng tracer fast path slows down tracer execution by 20%. In terms of buffer space usage, the *ring_buffer* infrastructure reserves precious event header bytes to encode the type and size of events (2 bits for event type, 3 bits for event length and 27 bits for time delta). The event payload itself is a multiple of 32 bits. The locking mechanism currently used by Ftrace is a per-cpu spinlock with interrupts disabled; this en-

sures that the fast-path will not produce cache-line bouncing between CPUs, but needs to use synchronized atomic operations and interrupt disabling, adding a non-negligible performance impact[5].

Work to provide a lockless buffering mechanism is underway, but it has not yet been published by the author nor reviewed due to patent application delays. Ftrace is also specialized for a single-user use on a development machine. Only one tracer can be activated at a time, which makes it hard for machines with multiple users to use different tracers.

LTTng presents the buffer layout as a contiguously addressable circular buffer, which supports writing event of variable size, with payload up to the size of sub-buffers. It is suitable for a wider variety of uses than Ftrace, including scenarios requiring larger events like network packet monitoring. The buffering mechanism is layered in such a way that it permits compiling-in various memory backends to hold the circular buffers. It uses, by default, pages from the page allocator, but can be trivially adapted to use video memory (which survives hot reboot, for kernel crash trace extraction) or statically allocated contiguous memory. LTTng offers this level of flexibility without sacrifying performance by using the minimum number of function calls in the fast-path. This is possible by building different objects including the same headers to build the various pieces with different options. Switching from one back-end to another could then be done by loading a different module.

LTTng also aims at providing a large instrumentation coverage. This relates to the amount of kernel code that could not be instrumented using the kernel tracer due to re-entrancy issues. LTTng uses a lockless, formally verified, buffer concurency management algorithm to support instrumentation of code executed from NMI context. The locklessness of the buffer algorithm also improves performances significantly[5].

LTTng design makes it very kernel-independent, which ensures that the tracer is easy to port to different contexts. For instance, a working user-space tracing port of LTTng is currently undergoing final review and awaiting LGPL licensing agreements from other concerned parties. A port of LTTng to the Xen hypervisor has also been done previously without requiring much effort. This design guide-line is not shared with the Ftrace project, which is very Linux-kernel specific. Low dependency on kernel behavior assumptions makes LTTng more solid when the kernel behaves incorrectly and more suitable for system-wide tracing, which includes hypervisors, kernel and user-space.

We can therefore see that the main difference between Ftrace and LTTng comes from different target use-cases and user requirements. Therefore, the question that prevails is whether is makes sense for those tracers to share low-level transport infrastructure. This happens to be very hard to do if one party does not consider the other's user requirements. A second worthwhile question is if LTTng could gain from moving to a different tracing infrastructure such as Ftrace *ring_buffer*. Given the community review, testing on various platforms, usage in the field and formal verification of the lockless algorithms performed in its four years of development, LTTng would actually be regressing in terms of maturity and testing without any added value. This is without even considering the work involved in doing the adaptation, that would postpone availability of the tracer.

It is important to specify that the respective Ftrace and LTTng team members are working in collaboration to share the maximum amount of knowledge and infrastructure between the projects and may eventually converge on pieces of tracing infrastructure as development moves

forward without important visible impact for users.

4.2 LTTng Features

We can now focus on the major features LTTng has that makes it interesting to its user-base.

LTTng focuses on system-wide tracing of the overall system behavior to give Linux end-users the ability to identify the root cause of performance degradation or high latency. It provides very good re-entrancy using lockless concurrency management algorithms. It supports kernel-wide instrumentation by being as isolated as possible from the rest of the kernel and by supporting re-entrancy from all kernel execution contexts. It contains a solid monotonic time-base which ensures sane upper bounds on the time-stamp precision error and especially on the trace event partial order, which lets trace analysis tools and users be confident that the tracer provides correct information.

LTTng is very low-overhead[6], has an architecture-agnostic core, supports the kernel Tracepoints, Linux Kernel Markers and Kprobes instrumentation infrastructures to provide an easily extensible instrumentation.

For multi-user systems, where various teams may have to share and monitor different aspects of common hardware ressources, LTTng supports multiple tracing sessions.

4.3 LTTng Users and Contributors

LTTng currently counts many users and contributors which either use Linux as their operating system or actively contribute to Linux.

Google[2] and IBM[10] have contributed and used LTTng in their systems to pinpoint the root cause of performance degradations. Autodesk used LTTng to identify high latency incurred by the Linux kernel in the development phase of their products[2]. Ericsson is contributing to LTTng development and actively involved with the Eclipse community to design an IDE able to handle traces generated by LTTng. Fujitsu is actively contributing to the LTTng project, both in term of code addition and support for the LTTng tracer on the mailing lists. Siemens is using LTTng in their production systems to provide "flight recorder" traces to diagnose problems. Nokia is working on and funding the LTTng ARM OMAP3 port to have precise tracing on their embedded systems. Sony, Samsung and Boeing are LTTng users as well.

In terms of distributions, Novell SuSe Linux Enterprise Server (Real-Time)[3] includes the full LTTng tracer, while the standard SLES 11 includes only core parts of the LTTng instrumentation. WindRiver has been distributing LTTng in their Linux distribution for a few years, supporting it with their WindRiver Workbench 2.6. Montavista is also shipping LTTng in their Carrier Grade Linux 5.0.

Therefore, we can see that both large Linux users and distributions show a clear need for the features LTTng provides.

5 Conclusion

The intent of the Linux community is to make it easy for non-core (driver) code to be pushed into the mainline Linux kernel, especially when it does not impact other subsystems. However, there still seems to be some disagreement about duplication of driver-like parts of tracing functionality in the kernel.

[3] SLES 11 and SLES 11 real-time

This paper presented how kernel tracing differs when targeting either kernel developers or Linux developers, system administrators, technical support staff and end users. It has then shown how it impacts tracer design decisions from the high-level (importance of reliability, low performance impact, multi-sessions support) down to the low-level implementation choices that follow those high-level design choices.

Some of the various LTTng users, Google, IBM, Autodesk, Ericsson, Fujitsu, Siemens, Nokia, Sony, Samsung, and Boeing as well as some distributions (SLES 11, WindRiver Linux and Montavista Carrier Grade Linux 5.0) express the strong need for such user-oriented tracer. Their main complaint in the past years about the LTTng tracer has been that it is not present in the mainline kernel. Perhaps it is time for the Linux community to address it ?

References

[1] Jens Axboe. Linux block io present and future. In *OLS (Ottawa Linux Symposium 2004)*, 2004.

[2] Martin Bligh, Rebecca Schultz, and Mathieu Desnoyers. Linux kernel debugging on google-sized clusters. In *OLS (Ottawa Linux Symposium) 2007*, 2007.

[3] Jonathan Corbet. Kernel markers. Linux Weekly News, `http://lwn.net/Articles/245671/`, August 2007.

[4] Jonathan Corbet. Tracing: no shortage of options. Linux Weekly News, `http://lwn.net/Articles/291091/`, July 2008.

[5] Mathieu Desnoyers and Michel Dagenais. Synchronization for fast and reentrant operating system kernel tracing. (To appear).

[6] Mathieu Desnoyers and Michel Dagenais. The lttng tracer : A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium) 2006*, 2006.

[7] Jake Edge. Elc/lfcs2009: A tale of two panels. Linux Weekly News, `http://lwn.net/Articles/327938/`, April 2009.

[8] Jake Edge. A look at ftrace. Linux Weekly News, `http://lwn.net/Articles/322666/`, March 2009.

[9] Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston, Anil Keshavamurthy, and Masami Hiramatsu. Probing the guts of kprobes. In *Proceedings of the Ottawa Linux Symposium 2006*, 2006.

[10] Robert W. Wisniewski, Reza Azimi, Mathieu Desnoyers, Maged M. Michael, Jose Moreira, Doron Shiloach, and Livio Soares. Experiences understanding performance in a commercial scale-out environment. In *Europar 2007*, 2007.