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

The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux

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

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

Abstract

Efficient tracing of system-wide execution, allowing integrated analysis of both kernel space and user space, is something difficult to achieve. The following article will present you a new tracer core, Linux Trace Toolkit Next Generation (LTTng), that has taken over the previous version known as *LTT*. It has the same goals of low system disturbance and architecture independance while being fully reentrant, scalable, precise, extensible, modular and easy to use. For instance, LTTng allows tracepoints in NMI code, multiple simultaneous traces and a flight recorder mode. LTTng reuses and enhances the existing LTT instrumentation and RelayFS.

This paper will focus on the approaches taken by LTTng to fulfill these goals. It will present the modular architecture of the project. It will then explain how NMI reentrancy requires atomic operations for writing and RCU lists for tracing behavior control. It will show how these techniques are inherently scalable to multiprocessor systems. Then, time precision limitations in the kernel will be discussed, followed by an explanation of LTTng's own monotonic timestamps motives.

In addition, the template based code generator for architecture agnostic trace format will be presented. The approach taken to allow nested types, variable fields and dynamic alignment of data in the trace buffers will be revealed. It will show the mechanisms deployed to facilitate use and extension of this tool by adding custom instrumentation and analysis involving kernel, libraries and user space programs.

It will also introduce LTTng's trace analyzer and graphical viewer counterpart: Linux Trace Toolkit Viewer (LTTV). The latter implements extensible analysis of the trace information through collaborating text and graphical plugins.[1] It can simultaneously display multiple multi-GBytes traces of multi-processor systems.

1 Tracing goals

With the increasing complexity of newer computer systems, the overall performance of applications often depends on a combination of several factors including I/O subsystems, device drivers, interrupts, lock contention among multiple CPUs, scheduling and memory management. A low impact, high performance, tracing system may therefore be the only tool capable of collecting the information produced by instrumenting the whole system, while not

[1] Project website: http://ltt.polymtl.ca.

changing significantly the studied system behavior and performance.

Besides offering a flexible and easy to use interface to users, an efficient tracer must satisfy the requirements of the most demanding application. For instance, the widely used `printk` and `printf` statements are relatively easy to use and are correct for simple applications, but do not offer the needed performance for instrumenting interrupts in high performance multiprocessor computer systems and cannot necessarily be used in some code paths such as non maskable interrupts (NMI) handlers.

An important aspect of tracing, particularly in the real-time and high performance computing fields, is the precision of events timestamps. Real-time is often used in embedded systems which are based on a number of different architectures (e.g. ARM, MIPS, PPC) optimized for various applications. The challenge is therefore to obtain a tracer with precise timestamps, across multiple architectures, running from several MHz to several GHz, some being multi-processors.

The number of ad hoc tracing systems devised for specific needs (several Linux device drivers contain a small tracer), and the experience with earlier versions of LTT, show the needs for a flexible and extensible system. This is the case both in terms of adding easily new instrumentation points and in terms of adding plugins for the analysis and display of the resulting trace data.

2 Existing solutions

Several different approaches have been taken by performance monitoring tools. They usually adhere to one of the following two paradigms. The first class of monitor, *post-processing*, aims to minimize CPU usage during the execution of the monitored system by collecting data for later off-line analysis. As the goal is to have minimum impact on performance, static instrumentation is habitually used in this approach. Static instrumentation consists in modifying the program source code to add logging statements that will compile with the program. Such systems include LTT [7], a Linux kernel Tracer, K42 [5], a research operating system from IBM, IrixView and Tornado which are commercial proprietary products.

The second class of monitor aims at calculating well defined information (e.g. I/O requests per seconds, system calls per second per PID) on the monitored CPU itself: it is what is generally called a *pre-processing* approach. It is the case of SystemTAP [3], Kerninst [4], Sun's dtrace [1] and IBM's Performance and Environment Monitoring (PEM) [6]. All except PEM use a dynamic instrumentation approach. Dynamic instrumentation is performed by changing assembly instructions for breakpoints in the program binary objects loaded in memory, like the gdb debugger does. It is suitable to their goal because it generally has a negligible footprint compared to the pre-processing they do.

Since our goal is to support high performance and real-time embedded systems, the dynamic probe approach is too intrusive, as it implies using a costly breakpoint interrupt. Furthermore, even if the pre-processing of information can sometimes be faster than logging raw data, it does not allow the same flexibility as post-processing analysis. Indeed, almost every aspect of a system can be studied once is obtained a trace of the complete flow of the system behavior. However, pre-processed data can be logged into a tracer, as does PEM with K42, for later combined analysis, and the two are therefore not incompatible.

3 Previous Works

LTTng reuses research that has been previously done in the operating system tracing field in order to build new features and address currently unsolved questions more thoroughly.

The previous Linux Trace Toolkit (LTT) [7] project offers an operating system instrumentation that has been quite stable through the 2.6 Linux kernels. It also has the advantage of being cross-platform, but with types limited to fixed sizes (e.g. fixed 8, 16, 32, or 64-byte integers compared to host size byte, short, integer, and long). It also suffers from the monolithic implementation of both the LTT tracer and its viewer which have proven to be difficult to extend. Another limitation is the use of the kernel NTP corrected time for timestamps, which is not monotonic. LTTng is based on LTT but is a new generation, layered, easily extensible with new event types and viewer plugins, with a more precise time base and that will eventually support the combined analysis of several computers in a cluster [2].

RelayFS [8] has been developed as a standard high-speed data relay between the kernel and user space. It has been integrated in the 2.6.14 Linux kernels. It offers hooks for kernel clients to send information in large buffers and interacts with a user space daemon through file operations on a memory mapped file.

IBM, in the past years, has developed K42 [5], an open source research kernel which aims at full scalability. It has been designed from the ground up with tracing being a necessity, not an option. It offers a very elegant lockless tracing mechanism based on the atomic compare-and-exchange operation.

The Performance and Environment Monitoring (PEM) [6] project shares a few similarities with LTTng and LTTV since some work have been done in collaboration with members of their team. The XML file format for describing events came from these discussions, aiming at standardizing event description and trace formats.

4 The LTTng approach

The following subsections describe the five main components of the LTTng architecture. The first one explains the control of the different entities in LTTng. It is followed by a description of the data flow in the different modules of the application. The automated static instrumentation will thereafter be introduced. Event type registration, the mecanism that links the extensible instrumentation to the dynamic collection of traces, will then be presented.

4.1 Control

There are three main parts in LTTng: a user space command-line application, *lttctl*; a user space daemon, *lttd*, that waits for trace data and writes it to disk; and a kernel part that controls kernel tracing. Figure 1 shows the control paths in LTTng. *lttctl* is the command line application used to control tracing. It starts a *lttd* and controls kernel tracing behavior through a library-module bridge which uses a netlink socket.

The core module of LTTng is *ltt-core*. This module is responsible for a number of LTT control events. It controls helper modules *ltt-heartbeat*, *ltt-facilities*, and *ltt-statedump*. Module *ltt-heartbeat* generates periodic events in order to detect and account for cycle counters overflows, thus allowing a single monotonically increasing time base even if shorter 32-bit (instead of 64-bit) cycle counts are stored in each event. *Ltt-facilities* lists the facilities

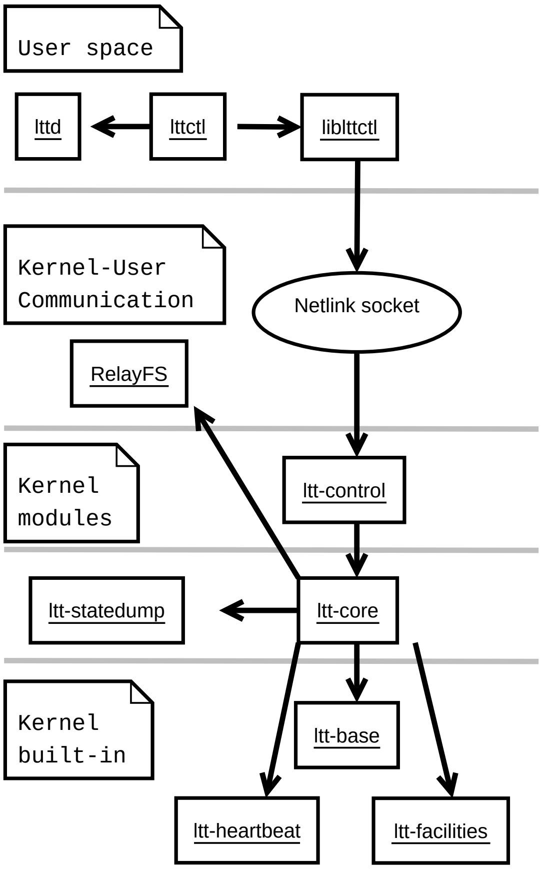

Figure 1: LTTng control architecture

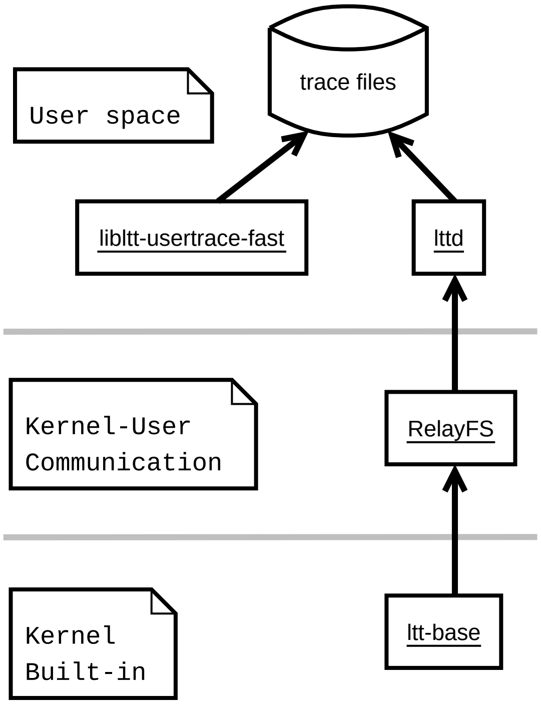

Figure 2: LTTng data flow

(collection of event types) currently loaded at trace start time. Module *ltt-statedump* generates events to describe the kernel state at trace start time (processes, files…). A builtin kernel object, *ltt-base*, contains the symbols and data structures required by builtin instrumentation. This includes principally the tracing control structures.

4.2 Data flow

Figure 2 shows the data flow in LTTng. All data is written through *ltt-base* into RelayFS circular buffers. When subbuffers are full, they are delivered to the *lttd* disk writer daemon.

Lttd is a standalone multithreaded daemon which waits on RelayFS channels (files) for data by using the poll file operation. When it is awakened, it locks the channels for reading by using a *relay buffer get* ioctl. At that point, it has exclusive access to the subbuffer it has reserved and can safely write it to disk. It should then issue a *relay buffer put* ioctl to release it so it can be reused.

A side-path, *libltt-usertrace-fast*, running completely in user space, has been developed for high throughput user space applications which need high performance tracing. It is explained in details in Section 4.5.4.

Both *lttd* and the *libltt-usertrace-fast* companion process currently support disk output, but should eventually be extended to other media like network communication.

4.3 Instrumentation

LTTng instrumentation, as presented in Figure 3, consists in an XML event description that

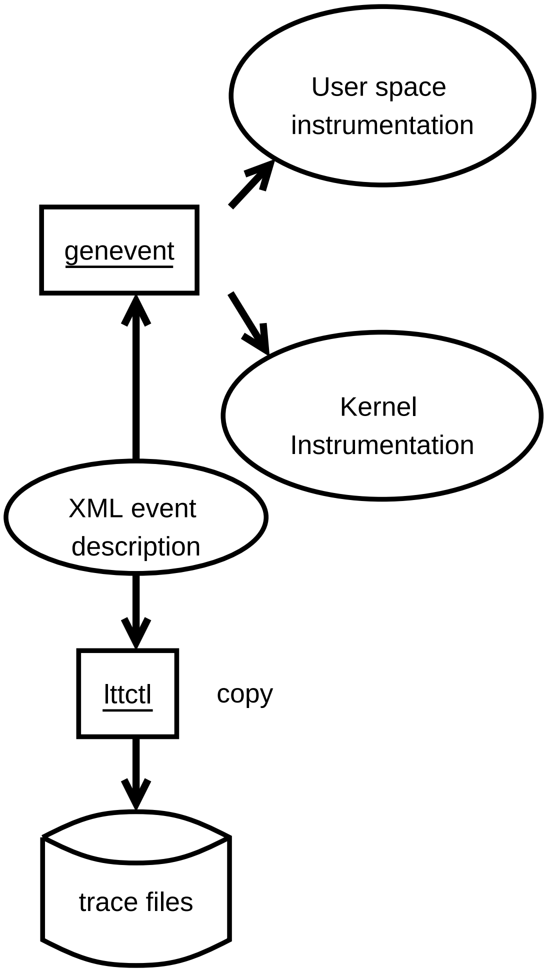

Figure 3: LTTng instrumentation

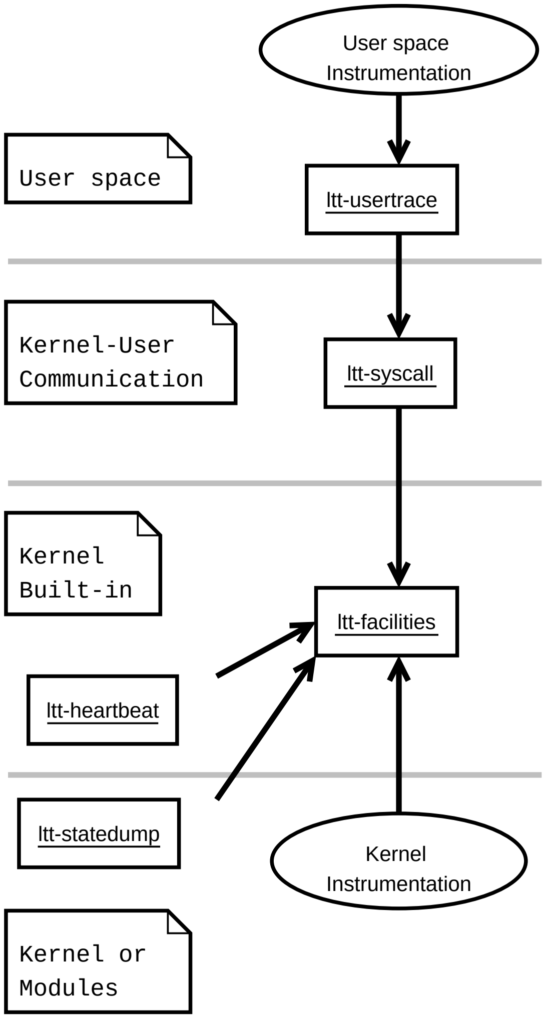

Figure 4: LTTng event type registration

is used both for automatically generating *tracing headers* and as *data metainformation* in the trace files. These tracing headers implement the functions that must be called at instrumentation sites to log information in traces.

Most common types are supported in the XML description: fixed size integers, host size integers (int, long, pointer, size_t), floating point numbers, enumerations, and strings. All of these can be either host or network byte ordered. It also supports nested arrays, sequences, structures, and unions.

The tracing functions, generated in the tracing headers, serialize the C types given as arguments into the LTT trace format. This format supports both packed or aligned data types.

A record generated by a probe hit is called an *event*. Event types are grouped in facilities. A *facility* is a dynamically loadable object, either a kernel module for kernel instrumentation or a user space library for user space instrumentation. An object that calls instrumentation should be linked with its associated facility object.

4.4 Event type registration

Event type registration is centralized in the *ltt-facilities* kernel object, as shown in Figure 4. It controls the rights to register specific type of information in traces. For instance, it does not allow a user space process using the *ltt-usertrace* API to register facilities with names conflicting with kernel facilities.

The *ltt-heartbeat* built-in object and the *ltt-statedump* also have their own instrumentation to log events. Therefore, they also register to *ltt-facilities*, just like standard kernel instrumentation.

Registered facility names, checksums and type sizes are locally stored in *ltt-facilities* so they can be dumped in a special low traffic channel at trace start. Dynamic registration of new facilities, while tracing is active, is also supported.

Facilities contain information concerning the type sizes in the compilation environment of the associated instrumentation. For instance, a facility for a 32-bit process would differ from the same facility compiled with a 64-bit process from its long and pointer sizes.

4.5 Tracing

There are many similarities between Figure 4 and Figure 5. Indeed, each traced information must have its metainformation registered into *ltt-facilities*. The difference is that Figure 4 shows how the metainformation is registered while Figure 5 show the actual tracing. The tracing path has the biggest impact on system behavior because it is called for every event.

Each event recorded uses *ltt-base*, container of the active traces, to get the pointers to RelayFS buffers. One exception is the *libltt-usertrace-fast* which will be explained at Subsection 4.5.4.

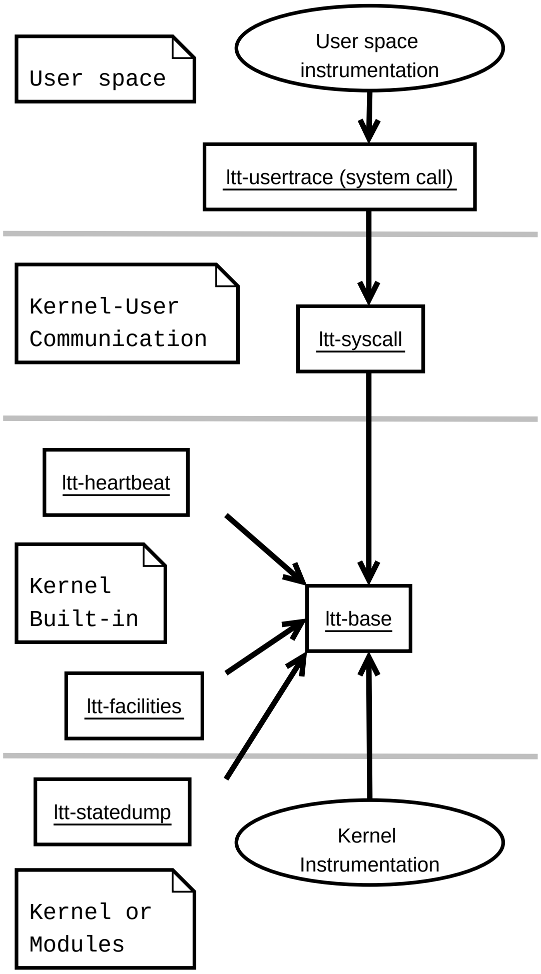

Figure 5: LTTng tracing

The algorithms used in these tracing sites which make them reentrant, scalable, and precise will now be explained.

4.5.1 Reentrancy

This section presents the lockless reentrancy mechanism used at LTTng instrumentation sites. Its primary goal is to provide correct tracing throughout the kernel, including non-

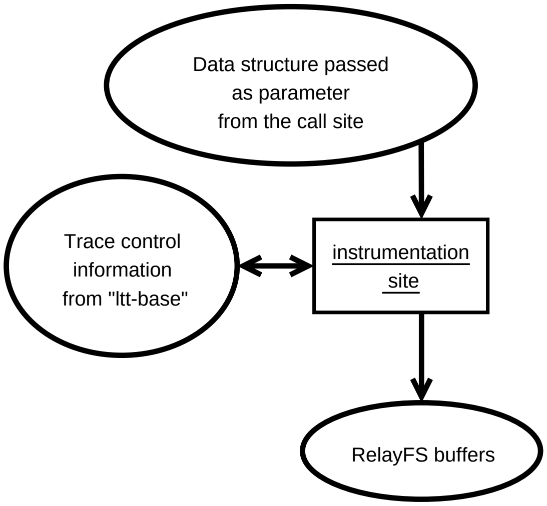

Figure 6: LTTng instrumentation site

maskable interrupts (NMI) handlers which cannot be disabled like normal interrupts. The second goal is to have the minimum impact on performance by both having a fast code and not disrupting normal system behavior by taking intrusive locks or disabling interrupts.

To describe the reentrancy mechanism used by the LTTng instrumentation site (see Figure 6), we define the *call site*, which is the original code from the instrumented program where the tracing function is called. We also define the *instrumentation site*, which is the tracing function itself.

The instrumentation site found in the kernel and user space instrumentation has very well defined inputs and outputs. Its main input is the call site parameters. The call site must insure that the data given as parameter to the instrumentation site is properly protected with its associated locks. Very often, such data is already locked by the call site, so there is often no need to add supplementary locking.

The other input that the instrumentation site takes is the global trace control information. It is contained in a RCU list of active traces in the *ltt-base* object. Note that the instrumentation site uses the *trace control information* both as an input and output: this is both how tracing behavior is controlled and where variables that control writing to RelayFS buffers are stored.

The main output of the instrumentation site is a serialized memory write of both an event header and the instrumentation site parameters to the per-CPU RelayFS buffers. The location in these buffers is protected from concurrent access by using a lockless memory write scheme inspired from the one found in K42 [5]:

First, the *amount of memory space* necessary for the memory write is computed. When the data size is known statically, this step is quite fast. If, however, variable length data (string or sequence) must be recorded, a first size calculation pass is performed. *Alignment* of the data is taken care of in this step. To speed up data alignment, the start address of the variable size data is always aligned on the architecture size: it makes it possible to do a compile time alignement computation for all fixed size types.

Then, a memory region in the buffers is *reserved* atomically with a compare-and-exchange loop. The algorithm retries the reservation if a concurrent reserve occurs. The timestamp for the event is taken inside the compare-and-exchange loop so that it is monotically incrementing with buffer offsets. This is done to simplify data parsing in the post-processing tool.

A reservation can fail on the following conditions. In *normal* tracing mode, a buffer full condition causes the reservation to fail. On the other hand, in *flight recorder* mode, we overwrite non-read buffers, so it will never fail. When the reservation fails, the event lost counter is incremented and the instrumentation site will return without doing a *commit*.

The next step is to *copy* data from the instrumentation site arguments to the RelayFS reserved memory region. This step must preserve the same data alignment that has been calculated earlier.

Finally, a *commit* operation is done to release the reserved memory segment. No information is kept on a per memory region basis. We only keep a count of the number of reserved and committed bytes per subbuffer. A subbuffer is considered to be in a consistent state (non-corrupted and readable) when both counts are equal.

Is is possible that a process die between the slot reservation and commit because of a kernel OOPS. In that case, the lttd daemon will be incapable of reading the subbuffer affected by this condition because of unequal reserve and commit counts. This situation is resolved when the reservation algorithm wraps to the faulty subbuffer: if the reservation falls in a new buffer that has unequal reserve and commit counts, the reader (lttd) is pushed to the next subbuffer, the subbuffers lost counter is incremented, and the subbuffer is overwritten. To insure that this condition will not be reached by normal out of order commit of events (caused by nested execution contexts), the buffer must be big enough to contain data recorded by the maximum number of out of order events, which is limited by the longest sequence of events logged from nestable contexts (softirq, interrupts, and NMIs).

The *subbuffer delivery* is triggered by a flag from the call site on subbuffer switch. It is periodically checked by a timer routine to take the appropriate actions. This ensures atomicity and a correct lockless behavior when called from NMI handlers.

Compared to `printk`, which calls the scheduler, disables interrupts, and takes spinlocks, LTTng offers a more robust reentrancy that makes it callable from the scheduler code and from NMI handlers.

4.5.2 Scalability

Scalability of the tracing code for SMP machines is insured by use of per-CPU data and by the lockless tracing mechanism. The inputs of the instrumentation site are scalable: the data given as parameter is usually either on the caller's stack or already properly locked. The global trace information is organized in a RCU list which does not require any lock from the reader side.

Per-CPU buffers eliminate the false sharing of cachelines between multiple CPUs on the memory write side. The fact that input-output trace control structures are per-CPU also eliminates false sharing.

To identify more precisely the performance cost of this algorithm, let's compare two approaches: taking a per-CPU spinlock or using an atomic compare-and-exchange operation. The most frequent path implies either taking and releasing a spinlock along with disabling interrupts or doing a compare-and-exchange, an atomic increment of the reserve count, and an atomic increment of the commit count.

On a 3GHz Pentium 4, a compare-and-exchange without LOCK prefix costs 29 cycles. With a LOCK prefix, it raises to 112 cycles. An atomic increment costs respectively 7 and 93 cycles without and with a LOCK prefix. Using a spinlock with interrupts disabled costs 214 cycles.

As LTTng uses per-CPU buffers, it does not need to take a lock on memory to protect from other CPU concurrent access when performing these operations. Only the non-locked versions of compare-and-exchange and atomic increment are then necessary. If we consider only the time spent in atomic operations, using compare-and-exchange and atomic increments takes 43 cycles compared to 214 cycles for a spinlock.

Therefore, using atomic operations is five times faster than an equivalent spinlock on this architecture while having the additionnal benefit of being reentrant for NMI code and not disturbing the system behavior, as it does not disable interrupts for the duration of the tracing code.

4.5.3 Time (im)precision in the Linux kernel

Time precision in the Linux kernel is a research subject on its own. However, looking at the Linux kernel x86 timekeeping code is very enlightening on the nanosecond timestamps accuracy provided by the kernel. Effectively, it is based on a CPU cycle to nanosecond scaling factor computed at boot time based on the timer interrupt. The code that generates and uses this scaling factor takes for granted that the value should only be precise enough to keep track of scheduling periods. Therefore, the focus is to provide a fast computation of the time with shifting techniques more than providing a very accurate timestamp. Furthermore, doing integer arithmetic necessarily implies a loss of precision.

It causes problems when a tool like LTTng strongly depends on the monotonicity and precision of the time value associated with timestamps.

To overcome the inherent kernel time precision limitations, LTTng directly reads the CPU timestamp counters. It uses the `cpu_khz` kernel variable which contains the most precise calibrated CPU frequency available. This value will be used by the post-processing tool, LTTV, to convert cycles to nanoseconds in a precise manner with double precision numbers.

Due to the importance of the CPU timestamp counters in LTTng instrumentation, a workaround has been developed to support architectures that only have a 32-bit timestamp counter available. It uses the *ltt-heartbeat* module periodic timer to keep a full 64-bit timestamp counter on architectures where it is missing by detecting the 32-bit overflows in an atomic fashion; both the previous and the current TSC values are kept, swapped by a pointer change upon overflow. The read-side must additionnaly check for overflows.

It is important to restate that the time base used by LTTng is based neither on the kernel `do_gettimeofday`, which is NTP corrected and thus non monotonic nor on the kernel monotonic time, which suffers from integer arithmetic imprecision. LTTng uses the CPU timestamp counter and its most accurate calibration.

4.5.4 User space tracing

User space tracing has been achieved in many ways in the past. The original LTT [7] did use write operations in a device to send events to the kernel. It did not, however, give the same performances as in kernel events, as it needs a round-trip to the kernel and many copies of the information.

K42 [5] solves this by sharing per-CPU memory buffers between the kernel and user space processes. Although this is very performant, it does not insure secure tracing, as a given process can corrupt the traces that belong to other processes or to the kernel. Moreover, sharing memory regions between the kernel and user space might be acceptable for a research kernel, but for a production kernel, it implies a weaker traceability of process-kernel communications and might bring limitations on architectures with mixed 32- and 64-bit processes.

LTTng provides user space tracing through two different schemes to suit two distincts categories of instrumentation needs.

The first category is characterized by a very low event throughput. It can be the case of an event than happens rarely to show a specific error condition or periodically at an interval typically greater or equal to the scheduler period. The "slow tracing path" is targeted at this category.

The second category, which is addressed by the "fast tracing path," is much more demanding. It is particularly I/O intensive and must be close to the performance of a direct memory write. This is the case when instrumenting manually the critical path in a program or automatically every function entry/exit by gcc.

Both mecanisms share the same facility registration interface with the kernel, which passes through a system call, as shown in Figure 4. Validation is done by limiting these user space facilities to their own namespace so they cannot imitate kernel events.

The *slow path* uses a costly system call at each event call site. Its advantage is that it does not require linking the instrumented program against any library and does not have any thread startup performance impact like the *fast path* explained below. Every event logged through the system call is copied in the kernel tracing buffers. Before doing so, the system call verifies that the facility ID corresponds to a valid user space facility.

The *fast path*, *libltt-usertrace-fast* (at Figure 2) library consists in a per thread *companion* process which writes the buffers directly to disk. Communication between the thread and the library is done through the use of circular buffers in an anonymous shared memory map. Writing the buffers to disk is done by a separate *companion* process to insure that buffered data is never lost when the traced program terminates. The other goal is to account the time spent writing to disk to a different process than the one being traced. The file is written in the filesystem, arbitrarily in `/tmp/ltt-usertrace`, in files following this naming convention: `process-tid-pid-timestamp`, which makes it unique for the trace. When tracing is over, the `/tmp/ltt-usertrace` must be manually moved into the kernel trace. The trace and usertrace do not have to coincide: although it is better to have the usertrace time span included in the kernel trace interval to benefit from the scheduler information for the running processes, it is not mandatory and partial information will remain available.

Both the slow and the fast path reuse the lockless tracing algorithm found in the LTTng kernel tracer. In the fast path, it ensures reentrancy with signal handlers without the cost of disabling signals at each instrumentation site.

5 Graphical viewer: LTTV

LTTng is independent of the viewer, the trace format is well documented and a trace-reading library is provided. Nonetheless, the associated viewer, LTTV, will be briefly introduced. It implements optimised algorithms for random access of several multi-GB traces, describing the behavior of one or several uniprocessor or multi-processor systems. Many plugin views can be loaded dynamically into LTTV for the display and analysis of the data. Developers can thus easily extend the tool by creating their own instrumentation with the flexible XML description and connect their own plugin to that information. It is layered in a modular architecture.

On top of the LGPL low-level trace files reading library, LTTV recreates its own representation of the evolving kernel state through time and keeps statistical information into a generic hierarchical container. By combining the kernel state, the statistics, and the trace events, the viewers and analysis plugins can extend the information shown to the user. Plugins are kept focused (analysis, text, or graphical display, control...) to increase modularity and reuse. The plugin loader supports dependency control.

LTTV also offers a rich and performant event filter, which allows specifying, with a logical expression, the events a user is interested to see. It can be reused by the plugins to limit their scope to a subset of the information.

For performance reasons, LTTV is written in C. It uses the GTK graphical library and glib. It is distributed under the GPLv2 license.

6 Results

This section presents the results of several measurements. We first present the time overhead on the system running microbenchmarks of the instrumentation site. Then, taking these results as a starting point, the interrupt and scheduler impact will be discussed. Macrobenchmarks of the system under different loads will then be shown, detailing the time used for tracing.

The size of the instrumentation object code will be discussed along with possible size optimisations. Finally, time precision calibration is performed with a NMI timer.

6.1 Test environment

The test environment consists of a 3GHz, uniprocessor Pentium 4, with hyperthreading disabled, running LTTng 0.5.41. The results are presented in cycles; the exact calibration of the CPU clock is 3,000.607 MHz.

6.2 Microbenchmarks

Table 1 presents probe site microbenchmarks. Kernel probe tests are done in a kernel module with interrupts disabled. User space tests are influenced by interrupts and the scheduler. Both consist in 20,000 hits of a probe that writes 4 bytes plus the event header (20 bytes). Each hit is surrounded by two timestamp counter reads.

When compiling out the LTTng tracing, calibration of the tests shows that the time spent in the two TSC reads varies between 97 and 105 cycles, with an average of 100.0 cycles. We therefore removed this time from the raw probe time results.

As we can see, the best case for kernel tracing is a little slower than the *ltt-usertrace-fast* library: this is due to supplementary operations that must be done in the kernel (preemption disabling for instance) that are not needed in user space. The maximum and average values of time spent in user space probes does not mean much because they are sensitive to scheduling and interrupts.

The key result in Table 1 is the average 288.5 cycles (96.15ns) spent in a probe.

LTTng probe sites do not increase latency because they do not disable interrupts. However, the interrupt entry/exit instrumentation itself does increase interrupt response time, which

Probe site	Test Series	Time spent in probe (cycles)		
		min	average	max
Kernel	Tracing dynamically disabled	0	0.000	338
Kernel	Tracing active (1 trace)	278	288.500	6,997
User space	*ltt-usertrace-fast* library	225	297.021	88,913
User space	Tracing through system call	1,013	1,042.200	329,062

Table 1: LTTng microbenchmarks for a 4-byte event probe hit 20,000 times

therefore increases low priority interrupts latency by twice the probe time, which is 577.0 cycles (192.29ns).

The scheduler response time is also affected by LTTng instrumentation because it must disable preemption around the RCU list used for control. Furthermore, the scheduler instrumentation itself adds a task switch delay equal to the probe time, for a total scheduler delay of twice the probe time: 577.0 cycles (192.29ns). In addition, a small implementation detail (use of `preempt_enable_no_resched()`), to insure scheduler instrumentation reentrancy, has a downside: it can possibly make the scheduler miss a timer interrupt. This could be solved for real-time applications by using the *no resched* flavour of preemption enabling only in the scheduler, wakeup, and NMI nested probe sites.

6.3 Macrobenchmarks

6.3.1 Kernel tracing

Table 2 details the time spent both in the instrumentation site and in *lttd* for different loads. Time spent in instrumentation is computed from the average probe time (288.5 cycles) multiplied by the number of probe hits. Time spent in *lttd* is the CPU time of the *lttd* process as given in the LTTV analysis. The load is computed by subtracting the time spent in *system call* mode in process 0 (idle process) from the wall time.

It is quite understandable that the probes triggered by the ping flood takes that much CPU time, as it instruments a code path that is called very often: the *system call* entry. The total cpu time used by tracing on a busy system (medium and high load scenarios) goes from 1.54 to 2.28%.

6.3.2 User space tracing

Table 3 compares the *ltt-usertrace-fast* user space tracer with gprof on a specific task: the instrumentation of each function entry and exit of a gcc compilation execution. You will see that the userspace tracing of LTTng is only a constant factor of 2 slower than a gprof instrumented binary, which is not bad considering the amount of additional data generated. The factor of 2 is for the ideal case where the daemon writes to a `/dev/null` output. In practice, the I/O device can further limit the throughput. For instance, writing the trace to a SATA disk, LTTng is 4.13 times slower than gprof.

The next test consists in running an instrumented version of gcc, itself compiled with option `-finstrument-functions`, to compile a 6.9KiB C file into a 15KiB object, with level 2 optimisation.

As Table 3 shows, gprof instrumented gcc takes 1.73 times the normal execution time. The

Load size	Test Series	CPU time (%)			Data rate (MiB/s)	Events/s
		load	probes	lttd		
Small	mozilla (browsing)	1.15	0.053	0.27	0.19	5,476
Medium	find	15.38	1.150	0.39	2.28	120,282
High	find + gcc	63.79	1.720	0.56	3.24	179,255
Very high	find + gcc + ping flood	98.60	8.500	0.96	16.17	884,545

Table 2: LTTng macrobenchmarks for different loads

gcc instrumentation	Time (s)	Data rate (MiB/s)
not instrumented	0.446	
gprof	0.774	
LTTng (null output)	1.553	153.25
LTTng (disk output)	3.197	74.44

Table 3: gcc function entry/exit tracing

Instrumentation	object code size (bytes)
log 4-byte integer	2,288
log variable length string	2,384
log a structure of int, string, sequence of 8-byte integers	2,432

Table 4: Instrumentation object size

fast userspace instrumentation of LTTng is 3.22 times slower than normal. Gprof only extracts sampling of function time by using a periodical timer and keeps per function counters. LTTng extracts the complete function call trace of a program, which generates an output of 238MiB in 1.553 seconds (153.25 MiB/s). The execution time is I/O-bound, it slows down to 3.197s when writing the trace on a SATA disk through the operating system buffers (74.44 MiB/s).

6.4 Instrumentation objects size

Another important aspect of instrumentation is the size of the binary instructions added to the programs. This wastes precious L1 cache space and grows the overall object code size, which is more problematic in embedded systems. Table 4 shows the size of stripped objects that only contain intrumentation.

Independently of the amount of data to trace, the object code size only varies in our tests of a maximum of 3.3% from the average size. Adding 2.37kB per event might be too much for embedded applications, but a tradeoff can be done between inlining of tracing sites (and reference locality) and doing function calls, which would permit instrumentation code reuse.

A complete L1 cache hit profiling should be done to fully see the cache impact of the instrumentation and help tweak the inlining level. Such profiling is planned.

6.5 Time precision

Time precision measurement of a timestamp counter based clock source can only be done relatively to another clock source. The following test traces the NMI watchdog timer, using it as a comparison clock source. It has the advantage of not being disturbed by CPU load as these interruptions cannot be deactivated. It is, however, limited by the precision of

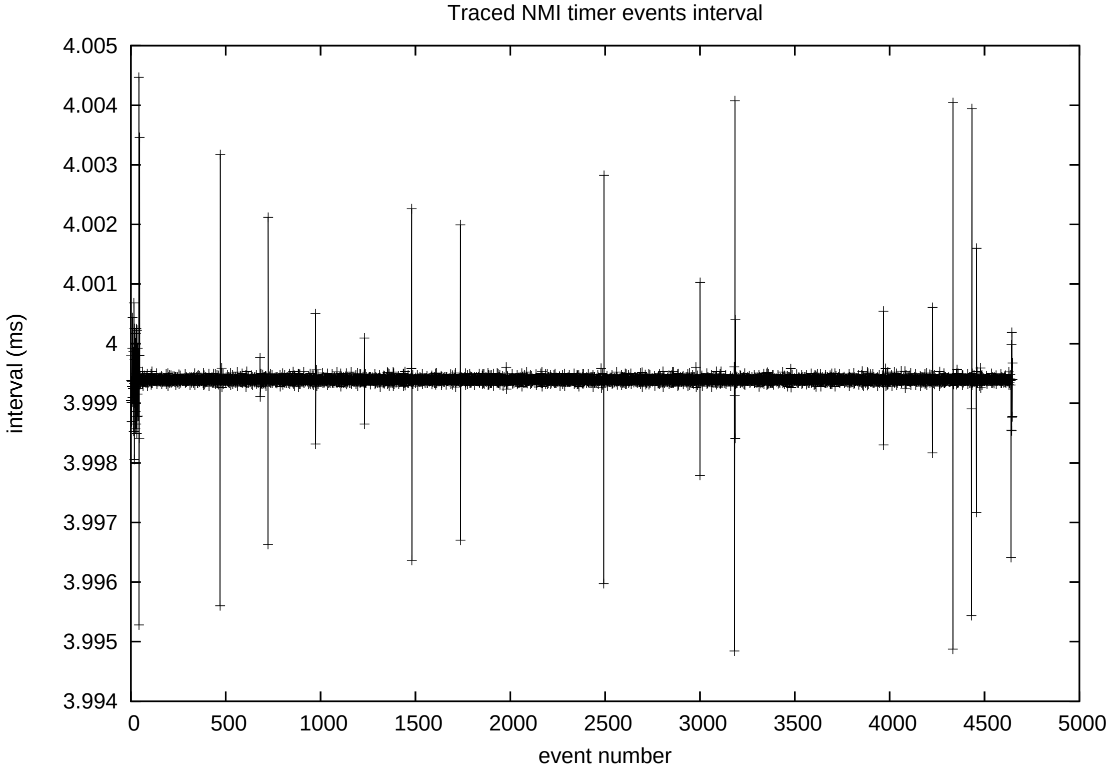

Figure 7: Traced NMI timer events interval

the timer crystal. The hardware used for these tests is an Intel D915-GAG motherboard. Its timer is driven by a TXC HC-49S crystal with a ±30 PPM precision. Table 5 and Figure 7 show the intervals of the logged NMI timer events. Their precision is discussed.

This table indicates a standard deviation of 52ns and a maximum deviation of 5,075ns from the average. If we take the maximum deviation as a worse case, we can assume than we have a ±5.075μs error between the programmable interrupt timer (PIT) and the trace time base (derived from the CPU TSC). Part of it is due to the CPU cache misses, higher priority NMIs, kernel minor page faults, and the PIT itself. A 52ns standard deviation each 4ms means a 13μs error each second, for a 13 PPM frequency precision which is within the expected limits.

Statistic	value (ns)
min	3,994,844
average	3,999,339
max	4,004,468
standard deviation	52
max deviation	5,075

Table 5: Traced NMI timer events interval

7 Conclusion

As demonstrated in the previous section, LTTng is a low disturbance tracer that uses about 2% of CPU time on a heavy workload. It is entirely based on atomic operations to insure reentrancy. This enables it to trace a wide range of code sites, from user space programs and libraries to kernel code, in every execution context, including NMI handlers.

Its time measurement precision gives a 13 PPM frequency error when reading the programmable interrupt timer (PIT) in NMI mode, which is coherent with the 30 PPM crystal precision.

LTTng proves to be a performant and precise tracer. It offers an architecture independent instrumentation code generator, from templates, to reduce instrumentation effort. It provides efficient and convenient mechanisms for kernel and user space tracing.

A plugin based analysis tool, LTTV, helps to further reduce the effort for analysis and visualisation of complex operating system behavior. Some work is actually being done in time synchronisation between cluster nodes, to extend LTTV to cluster wide analysis.

You are encouraged to use this tool and create new instrumentations, either in user space or in the kernel. LTTng and LTTV are distributed under the GPLv2 license.[2]

References

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

[2] Michel Dagenais, Richard Moore, Robert Wisniewski, Karim Yaghmour, and Thomas Zanussi. Efficient and accurate tracing of events in linux clusters. In *Proceedings of the Conference on High Performance Computing Systems (HPCS)*, 2003.

[3] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. In *OLS (Ottawa Linux Symposium) 2005*, 2005.

[4] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *3rd Symposium on Operating Systems Design and Implementation*, February 1999.

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

[6] Robert W. Wisniewski, Peter F. Sweeney, Kartik Sudeep, Matthias Hauswirth, Evelyn Duesterwald, Calin Cascaval, and Reza Azimi. Pem performance and environment monitoring for whole-system characterization and optimization. In *PAC2 (Conference on Power/Performance interaction with Architecture, Circuits, and Compilers)*, 2004.

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

[8] Tom Zanussi, Karim Yaghmour Robert Wisniewski, Richard Moore, and Michel Dagenais. relayfs: An efficient unified approach for transmitting data from kernel to user space. In *OLS (Ottawa Linux Symposium) 2003*, pages 519–531, 2003.

[2] Project website: `http://ltt.polymtl.ca`