Lockless Multi-Core High-Throughput Buffering Scheme for Kernel Tracing

Mathieu Desnoyers
EfficiOS Inc.
mathieu.desnoyers@efficios.com

Michel R. Dagenais
Dept. of Computer and Software Eng.
Ecole Polytechnique de Montreal
P.O. Box 6079, Station Downtown
Montreal, Quebec, Canada, H3C 3A7
michel.dagenais@polymtl.ca

ABSTRACT

Studying execution of concurrent real-time online systems, to identify far-reaching and hard to reproduce latency and performance problems, requires a mechanism able to cope with voluminous information extracted from execution traces. Furthermore, the workload must not be disturbed by tracing, thereby causing the problematic behavior to become unreproducible.

In order to satisfy this low-disturbance constraint, we created the `LTTng` kernel tracer. It is designed to enable safe and race-free attachment of probes virtually anywhere in the operating system, including sites executed in non-maskable interrupt context.

In addition to being reentrant with respect to all kernel execution contexts, `LTTng` offers good performance and scalability, mainly due to its use of per-CPU data structures, *local atomic operations* as main buffer synchronization primitive, and `RCU` (*Read-Copy Update*) mechanism to control tracing.

Given that kernel infrastructure used by the tracer could lead to infinite recursion if traced, and typically requires non-atomic synchronization, this paper proposes an asynchronous mechanism to inform the kernel that a buffer is ready to read. This ensures that tracing sites do not require any kernel primitive, and therefore protects from infinite recursion.

This paper presents the core of `LTTng`'s buffering algorithms and measures its performance.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids—*Device driver tracing*

; C.4 [**Computer Systems Organization**]: PERFORMANCE OF SYSTEMS—*Measurement technique*

; D.4.7 [**OPERATING SYSTEMS**]: Organization and Design—*Real-time systems embedded systems, Distributed system*

; D.4.8 [**OPERATING SYSTEMS**]: Performance—*Measurements, Monitors, Operational analysis*

General Terms

Performance, Measurement, Algorithms, Design, Experimentation

Keywords

kernel, tracing, lockless, atomic, modular arithmetic, Linux, LTTng

1. INTRODUCTION

Performance monitoring of multiprocessor high-performance computers, deployed as production systems (e.g. Google platform), requires tools reporting detailed system execution. This provides better understanding of complex multi-threaded and multi-processes application interactions with the kernel.

Tracing the most important kernel events has been done for decades in the embedded field to reveal useful information about program behavior and performance. The main distinctive aspect of multiprocessor system tracing is the complexity added by time-synchronization across cores. Additionally, tracing of interactions between processes and the kernel generates a high volume of information. Further challenges emerge from the requirements for a tracing facility appropriate for use on live production servers. These factors steer design choices towards low-impact monitoring through data extraction, leaving out debugger-like approaches that change the system behavior, e.g., stopping processes or changing the process memory.

Allowing wide instrumentation coverage of the kernel code is especially tricky, given the concurrency of multiple execution contexts and multiple processors. In addition to being able to trace a large portion of the executable code, another key element expected from a kernel tracer is low-overhead and low-disturbance of the normal system behavior. Ideally, a problematic workload should be repeatable both under normal conditions and under tracing, without suffering from the observer effect caused by the tracer. The `LTTng` [Desnoyers and Dagenais 2006] tracer (available at: `http://www.lttng.org`) has been developed with these two main goals in mind: provide good instrumentation coverage and minimize observer effect. The work on `LTTng` started back in 2005.

A state of the art review is first presented in Section 2. It shows how the various tracer requirements influence their design and core synchronization primitive choices, leading them to differ in many aspects from LTTng. The K42 tracer will be studied in detail, given the significant contribution of this research-oriented operating system. This paper will discuss some limitations present in the K42 lockless algorithm, which motivates the need for a new buffer management model. The design of the LTTng tracer will be presented in Section 3. The equations and algorithms required to manage the buffers, ensuring complete atomicity of the probe, will then be detailed in Section 4. The scalability of the approach is discussed, explaining the motivation behind the choice of per-CPU data structures to provide good processor cache locality. Finally, in Section 5 performance tests show how the tracer performs under various workloads at the macro-benchmark and micro-benchmark levels.

2. STATE OF THE ART

In this section, we first present a review of the tracing requirements from the target LTTng user-base. This is a summary of field work done to identify those requirements from real-world Linux users. Then, we present state-of-the-art open source tracers. For each of these, their target usage scenarios are presented along with the requirements imposed. Finally, we study in more depth the tracer in K42, which is the closest to LTTng requirements, explaining where LTTng brings new contributions.

Previous work published in 2007 at the Linux Symposium [Bligh et al. 2007] and Europar [Wisniewski et al. 2007] presented the user-requirements for kernel tracing that are driving the LTTng effort. They explain how tracing is expected to be used by Linux end-users, developers, technical support providers and system administrators. The following list summarizes this information and lists which Linux distributions integrate LTTng:

- Large online service companies such as Google need a tool to monitor their production servers and to help them solve hard to reproduce problems. Google had success with such tracing approaches to fix rarely occurring disk delay issues and virtual memory related issues. They need the tracer to have a minimal performance footprint.
- IBM Research looked into debugging of commercial scale-out applications, which are increasingly used to split large server workloads. They used LTTng successfully to solve a distributed filesystem-related issue.
- Autodesk, in the development of their next-generation Linux audio/video edition applications, used LTTng extensively to solve soft real-time issues.
- Wind River includes LTTng in their Linux distribution so their clients, already familiar with Wind River VxWorks tracing solutions, can benefit from the same kind of features they have relied on for a long time.
- Montavista has integrated LTTng in their Carrier Grade Linux Edition 5.0 for the same reasons.
- SuSE is currently integrating LTTng in their SLES real-time distribution, because their clients, asking for solutions supporting a real-time kernel, need such tools to debug their problems.
- A project between Ericsson, Defence R&D Canada, NSERC and various universities aims at monitoring and debugging multi-core systems, providing tools to automate system behavior analysis.
- Siemens has been using LTTng internally for quite some time [Hillier 2008].
- Sony, Freescale Semiconductors, Nokia, Mentor Graphics, and others are using or distributing LTTng as well.

In theory, a number of lock-free data structures (e.g. queues) could be used for buffering trace data. However, the volume of tracing data, and the importance of maintaining a minimal overhead, rules out these generic solutions. Tracing requires event data insertion atomicity but removal is performed asynchronously in bulk. Preallocation of buffer memory with a bounded capacity is preferred in tracing over dynamic allocation and deallocation at each insertion and removal. Furthermore, tracing brings a number of specific requirements such as working even in NMI context and discarding events on buffer full condition rather than blocking.

We now look at existing tracing solutions for which detailed design and implementation documentation is publicly available. This study focuses on tracers available under an open-source license, given that closed-source tracers do not provide such detailed documentation. The requirements fulfilled by each tracer as well as their design choices are exposed. Areas in which LTTng requirements differ from these tracers are outlined.

DTrace [Cantrill et al. 2004], first made available in 2003 and formally released as part of Sun's Solaris 10 in 2005, aims at providing information to users about the way their operating system and applications behave by executing scripts performing specialized analysis. It also provides the infrastructure to collect the event trace into memory buffers, but aims at moderate event production rates. It disables interrupts to protect the tracer from concurrent execution contexts on the same processor, and uses a sequence lock to protect the clock source from concurrent modifications. Therefore, while very flexible, DTrace cannot provide the same coverage (e.g., applicable in NMI context) or the same low overhead (lockless tracing) as LTTng.

SystemTAP [Prasad et al. 2005] provides scriptable probes which can be connected on top of Linux kernel statically defined tracepoints (Markers or Tracepoints) or dynamically inserted (Kprobes [Mavinakayanahalli et al. 2006]) tracepoints. It is designed to provide a safe language to express the scripts to run at the instrumentation site, but does not aim at optimizing probe performance for high data volume. Indeed, it was originally designed to gather information exclusively from Kprobes breakpoints and therefore expects the user to carefully filter out the unneeded information to diminish the probe effect. It disables interrupts and takes a busy-spinning lock to synchronize concurrent tracing site execution. The LKET project (Linux Kernel Event Tracer) re-used the SystemTAP infrastructure to trace events, but reached limited performance results given the fact that it shared much of SystemTAP's heavy synchronization. SystemTap is very similar to DTrace in several respects, and suffers from the same limitations as compared to LTTng. One key difference is that the probe scripts are compiled into more efficient native code in SystemTap, instead of bytecode in DTrace.

Ftrace, started in 2009 by Ingo Molnar, aims primarily at kernel tracing suited for kernel developer's needs. It is mostly based on specialized trace analysis modules run in kernel-space to generate either a trace or analysis output, available to the user in text format. It also integrates binary buffer data extraction to provide efficient data output. Until recently, it was based on per-cpu busy-spinning locks, and interrupt disabling, to protect the tracer against concurrent execution contexts. It now implements a lockless buffering scheme similar to that already used by LTTng.

The K42 [Krieger et al. 2006] project is a research operating system developed mostly between 1999 and 2006 by IBM Research. It targeted primarily large multiprocessor machines with high scalability and performance requirements. It contained a built-in tracer named "trace", which was an element integrated to the kernel design. The scale of systems targeted by K42, as well as their use of lockless buffering algorithms with atomic operations, make its an inevitable part of the state of the art that needs to be taken into consideration before introducing LTTng.

From a design point of view, a major difference between this research-oriented tracer and LTTng is that the latter aims at deployments on multi-user Linux systems, where security is a concern. Therefore, simply sharing a per-CPU buffer, available both for reading and writing by the kernel and any user process, would not be acceptable on production systems. Also, in terms of synchronization, K42's tracer implementation ties trace extraction daemon threads to the processor on which the information is collected. Although this removes needs for synchronization, it also implies that a relatively idle processor cannot contribute to the overall tracing effort when some processors are busier. Regarding CPU hotplug support, which is present in Linux, an approach where the only threads able to extract the buffer data would be tied to the local processor would not allow trace extraction in the event a processor would go offline. Adding support for cross-CPU data reader support would involve adding the proper memory barriers to the tracer.

Then, more importantly for the focus of this paper, studying in depth the lockless atomic buffering scheme found in K42 indicates the presence of a race condition where data corruption is possible. It must be pointed out that, given the fact that the K42 tracer uses large buffers compared to the typical event size, this race is unlikely to happen, but could become more frequent if the buffer size is made smaller or larger events were written, which LTTng tracer's flexibility permits.

The K42 tracer [Wisniewski and Rosenburg 2003] divides the memory reserved for tracing a particular CPU into buffers (which are called "K42 buffers" throughout this paper). This maps to the sub-buffer concept which will be presented in the LTTng design 3. In comparison, LTTng uses the term "buffer" to identify the set of sub-buffers which are part of the circular buffer. The K42 scheme uses a lockless buffer-space management algorithm based on reserve-commit semantic. Space is first reserved atomically in the K42 buffer, then both data write and the commit counter update are performed out-of-order with respect to local interrupts. It uses a *buffersProduced* count, which counts the number of K42 buffers produced by the tracer, a *buffersConsumed* counter, tracking the number of K42 buffers read, and a per-buffer *bufferCount*, counting the amount of data committed into each K42 buffer.

In the K42 scheme, the *buffersProduced* counter is incremented upon buffer space *reservation* for an event crossing a K42 buffer boundary. If other out-of-order writes are causing the current and previous K42 buffer's commit counters to be a multiple of buffer size (because they would still be fully uncommitted), the user-space data consumption thread can read non-committed (invalid) data because the *buffersProduced* would make an uncommitted K42 buffer appear as fully committed. This is a basic algorithmic flaw that LTTng fixes by using a free-running per sub-buffer *commit count* and by introducing a new *buffer full* criterion which depends on the difference between the *write count* (global to the whole buffer) and its associated per-sub-buffer *commit count*, as detailed in Equation (1) in Section 4.2. The algorithmic flaws identified here in the K42 buffering mechanism, which differ from the minor limitations mentioned in [Wisniewski and Rosenburg 2003], require a complete re-design the lock-less counter scheme.

The formal verification performed by modeling LTTng algorithms, and using the Spin model-checker [Desnoyers 2009], increases the level of confidence that such corner-cases are correctly handled.

3. DESIGN OF LTTNG

Tracing an operating system kernel brings interesting problems related to the *observer effect*. In fact, tracing performed at the software level requires modifying the execution flow of the traced system and therefore modifies its behavior and performance. Each execution context concerned must be taken into account in order to decide what code is allowed to be executed when the instrumentation is reached.

This section describes how LTTng is designed to deal with kernel tracing, satisfying the constraints associated with *synchronization* of data structures while running in any *execution context*, avoiding *kernel recursion* and inducing a very small *performance impact*. It details a complete buffer synchronization scheme.

This section starts with a high-level overview of the tracer design. It is followed by a more detailed presentation of the *Channel* component, an highly-efficient data transport pipe. The *Data Flow* presentation, as seen from the tracing probe perspective, is then exposed. This leads to the synchronization of trace *Control* data structures, allowing tracing configuration. Finally, the *Atomic Buffering Scheme* section details the core of LTTng concurrency management, which brings innovative algorithms to deal with write concurrency in circular memory buffers.

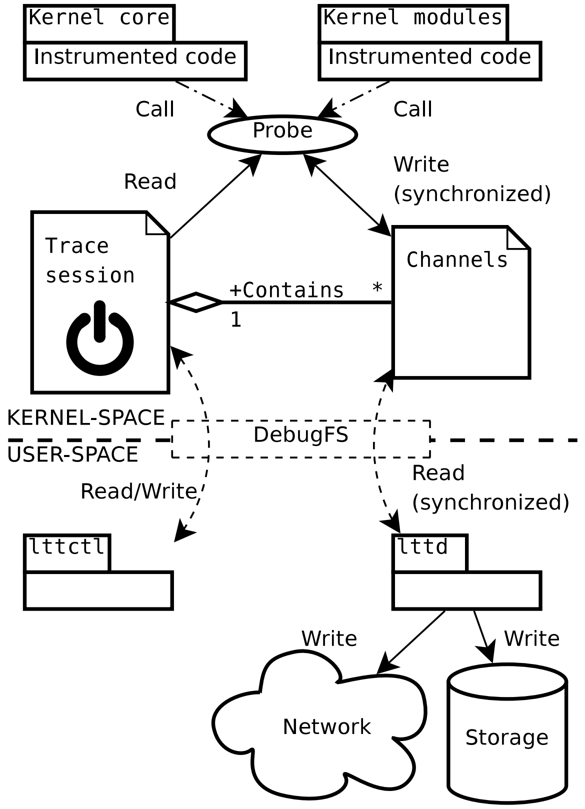

Figure 1: Tracer components overview

3.1 Components overview

Starting with a high-level perspective on the tracer design, Figure 1 presents the component interactions across kernel-space to user-space boundary.

Kernel core and kernel modules are *instrumented* either statically, at the source-code level with the Linux Kernel Markers and Tracepoints, or dynamically with Kprobes. Each instrumentation site identifies kernel and module code which must be traced upon execution. Both static and dynamic instrumentation can be activated at runtime on a per-site basis to individually enable each event type. An event is semantically tied to a set of functionally equivalent instrumentation sites.

When an instrumented code site is executed, the `LTTng` *probe* is called if the instrumentation site is activated. The probe reads the *trace session* status and writes an event to the *channels*.

Trace sessions contain the tracing configuration data and pointers to multiple *channels*. Although only one session is represented in Figure 1, there can be many `trace sessions` concurrently active, each with its own trace configuration and set of `channels`. Configuration data determines if the trace session is active or not and which event filters should be applied.

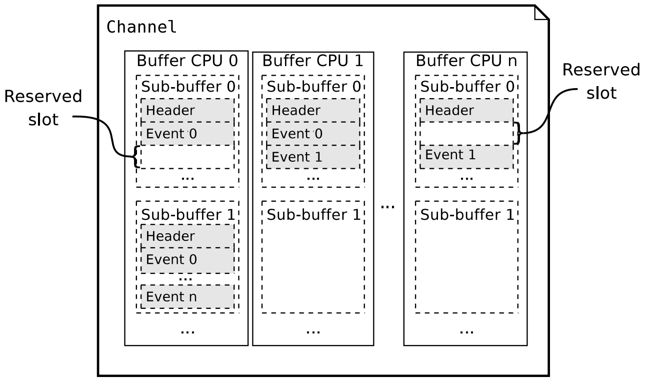

Figure 2: Channel components

From a high-level perspective, a channel can be seen as an information pipe with specific characteristics configured at trace session creation time. Various configuration options can be specified on a per-channel basis, such as: buffer size, tracing mode, which specify how buffer full situations should be handled, and buffer flush period. These options will be detailed in Section 3.2.

`DebugFS` is a virtual filesystem providing an interface to control kernel debugging and export data from kernel-space to user-space. The trace session and channel data structures are organised as `DebugFS` files to let `lttctl` and `lttd` interact with them.

The user-space program `lttctl` is a command-line interface interacting with the `DebugFS` file system to control kernel tracing. It configures the trace session before tracing starts and is responsible for starting and stopping trace sessions.

The user-space daemon `lttd` also interacts with `DebugFS` to extract the channels to disk or network storage. This daemon is only responsible for data extraction; this daemon has absolutely no direct interaction with trace sessions.

3.2 Channels

After the high-level tracer presentation, let's focus on the *Channel* components. They are presented in Figure 2.

A channel is a pipe between an information producer and consumer (producer and writer as well as consumer and reader will be respectively used as synonyms through this paper). It serves as a buffer to move data efficiently. It consists of one buffer per CPU to ensure cache locality and eliminate false-sharing. Each buffer is made of many sub-buffers where *slots* are reserved sequentially. This way, while one sub-buffer is being filled, one or more other sub-buffers are full and queued to be written to disk, or are free and available to be filled. Each sub-buffer is exported by the `lttd` daemon to disk or through network.

A *slot* is a sub-buffer region reserved for exclusive write access by a probe. This space is reserved to write either a *sub-buffer header* or an *event header and payload.* Figure 2 shows space being reserved. On CPU 0, space is reserved in sub-buffer 0 following event 0. In this buffer, the *header* and *event 0* elements have been completely written to the buffer. The grey area represents *slots* for which associated *commit count* increment has been done. Committing a reserved *slot* makes it available for reading. On CPU n, a slot is reserved in sub-buffer 0 but is still uncommitted. It is however followed by a committed event. This is possible due to the non-serial nature of event write and commit operations. This situation happens when an event must be written by an interrupt handler nested between space reservation and *commit count* update of another event. Sub-buffer 1, belonging to CPU 0, shows a fully committed sub-buffer ready for reading. The consumer is allowed to read from *slots* only after the sub-buffer containing them has been filled, which ensures no holes are encountered.

Events written in a reserved *slot* are made of a *header* and a variable-sized *payload*. The *header* contains the time stamp associated with the event and the event type (an integer identifier). The event type information allows parsing the *payload* and determining its size. The maximum *slot* size is bounded by the sub-buffer size. A data producer is performing what we define as *sub-buffer switch* when it stops using a sub-buffer for reserving space and passes to the following sub-buffer.

Channels are configured in a tracing mode that specify how buffer full conditions should be handled. *Flight recorder* tracing overwrites the oldest buffer data when a buffer is full. Conversely, *discard* tracing discards (and counts) events when a buffer is full. Those discarded events are counted to evaluate tracing accuracy. These counters are recorded in each sub-buffer header to allow identifying which trace region suffered from event loss. The former mode is made to capture a snapshot of the system preceding execution at a given point. The latter is made to collect the entire execution trace over a period of time.

3.3 Probe Data Flow

The tracing data flow from the probe perspective is illustrated in Figure 3. This figure includes all data sources and sinks, including those which are not part of the tracer per se, such as kernel data structures and hardware time stamps.

A probe takes event data from registers, the stack, or from memory every time the instrumented kernel execution site is reached. A time stamp is then associated with this information to form an event, identified by an event type ID. The tracing control information is read to know which channel is concerned by the information. Finally, the resulting event is serialized and written to a circular buffer to be later exported outside of kernel-space. The channels follow a producer-consumer semantic.

Instrumentation can be inserted either statically, at the source-code level, or dynamically, using a breakpoint. The former allows building instrumentation into the software and therefore identify key instrumentation sites, maintaining a stable API. It can also restrain the compiler from optimizing away variables needed at the instrumented site. However, in order to benefit from flexible live instrumentation insertion, without recompilation and reboot, it might be adequate to pay the performance cost associated with a breakpoint, but one must accept that the local variables might be optimized away and that the kernel debug information must be kept around.

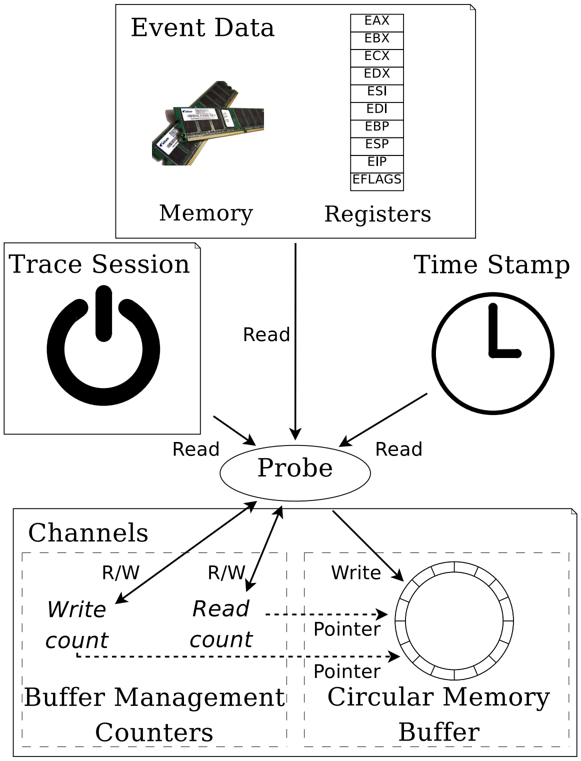

Figure 3: Probe data flow

Source-code level instrumentation, enabled at runtime, is currently provided by *Tracepoints* [Corbet 2008] and *Linux Kernel Markers* [Corbet 2007a], developed as part of the `LTTng` project and merged into the mainline Linux kernel. Dynamic instrumentation, based on breakpoints, is provided in the Linux kernel by *Kprobes* [Mavinakayanahalli et al. 2006] for many architectures. `LTTng`, `SystemTAP` and `DTrace` all use a combination of dynamic and static instrumentation. The details about the different instrumentation mechanisms are not, however, the focus of this paper.

3.4 Control

This section presents interactions with the *trace session* data structure depicted in Figure 1 along with the required synchronization.

Tracing control under `LTTng` is organized around the concept of "tracing session". This defines the configuration of a set of buffers and the subset of the system events that needs to be collected into each channel. Multiple tracing sessions can be active in parallel, each with different buffer configurations and collecting different system events.

Tracing control operates on a list of tracing sessions. Available actions include creating a new trace session, starting or stopping tracing, and freeing a trace session. Upon new trace session creation, parameters must be set such as channel's buffer size, number of sub-buffers per buffer, and tracing mode (*flight recorder* or *discard*). This allows end-users to tune the tracer configuration appropriately for their workload.

Event collection can be enabled/disabled on a per-tracing-session basis. It is useful to start/stop collection of a set of events atomically. Events to be collected into each channel are selected by hooking callbacks onto the instrumentation sites (this selection is therefore performed in multiple memory accesses). The single per-session flag starts/stops tracing at once with a single memory access.

The linked list containing tracing session nodes is protected against concurrent modifications with a global mutex and synchronized with respect to read-side traversal using RCU (Read-Copy Update) [McKenney 2004, Desnoyers et al. 2012]. Two types of data structure modifications can be done: configuration elements within a session can be updated atomically, in which case it is safe to perform the modification without copying the complete trace session data structure as long as the mutex is held. Non-atomic updates must be done on a copy of the trace session structure, followed by a replacement of the old copy in the list by two successive pointer changes in this precise order: first setting the pointer to next element within the new copy and then setting the pointer to the new copy in the previous element. Then the update must wait for quiescent state, which allows memory reclamation of the old data structure. This ensures no active data structure readers, the probes, still hold a reference to the old tracing session structure when it is freed.

Although some tracing session configuration options are allowed to be modified while tracing is active, modification of buffer structures configuration per se is only allowed upon new trace session creation, before tracing is started, and upon deletion, after tracing has been stopped and no consumers are using the buffers anymore. This ensures only the data producers and consumers will touch the buffer management structures, thus keeping complexity of interaction between producer and consumer relatively low.

In order to provide the ability to export tracing information as a live stream, we need to provide an upper bound on the maximum latency between the moment the event is written to the memory buffers and the moment it is ready to be read by the consumer. However, because the information is only made available for reading after a sub-buffer has been filled, a low event rate channel might never be ready for reading until the final buffer flush is done when tracing is stopped.

To get around this problem, LTTng implements a per-CPU sub-buffer flush function which can be executed concurrently with tracing. It shares many similarities with tracing an event. However, it won't flush an empty sub-buffer because there is no information to send and it does not reserve space in the buffer. The only supplementary step required to stream the information is to call the buffer flush for each channel periodically in a per-CPU timer interrupt.

The following section presents channel ring-buffer synchronization.

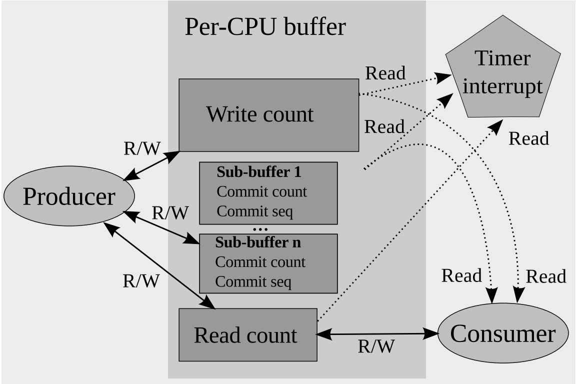

Figure 4: Producer-consumer synchronization

4. ATOMIC BUFFERING SCHEME

The atomic buffering scheme implemented in LTTng allows the probe to produce data in circular buffers with a buffer-space reservation mechanism which ensures correct reentrancy with respect to asynchronous event sources. These include maskable and non-maskable interrupts (NMIs). Preemption[1] is temporarily disabled around the tracing site to make sure no thread migration to a different CPU can occur in the middle of probe execution[2].

Section 4.1 first presents the data structures used to synchronize the buffering scheme. Then, algorithms performing interactions between producer and consumer are discussed respectively in sections 4.2, 4.3, 4.3.1, 4.3.2 and 4.3.3.

4.1 Atomic data structures

Figure 4 shows the per-CPU data structures keeping track of the number of bytes reserved (*write count*) by the producer and consumed (*read count*) by the consumer for a buffer. An array of *commit counts* and *commit seq* counters[3] is needed to know when data written by the producer is ready to be consumed. The *commit count* and *commit seq* counters keep track of the number of bytes committed in a sub-buffer.

On SMP (*Symmetric Multiprocessing*) systems, some instructions are designed to update data structures in one single indivisible step. Those are called *atomic operations*. If atomicity is only needed in the context of a single processor, for instance to protect per-CPU data structure updates against preemption, less expensive *local atomic operations* may be available.

[1] With fully-preemptible Linux kernels (*CONFIG_PREEMPT=y*), the scheduler can preempt threads running in kernel context to run another thread.

[2] This requirement on disabling thread migration is only necessary if atomic operations are not SMP-aware. User-level tracing, where migration cannot be disabled without significant impact on execution, can be performed with the algorithms presented here without disabling preemption by using SMP-aware atomic operations

[3] The size of this array is the number of sub-buffers.

To properly implement the semantic of SMP-safe atomic operations, memory barriers are required on some architecture (this is the case for PowerPC and ARMv7 for instance). For the x86 architecture family, these memory barriers are implicit. However, a special lock prefix is required before these instructions to synchronize multiprocessor accesses on x86. Use of per-CPU data allows us to diminish performance overhead of the tracer fast-path, as we can remove memory barriers and use non bus-locking atomic operations. These *local* atomic operations, only synchronized with respect to the local processor, have a lower overhead than those synchronized across cores. Those are the only instructions we use to modify the per-CPU counters, which ensures reentrancy with `NMI` context.

The main restriction that must be observed when using such operations is to disable preemption between reading the current CPU number and access to these variables, thus ensuring no thread is migrated from one core to another between the moment the reference is read and the atomic access. This ensures no remote core accesses the variable with SMP-unsafe operations.

The three atomic instructions used are the `CAS` (*Compare-And-Swap*), a local `Compare-And-Swap` (thereafter referred to as `LCAS`), and a local atomic increment. The `LCAS` instruction is used on the *write count* to update the counter of reserved buffer space. This operation ensures space reservation is done atomically with respect to other execution contexts running on the same CPU. The atomic add instruction is used to increment the per sub-buffer *commit count*, which identifies how much information has actually been written in each sub-buffer. The *commit count* counters are updated with a lightweight local increment instruction. The *commit seq* counters are updated with a concurrency-aware `CAS` each time a sub-buffer is filled. The consumer is only allowed to read from a sub-buffer after its *write count* and *commit seq* counter shows that it has been filled, thus ensuring that no holes are present.

The sub-buffer size and the number of sub-buffers within a buffer are limited to powers of 2 for two reasons. First, using bitwise operations to access the sub-buffer offset and sub-buffer index is faster than the modulo and division. The second reason is more subtle: although the `LCAS` operation could detect 32 or 64-bits overflows and deal with them correctly before they happen by resetting to 0, the *commit count* atomic add will eventually overflow the 32 or 64-bits counters, which adds an inherent power of 2 modulo that would be problematic, would the sub-buffer size not be a power of 2.

On the reader side, the *read count* is updated using a standard SMP-aware `CAS` operation. This is required because the reader thread can read sub-buffers from buffers belonging to a remote CPU. It is designed to ensure that a traced workload executed on a very busy CPU can be extracted by other CPUs which have more idle time. Having the reader on a remote CPU requires SMP-aware `CAS`. This allows the writer to push the reader position when the buffer is configured in *flight recorder* mode. The performance cost of the SMP-aware operation is not critical because updating the *read count* is only done once a whole sub-buffer has been read by the consumer, or when the writer needs to push the reader at sub-buffer switch, when a buffer is configured in *flight recorder* mode. Concurrency between many reader threads is managed by using a reference count on file open/release, which only lets a single process open the file, and by requiring that the user-space application reads the sub-buffers from only one execution thread at a time. Mutual exclusion of many reader threads is left to the user-space caller, because it must encompass a sequence of multiple system calls. As a matter of fact, holding a kernel mutex is not allowed when returning to user-space.

4.2 Equations

This section presents equations determining buffer state. These are used by algorithms presented in Section 4.3.

These equations extensively use modulo arithmetic to consider physical counter overflows. On 64-bits architectures, equations are in modulo 2^{64}. On 32-bits architectures, they are modulo 2^{32}.

We first define the following basic operations. Let's define

- $|x|$ as length of x.
- $a \bmod b$ as modulo operation (remainder of $\frac{a}{b}$).
- $\overset{n}{\underset{m}{\mathcal{M}}}(x)$ as x bitwise AND $00\ldots0\underbrace{11\ldots1}_{n-m}\underbrace{00\ldots0}_{m}$, formally: $(x \bmod 2^n) - (x \bmod 2^m)$.

We define the following constants. Let

- $|\texttt{sbuf}|$ be the size of a sub-buffer.
(power of 2)
- $|\texttt{buf}|$ be the size of a buffer.
(power of 2)
- $sbfbits = \lg_2(|\texttt{sbuf}|)$.
- $bfbits = \lg_2(|\texttt{buf}|)$.
- $nsbbits = bfbits - sbfbits$.
- $wbits$ be the architecture word size in bits.
(32 or 64 bits)

We have the following variables. Let

- $wcnt$ be write counter mod 2^{wbits}.
- $rcnt$ be read counter mod 2^{wbits}.
- $wcommit$ be the commit counter *commit seq* mod 2^{wbits} belonging to the sub-buffer pointed to by $wcnt$.
- $rcommit$ be the commit counter *commit seq* mod 2^{wbits} belonging to the sub-buffer pointed to by $rcnt$.

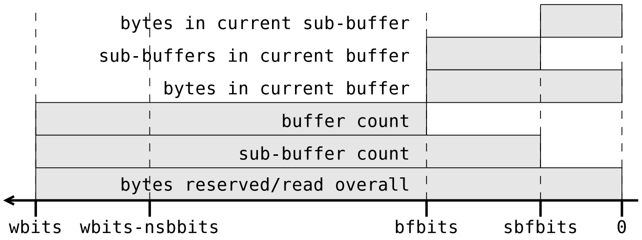
bytes in current sub-buffer
sub-buffers in current buffer
bytes in current buffer
buffer count
sub-buffer count
bytes reserved/read overall
wbits
wbits-nsbbits
bfbits
sbfbits
0

Figure 5: Write and read counter masks

Less than one complete sub-buffer is available for writing when Equation (1) (*Buffer Full*) is satisfied. It verifies that the difference between the number of sub-buffers produced (multiplied by |`sbuf`|) and the number of sub-buffers consumed (multiplied by |`sbuf`|) in the ring buffer is greater or equal to the number of sub-buffers per buffer (also multiplied by |`sbuf`|). The implication of this equation being satisfied at buffer switch is that the buffer is full. Both side of this equation are treated as-is, without shifting to divide by |`sbuf`|, to naturally apply the *wbits* overflow to the subtraction operation.

$$\mathop{\mathcal{M}}\limits_{sbfbits}^{wbits}(wcnt) - \mathop{\mathcal{M}}\limits_{sbfbits}^{wbits}(rcnt) \geq |\texttt{buf}| \tag{1}$$

Write counter and read counter masks are illustrated in Figure 5. These masks are applied to *wcnt* and *rcnt*.

A buffer contains at least one sub-buffer ready to read when Equation (2) (*Sub-buffer Ready*) is satisfied. The left side of this equation takes the number of buffers consumed so far (multiplied by |`buf`|), masks out the current buffer offset and divides the result by the number of sub-buffers per buffer. This division ensures the left side of the equation represents the number of sub-buffers reserved. The right side of this equation takes the *commit count* to which *rcnt* points and subtracts |`sbuf`| from it. It is masked to clear the top bits, which ensures both sides of the equation overflow at the same value. This is required because *rcnt* reaches a 2^{wbits} overflow *sbfnb* times more often than the per-sub-buffer *rcommit* counters. |`sbuf`| is subtracted from *rcommit* because we need to know when the *commit seq* is one whole sub-buffer ahead of the read count.

$$\frac{\mathop{\mathcal{M}}\limits_{bfbits}^{wbits}(rcnt)}{2^{nsbbits}} = \mathop{\mathcal{M}}\limits_{0}^{wbits-nsbbits}(rcommit - |\texttt{sbuf}|) \tag{2}$$

The sub-buffer corresponding to *wcnt* is in a fully committed state when Equation (3) (*Fully Committed*) is satisfied. Its negation is used to detect a situation where an amount of data sufficient to overflow the buffer is written by concurrent execution contexts running between a reserve-commit pair.

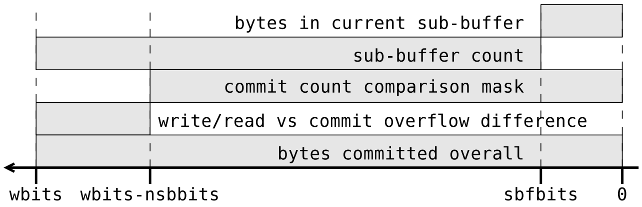
bytes in current sub-buffer
sub-buffer count
commit count comparison mask
write/read vs commit overflow difference
bytes committed overall
wbits
wbits-nsbbits
sbfbits
0

Figure 6: Commit counter masks

$$\frac{\mathop{\mathcal{M}}\limits_{bfbits}^{wbits}(wcnt)}{2^{nsbbits}} = \mathop{\mathcal{M}}\limits_{0}^{wbits-nsbbits}(wcommit) \tag{3}$$

Commit counter masks are illustrated in Figure 6. These masks are applied to *rcommit* and *wcommit*.

The sub-buffer corresponding to *rcnt* is being written when Equation (4) (*Sub-buffer Written*) is satisfied. It verifies that the number of sub-buffers produced and consumed are equal.

$$\mathop{\mathcal{M}}\limits_{sbfbits}^{wbits}(wcnt) = \mathop{\mathcal{M}}\limits_{sbfbits}^{wbits}(rcnt) \tag{4}$$

4.3 Algorithms

Algorithms used to synchronize the producer and consumer are presented in this section. It is followed by a presentation of the asynchronous buffer delivery algorithm. Algorithms presented in this section refer to equations presented in Section 4.2.

4.3.1 Producer

This section presents the algorithms used by the information producer, the *probe*, to synchronize its *slot* reservation within the *channels*.

The overall call-graph presented in this section can be summarized as follow. When an event is to be written, space is reserved by calling ReserveSlot(), which calls TryReserveSlot() in a loop until it succeeds. Then, PushReader(), SwitchOldSubbuf(), SwitchNewSubbuf() and EndSwitchCurrent() (not expanded in this paper for brevity) are executed out-of-order to deal with sub-buffer switch. After the event data is written to the slot, CommitSlot() is called to increment the commit counter.

The *write count* and *read count* variables have the largest size accessible atomically by the architecture, typically 32 or 64 bits. Since, by design, the sub-buffer size and the number of sub-buffers within a buffer are powers of two, a LSB (Least Significant Bit) mask can be used on those counters to extract the offset within the buffer. The MSBs (Most Significant Bits) are used detecting the improbable occurrence of a complete buffer wrap-around nested on top of the `LCAS` loop in *flight recorder* mode. Such overflow, if undetected, could make time-stamps appear to go backward in a buffer when moving forward between two physically contiguous events.

Algorithm 1 TryReserveSlot(*payload_size*)

```
Require: An integer $payload\_size \geq 0$.
Require: Disabled preemption, ensuring event lost count
    LCAS apply on local CPU counters.
Read $write\_count$
Read $time\_stamp\_counter$
Calculate required $slot\_size$
Calculate $slot\_offset$
if $slot\_offset$ is at beginning of sub-buffer then
    if Not Fully Committed (would overflow) then
        Increment event lost count
        $slot\_size = FAIL$
        return $slot\_size$
    end if
    if Buffer Full && (in discard mode) then
        Increment event lost count
        $slot\_size = FAIL$
        return $slot\_size$
    end if
end if
Update $buffer\_switch\_flags$
return < $slot\_size$, $slot\_offset$, $buffer\_switch\_flags$,
    $write\_count$, $time\_stamp\_counter$ >
```

Algorithm 2 ReserveSlot(*payload_size*)

```
Require: An integer $payload\_size \geq 0$
Require: Disabled preemption, ensuring $write\_count$ LCAS
    apply on local CPU counters.
Ensure: $slot\_offset$ is the only reference to the slot during
    all the reserve and commit process, the slot is reserved
    atomically, time stamps of physically consecutive slots
    are always incrementing.
repeat
    <$slot\_size$, $slot\_offset$, $buffer\_switch\_flags$,
    $time\_stamp\_counter$>
    = TryReserveSlot($payload\_size$)
    if $slot\_size = FAIL$ then
        return FAIL
    end if
until LCAS of $write\_count$ succeeds
PushReader($buffer\_switch\_flags$)
Set reference flag in pointer to current sub-
    buffer. Indicates that the writer is using
    this sub-buffer.
SwitchOldSubbuf($buffer\_switch\_flags$)
SwitchNewSubbuf($buffer\_switch\_flags$)
EndSwitchCurrent($buffer\_switch\_flags$)
return <$slot\_size$, $slot\_offset$, $time\_stamp\_counter$>
```

Such wrap-around could happen if many interrupts nest back-to-back on top of a LCAS loop. A worse-case scenario would be to have back-to-back nested interrupts generating enough data to fill the buffer (typically 2 MiB in size) and bring the write count back to the same offset in the buffer. The LCAS loop uses the most significant counter bits to detect this situation. On 32-bits architectures, it permits to detect counter overflow up to 4 GiB worth of buffer data. On 64-bits architectures, it detects up to 16.8 million TiB worth of data written while nested over a LCAS loop execution. Given that this amount of trace data would have to be generated by interrupt handlers continuously interrupting the probe, we would consider an operating system facing such interrupt rate to be unusable. As an example of existing code doing similar assumptions, the Linux kernel sequence lock, used to synchronize the time-base, is made of a sequence counter also subject to overflow.

Slot reservation, presented in TryReserveSlot() and ReserveSlot() is performed as follow. From a high-level perspective, the producer depends on the *read count* and *write count* difference to know if space is still available in the buffers. If no space is available in *discard* mode, the event lost count is incremented and the event is discarded. In *flight recorder* mode, the next sub-buffer is overwritten by *pushing* the reader. Variables *write count*, *read count* and the *commit seq* array are used to keep track of the respective position of the writer and the reader gracefully with respect to counter overflow. Equations (1), (2), (3) and (4) are used to verify the state of the buffer.

The *write count* is updated atomically by the producer to reserve space in the sub-buffer. Slots need to be written with a time-stamp to allow a posteriori reordering of events across buffers. Those time-stamps must be ordered according to the order the slots were allocated to lessen the complexity of the (a posteriori) reordering operation: it allows reordering events across buffers with a single-pass merge. This can be achieved by assigning time-stamps to each successful LCAS. In order to apply monotonically increasing time stamps to events which are physically consecutive in the buffer, the time stamp is read within the LCAS loop. This ensures that no space reservation succeeds between the time-stamp register read and the atomic space reservation, and therefore ensures that a successful buffer-space reservation and time-stamp read are indivisible from one another from a CPU's perspective. Such mechanisms to make many instructions appear to execute atomically is however limited to operations not having side-effects outside of the variables located on the stack or in registers which can be re-executed upon failure, except for the single LCAS operation which has side-effects when it succeeds. It is therefore mostly limited to read operations and the computation of the required *slot* size for the event.

Keeping accurate time-stamping information with each event allows following a thread execution across migration between processors. Since thread migration typically takes an order of magnitude longer than the maximum combined error due to CPU cycle-counter precision and memory accesses reordering, event causality within a traced thread will never be reversed. For instance, if function_A() is always executed before function_B() in a given thread, they will never appear to execute in reverse order due to thread migration between processors. However, causality between threads is not guaranteed, as their execution would need to be modified to provide the level of granularity required to keep track of synchronization performed through shared memory.

Once space is reserved, the remaining operations are done out-of-order. This means that if an interrupt nests over a probe, it will reserve a buffer slot next to the one being written to by the interrupted thread, will write its event data

Algorithm 3 COMMITSLOT(*slot_size, slot_offset*)

```
Require: An integer $slot\_size > 0$ and the $slot\_offset$
Require: Disabled preemption, ensuring commit_count local_add() apply on local CPU counters.
Compiler barrier[4]
Issue local_add() to increment commit_count of slot_size
if Fully Committed then
    commit_seq_old = commit_seq
    while commit_seq_old < commit_count do
        try CAS of commit_seq. Expect commit_seq_old, new value written is commit_count. Save value read to commit_seq_old.
    end while
end if
```

Algorithm 4 FORCESWITCH()

```
Ensure: Buffer switch is done if sub-buffer contains data
Require: Disabled preemption, ensuring write_count LCAS apply on local CPU counters.
repeat
    Calculate the commit_count needed to fill the current sub-buffer.
until LCAS of write_count succeeds
PUSHREADER()
Set reference flag in pointer to current sub-buffer. Indicates that the writer is using this sub-buffer.
SWITCHOLDSUBBUF()
SWITCHNEWSUBBUF()
```

in its own reserved slot and will atomically increment the *commit count* before returning to the previous probe stack. When a slot has been completely written to, the COMMITSLOT() algorithm is used to update the *commit count*. It is also responsible for clearing the sub-buffer reference flag if the sub-buffer is filled and updating *commit seq*.

There is one *commit seq* per sub-buffer. It also increments forever in the same way the *write count* does, with the difference that it only counts the per-sub-buffer bytes committed rather than the number of bytes reserved for the whole buffer. The difference between the `write count` MSBs divided by the number of sub-buffers and the *commit seq* MSBs (with the highest bits corresponding to the number of sub-buffers set to zero) indicates if the commit count LSBs represent an empty, partially or completely full sub-buffer.

As shown at the end of RESERVESLOT(), switching between sub-buffers is done out-of-order. It consists of two phases: the first detects, within the `LCAS` loop, if a buffer switch is needed. If this is the case, flags are set on the probe stack to make the out-of-order code, following the loop, increment the sub-buffer *commit counts* of the sub-buffer we are switching out from and the sub-buffer we are switching into. The sub-buffer switched out from will therefore have its *commit count* incremented by the missing amount of bytes between the number of bytes *reserved* (and thus monotonically incrementing) and the sub-buffer size. Switching to a new sub-buffer adds the new sub-buffer header's size to the new sub-buffer's *commit count*. Another case is also possible, namely when there is exactly enough event data to fit perfectly in the sub-buffer. In this case, an *end switch current* flag is raised so the header information is finalized. All these buffer switching cases also populate the sub-buffer headers with information regarding the current time stamp and padding size at the end of the sub-buffer, prior to incrementing the *commit count*. SWITCHOLDSUBBUF(), SWITCHNEWSUBBUF() and ENDSWITCHCURRENT() are therefore responsible for incrementing the *commit count* of the amount of padding added at the end of a sub-buffer, clearing the reference flag when the sub-buffer is filled and updating *commit seq*.

Pushing a reader, represented by PUSHREADER(), is done by a writer in *flight recorder* mode when it detects that the buffer is full. In that case, the writer sets the *read count* to the beginning of the following sub-suffer.

Flushing the buffers while tracing is active, as done by the pseudo-code presented in Algorithm FORCESWITCH(), is required to permit streaming of information with a bounded latency, between the time events are written into the buffers and event delivery to user-space. It is a special-case of normal space reservation which does not reserve space in the sub-buffer, but forces a buffer switch if the current sub-buffer is non-empty. Buffer switch is called from a periodical timer, configurable by the user to select how often buffer data must be flushed.

4.3.2 Consumer

The consumer, `lttd`, uses two system calls, *poll()* and *ioctl()*, to control interaction with memory buffers, and *splice()* as a mean to extract buffers to disk or through network without extra copy. At kernel-level, we specialize those three system calls for the virtual files presented by *DebugFS*. The daemon waits for incoming data using *poll()*. This system call waits to be woken up by the timer interrupt (see the ASYNCWAKEUPREADERSTIMER() pseudo-code in Algorithm 8). Once data is ready, it returns the poll priority to user-space. If the tracer is currently writing in the last available sub-buffer of the buffer, a high priority is returned. Pseudo-code READPOLL() summarizes the actions taken by the *poll()* system call.

Once control has returned to user-space from the *poll()* system call, the daemon takes a user-space mutex on the buffer and uses the *ioctl()* system call to perform buffer locking operations. Its implementation uses the READGETSUBBUF() and READPUTSUBBUF() algorithms. The former operation, detailed in Algorithm 6, reserves a sub-buffer for the reader and returns the *read count*. If the lower-level buffer writing scheme would allow concurrent accesses to the reserved sub-buffer between the reader and the writer, this value could be used to verify, in the READPUTSUBBUF() operation, detailed in Algorithm 7, that the reader has not been pushed by a writer dealing with buffers in *flight recorder* mode. However, as we present below, this precaution is unnecessary because the underlying buffer structure does not allow such concurrency.

[4] The compiler barrier will be promoted to a write memory barrier by an interprocessor interrupt sent by the read-side READGETSUBBUF(), as explained thoroughly in Section 4.4.

Algorithm 5 READPOLL()

```
Ensure: Returns buffer readability state and priority
Wait on read_wait wait queue.
if Sub-buffer Written then
    if Sub-buffer is finalized (freed by the tracer) then
        Hang up.
        return POLLHUP
    else
        No information to read.
        return OK
    end if
else
    if Buffer Full then
        High-priority read.
        return POLLPRI
    else
        Normal read.
        return POLLIN
    end if
end if
```

The specialized *ioctl()* operation is responsible for synchronizing the reader with the writer's buffer-space reservation and commit. It is also responsible for making sure the sub-buffer is made private to the reader, to eliminate any possible race in flight recorder mode. This is achieved by adding a supplementary sub-buffer, owned by the reader. A "sub-buffer table", with pointers to the sub-buffers being used by the writer, allows the reader to change the reference to each sub-buffer atomically. The READGETSUBBUF() algorithm is responsible for atomically exchanging the reference to the sub-buffer about to be read with the sub-buffer currently owned by the reader. If the `CAS` operation fails, the reader does not get access to the buffer for reading.

The reference to the sub-buffer consists of three values packed within a single word, updated atomically: the low-half of the sub-buffer reference keeps the index of the sub-buffer within the sub-buffer page table. The lowest bit of the sub-buffer reference high half encodes whether the sub-buffer is actively referenced by the writer. This ensures that the pointer exchange performed by the reader can never succeed when the writer is actively using the reference to write to a sub-buffer about to be exchanged by the reader. Finally, the rest of the high-half bits count the number of buffers produced, used as a "tag" in flight-recorder mode. This makes sure the consumer will always read the sub-buffers in the order they were produced.

4.3.3 Asynchronous buffer delivery

Because the probe cannot interact directly with the rest of the kernel, it cannot call the scheduler to wake up the consumer. Instead, this ready to read sub-buffer delivery is done asynchronously by a timer interrupt. This interrupt checks if each buffer contains a filled sub-buffer and wakes up the readers waiting in the *read wait* queue associated with each buffer accordingly. This mechanism is detailed in Algorithm 8.

Algorithm 6 READGETSUBBUF()

```
Ensure: Take exclusive reader access to a sub-buffer.
Read read_count.
Read the commit_seq corresponding to the read_count.
Issue a smp_mb() (Memory Barrier on multiprocessor) to
  ensure commit_seq read is globally visible before sending
  the IPI (Interprocessor Interrupt).
Send IPI to target writer CPU (if differs from the local
  reader CPU) to issue a smp_mb(). This ensures that
  data written to the buffer and write count update are
  globally visible before the commit seq write. Wait for
  IPI completion.
Issue a smp_mb() to ensure the write_count and buffer
  data read are not reordered before IPI execution.
Read write_count.
if No Sub-buffer Ready then
    return EAGAIN
end if
if Sub-buffer Written (Only flight recorder) then
    return EAGAIN
end if
if Writer is holding a reference to the sub-buffer about
  to be exchanged $\vee$ Exchange of reader/writer sub-buffer
  reference fails then
    return EAGAIN
end if
return read_count
```

Algorithm 7 READPUTSUBBUF(*arg_read_count*)

```
Require: read_count returned by READGETSUBBUF()
  (arg_read_count).
Ensure: Release exclusive reader access from a sub-buffer.
  Always succeeds even if the writer pushed the reader,
  because the reader had exclusive sub-buffer access.
$new\_read\_count = arg\_read\_count + subbuffer\_size$.
CAS expects arg_read_count, replaces with
  new_read_count
return OK
```

Algorithm 8 ASYNCWAKEUPREADERSTIMER()

```
Ensure: Wake up readers for full sub-buffers
for all Buffers do
    if Sub-buffer Ready then
        Wake up consumers waiting on the buffer
          read_wait queue.
    end if
end for
```

4.4 Memory Barriers

Although LTTng mostly keeps data local to each CPU, cross-CPU synchronization is still required at three sites:

- At boot-time and cpu hotplug time-stamp counters synchronization, performed at by either the BIOS (*Basic Input/Output System*) or the operating system. This heavy synchronization requires full control of the system.
- When the producer finishes writing to a sub-buffer, thus making it consumable by threads running on arbitrary CPUs. This involves using the proper memory barriers ensuring that all previously written buffer data is committed to memory before another CPU starts reading the buffer.
- At consumed data counter update, involving the appropriate memory barriers ensuring the data has been fully read before making the buffer available for writing.

The two points where a sub-buffer can pass from one CPU to another occur when it is exchanged between the producer and the consumer and when it goes back from the consumer to the producer, because the consumer may run on a different CPU than the producer. Good care must therefore be taken to ensure correct memory ordering between buffer management variables and the buffer data accesses. The condition which makes a sub-buffer ready for reading is represented by Eqn. (2), which depends on the *read count* and the *commit seq* counter corresponding to the *read count*. Therefore, SMP systems allowing out-of-order memory writes must write buffer data before incrementing *commit seq*, by means of a write memory barrier. On the read-side, a read memory barrier must be issued between reading *commit seq* and its associated buffer data. It ensures correct read ordering of counter and buffer data.

LTTng buffering uses an optimization over the classic memory barrier model. Instead of executing a write memory barrier before each *commit seq* update, a simple compiler optimization barrier is used to make sure data written to buffer and *commit seq* update happen in program order with respect to local interrupts. Given that the write order is only needed when the read-side code needs to check the buffer's *commit seq* value, Algorithm 6 shows how the read-side sends an IPI to execute a memory barrier on the target CPU between two memory barriers on the local CPU to ensure that memory ordering is met when the sub-buffer is passed from the writer to the reader. This IPI scheme promotes the compiler barrier to a memory barrier each time the reader needs to issue a memory barrier. Given the reader needs to issue such a barrier only once per sub-buffer switch, compared to a write memory barrier once per event, this improves performance by removing a barrier from the fast path at the added cost of an extra IPI at each sub-buffer switch, which happens relatively rarely. With an average event size of 8 bytes and a typical sub-buffer size of 1 MiB, the ratio is one sub-buffer switch each 131072 events. Given an IPI executing a write memory barrier on an Intel Core2 Xeon 2.0 GHz takes about 2500 cycles, and given that a write memory barrier takes 8 cycles, memory barrier synchronization speed is increased by a factor 419 to 1.

When the buffer is given back to the producer, a synchronized CAS is used to update the *read count*, which implies a full memory barrier before and after the instruction. The CAS ensures the buffer data is read before the *read count* is updated. Given that the writer does not have to read any data from the buffer and depends on reading the *read count* value to check if the buffer is full (in *discard* mode), only the *read count* is shared. The control dependency between the test performed on *read count* and write to the buffer ensures the writer never writes to the buffer before the reader has finished reading from it.

4.5 Buffer allocation

The lockless buffer management algorithm found in LTTng allows dealing with concurrent write accesses to *slots* (segments of a circular buffer) of variable length. This concurrency management algorithm does not impose any requirement on the nature of the memory backend which holds the buffers. The present section will expose the primary memory backends supported by LTTng as well as the backends planned for support in future versions.

The primary memory backend used by LTTng is a set of memory pages allocated by the operating system's page allocator. Those pages are not required to be physically contiguous. This ensures that page allocation is still possible even if memory is fragmented. There is no need to have any virtually contiguous address mapping, which is preferable given that there is a limited amount of kernel-addressable virtual address space (especially on 32-bits systems). These pages are accessed through a single-level page table which performs the translation from a linear address mapping (offset within the buffer) to a physical page address. Buffer *read()*, *write()* and *splice()* primitives abstract the non-contiguous nature of the underlying memory layout by providing an API which presents the buffer as a virtually contiguous address space.

LTTng buffers are exported to user-space through the *DebugFS* file system. It presents the LTTng buffers as a set of virtual files to user applications and allows interacting with those files using *open()*, *close()*, *poll()*, *ioctl()* and *splice()* system calls.

LTTng includes a replacement of *RelayFS* aimed at efficient zero-copy data extraction from buffer to disk or to the network using the *splice()* system call. Earlier LTTng implementations, using *RelayFS*, were based on mapping the buffers into user-space memory to perform data extraction. However, this comes at the expense of wasting precious TLB entries usually available for other use. The current LTTng implementation uses the *splice()* system call. Its usage requires creating a pipe. A *splice()* system call, implemented specifically to read the buffer virtual files, is used to populate the pipe source with specific memory pages. In this case, the parts of the buffer to copy are selected. Then, a second *splice()* system call (the standard pipe implementation) is used to send the pages to the output file descriptor, which targets either a file on disk or a network socket.

Separating the buffer-space management algorithm from the memory backend support eases the implementation of specialized memory backends, depending on the requirements:

- Discontiguous page allocation (presented above) requires adding a software single-level page table, but permits allocation of buffers at run-time when memory is fragmented.
- Early boot-time page allocation of large contiguous memory areas requires low memory fragmentation, but permits faster buffer page access because it does not need any software page-table indirection.
- Video memory backend can be used by reserving video memory for trace buffers. It allows trace data to survive hot reboots, which is useful to preserve trace data after a kernel crash.

5. EXPERIMENTAL RESULTS

This section presents the experimental results from the `LTTng` implementation under various workloads, and compares these with alternative existing technologies.

5.1 Methodology

To present the tracer performance characteristics, we first measure the overhead of the `LTTng` tracer for various workloads on different system types. Then, we compare this overhead to existing state-of-the-art approaches.

The probe CPU-cycles benchmarks, presented in section 5.2, demonstrate the `LTTng` probe overhead in an ideal scenario, where the data and instructions are already in cache.

Then, benchmarks representing real-life workloads, tbench and dbench, simulate the load of a Samba server for network traffic and for disk traffic, respectively. A tbench test on loopback interface shows the worse-case scenario of 8 client and 8 server tbench threads heavily using a traced kernel. Scalability of the tracer when the number of cores increases is tested on the heavy loopback tbench workload.

Yet another set of benchmarks uses `lmbench` to individually test tracing overhead on various kernel primitives, mainly system calls and traps, to show the performance impact of active tracing on those important system components.

Finally, a set of benchmarks runs a compilation of the Linux kernel 2.6.30 with and without tracing to produce a CPU intensive workload.

Probe CPU-cycles overhead benchmarks are performed on a range of architectures. Unless specified, benchmarks are done on an Intel Core2 Xeon E5405 running at 2.0 GHz with 16 GiB of RAM. Tests are executed on a 2.6.30 Linux kernel with full kernel preemption enabled. The buffers configuration used for high event-rate buffers is typically two 1 MiB sub-buffers, except for block I/O events, where per-CPU buffers of eight 1 MiB sub-buffers are used.

5.2 Probe CPU-cycles overhead

This test measures the cycle overhead added by `LTTng` probes. This provides us with a per-event overhead lower bound.

Architecture	Cycles	Core freq. (GHz)	Time (ns)
Intel Pentium 4	545	3.0	182
AMD Athlon64 X2	628	2.0	314
Intel Core2 Xeon	238	2.0	119
ARMv7 OMAP3	507	0.5	1014

Table 1: Cycles taken to execute a `LTTng` 0.140 probe, Linux 2.6.30

Test	Tbench output (MiB/s)	Overhead (%)	Trace output (KEvents/s)
Mainline Linux	12.45	0	–
Instrumented	12.56	0	–
Flight recorder	12.49	0	104
Tracing to disk	12.44	0	107

Table 2: `tbench` client network throughput tracing overhead

This is considered a lower-bound because this test is performed in a tight loop, therefore favoring cache locality. In standard tracer execution, the kernel usually trashes part of the data and instruction caches between probe executions.

The number of cycles consumed by calling a probe from a static instrumentation site passing two arguments, a long and a pointer, on Intel Pentium 4, AMD Athlon, Intel Core2 Xeon and ARMv7 is presented in Table 1. These benchmarks are done in kernel-space, with interrupts disabled, sampling the CPU time-stamp counter before and after looping 20,000 times over the tested case.

Given that one `LCAS` is needed to synchronize the tracing space reservation, based on the results published in [Desnoyers and Dagenais 2010], we can see that disabling interrupts instead of using the `LCAS` would add 34 cycles to these probes on Intel Core2, for an expected 14.3% slowdown. Therefore, not only is it interesting to use *local atomic operations* to protect against non-maskable interrupts, but it also improves the performance marginally. Changing the implementation to disable interrupts instead of using `LCAS` confirms this: probe execution overhead increases from 240 to 256 cycles, for a 6.6% slowdown.

5.3 tbench

The `tbench` benchmark tests the throughput achieved by the network traffic portion of a simulated Samba file server workload. Given it generates network traffic from data located in memory, it results in very low I/O and user-space CPU time consumption, and very heavy kernel network layer use. We therefore use this test to measure the overhead of tracing on network workloads. We compare network throughput when running the non instrumented *mainline Linux* kernel, the *instrumented* kernel (with inactive tracing), the traced kernel in *flight recorder* mode (events overwritten in memory), and *tracing to disk* in discard mode.

This set of benchmarks, presented in Table 2, shows that tracing has very little impact on the overall performance under network load on a 100 Mb/s network card. 8 `tbench` client threads are executed for a 120s warm up and 600s test

Test	Tbench output (MiB/s)	Overhead (%)	Trace output (KEvents/s)
Mainline Linux	2036.4	0	–
Instrumented	2047.1	-1	–
Flight recorder	1474.0	28	9768
Tracing to disk	–	–	–

Table 3: `tbench` localhost client/server throughput tracing overhead

execution. Trace data generated in flight recorder mode reaches 0.9 GiB for a 1.33 MiB/s trace data throughput. Data gathered in normal tracing to disk reaches 1.1 GiB. The supplementary data generated when writing trace-data to disk is explained by the fact that we also trace disk activity, which generates additional events. This very little performance impact can be explained by the fact that the system was mostly idle.

Now, given that currently existing 1 Gb/s and 10 Gb/s network cards can generate higher throughput, and given the 100 Mb/s link was the bottleneck of the previous `tbench` test, Table 3 shows the added tracer overhead when tracing `tbench` running with both server and client on the loopback interface on the same machine, which is a worst-case scenario in terms of generated throughput kernel-wise. This workload consists in running 8 client threads and 8 server threads.

The kernel instrumentation, when compiled-in but not enabled, actually accelerates the kernel for both cases shown in Tables 2 and 3. This can be attributed to modification of instruction and data cache layout. Flight recorder tracing stores 92 GiB of trace data to memory, which represents a trace throughput of 130.9 MiB/s for the overall 8 cores. Tracing adds a 28% overhead on this workload. Needless to say that trying to export such throughput to disk would cause a significant proportion of events to be dropped. This is why tracing to disk is excluded from this table. This type of workload shows the importance of tracer flexibility, allowing end-users to tweak the tracer configuration as required to cope with high-throughput workloads.

5.4 Scalability

To characterize the tracer overhead when the number of CPUs increases, we need to study a scalable workload where tracing overhead is significant. The localhost `tbench` test exhibits these characteristics. Figure 7 presents the impact of *flight recorder* tracing on the `tbench` localhost workload on the same setup used for Table 3. The number of active processors varies from 1 to 8 together with the number of `tbench` threads. We notice that the `tbench` workload itself scales linearly in the absence of tracing. When tracing is added, linear scalability is invariant. It shows that the overhead progresses linearly as the number of processors increases. Therefore, tracing with `LTTng` adds a constant per-processor overhead independent from the number of processors in the system.

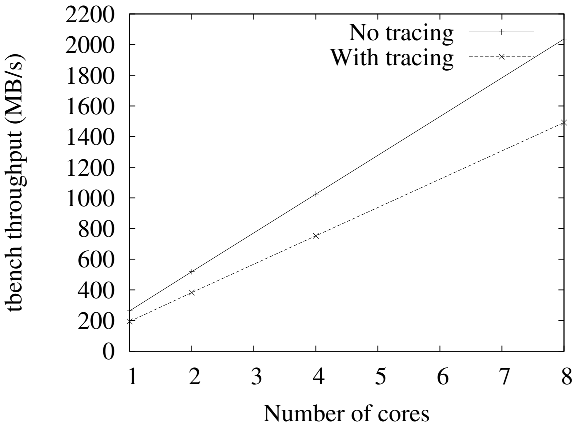

Figure 7: Impact of tracing overhead on localhost `tbench` workload scalability

Test	Dbench output (MiB/s)	Overhead (%)	Trace output (KEvents/s)
Mainline Linux	1334.2	0	–
Instrumented	1373.2	-2	–
Flight recorder	1297.0	3	2840
Tracing to disk	872.0	35	2562

Table 4: `dbench` disk write throughput tracing overhead

5.5 dbench

The `dbench` test simulates the disk I/O portion of a Samba file server. The goal of this benchmark is to show the tracer impact on such a workload, especially for *discard* tracing to disk.

This set of benchmarks, presented in Table 4, shows tracing overhead on a 8 thread `dbench` workload. Tracing in *flight recorder* mode causes a 3% slowdown on disk throughput while generating 30.2 GiB of trace data into memory buffers. Normal tracing to disk causes a 35% slowdown on heavy disk operations, but lack of disk bandwidth is causing a significant portion of trace events to be discarded.

Analysis of the buffer state in *flight recorder* mode shows that 30.2 GiB worth of data was generated in 720 seconds, for a sustained trace throughput of 43.0 MiB/s. In *discard* mode, the trace is written to the same disk `dbench` is using. The tracing throughput is therefore significant compared to the available disk bandwidth. It comes without surprise that only 23 GiB of trace data has been collected to disk in the *discard* trace, with a total of 21.8 million events lost. This trace size difference is caused both by the events lost (only lost about 244 MiB of data, or 1%, given an average event size of 12 bytes) and, mostly, to the behavior change generated by the added disk I/O activity for tracing. While the system is busy writing large chunks of trace data, it is not available to process smaller and more frequent `dbench` requests. This nicely shows how the tracer, in *discard* mode, can affect disk throughput in I/O-heavy workloads.

5.6 lmbench

The `lmbench` benchmarks test various kernel primitives by executing them in loops. We use this test to appropriately measure the tracer overhead on a per-primitive basis. Running `lmbench` on the mainline Linux kernel, *flight recorder* and *discard* tracing kernels, helps understanding the performance deterioration caused by tracing.

When running on a Intel Core2 Xeon E5405, the standard `lmbench` 3.0 OS test generates 5.41 GiB of trace data with the default `LTTng` instrumentation in 6 minutes for a throughput of 150 MiB/s. When writing to disk the total trace size reaches 5.5 GiB due to the added traced disk I/O overhead.

The "simple system call" test, which calls a system call with small execution time in a tight loop, takes 0.1752 μs per system call on the mainline Linux kernel. Compared to this, it takes 0.6057 μs on the *flight recorder* mode traced kernel. In fact, the benchmarks for *flight recorder* tracing and disk tracing are very similar, because the only difference is the CPU time taken by the `lttd` daemon and the added disk I/O.

The "simple system call" slowdown is explained by the fact that two sites are instrumented: system call entry and system call exit. Based on measurements from Table 1, we would expect each event to add at least 0.119 μs to the system call. In reality, they add 0.215 μs each to the system call execution. The reasons for this additional slowdown is because supplementary registers must be saved in the system call entry and exit paths and cache effects. The register overhead is the same as the well-known *ptrace()* debugger interface, secure computing and process accounting because these and `LTTng` all share a common infrastructure to extract these registers.

Some system calls have more specific instrumentation in their execution path. For instance, the file name is extracted from the *open()* system call, the file descriptor and size are extracted from the *read()* system call. The performance degradation is directly related to the number of probes executed. For the *read()* system call, the mainline Linux kernel takes 0.2138 μs, when the flight recorder tracing kernel takes 0.8043 μs. By removing the "Simple system call" tracing overhead, this leaves a 0.1600 μs, which corresponds to the added event in the *read()* system call.

The page fault handler, a frequently executed kernel code path, is instrumented with two tracepoints. It is very important due to the frequency at which it is called during standard operation. On workloads involving many short-lived processes, page faults, caused by copy-on-write, account for an important fraction of execution time (4% of a Linux kernel build). It runs in 1.3512 μs on the mainline Linux kernel and takes 1.6433 μs with flight recorder activated. This includes 0.146 μs for each instrumentation site, which is close to the expected 0.119 μs per event. Non-cached memory accesses and branch prediction buffer pollution are possible causes for such small execution time variation from expected results.

Instrumentation of such frequently executed kernel code path is the reason why minimizing probe execution time is critical to the tracer's usability on heavy workloads.

Test	Time (s)	Overhead (%)	Trace output (KEvents/s)
Mainline Linux	85	0	–
Instrumented	84	-1	–
Flight recorder	87	3	822
Tracing to disk	90	6	816

Table 5: Linux kernel compilation tracing overhead

Other `lmbench` results show that some instrumented code paths suffer from greater overhead. This is mostly due to use of a less efficient dynamic format-string parsing method to write the events into the trace buffers. For instance, the "Process fork+exit" test takes 211.5 μs to execute with tracing instead of 177.8 μs, for an added overhead of 33.7 μs for each entry/exit pair. Based on execution trace analysis of standard workloads, as of `LTTng` 0.140, events corresponding to process creation and destruction where not considered to be frequently used compared to page faults, system calls, interrupts and scheduler activity. If this becomes a concern, the optimized statically-compiled version of the event serializer could be used.

5.7 gcc

The `gcc` compilation test aims at showing the tracer impact on a workload where most of the CPU time is spent in user-space, but where many short-lived processes are created. Building the Linux kernel tree is such a scenario, where `make` creates one short-lived `gcc` instance per file to compile. This therefore shows mostly tracer impact on process creation. This includes page fault handler instrumentation impact, due to copy-on-write and lazy page population mechanisms when processes are created and when executables are loaded. This also includes instrumentation of scheduler activity and process state changes.

Table 5 presents the time taken to build the Linux kernel with `gcc`. This test is performed after a prior cache-priming compilation. Therefore, all the kernel sources are located in I/O buffer cache.

Tracing the kernel in *flight recorder* mode, with the default `LTTng` instrumentation, while compiling the Linux kernel, generates 1.1 GiB of trace data for a 3% slowdown. The results show, without surprise, that kernel tracing has a lower impact on user-space CPU-bound workloads than I/O-bound workloads. Tracing to disk generates 1.3 GiB of data output. This is higher than the trace data generated for flight recording due to the supplementary disk activity traced. Trace throughput, when tracing to disk, is lower than *flight recorder* mode, because the tracer disk activity generates fewer events per second than kernel compiling in the CPU time it consumes, hence reducing the number of events per second to record.

5.8 Comparison

Previous work on highly scalable operating systems took place at IBM Research resulting in the `K42` operating system [Krieger et al. 2006], which includes a built-in highly

Architecture	IRQ-off (ns)	Lockless (ns)	Speedup (%)
Intel Pentium 4	212	182	14
AMD Athlon64 X2	381	314	34
Intel Core2 Xeon	128	119	7
ARMv7 OMAP3	1108	1014	8

Table 6: Comparison of lockless and interrupt disabling LTTng probe execution time overhead, Linux 2.6.30

scalable kernel tracer based on a lockless buffering scheme. As presented in Section 2, K42's buffering algorithm contains rare race conditions which could be problematic, especially given LTTng buffer and event size flexibility. Being a research operating system, K42 does not support CPU hotplug, nor distributing tracing overhead across idle cores, and is limited to a subset of existing widely used hardware, which provides a 64-bit cycle counter synchronized across cores.

The instrumentation used in LTTng has been taken from the original LTT project [Yaghmour and Dagenais 2000]. It consists of about 150 instrumentation sites, some architecture-agnostic, others being architecture-specific. They have been ported to the "Linux Kernel Markers" [Corbet 2007a] and then to "Tracepoints" [Corbet 2008] developed as part of the LTTng project and currently integrated in the mainline Linux kernel. The original LTT and earlier LTTng versions, used RelayFS [Zanussi et al. 2003] to provide memory buffer allocation and mapping to user-space. LTTng re-uses part of the *splice()* implementation found in RelayFS.

To justify the choice of using static code-level instrumentation instead of dynamic, breakpoint-based instrumentation, we must explain the performance impact of breakpoints. These are implemented with a software interrupt triggered by a breakpoint instruction temporarily replacing the original instructions to instrument. The specialized interrupt handler executes the debugger or the tracer when the breakpoint instruction is executed. An interesting result of the work presented in this paper is that the LTTng probe takes less time to run than a breakpoint alone. Tests running an empty Kprobe (which includes a breakpoint and single-stepping) in a loop shows it has a performance impact of 4200 cycles, or 1.413 μs, on a 3 GHz Pentium 4. Compared to this, the overall time taken to execute an LTTng probe is 0.182 μs, which represents a 7.8:1 acceleration compared to the breakpoint alone.

It is also important to compare the lockless scheme proposed to an equivalent solution based on interrupt disabling. We therefore created an alternative implementation of the LTTng buffering scheme based on interrupt disabling for this purpose. It uses non-atomic operations to access the buffer state variables and is therefore not NMI-safe. Table 6 shows that the lockless solution is either marginally faster (7–8%) on architectures where interrupt disabling cost is low, or much faster (34%) in cases where interrupt disabling is expensive in terms of cycles per instruction.

Benchmarks performed on DTrace [Cantrill et al. 2004], the Solaris tracer, on a Intel Pentium 4 shows a performance impact of 1.18 μs per event when tracing all system calls to a buffer. LTTng takes 0.182 μs per event on the same architecture, for a speedup of 6.42:1. As shown in this paper, tracing a tbench workload with LTTng generates a trace throughput of 130.9 MiB/s, for approximately 8 million events/s with an average event size of 16 bytes. With this workload, LTTng has a performance impact of 28 %, for a workload execution time of 1.28:1. DTrace being 6.42 times slower than LTTng, the same workload should be expected to be slowed down by 180% and therefore have an execution time of 2.8:1. Therefore, performance-wise, LTTng has nothing to envy [Corbet 2007b]. This means LTTng can be used to trace workloads and diagnose problems outside of DTrace reach.

6. CONCLUSION

Overall, the LTTng kernel tracer presented in this paper has a wide kernel code instrumentation coverage, which includes tricky non-maskable interrupts, traps and exception handlers, as well as the scheduler code. It has a per-event performance overhead 6.42 times lower than the existing DTrace tracer. The performance improvements are mostly derived from the following atomic primitive characteristics: *local atomic operations*, when used on local per-CPU variables, are cheaper than disabling interrupts on many architectures.

The atomic buffering mechanism presented in this paper is very useful for tracing. The good reentrancy and performance characteristics demonstrated could be useful to other parts of the kernel, especially drivers. Using this scheme could accelerate buffer synchronization significantly and diminish interrupt latency.

A port of LTTng has already been done to the Xen hypervisor and as a user-space library to study merged traces taken from the hypervisor, the various kernels running in virtual machines, and user-space applications and libraries.

7. ACKNOWLEDGMENTS

We would like to thank the Linux Trace Toolkit, Linux and SystemTAP communities for their feedback, as well as NSERC, Google, IBM Research, Autodesk and Ericsson for funding parts of this work. We are indebted to Etienne Bergeron and Robert Wisniewski for reviewing this paper.

8. REFERENCES

[Bligh et al. 2007] Bligh, M., Schultz, R., and Desnoyers, M. 2007. Linux kernel debugging on Google-sized clusters. In *Proceedings of the Ottawa Linux Symposium.*

[Cantrill et al. 2004] Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H. 2004. Dynamic instrumentation of production systems. In *USENIX.* [Online]. Available: http://www.sagecertification.org/events/usenix04/tech/general/full_papers/cantrill/cantrill_html/index.html. [Accessed: October 19, 2009].

[Corbet 2007a] Corbet, J. 2007a. Kernel Markers. [Online]. Available: Linux Weekly News, http://lwn.net/Articles/245671/. [Accessed: October 19, 2009].

[Corbet 2007b] Corbet, J. 2007b. On DTrace envy. [Online]. Available: Linux Weekly News, http://lwn.net/Articles/244536/. [Accessed: October 19, 2009].

[Corbet 2008] Corbet, J. 2008. Tracing: no shortage of options. [Online]. Available: Linux Weekly News, http://lwn.net/Articles/291091/. [Accessed: October 19, 2009].

[Desnoyers 2009] Desnoyers, M. 2009. Low-impact operating system tracing. Ph.D. thesis, École Polytechnique de Montréal. [Online]. Available: http://www.lttng.org/pub/thesis/desnoyers-dissertation-2009-12.pdf.

[Desnoyers and Dagenais 2006] Desnoyers, M. and Dagenais, M. 2006. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Proceedings of the Ottawa Linux Symposium*.

[Desnoyers and Dagenais 2010] Desnoyers, M. and Dagenais, M. R. 2010. Synchronization for fast and reentrant operating system kernel tracing. *Software – Practice and Experience 40,* 12, 1053–1072.

[Desnoyers et al. 2012] Desnoyers, M., McKenney, P. E., Stern, A. S., Dagenais, M. R., and Walpole, J. 2012. User-level implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 23,* 2 (feb.), 375–382.

[Hillier 2008] Hillier, G. 2008. System and application analysis with LTTng. [Online]. Available: Siemens Linux Inside, http://www.hillier.de/linux/LTTng-examples.pdf. [Accessed: June 7, 2009].

[Krieger et al. 2006] Krieger, O., Auslander, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Da Silva, D., and al. 2006. K42: building a complete operating system. In *EuroSys '06: Proceedings of the 2006 EuroSys conference*. 133–145.

[Mavinakayanahalli et al. 2006] Mavinakayanahalli, A., Panchamukhi, P., Keniston, J., Keshavamurthy, A., and Hiramatsu, M. 2006. Probing the guts of kprobes. In *Proceedings of the Ottawa Linux Symposium*.

[McKenney 2004] McKenney, P. E. 2004. Exploiting deferred destruction: An analysis of read-copy-update techniques in operating system kernels. Ph.D. thesis, OGI School of Science and Engineering at Oregon Health and Sciences University. [Online]. Available: http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf. [Accessed: October 19, 2009].

[Prasad et al. 2005] Prasad, V., Cohen, W., Eigler, F. C., Hunt, M., Keniston, J., and Chen, B. 2005. Locating system problems using dynamic instrumentation. In *Proceedings of the Ottawa Linux Symposium*. [Online]. Available: http://sourceware.org/systemtap/systemtap-ols.pdf. [Accessed: October 19, 2009].

[Wisniewski and Rosenburg 2003] Wisniewski, R. and Rosenburg, B. 2003. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Supercomputing, 2003 ACM/IEEE Conference*. IEEE, 3–3.

[Wisniewski et al. 2007] Wisniewski, R. W., Azimi, R., Desnoyers, M., Michael, M. M., Moreira, J., Shiloach, D., and Soares, L. 2007. Experiences understanding performance in a commercial scale-out environment. In *European Conference on Parallel Processing (Euro-Par)*.

[Yaghmour and Dagenais 2000] Yaghmour, K. and Dagenais, M. R. 2000. The Linux Trace Toolkit. *Linux Journal*. [Online]. Available: http://www.linuxjournal.com/article/3829. [Accessed: October 19, 2009].

[Zanussi et al. 2003] Zanussi, T., Wisniewski, K. Y. R., Moore, R., and Dagenais, M. 2003. RelayFS: An efficient unified approach for transmitting data from kernel to user space. In *Proceedings of the Ottawa Linux Symposium*. 519–531. [Online]. Available: http://www.research.ibm.com/people/b/bob/papers/ols03.pdf. [Accessed: October 19, 2009].