Combined Tracing of the Kernel and Applications with LTTng

Pierre-Marc Fournier
École Polytechnique de Montréal
pierre-marc.fournier@polymtl.ca

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

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

Abstract

Increasingly complex systems are being developed and put in production. Developers therefore face increasingly complex bugs. Kernel tracing provides an effective way of understanding system behavior and debugging many types of problems in the kernel and in userspace applications. In some cases, tracing events that occur in application code can further help by providing access to application activity unknown to the kernel.

LTTng now provides a way of tracing simultaneously the kernel as well as the applications of a system. The kernel instrumentation and event collection facilities were ported to userspace. This paper describes the architecture of the new LTTng Userspace Tracer and how it can be used in combination with the kernel tracer. Results of some early performance tests are also presented.

1 Introduction

Technologies such as multi-core, clusters and embedded systems are used to build increasingly complex systems, which results in developers facing increasingly complex bugs. These bugs may for example occur only in production, disappear when probed, occur rarely or have for only symptom a slowdown of the system. These characteristics make traditional debugging tools ineffective against them. New debugging tools are therefore required.

The impact of these tools on system performance must be as small as possible, so they can run on systems in production, whose hardware is chosen to match the production load (and not debugging tools), or on which adding debugging tools may render certain bugs unreproducible.

Kernel tracing is one of these tools. It may be used to understand a great variety of bugs. Quite often, the kernel is aware of all the important activities of an application, because they involve system calls or traps. In certain cases however, kernel tracing is not sufficient. For example, the execution of applications that process a large number of requests or that have a large number of threads may be more difficult to follow from a kernel perspective. For this reason, applications need to be traceable too. It is moreover highly desirable that userspace trace events be correlatable with kernel events during the analysis phase.

LTTng[3], while providing a highly efficient kernel tracer, lacks a userspace tracer of equal performance. In this paper, the LTTng Userspace Tracer, a work in progress to fill this gap, is described. In the next sections, its architecture is presented. Afterward, performance results are shown, followed by proposals for future work.

2 Related Work

The classic `strace` tool provides a primitive form of userspace tracing. It reports system calls and signals in a process. Unfortunately, its usage induces a significant performance penalty. It is moreover limited to tracing system calls and signals, both of which are nowadays obtainable at a much lower cost through kernel tracing.

DTrace has statically defined tracing (SDT) that can be used inside userspace applications[6]. This implementation uses special support inside the runtime linker. Upon activation of an instrumentation point, NOP instructions

placed by the linker are replaced by an instruction that provokes a trap. Probes are limited to 4 arguments.

SystemTap has an implementation of SDT[2] that seems to be very similar to that of DTrace.

LTTng has offered several different userspace tracing technologies over the years. The first is called "system call assisted" userspace tracing. It declares two new system calls. The first is used to register an event type; it returns an event ID. The second is used to record an event; it requires an event ID and a payload to be passed as arguments.

The second, called "companion process" userspace tracing, requires no kernel support. Processes write their events in buffers in their own adress space. Each thread had a "companion" process, created by the tracing library, that shares the buffers (through a shared memory map). The companion consumes the buffers using a lockless algorithm.

After some refactoring of the LTTng core, these two approaches were dropped. Eventually, a quick replacement was devised, which consists in a simple system call taking a string as argument. Calling it produces an event whose argument is the string. The event always has the same name; an indication of the application generating the event needs to be prepended to the string.

Eventually, the feature was moved from a system call to a file in DebugFS (`/debug/ltt/write_event`). Writing a string to this file generates an event called `userspace_event` whose argument is the string.

Ftrace[7], another kernel tracer, has a similar feature using a file called `trace_marker`.

3 Architecture

The LTTng Userspace Tracer (UST) is a port of the LTTng static kernel tracer to userspace. This section describes the architecture of the UST, insisting on the particularities of the userspace implementation. Figure 1 shows an overview of the UST architecture.

Here are some of the important design goals of the UST, that influenced its architecture.

- It should be completely independant from the kernel tracer. Kernel and userspace traces should be correlated at analysis time.
- It should be completely reentrant, supporting multi-threaded applications and tracing of events in signal handlers.
- There should be no system call in the fast path.
- The trace data should never be copied.
- It should be possible to trace code in shared libraries as well as the executable.
- The instrumentation points should support an unlimited number of arguments[1].
- No special support from the linker or compiler should be required.
- The trace format should be compact.

3.1 Tracing Library

Programs that must be traced are linked with the tracing library (`libust`). They must also be linked with the Userspace Read-Copy-Update library (`liburcu`)[4], which is used for lockless manipulation of data structures. They must finally be linked with `libkcompat`[5], a library that provides a userspace version of some APIs available in the Linux kernel (atomic operations, linked list manipulation, kref-style mechanism, and others).

3.2 Time

There are no dependencies between the kernel and userspace tracers of LTTng. However, in order to do a combined analysis of a kernel trace and of userspace traces, timestamps of all traces must be coherent (e.g. they must come from the same time source).

An appropriate tracing time source must have a high resolution in addition to being coherent across cores. The cost of accessing this time source must be low in kernel space, but also in userspace (making a system call is too costly). Work is needed in the Linux kernel to make such a time source with all these characteristics available in all architectures.

The UST code currently works only on x86 (32 and 64 bits). Until a suitable time source is provided by the kernel, the TSC is read directly with the `rdtsc` instruction. This is the same time source used by the kernel tracer. It is quick to read and synchronized across cores in most variants of the architecture.

[1] The only constraint is that an event must fit in a sub-buffer.

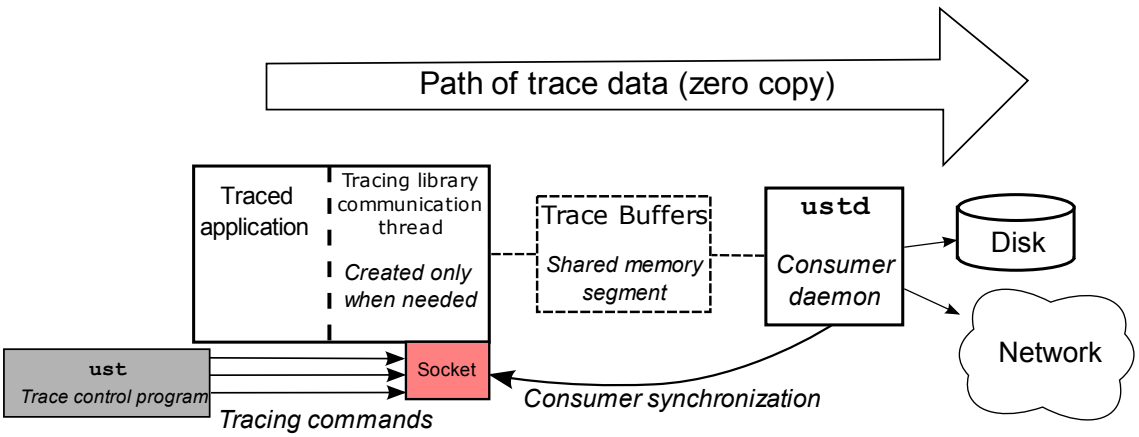

Figure 1: Overview of the LTTng Userspace Tracer architecture.

3.3 Instrumentation Points

Instrumentation points consist in ports of the two kernel instrumentation technologies LTTng uses: markers and tracepoints. Their usage is the same as in the kernel.

Inserting a marker is as simple as adding a single line of code at the point where the event must be recorded. Figure 2 shows an example of a marker. Markers include a format string that resembles `printf` format strings; they include the name of each argument, the format of the event in the trace and the type of the variable passed to `trace_mark`.

Tracepoints are designed to be more elegant and provide type checking. An example is shown in Figure 3. They do not include a format string in the call, but necessitate some declarations, typically in seperate C and header files. This makes them more suitable for permanent instrumentation. Markers are best suited for quickly adding instrumentation while debugging.

Markers and tracepoints list information about themselves in a special section of the executable or dynamic object. Each library and each executable that contains instrumentation must therefore register its markers/tracepoints section via a call to the tracing library. This is done by invoking a macro that adds a constructor that automatically does this, in one of the source files of every executable and every dynamic object.

Each of the instrumentation points may be enabled or disabled by the user, even while the trace is active. Each time the control flow passes on an instrumentation point, a global variable is tested to verify whether it is enabled.

3.4 Buffering Mechanism

The buffering mechanism is a port of the lockless LTTng algorithm. Its design reuses many ideas from the K42 operating system and the Linux kernel Relay[8] system.

Events are written in a circular, per-process buffer, which is divided in sub-buffers. By default, when a sub-buffer is full, it is consumed by a consumer daemon. In another operating mode called *flight recorder*, the circular buffers are constantly overwritten, until the buffers are flushed, either by the user or by a program. This is useful to wait until an infrequent bug occurs in the application.

Each event is associated with a channel. Each process has a distinct buffer for each channel. Having several channels allows to choose the size of sub-buffers per channel. It also allows to keep some events for a longer period of time by putting them in a low-rate buffer when operating in flight recorder mode.

The buffers are allocated inside System V shared memory segments so the consumer daemon can map them in its address space.

Writes to the buffers are done using a lockless algorithm whose correctness was formally verified. It is therefore reentrant and thread- and signal-safe.

3.5 Trace Control

There needs to be a way to command an application to start or stop tracing, to enable, disable or list instrumentation points and to control trace parameters such as sub-buffer size and count.

```
trace_mark(main, myevent, "firstarg %d secondarg %s", v, st);
```

Figure 2: Example of a marker. The first argument is the channel and the second is the event name. The third is the format string of the event arguments while the last are the event arguments. In the format string, each argument name is followed by its format.

```
trace_main_myevent(v, st);
```

Figure 3: Example of a tracepoint.

A helper application called `ust` is used for this purpose. It communicates with the traceable application through a Unix socket. Communications are handled by a special thread in the traced process.

In order for the UST to be as minimally invasive as possible, this thread is not launched automatically when the application starts. Instead, the tracing library constructor registers a signal handler for a particular signal. When that signal is received, a listener thread is started. This thread creates a named socket in a predefined directory. The name of the socket is the process ID.

For now, the SIGIO signal is used. Although this signal is used for other purposes on occasion, the siginfo_t structure allows to determine whether the signal was sent by a process or the kernel.

3.6 Data Collection

A single process collects trace data for all processes being traced on the system. This process is called `ustd`. It opens a named socket, called `ustd` and located in the same directory as the applications' sockets. Through it, `ustd` can be commanded to collect the trace data of a certain buffer of a given PID.

Upon receiving this command, `ustd` creates a new thread that connects to the socket of the tracing process, first sending it the SIGIO signal if the socket is not yet available. It then requests the shared memory segment IDs for the buffers and maps them.

Still using the socket of the traced application, this consumer thread sends a command requesting access to the next sub-buffer. When the next unconsumed sub-buffer is full, a reply is sent, and the consumer thread writes its data to the trace file, reading from the shared memory segment. Because the memory area passed to `write()` is in the shared memory segment, no copying in RAM occurs.

3.7 Early and Late Tracing

A few complicating factors must be taken into account when tracing very early or late in the program lifespan.

3.7.1 Tracing from program start

Sometimes, it is important to trace the program from its beginning. One can try to start the program, and then enable tracing. But chances are by the time the SIGIO signal is sent and received, and the command to start tracing is sent through the socket, some events will have been lost. In some cases, the program may have already ended.

Therefore the UST has a special mechanism for tracing from the beginning of the program execution. To trace a program from its beginning, the user can run the program with two environment variables defined. These variables are parsed by the tracing library constructor. Defining both these variables guarantees that by the time the program enters its `main()` function, tracing will have started.

`UST_TRACE=1` Automatically activate tracing on program start.

`UST_AUTOPROBE=1` Automatically enable all instrumentation points.

3.7.2 Tracing until the end of the program

Things are also slightly more complicated when tracing near the end of a program. The program can crash and

be unable to notify `ustd` that its last sub-buffer should be consumed. Worse, it may end before `ustd` is able to map its buffers. In the former case, the end of the trace will be lost. In the latter, the full trace is lost, since the kernel deallocates shared memory segments when their last user disconnects from them. The following describes how the UST deals with these issues.

When a program crashes, its socket connections are closed by the kernel. `ustd` can detect this and run a crash recovery procedure on the buffer. The recovery procedure identifies which sub-buffers contain data that is not yet consumed, and how much data can be recovered in each one of them. This data is appended to the trace file. The procedure guarantees that all the events up to the last that is recovered are valid and that none was skipped (provided there are no lost events in the buffer due to overflow). It is possible to determine what data in each sub-buffer is valid, because some counters used in the atomic algorithm are mapped along with the buffer in the shared memory segment.

When the program lifetime is too short for `ustd` to have time to map its memory, a different problem is encountered. Although the UST does not yet support this case, it is planned to use a destructor to handle this case. If the destructor of the trace library detects that a trace is being recorded and that its buffers have not yet been mapped, it will extend the life of the process slightly to give time to `ustd` to map them.

4 Trace Analysis

LTTV[1], the LTTng Viewer, is a graphical trace viewer for LTTng traces. LTTV provides a number of graphical, statistical and text-based views for traces. Furthermore, it has the ability to display concurrently the events of several traces that were recorded simultaneously. This is useful for viewing traces recorded in virtual machines at the same time as a trace of the host system.

This feature can also be used to display a kernel trace at the same time as userspace traces. In the event list, the events of all the traces are then interleaved. This allows to get a better grasp of problems that involve both the userspace and kernel side. The usage of a precise and common time source ensures events in the list are correctly ordered even if they are produced on different cores or on different sides of the kernel/userspace border.

5 Performance

This section presents some early performance measurements for the UST, as well as a comparison with the performance of DTrace SDT for an equivalent tracing task.

The tests were run cache-hot on a dual quad-core Xeon 2 GHz with 8 GB of RAM. DTrace was run under OpenSolaris. The test consisted in running 60 times the command `find /usr -regex '.*a'`. This regular expression was chosen arbitrarily to provoke `malloc`/`free` activity.

The calls to `malloc` and `free` made by `find` were instrumented. This was done by intercepting the calls to them using a shared library loaded with `LD_PRELOAD`. The intercepting functions contained the actual instrumentation points and called the real version of the function. The `malloc`/`free` interception was active for all tests, even when not tracing. The `malloc`/`free` arguments and return values were recorded by the instrumentation.

Event counts vary between DTrace and UST tests because the `/usr` directory contained more files in the Linux system (for UST tests) than in the OpenSolaris system (used for DTrace tests).

The DTrace performance (Table 1) was first measured with tracing disabled. Then, it was measured with tracing enabled, with two different scripts. One (*printing probe*) printed the function name (`malloc` or `free`), its arguments and its return value. The output was redirected to a file. The other (*simple probe*), only counted the number of events. Its aim was to verify how much time is due to the actual printing operation. The cost per event was obtained by taking the time in excess of the time with tracing disabled and dividing it by the number of events.

The UST performance (Table 2) was measured first with the instrumentation not compiled in and then compiled in. The difference between these two measures was not significant. In fact, in these tests, the execution time diminished when compiling in the instrumentation. With probes connected but tracing not active, the execution time was slightly higher. In this mode of operation, a function call is made upon hitting an instrumentation point, but the function returns almost immediately, after finding out tracing is disabled. Finally, with tracing

Test	Exec. time	Nb. of events	Cost / event
Not tracing	53.29 s	–	–
Tracing, simple probe	251.81 s	44,085,780	4.5 μs
Tracing, printing probe	274.51 s	44,085,780	5.0 μs

Table 1: DTrace results.

Test	Exec. time	Nb. of events	Cost / event
Not tracing, instrumentation not compiled in	92.61 s	–	–
Not tracing, instrumentation compiled in	92.18 s	145,168,560	≈ 0
Not tracing, probes connected	99.25 s	145,168,560	46 ns
Tracing	193.94 s	145,168,560	698 ns

Table 2: LTTng Userspace Tracer results.

enabled, a cost per event of 698 ns was obtained. The cost per event was calculated by taking the time in excess of the time with instrumentation not compiled in and dividing it by the number of events.

The LTTng UST had a cost per event more than 7 times lower than DTrace. This difference is explained by the method used by each tracer to record events. While DTrace executes a trap at each event, the UST writes the event in a buffer in the program memory, saving a round-trip to the kernel.

The UST has a low per event cost, while having no apparent impact while disabled. This makes it particularly useful in production systems, and other systems where affecting performance as little as possible is critical. Its compact trace format further limits its impact by limiting the disk and network usage.

As the UST becomes more mature, it is likely that new optimizations will result in an even lower cost per event. The Future Work sections mentions a few possibilities to this effect.

6 Future Work

The current per-process buffers were a simple first step for a port. However, this approach has an important limitation. It induces cacheline bouncing on multi-threaded applications. Using per-thread buffers would fix this problem.

In the kernel, the most optimized variant of the markers uses *immediate values*, a technique that modifies an instruction at the instrumentation point site when enabling or disabling markers. This code modification consists in changing the immediate value in a *load immediate* instruction. This instruction is immediately followed by a test of the register in which the value was loaded. Depending on the result of the test, the event is recorded or not. Although this approach is faster than the current test of a global variable, is much more architecture-dependant.

`UST_AUTOPROBE` should allow the specification of a list or pattern of markers. Its current limitation of activating all of them at once may cause a performance penalty that is higher than necessary on programs where markers encountered extremely often are compiled in but not needed for the specific problem being debugged.

Complex programs that necessitate userspace tracing are often written in high-level languages. Therefore the UST should be available to these languages. For example, a Java API using the JNI to interface the C API would be straightforward to implement.

Work is currently in progress to enhance the daemon so it can send traces over a network. This is particularly useful on special purpose systems with little or no disk space available.

References

[1] LTTV. `http://lttng.org`.

[2] Systemtap static probes. `https://fedoraproject.org/wiki/Features/SystemtapStaticProbes`.

[3] Mathieu Desnoyers and Michel R. Dagenais. The LTTng tracer: A low impact performance and

behavior monitor for GNU/Linux. In *Linux Symposium*, Ottawa, Ontario, Canada, June 2006.

[4] Mathieu Desnoyers and Paul E. McKenney. Userspace Read-Copy-Update Library. `http://ltt.polymtl.ca/cgi-bin/gitweb.cgi?p=userspace-rcu.git`.

[5] Pierre-Marc Fournier and Jan Blunck. libkcompat. `http://git.dorsal.polymtl.ca/?p=libkcompat.git`.

[6] Frank Hofmann. The DTrace backend on Solaris for x86/x64. `http://opensolaris.org/os/project/czosug/events_archive/czosug2_dtrace_x86.pdf`.

[7] Steve Rostedt. ftrace. `http://lwn.net/Articles/290277/`.

[8] Karim Yaghmour, R Wisniewski, R Moore, and M Dagenais. relayfs: An efficient unified approach for transmitting data from kernel to user space. In *Linux Symposium*, Ottawa, Ontario, Canada, 2003.