SOFTWARE—PRACTICE AND EXPERIENCE
*Softw. Pract. Exper.* 2009; **00**:1–20 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

# Synchronization for fast and reentrant operating system kernel tracing

Mathieu Desnoyers* and Michel R. Dagenais

*École Polytechnique de Montréal, Dept. of Computer and Software Eng., C.P. 6079, succ. Centre-ville, Montréal, Québec, Canada, H3C 3A4.*

**SUMMARY**

**To effectively trace an operating system, a performance monitoring and debugging infrastructure needs the ability to trace various execution contexts. These contexts range from kernel running as a thread to** `NMI` **(*Non-Maskable Interrupt*) contexts.**

**Given that any part of kernel infrastructure used by a kernel tracer could lead to infinite recursion if traced, and because most kernel primitives require synchronization unsuitable for some execution contexts, all interactions of the tracing code with the existing kernel infrastructure must be considered in order to correctly inter-operate with the existing operating system kernel.**

**This paper presents a new low overhead tracing mechanism and motivates the choice of synchronization sequences suited for operating system kernel tracing, namely** *local atomic instructions* **as main buffer synchronization primitive and the** `RCU` **(*Read-Copy Update*) mechanism to control tracing. It also proposes a wait-free algorithm extending the time-base needed by the tracer to 64-bit on architectures that lack hardware 64-bit time-base support. Copyright © 2009 John Wiley & Sons, Ltd.**

KEY WORDS: tracing; operating system; synchronization; wait-free algorithms; low overhead; clock

## 1. INTRODUCTION

As computers are becoming increasingly multi-core, the need for debugging facilities that will help identify timing related problems, across execution layers and between processes, is rapidly increasing [1, 2, 3]. Such facilities must allow information gathering about problematic workloads without affecting the behavior being investigated in order to be useful for problem analysis purposes.

*Correspondence to: Mathieu Desnoyers, École Polytechnique de Montréal, Dept. of Computer and Software Eng., École Polytechnique de Montréal, C.P. 6079, succ. Centre-ville, Montréal, Québec, Canada, H3C 3A4.
E-mail: mathieu.desnoyers@polymtl.ca
Contract/grant sponsor: Natural Sciences and Engineering Research Council of Canada
Contract/grant sponsor: Google Inc.

The variety of execution contexts reached during kernel execution complicates efficient exportation of trace data out of the instrumented kernel. The general approach used to deal with this level of concurrency is to either provide good protection against other execution contexts, for instance by disabling interrupts, or to adopt a fast, but limited mechanism by shrinking the instrumentation coverage (e.g. by disallowing interrupt handler instrumentation). This trade-off often means that either performance or instrumentation coverage is sacrificed. However, as this paper will show, this trade-off is not required if the appropriate synchronization primitives are chosen.

We propose to extend the `OS`[1] kernel instrumentation coverage compared to other existing tracers by dealing with the variety of kernel execution contexts. Our approach is to consider reentrancy from `NMI`[2] execution context, which presents particular constraints regarding execution atomicity due to the inability to create critical sections by disabling interrupts. In this article, we show that in a multiprocessor `OS`, the combination of synchronized time-stamp counters, cheap single-CPU atomic operations and trace merging, provides an effective and efficient tracing mechanism which supports tracing in `NMI` contexts.

Section 3 will first present the synchronization primitives used by the state-of-the-art open source tracers. In Section 4, we outline the `LTTng`[3] design requirements which aim at high scalability and minimal real-time response disruption.

Recursion between the tracer probe and the kernel will be discussed in Section 5, where the reasons why a kernel tracer cannot call standard kernel primitives without limiting the instrumentation coverage will be illustrated by concrete examples in Sections 5.1 and 5.2. We then propose an algorithm that synchronizes data structures to provide a 64-bit tracer clock on architectures that lack hardware 64-bit time-base support in Section 5.3.

In Section 6 we show how to achieve both good performance and instrumentation coverage by choosing primitives that provide the right reentrancy characteristics and high performance. Overhead measurements comparing the `RCU`[4] [4] mechanism and *local atomic operations* primitives to alternative synchronization methods will support this proposal.

This opens a wide perspective for the design of fully reentrant, wait-free, high-performance buffering schemes.

## 2. INTRODUCTION TO TRACING

This section introduces to the background required to understand the following state of the art. The tracer impact on real-time response is first discussed, followed by a description of tracer concepts.

Real-time impact of algorithms can be categorized following the guarantees they provide. The terms used to identify such guarantees evolved through time in literature [5, 6]. The terminology used in this article will follow [6]. The strongest non-blocking guarantee is wait-free, which ensures each thread can always make progress and is therefore never starved. A non-blocking lock-free algorithm

[1]`OS`: Operating System
[2]`NMI`: Non-Maskable Interrupt
[3]`LTTng`: Linux Trace Toolkit Next Generation
[4]`RCU`: Read-Copy Update.

only ensures that the system as a whole always makes progress. This is made possible by ensuring that at least one thread is progressing when concurrent access to a data structure is performed. An obstruction-free algorithm offers an even weaker guarantee than lock-free: it only guarantees progress of any thread executing in isolation, meaning that all competing concurrent accesses may be aborted. Finally, a blocking algorithm does not provide any of these guarantees.

This article specifically discusses operating system kernel tracers. A tracer consists in a mechanism collecting an execution trace from a running system. A trace is a sequence of event records, each identifying that the kernel executed a pre-identified portion of its code.

Mapping between execution sites and events is made possible by instrumentation of the kernel. Instrumentation can be either declared statically at the source-code level or dynamically added to the running kernel. Statically declared instrumentation can be enabled dynamically.

A tracer probe is the tracer component called when enabled instrumentation is executed. This probe is responsible for fetching all the data to write in the event record, namely an event identifier, a time-stamp and, optionally, an event-specific payload. Time-stamps are monotonically increasing values representing the time-flow on the system. They are typically derived from an external timer or from a `TSC`[5] register on the processor.

To amortize the impact of `I/O`[6] communication, event records are saved in memory buffers. Their extraction through an `I/O` device is therefore delayed. To ensure continuous availability of free buffer space, a ring buffer with at least two sub-buffers can be used. One is used by tracer probes to write events while the other is extracted through `I/O` devices.

## 3. STATE OF THE ART

This section reviews the synchronization primitives used in the state-of-the-art open source tracers currently available, namely: the original LTT tracer [7], the wait-free write-side tracing solution found in `K42` [8, 9], a highly-scalable research operating system created by IBM, `DTrace` [2] from Sun's OpenSolaris, `SystemTAP` [10] from RedHat, providing scripting hooks for the Linux kernel built as external modules, `KTAU` [11] from University of Oregon and `Ftrace`, started by Linux kernel maintainer Ingo Molnar. The following study details the synchronisation mechanisms used in each of these projects.

The original `LTT` (Linux Trace Toolkit) [7] project started back in 1999. Karim Yaghmour, its author, aimed at creating a kernel tracer suitable for the Linux kernel with a static instrumentation set targeting the most useful kernel execution sites. `LTT` uses the architecture time-stamp counter register when available to interpolate the time between the time-stamps taken at sub-buffer boundaries with *do_gettimeofday()*. This leads to problems with `NTP` (Network Time Protocol) correction, where the time-base at the sub-buffer boundaries could appear to go backward. Regarding synchronization, the *do_gettimeofday()* function uses a sequence counter locking on the read-side for ensuring time-base data consistency, which can cause deadlocks if `NMI` handlers were instrumented. One of the early buffering scheme used was based on spin lock (busy-waiting lock) disabling interrupts for buffers

[5]`TSC`: Time-Stamp Counter
[6]`I/O`: Input/Output

shared between the CPUs. Per-CPU buffers support, provided by RelayFS, uses interrupt disabling to protect from interrupt handlers. Karim worked, in collaboration with Tom Zanussi and Robert Wisniewski from IBM, on the integration of some lockless buffering ideas from the K42 tracer into RelayFS.

K42 [8, 9] is a research operating system developed by IBM, mostly between 1999 and 2006. According to the authors, its code-base should be considered as a prototype. It focuses on large multi-processor machine scalability and therefore uses data structures and operations local to each CPU as much as possible. It brings some very interesting ideas for tracing synchronization, namely the use of atomic operations to synchronize buffer space reservation. The tracing facility found in K42 is built into the kernel. It uses per-CPU buffers to log tracing data and limits the consumption of data to user-space threads tied to the local CPU. This first design constraint could be problematic in a production OS, because if the workload is not equally shared amongst all CPUs, those with the most idle time will not be able to collaborate with the busier CPUs to help them extract the trace streams to disk or over the network. It uses a wait-free algorithm based on the *CAS* (*compare-and-swap*) operation to manage space reservation from multiple concurrent writers. It adds compiler optimisation restriction barriers to order instructions with respect to the local instruction stream, but does not add memory barriers, since all data accesses are local. Once the space reservation is performed, the data writes to the buffer and the commit count increments are done out-of-order. A *buffers produced* count and a *buffers consumed* count are updated to keep track of the buffers available for consumption by the user-space thread. For time-base synchronization, K42 only supports architectures with 64-bit time-stamp counters (PowerPC and amd64) and assumes that those counters are synchronized across all CPUs. Therefore, a simple register read is sufficient to provide the time-base at the tracing site, and no synchronization is required after system boot.

The DTrace [2] tracer has first been made available in 2003, and formally released as part of Sun's Solaris 10 in 2005 under the CDDL[7] license. It aims at providing information to users about their system's behavior by executing scripts at the kernel level when instrumentation sites are reached. It has since then been ported to FreeBSD and Apple Mac OS X 10.5 Leopard. A port to the Linux kernel is under development, but involves license issues between CDDL and GPL[8].

The DTrace tracer[9] disables interrupts around iteration on the probe array before proceeding to their invocation. Therefore, the whole tracer site execution is protected from interrupts coming on the local CPU. Disabling interrupts also serves as a means to mark a RCU read-side critical section. Trace control synchronization is based on a RCU-like [4] mechanism which waits for a grace period before tracing sites can be considered having reached a quiescent state. This is performed by executing a thread on every CPU waiting for all the currently active tracing sites to complete. Given that such threads are not allowed to execute while interrupts are disabled, this permits detection of tracing sites quiescent states.

DTrace also uses a per-thread flag, *T_DONTDTRACE*, ensuring that critical kernel code dealing with page mappings does not call the tracer. It does not seem, however, to apply any thread flag to NMI handler execution. In OpenSolaris, NMIs are primarily used to enter the kernel debugger, which

[7] CDDL: Common Development and Distribution License.
[8] GPL: General Public License.
[9] Version reviewed: OpenSolaris 20090330.

is not allowed to run at the same time as DTrace. Therefore, the following discussion applies to a situation where the same algorithms and structures would be used in an operating system like Linux, where NMIs can execute code contained in various subsystems, including the Oprofile [12] profiler.

DTrace calls the *dtrace_gethrtime()* primitive to read the time source. On the x86 architecture, this primitive uses a locking mechanism similar to the sequence lock in Linux. A sequence lock is a type of lock which lets the reader retry the read operation until the writer exits its critical section. The particularity of the sequence lock found in DTrace is that if it spins twice waiting for the lock, it assumes that it is nested over the write lock, so a time value previously copied by the time-base tick update will be returned. This shadow value is protected by its own sequence lock. In the x86 implementation, this leaves room for a 4-way deadlock on 2 CPUs involving the NTP correction update routines, *tsc_tick()* and two nesting *dtrace_gethrtime()* calls in interrupt handlers.

Although this deadlock should never cause harm due to specific and controlled use of NMIs in OpenSolaris, porting this tracer to a different operating system or loading specific drivers using NMIs could become a problem. Discussion with Bryan Cantrill, author of DTrace, with Mike Shapiro and Adam Leventhal, led to notice that an appropriate NMI-safe implementation based on two sequence locks taken successively from a **single** thread already exists in *dtrace_gethrestime()*, but is not used in the lower-level x86 primitive. It requires that only a single execution thread takes the two sequence locks successively. Using it in the lower-level code would require modification of the NTP adjustment code[10]. This example taken from a widely distributed tracer shows that it is far from trivial to design tracing clock source synchronization properly, especially for a flexible open source operating system like Linux.

Considering real-time guarantees, a sequence lock should be categorized as a blocking algorithm. If an updater thread is stopped in the middle of an update, no reader thread can progress. Therefore, a sequence lock does not provide non-blocking guarantees. This means real-time behavior can be affected significantly by the execution of DTrace.

The SystemTAP [10] project from Redhat, first made available in 2006, aims at letting system administrators run scripts connected at specific kernel sites to gather information and statistics about the system behavior and investigate problems at a system-wide level. It aims at providing features similar to DTrace[11] in the Linux operating system. Its first aim is not to export the whole trace information flow, but rather to execute scripts which can either aggregate the information, perform filtering on the data input or write data into buffers along with time-stamps. The focus is therefore not to have a very high-performance capable data extraction mechanism, given this is not their main target use-case. SystemTAP uses a heavy locking mechanism at the probe site. It disables interrupts and takes a global spin lock[12] twice in the write path. The first critical section surrounded by interrupt disabling and locking is used to manage the free buffer pool. The second critical section, similar to the former but using a distinct lock, is needed to add the buffer ready for consumption to a *ready queue*.

[10]Based on review of the DTrace code-base, we recommend using a standard mutex to ensure mutual exclusion around the two write sequence locks should allow to permit using the same locking mechanism for both *dtrace_gethrestime()* and *dtrace_gethrtime()*, which would allow updates from NTP and from the *tsc_tick()* routine.
[11]According to http://sourceware.org/systemtap/wiki/SystemtapDtraceComparison.
[12]A spin lock is a type of busy-waiting lock in the Linux kernel.

SystemTAP assumes it is called from Kprobes [13], a Linux kernel infrastructure permitting connection of breakpoint-based probes at arbitrary addresses in the kernel. Kprobes disables interrupts around handler execution. Therefore, SystemTAP assumes interrupt disabling is done by the caller, which is not the case for static instrumentation mechanisms like the Linux Kernel Markers and Tracepoints. In those cases, if events come nested over the tracing code, caused by recursion or coming from NMIs, SystemTAP will consider this as an error condition and will silently discard the event until the number of events discarded reaches a threshold. At that point, it will stop tracing entirely. SystemTAP modules can use the *gettimeofday()* primitive exported by the Linux kernel as time source. It uses a sequence lock to ensure the time-base coherency. This fails in a NMI context because it would cause a deadlock if a probe in a NMI nests over a sequence writer lock. Therefore, SystemTAP's internals disallows instrumentation of code reached from NMI context. It also depends on interruptions being disabled by the lower-level instrumentation mechanism.

The KTAU (Kernel Tuning and Analysis Utilities) [11] project, available since 2006, allows to either profile or trace the Linux kernel on a system-wide or per-process basis. It allows detailed per-process collection of trace events to memory buffers, but deals with kernel system-wide data collection by aggregating performance information of the entire system. The motivation for using aggregation to deal with system-wide data collection is that exporting the full information flow into tracing buffers would consume too much system resources. Conversely, the hypothesis the LTTng approach is trying to verify is that it is possible to trace a significant useful subset of operating system's execution in a detailed manner without prohibitive impact on the workload behavior. Therefore, we have to consider if the KTAU process-centric tracing approach would deal with system-wide tracing appropriately.

Some design decisions indicate that detailed process tracing is not meant to be used for system-wide tracing. KTAU keeps buffers and data structures local to each thread, which can lead to significant memory usage on workloads containing multiple threads. Workloads consisting of many mostly inactive threads and few very active threads risk overflowing the buffers if they are too small, or consuming a lot of memory if all buffers are made larger. KTAU allows tweaking the size of specific thread's buffers, but it can be difficult to tune if the threads are short-lived. We can also notice that the kernel idle loop, which includes swap activity, and all interrupts and bottom halves nested over this idle loop, are not covered by the tracer, which silently drops the events.

For synchronization, KTAU permits choosing at compilation time between IRQ or bottom half (lower priority interrupts) disabling and uses a per-thread spin lock to protect its data structures. The fact that the data can stay local to each thread ensures that no unnecessary cache-line bouncing between the CPUs will occur. Those spin locks are used therefore mainly to synchronize the data producer with the consumer. This protection mechanism is thus not intended to trace NMIs because the handler could deadlock when taking a spin lock if it nests over code already holding the lock.

Regarding kernel reentrancy, KTAU uses vmalloc (kernel virtual memory) to allocate the trace buffers. Given that the Linux kernel populates the TLB (Translation Lookaside Buffer) entries of those pages lazily on x86, the tracing code will trigger page faults the first time those pages are accessed. Therefore, the page fault handler should be instrumented with great care. KTAU only supports x86 and PowerPC and uses the time-stamp counter register as a time source, which does not require any synchronization per se. On the performance impact side, allocation of tracing buffers at each thread creation could be problematic on workloads consisting of many short-lived threads, because thread creation is normally not expected to be slowed down by multiple page allocations, since threads usually share the same memory pages.

Ftrace, a project started in 2009 by Ingo Molnar, grew from the `IRQ` tracer, which traces long interrupt latencies, to incrementally integrate the wake up tracer, providing information about the scheduler activity, the function tracer, which instruments the kernel function entry at low-cost and an actively augmented list of tracers. Its goal is to provide system-wide, but subsystem-oriented tracing information primarily useful to kernel developers. It uses the `Tracepoint` mechanism, which comes from the `LTTng` project, as primary instrumentation mechanism.

`Ftrace`, in its current implementation, disables interrupts and takes per-buffer (and thus per-CPU) spin locks. The advantage of taking a per-CPU spin lock over a global spin lock is that it does not require to transfer the spin lock cache-line between CPUs when the lock has to be taken, which improves the scalability when the number of CPUs increases. `Ftrace`, as of its Linux 2.6.29 implementation, does not handle `NMI`s gracefully. If instrumentation is added in a code path reached by `NMI` context, a deadlock may occur due to the use of spin locks. `Ftrace` relies on the scheduler clock for timekeeping, which does not provide any locking against non-atomic *jiffies*[13] counter updates. Although this time source is statistically correct for scheduler purposes, it can result in incorrect timing data when the tracer races with the *jiffies* update. Dropping events coming from nested `NMI` handlers will be the solution integrated in the 2.6.30 kernels. Improvement is expected in a near future regarding tracing buffer ability to handle `NMI`s gracefully using a lock-free kernel-specific buffering scheme submitted for U.S. and international patent in early 2009 by Steven Rostedt[14].

## 4. LINUX TRACE TOOLKIT NEXT GENERATION

The purpose of the study presented in this paper is to be used as a basis for developing the `LTTng` kernel tracer. This tracer aims at tracing the Linux kernel while providing these guarantees:

- Provide a wide instrumentation coverage.
- Provide probe reentrancy for all kernel execution contexts, including `NMI`s and `MCE` (Machine Check Exception) handlers.
- Record very high-frequency kernel events.
- Impose small overhead to typical workloads.
- Scale to large multiprocessor systems.
- Change the system real-time response in a predictable way.

Earlier work presented an overview of the `LTTng` tracer design [14] and industry use-case scenarios in the industry [1, 15, 16, 17]. That work presents an in-depth analysis of synchronization primitives and new algorithms required to deal with some widely used 32-bit architectures.

The `LTTng` tracer probe needs, as input, a clock source to provide timestamps, trace control information to know if tracing is enabled or if filters must be applied, and the input data identified by the instrumentation. The result of its execution is to combine its inputs to generate an event written to a ring buffer.

[13] The *jiffies* counter increments at each timer tick, at a frequency typically between 100 and 1000 HZ.
[14] As stated in the `Ftrace` presentation at the Linux Foundation Collaboration Summit 2009.

In order to provide good scalability when the number of CPU increases, LTTng uses per-CPU buffers and buffer management counters to eliminate cache misses and false-sharing. It diminishes the impact of the tracer on the overall system performance. Nevertheless, cross-CPU synchronization is still required when information is exchanged from a producer to a consumer CPU.

This paper will justify LTTng's use of the RCU mechanism to synchronize control information read from the probe, local CAS and proper memory barriers to synchronize ring buffer output and present a custom trace clock scheme used to deal with architectures lacking 64-bit hardware clock source.

## 5. TRACING SYNCHRONIZATION

In this section we describe the *atomic primitives* and RCU mechanisms used by LTTng [14, 18] to deal with the constraints associated with *synchronization* of data structures while running in any *execution context*, avoiding *kernel recursion*. We then present an RCU-like trace clock infrastructure required to provide 64-bit time-base on many 32-bit architectures. The associated *performance impact* of the synchronization primitives will be studied thereafter, which will lead to the subsequent benchmark section.

### 5.1. ATOMIC PRIMITIVES

This section presents synchronization considerations for kernel data read from the tracing probe, followed by inner tracer synchronization for the control data structures read using RCU and buffer space reservation performed with atomic operations.

Because any execution context, including NMIs, can execute the probe, any data accessed from the probe must be consistent when it runs. Kernel data identified by the instrumentation site is expected to be coherent when read by every execution contexts associated with the given site. It is therefore the instrumentation site's responsibility to correctly synchronize with those kernel data structures.

Data read by the probe can be classified into two types. The first type contains global and static shared variables read from kernel memory. The second type includes data accessed locally by the processor, contained either in registers, on the thread or interrupt stack, or in per-CPU data structures when preemption[15] is temporarily disabled.

Synchronization of shared data structures is ensured by static instrumentation because the data input identification is located within the source code which carries the correct locking semantic. Conversely, dynamic instrumentation offers no guarantee that global or static variables read by the probe will be appropriately synchronized. For instance, Kprobes [13] do not export specific data at a given instrumentation site. Therefore, it does not guarantee locking other than what is being done in the kernel around the breakpoint instruction. Given that there are not necessarily any data dependency between the instruction being instrumented and the data accessed within the probe, subtle race conditions may occur if locking is not performed appropriately within the probe.

[15]User space preemption naturally occurs when the scheduler interrupts a thread executing in user space context and replaces it by another runnable thread. At kernel-level, with fully-preemptible Linux kernels (*CONFIG_PREEMPT=y*), the scheduler can preempted threads running in kernel context as well.

Local data accessed by its owner execution context, however, do not have such locking requirement because it is normally modified only by the local execution context. The probe which accesses this data executes either in the same execution context owning this data or in a trap generated by instructions within the owner context. However, compiler optimizations do not guarantee to keep local variables live at the probe execution site with Kprobes. Static instrumentation can make sure that the compiler keeps the data accessed live at a specific instruction.

Information controlling tracing behavior is accessed directly from the probe, without any consideration regarding the context in which it is executed. This information includes the buffer location, produced and consumed data counters and a flag to specify if a specific set of buffers is active for tracing. This provides flexibility so users can tune the tracer following their system's workload.

`LTTng` uses the `RCU` mechanism to manage trace-control data structure. This synchronization mechanism provides very fast and scalable data structure read access by keeping copies of the protected data structure when a modification is performed. It gradually removes an outdated data structure by first replacing all pointers to it by pointers to the new version. It keeps all data copies in place until a grace period has passed, which identifies a read-side quiescent state and therefore permits reclamation of the data structure. A `RCU` read-side is wait-free, but the write-side can block if no more free memory is available. Moreover, the write-side may either block waiting for a grace period to end, or queue memory reclamation as a `RCU` callback to execute after the current grace period. In this latter case, reclamation is performed in batch after the current grace period ends. It therefore provides very predictable read-side real-time response. Given that the trace control data structure updates are rare, this operation can afford to block. `LTTng` marks the read-side critical sections by disabling preemption because this technique is self-contained (it does not use other kernel primitives) and due to its low overhead. The `LTTng` trace-control write-side waits for readers to complete execution to provide guarantees to the trace-control caller. Therefore, when the operation *start trace* completes, the caller knows all current and new tracer probes are seeing an active trace. The opposite applies when tracing stops.

The tracing information is organized as a `RCU` list of trace structures, and is only read by the probe to control its behavior. Since the probe is executed with preemption disabled, updates to this structure can be done on a copy of the original while the two versions are presented to the probes when the list is updated: probes holding a pointer to the old structure still use the old one, while the newly executing probes use the new one. A quiescent state is reached when all processors have executed the scheduler. It guarantees that all preemption-disabled sections holding a pointer to the old structure finished their execution. It is thus safe, from that point, to free the old data structure.

With the `RCU` mechanism, the write-side must use preemptible mutexes to exclude other writers and has to wait for quiescent states. Luckily, such trace data structure updates are rare (e.g. starting a trace session), so update performance is not an issue.

Because the `RCU` mechanism wait-free guarantees apply only for the read-side, `LTTng` cannot leverage `RCU` primitives to deal with reentrancy coming from any execution context to synchronize memory buffer space reservation, which includes updating a data structure. Primitives, allowing protection from concurrent execution contexts performing buffer space reservation on the local CPU, need to execute atomically with respect to interrupts and `NMI`s, which implies that *atomic operations* must be used to perform atomic data accesses.

Given that the cross-CPU synchronization points are clearly identified and occur only when sub-buffers can pass from a producer CPU to a consumer CPU at sub-buffer boundaries, the performance impact of synchronization primitives required for each event should be characterized to find out which set of primitives are adequate to protect the tracer data structures from use in concurrent execution contexts.

On modern architectures such as Intel Pentium and above, AMD, PowerPC and MIPS, using atomic instructions, synchronized to modify shared variables in a `SMP` (Symmetric Multi-Processor) system, incurs a prohibitive performance degradation due to the synchronized variant of the instructions used (for Intel and AMD) or to the memory barriers which must be used on PowerPC, MIPS and modern ARM processors. Given that several atomic operations are often required to perform the equivalent synchronization of what would otherwise be done by disabling interrupts a single time, the latter method is often preferred. The wait-free write-side tracing algorithm used in `LTTng`[16] needs a single `CAS` operation to update the write count (amount of space reserved for writing) and an atomic increment to update the commit count (amount of information written in a particular sub-buffer).

Given that the tracing operations happen, by design, only on per-CPU data, their single-CPU atomic primitives can be safely used. This means Intel and AMD x86 do not need *LOCK* prefix to synchronize these atomic operations with concurrent CPU access, while PowerPC, MIPS and modern ARM processors do not require them to be surrounded by memory barriers to ensure correct memory order, since the only order that matters is from the point of view of a single CPU. Therefore, those lightweight primitives, faster than disabling interrupts on many architectures, can be used. Section 6 will present benchmarks supporting these claims.

## 5.2. RECURSION WITH THE OPERATING SYSTEM KERNEL

The instrumentation coverage depends directly on the amount of interaction the probe has with the rest of the kernel. In fact, the tracer code itself cannot be instrumented because it would lead to infinite probe recursion. The same applies to any kernel function used by the probe[17].

In the Linux kernel, the x86 32 and 64-bit architectures rely on page faults to populate the page table entries of the virtual memory mappings created with *vmalloc()* or *vmap()*. Since the kernel modules are allocated in this area, any access to module instructions and data might cause a minor kernel page fault. Care must therefore be taken to call the *vmalloc_sync_all()* primitive which populates all the kernel virtual address space reserved for virtual mappings with the correct page table entries between module load and use of this module at the tracing site. This ensures that no recursive page fault will be triggered by the page fault handler instrumentation.

In the context of the probe, the most important limitation regarding operating system recursion is the inability to wake up a process when the buffers are ready to be read. Instrumenting thread wake-ups provides very useful information about the inner scheduler behavior. However, instrumentation of this scheduler primitive forbids using it in the tracer probe. This problem is solved by adding a periodic timer which samples the buffer state and wakes up the consumers appropriately. Given that the operating system already executes a periodic timer interrupt to perform scheduling and manage its internal state, the performance impact of this approach is in the same order of magnitude as adding a callback to the timer interrupt. The impact on low power-consumption modes is kept small by ensuring

[16] `LTTng` kernel tracing algorithm with wait-free write-side will be presented in a forthcoming paper.
[17] A particularly unobvious example is the page fault handler instrumentation.

that these per-processor polling timers are delayed while the system is in these low-power modes. Therefore, polling is only performed when the system is active, and thus generating trace data.

As a general guideline, the probe site only touches its own variables atomically, so it requires absolutely no higher-level synchronization with the OS. On the OS side, any operation done on those shared variables is also performed atomically. It results in an hermetic interface between the probe and the kernel which makes sure the probe calls no OS primitive.

Because preemption must be disabled around probe execution, primarily to allow the `RCU`-based data structures reads, care must be taken not to use an instrumented version of the preemption disabling macros. It can be done by using the untraced implementation *preempt_disable_notrace()*.

## 5.3. TIMEKEEPING

Time-stamping must also be done by the probe. It therefore has to read a time-base. In the Linux kernel, the standard *gettimeofday()* or other clock sources are synchronized with a sequence lock (*seqlock*), which consists of a busy loop on the read-side, waiting for writers to finish modifying the data structure and checking for a sequence counter modification prior to and after reading the data structure. However, this is problematic when `NMI`s need to execute the read-side, because nesting over the write lock would result in a deadlock; the `NMI` would wait endlessly for the writer to complete its modification, but would do so while being nested over the writer. Normal use of this synchronization primitive requires interrupt disabling, which explains why it is generally correct, except in this specific case. Another issue is that the sequence lock is a blocking synchronization algorithm, because the updater threads have the ability to inhibit reader progress for an arbitrarily long period of time. Therefore, the CPU time-stamp register, when available, is used to read the time-base rather than accessing any kernel infrastructure.

Some architectures provide a 64-bit time-base. This is the case for the cycle counter read with *rdtsc* on x86 32 and 64-bit [19], the PowerPC time-base register [20] and the 64-bit MIPS [21]. A simple atomic register read permit reading a full 64-bit time-base. However, architectures like the 32-bit MIPS and ARM OMAP3 [22] only provide a 32-bit cycle counter. Other architecture which lack proper cycle counter register support must read external timers. For instance, earlier ARM processors must read the time-base from an external timer through memory mapped I/O. Memory-mapped I/O timers usually overflow every 32-bit count or even more often, although some exceptions, like the Intel `HPET` [23], permits reading a 64-bit value atomically in some modes.

The number of bits used to encode time has a direct impact on the ability of the time-base to accurately keep track of time during a trace session. A 64-bit time-base is guaranteed not to overflow for 3 thousand years at 4 GHz, which should be enough for any foreseeable use. However, at a 500 MHz frequency, typical for embedded systems, 32-bit overflows occur every 8 seconds.

Tracing-specific approaches to deal with time-stamp overflow has been explored in the past, all presenting their own limitations. The sequence lock inability to deal with `NMI` context has been presented above, although one could imagine porting the `DTrace` double-sequence lock implementation to address this problem. This approach is however slower than the `RCU` read-side and implies using a blocking sequence lock, which fails to provide good real-time guarantees.

Alternatively, an approach based on a posteriori analysis of the event sequence presented in the buffers could permit detecting overflows, but this requires a guaranteed maximum time delta between two events, which could be hard to meet due to its dependency on the workload and events traced. Low-power consumption systems with deep sleep states are good examples of such workloads. Periodically writing a resynchronization time-stamp read from a lower-frequency time-source would diminish the precision of time-stamps to the precision of the external time-source.

If, instead of writing such resynchronization event periodically, it was written in a header to the buffer containing the events, this would again either impose limits minimum event flow expected, otherwise a buffer covering a too long time period could contain undetectable 32-bit overflows. Also, given that the buffer is naturally expected to present the events in an order in which time monotonically increases, performing adjustments based on a different time-source at the buffer boundary can make time go backward because the two clocks are not perfectly synchronized. One clock going too fast could make the last buffer events overlap the time window of the following buffer. Simply using a `CAS` instruction would not solve the issue, given that the architectures we are dealing with only have a 32-bit cycle counter and are typically limited to 32-bit atomic operations.

There is already an existing approach in the Linux kernel, created initially for the ARM architecture, to extend a 32-bit counter to 63 bits. This infrastructure, named *cnt32_to_63*, keeps a 32-bit (thus atomically updated) value in memory. Its lower 31 bits are used to represent the extended counter top 31 bits. A single bit is used to detect overflow by keeping track of the low-order 32nd bit. Update is performed atomically in the reader context when a 32-bit overflow is detected. Assuming the code is run at least twice per low-order 32-bit overflow period, this algorithm detects the 32-bit overflows and updates the high-order 31-bit count accordingly. This approach has the benefit of requiring a very small amount of memory data (only 32-bit) and being fast: given the snapshot is updated on the reader-side as soon as the overflow is detected, the branch verifying this condition only needs to be taken very rarely. This approach, however, has some limitations: it only permits to keep an amount of data smaller than the architecture word size. Therefore, it is not extensible: it would not be possible to return the full 64-bit, because the top bit must be cleared to zero, and it could not support addition of `NTP` or CPU frequency scaling information. This infrastructure assumes that the hardware time-source will always appear to go forward. Therefore, with slightly buggy timers or if the execution or memory accesses are not performed in order, this would cause time to jump forward of a whole 32-bit period if the time-source appears to slightly decrement at the same time an overflow occurs. This could be fixed by reserving one more bit to also keep track of the low-order 31st bit and require the code to be called 4 times per counter overflow period. Those two bits could be used together to distinguish between overflow and underflow. This would however be at the expense of yet another high-order information bit and only permit returning a 62-bit time-base.

Therefore, a new mechanism would be welcome to generically extend these 32-bit counters to 64-bit while still allowing a time-base read from `NMI` context. The algorithm we created to solve this problem extends a counter containing an arbitrary number of bits to 64 bits. The data structure used is a per-CPU array containing two 64-bit counts. A pointer to either the first or the second array entry is updated atomically, which permits to atomically read the odd counter while the even is being updated and conversely (as shown in Figure 1). The reader atomically reads the pointer to the current array parity and then reads the last 64-bit value updated by the periodic timer. It then detects the possible overflows by comparing the current value of the time source least significant bits with the low-order bits of the 64-bit value. It returns the 64 bits corresponding to the current count, with high-order bits incremented if a low-order bit overflow is detected (as shown in Figure 2).

The algorithm for synthetic clock read-side is shown in Figure 4. At line 1, `TC_HW_BITS` is defined as the number of bits provided by the clock source, represented by a call to `hw_clock_read()`. The main limitation on the minimum number of bits required from the clock source is that it must be

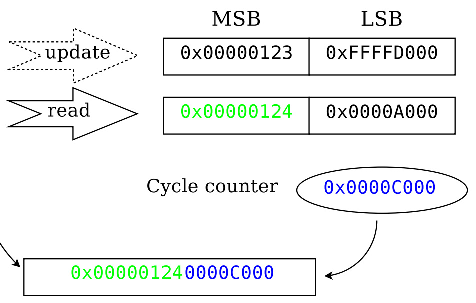

Figure 1. Trace clock read (*no* 32nd bit overflow)

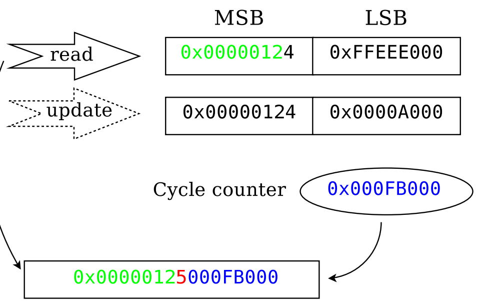

Figure 2. Trace clock read (32nd bit overflow)

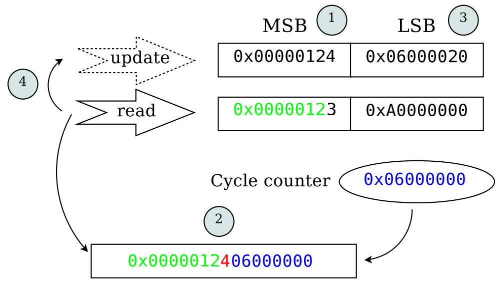

Figure 3. Trace clock update (1, 3, 4) interrupted by a read (2)

```
 1 #define HW_BITMASK            ((1ULL << TC_HW_BITS) - 1)
 2 #define HW_LS(hw)             ((hw) & HW_BITMASK)
 3 #define SW_MS(sw)             ((sw) & ~HW_BITMASK)
 4
 5 struct synthetic_tsc_struct {
 6   u64 tsc[2];
 7   unsigned int index;
 8 };
 9
10 static DEFINE_PER_CPU(struct synthetic_tsc_struct, synthetic_tsc);
11
12 static inline notrace u64 sw_tsc_read(u64 old_sw_tsc)
13 {
14       u64 hw_tsc, new_sw_tsc;
15
16       hw_tsc = (u64)hw_clock_read();
17       new_sw_tsc = SW_MS(old_sw_tsc) | hw_tsc;
18
19       if (unlikely(hw_tsc < HW_LS(old_sw_tsc)))
20         new_sw_tsc += 1ULL << TC_HW_BITS;
21
22       return new_sw_tsc;
23 }
24
25 u64 notrace trace_clock_read_synthetic_tsc(void)
26 {
27       struct synthetic_tsc_struct *cpu_synth;
28       unsigned int index;
29       u64 sw_tsc;
30
31       preempt_disable_notrace();
32       cpu_synth = &per_cpu(synthetic_tsc, smp_processor_id());
33       index = ACCESS_ONCE(cpu_synth->index);
34       sw_tsc = sw_tsc_read(cpu_synth->tsc[index]);
35       preempt_enable_notrace();
36
37       return sw_tsc;
38 }
```

Figure 4. Synthetic clock read-side

larger than the sum of timer interrupt period and maximum interrupt latency. This ensures that a timer interrupt is executed at least once per counter overflow period. Lines 2–3 present the `HW_LS()` and `HW_MS()` macros, to select the least and most significant bits of a counter, respectively corresponding to the hardware clock source and the bits counting the clock-source overflows. Lines 5–8 declare a structure containing two 64-bit `tsc` values and an index to the current `tsc` value to read. Line 10 defines a per-CPU variable, `synthetic_tsc`, holding the current `tsc` value for each processor.

The inline function `sw_tsc_read` is detailed at lines 12–23. The `notrace` keyword is a macro expanding to a gcc attribute indicating that the function must not be traced, in the unlikely event gcc decides not to inline the function. It receives as parameter the last 64-bit clock value saved in the data structure and returns the current 64-bit clock value. The current source clock value is read at line 16. The current 64-bit clock value is then derived from the old 64-bit clock most significant bits and the source clock bits. If an overflow is detected by line 19, the 64-bit clock value is incremented of the power of two value corresponding to the overflow at line 20.

Lines 25–37 show the execution context considerations taken around the execution of the trace clock read. Lines 31 and 35 disable and re-enable preemption, therefore inhibiting the scheduler during this execution phase. This ensures that no thread migration occurs, therefore ensuring local access to per-CPU data. It also ensures that the thread is not scheduled out for a long period of time between the moment it reads the index, reads the clock source and accesses the array. Long preemption between these operations could cause the current clock value to be more than a clock-source overflow apart from the previously read last 64-bit clock value when the thread resumes. To overcome such problem, the maximum duration for which this code can be interrupted is bounded by the maximum interrupt handler execution time, which must be an order of magnitude lower than the overflow period. Line 32 uses the `per_cpu` inline, a primitive which gets a pointer to the CPU-local instance of `synthetic_tsc`. `ACCESS_ONCE()` is used at line 33 to read the current index through a volatile access, which informs the compiler to treat this as an access to a memory-mapped hardware device, therefore not permitting re-fetching nor reading in multiple segments. Line 34 invokes the `sw_tsc_read()` inline explained above, which returns the current 64-bit clock value.

The update is performed periodically, at least once per overflow period, by a per-CPU interrupt timer. It detects the low-order bits overflows and increments the upper bits, and then flips the current array entry parity (as shown in Figure 3). Readers still use the previous 64-bit value while the update is done until the update completes with the parity flip.

As pointed out earlier, the read-side must disable preemption to ensure that it only holds a reference to the current array parity for a bounded amount of cycles, much lower than the periodic timer period. This upper bound is provided by the maximum number of cycles spent in this short code path increased by the worse interrupt response time expected on the system. It is assumed that no interrupt flood will hold the code path active for a whole timer period. If this assumption is eventually proven to be wrong, disabling interrupts around the algorithm execution could help not experiencing this type of problem, but delaying of timer interrupt would still leave room for overflow miss.

The update-side algorithm is detailed in Figure 5. The function `update_synthetic_tsc()` must be executed periodically on each processor. It is expected to be executed in interrupt context (therefore with preemption already disabled) at least once per overflow period. Line 6 gets a pointer to the CPU-local `synthetic_tsc`. Line 7 flips the current index back and forth between 0 and 1 at each invocation. Lines 8–9 invoke `sw_tsc_read()` to read the current 64-bit `TSC` value, using the last synthetic TSC value saved in the data structure by the previous `update_synthetic_tsc()` execution. The current 64-bit `TSC` value is saved in the free array entry, unused at that moment. Line 10 is a compiler barrier, ensuring that the index update performed on line 11 is not reordered before line 8 by the compiler. This makes sure concurrent interrupts and `NMI`s are never exposed to corrupted data.

If processors need to be kept in low-power mode to save energy, the per-processor interrupt needed to update the current 64-bit synthetic `TSC` value can be disabled in such low-power mode, replaced by a resynchronization on an external timer counter upon return to normal processor operation.

```
static void update_synthetic_tsc(void)
{
      struct synthetic_tsc_struct *cpu_synth;
      unsigned int new_index;

      cpu_synth = &per_cpu(synthetic_tsc, smp_processor_id());
      new_index = 1 - cpu_synth->index;
      cpu_synth->tsc[new_index] =
        sw_tsc_read(cpu_synth->tsc[cpu_synth->index]);
      barrier();
      cpu_synth->index = new_index;
}
```

Figure 5. Synthetic clock periodic update

The amount of data which can be placed in the per-CPU array is not limited by the architecture size. This could therefore be extended to support time-base correction for CPU frequency scaling and NTP correction. If the hardware time-source is expected to appear to run slightly backward (due to hardware bugs or out-of-order execution), the algorithm presented above could additionally check the 31st bit to differentiate between overflow or underflow in order to support non-perfectly monotonic time-sources and still keep the ability to return the full 64 bits.

Given that each read-side and write-side thread will complete in a bounded amount of cycles without waiting, this time-base enhancement algorithm can be considered as wait-free, which ensures that no thread starvation can be caused by this algorithm.

It must be understood, however, that this proposed algorithm does not replace a proper 64-bit time-stamp counter implemented by hardware. Indeed, if a faulty device holds the bus or if a driver disables interrupts for more than a cycle-counter overflow period, it would lead to time-base inaccuracy due to miss of one (or more) cycle-counter overflow. Making sure that this situation does not happen would imply reading an external clock source in addition to the cycle counter, which does not meet our efficiency constraints. Therefore, given that it is of utmost importance to be able to rely on core debugging facilities like kernel tracers, it is highly recommended to use hardware providing full 64-bit cycle counters. However, given that software must often adapt to hardware limitations rather than the opposite, the algorithm proposed should work correctly, unless some hardware or driver is doing something *really* bad like holding the bus or disabling interrupts for a few seconds.

## 6. BENCHMARKS

This section will present the benchmarks used to choose the right synchronization primitives for tracing, given their respective performance impact on many of the mainstream architectures, namely Intel and AMD x86, PowerPC, ARM, Itanium and SPARC. The goal of the present section is to show that it is possible to use local atomic operations without adding a prohibitive performance impact compared to interrupt disabling. It will be demonstrated that, on most architectures, it is even faster to use local atomic operations than to disable interrupts.

A comparison between benchmarks realized only with synchronization primitives, and with added operations within the synchronization is presented at Table I. The added operation consists in 10 word-sized reads and one word write. It shows that simple operations account for a negligible amount of cycles compared to the synchronization cost, and that costly synchronization primitives such as synchronized `CAS` are made even slower by the added operations, probably due to pipeline stalls caused by the serializing instruction. Therefore, the following benchmarks only take into account the synchronization primitive execution time. The term *speedup* is used to represent the acceleration of one synchronization primitive compared to another. Assuming cache-line effects as small compared to the synchronization cost makes combination of synchronization primitives more straightforward. The assembly listings for the following Intel Xeon benchmarks are presented in Figures 6, 7, 8 and 9.

Let's first focus on performance testing of the `CAS` operation. Table II presents benchmarks comparing disabling interrupts to local `CAS` on various architectures. When comparing the synchronization done with local `CAS` to disabling local interrupts alone, a speedup between 4.60 and 5.37 is reached on x86 architectures. On PowerPC, the speedup range is between 1.77 and 4.00. Newer PowerPC generations seems to provide better interrupt disabling performance than the older ones. Itanium, for both older single-core and newer dual-core 9050 processor, has a small speedup of 1.33. Conversely, UltraSPARC atomic CAS seems inefficient compared to interrupt disabling, which makes the latter option about twice faster. As we will discuss below, besides the performance considerations, all those architectures allow `NMI`s to execute. Those are, by design, unprotected by interrupt disabling. Therefore, unless the macroscopic impact of atomic operations becomes prohibitive, the tracer robustness, and ability to instrument code executing in `NMI` context, favors use of atomic operations.

Tables III, IV and V present the different synchronization schemes that could be used at the tracing site. Table III shows the individual elementary operations performed when taking a spin lock (busy-waiting loop) with interrupts disabled. These numbers are a "best case", because they do not consider the non-scalability of this approach. Indeed, the spin lock atomic variable must be shared between all CPUs, which leads to performance degradation when the variable must be alternately owned by different CPU's caches, a phenomenon known as cache-line bouncing.

Table IV presents the equivalent synchronization performed using a sequence counter lock and a fully synchronized atomic operation. Sequence counter locks are used in the kernel time-keeping infrastructure to make sure reading the 64-bit jiffies is consistent on 32-bit architectures and also to ensure the monotonic clock and the clock adjustment are read consistently. A synchronized `CAS` operation is needed because preemption is kept enabled, which allows migration. Therefore, given that the probe could be preempted and migrated between the moment it reads the processor ID and the moment it performs the atomic data access, concurrency between CPUs must be addressed by a SMP-aware atomic operation. If preemption is left enabled, per-CPU data would be accessed by the local CPU most of the time, so it would statistically provide a good cache locality, but, in cases where a thread is migrated to a different CPU between reading the pointer to the data structure and the write to the reserve or commit counters, we could have concurrent writes in the same structure from two processors. Therefore, the synchronized version of `CAS` and increment should be used if preemption is left enabled. It is interesting to note that ARMv7 OMAP3 shows a significant slowdown for the sequence lock. This is caused by the requirement for read barriers before and after the sequence number read due to lack of address [22] or control dependency between the sequence lock and the data to access. ARMv7 does not have weaker read-side only memory barriers and therefore requires two *dmb* (Data Memory Barrier) instructions, which decreases performance significantly.

Table V presents a RCU approach to synchronization. It involves disabling preemption around the read-side critical section, keeping a copy of the old data structures upon update and making sure the write-side waits for a grace-period to pass before the old data structure can be considered private and memory can be reclaimed. Disabling preemption, in this scheme, also has an effect on the scheduler: it ensures that the whole critical section is not preempted nor migrated to a different CPU, which permits to use the faster local CAS.

Table VI presents the overall speedup of each synchronization approach compared to the baseline: Spin lock disabling interrupts.

If we would only care about the read-side, the sequence counter lock approach is the fastest: it only takes 3-4 cycles on the x86 architecture family to read the sequence counter and to compare it after the data structure read. This is faster than disabling preemption, which takes 8-9 cycles on x86. Preemption disabling is the cost of RCU read-side synchronization. Therefore, in preemptible kernels, a RCU read-side could be slightly slower than a sequence lock. On non-preemptible kernels, however, the performance cost of RCU falls down to zero and outperforms the sequence lock. But the synchronization requirements we have also involve synchronizing concurrent writes to data structures.

In our specific tracing case, in addition to read tracing control information, we also have to synchronize for writing to buffers by using CAS to update the write counter and by using an atomic increment to keep track of the number of bytes committed in each sub-buffer. Choosing between a *seqlock* and RCU has a supplementary implication: the seqlock outperforms RCU on preemptible kernels only because preemption is left enabled. However, this implies that a fully synchronized CAS and atomic add must be used to touch per-CPU data to prevent migration.

The speedup obtained by using the RCU approach rather than the sequence lock ranges between 1.2 and 2.53 depending on the architectures, as presented in Table VI. This is why, overall, the RCU and local atomic operations solution is preferred over the solution based on read-side sequence lock and synchronized atomic operations. Moreover, in addition to execute faster, the RCU approach is reentrant with respect to NMIs. The read sequence lock would deadlock if an NMI nests over the write lock.

## 7. LEAST PRIVILEDGED EXECUTION CONTEXTS

The discussion presented above focused on tracing the kernel execution contexts. It it however important to keep in mind that different execution contexts, namely user-space, have different constraints. The main distinction comes from the fact that it is a bad practice to let user-space code modify data structures shared with the kernel without going through a system call, because this would pose a security threat and lead to potential privilege escalation.

If we were to port the tracing probe to perform user-space tracing, the trade-off would differ. The main downside of the RCU approach, for both the scheduler-based and preemptible versions, is that it requires the writer to wait for reader quiescent state before the old memory can be reclaimed. This could be a problem when exporting data from kernel-space to user-space, (e.g. time-keeping data structures) where the write-side is the kernel and the reader is user-space. When synchronizing between different privilege levels (kernel vs user-space), the highest privilege level must never wait or synchronize on the least-privileged execution context, otherwise resource exhaustion could be triggered by the lower privilege context.

## 8. CONCLUSION

As this paper has demonstrated, the current state of the art in tracing involves either instrumentation coverage limitations, synchronization flaws or limitation of the architectures supported to those which have synchronized 64-bit time-stamp counters.

A set of synchronization primitives has been proposed which fulfill the instrumentation coverage requirements of kernel tracing, adding code executed in `NMI` handler context, which was not properly handled by state-of-the-art tracers. Those primitives are the local `CAS` instruction and the `RCU` mechanism along with preemption disabling around the tracing code execution.

A wait-free algorithm, to extend a time-base providing less than 64-bit (which overflows periodically during the trace) to a full 64-bit counter by software, has been detailed. It should help tracers implement time-bases without the flaws caused by incorrect use of the sequence lock and improving the real-time guarantees compared to the sequence lock.

Finally, benchmarks have demonstrated that, on almost all architectures (except SPARC), using local `CAS` for synchronization rather than disabling interrupts is actually faster. It shows that using atomic primitives over interrupt disabling allows to grow the instrumentation coverage, including code executed from `NMI` handler context, without sacrificing performance.

This will open the door to the design of fully reentrant, wait-free, high-performance buffering schemes and to speedups in kernel primitives currently using interrupt disabling to protect their execution fast path, such as the memory allocator.

**ACKNOWLEDGEMENTS**

The authors would like to thank the Linux Trace Toolkit, Linux Kernel and SystemTAP communities for their feedback, as well as NSERC, Google, IBM Research and Autodesk for funding parts of this work.

We are indebted to Robert Wisniewski, Robert Roy, Paul E. McKenney, Bryan Cantrill, Simone Winkler and Etienne Bergeron for their constructive comments on this paper. Special thanks to David Miller, Josh Boyer, Alan D. Brunelle and Andika Triwidada for helping running benchmarks on various architectures.

**REFERENCES**

1. Bligh M, Schultz R, Desnoyers M. Linux kernel debugging on Google-sized clusters. *Proceedings of the Ottawa Linux Symposium*, 2007.
2. Cantrill BM, Shapiro MW, Leventhal AH. Dynamic instrumentation of production systems. *USENIX*, 2004. `http://www.sagecertification.org/events/usenix04/tech/general/full_papers/cantrill/cantrill_html/index.html`.
3. Corbet J. On DTrace envy. Linux Weekly News, `http://lwn.net/Articles/244536/` August 2007.
4. McKenney PE. Exploiting deferred destruction: An analysis of read-copy-update techniques in operating system kernels. `http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf` July 2004.
5. *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*.
6. *Obstruction-free synchronization: Double-ended queues as an example*, IEEE Computer Society, 2003.

7. Yaghmour K, Dagenais MR. The Linux Trace Toolkit. *Linux Journal* May 2000; URL `http://www.linuxjournal.com/article/3829`.
8. Krieger O, Auslander M, Rosenburg B, Wisniewski RW, Xenidis J, Da Silva D, al. K42: building a complete operating system. 2006; 133–145.
9. Wisniewski RW, Rosenburg B. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. *Supercomputing, ACM/IEEE Conference*, 2003. URL `http://www.research.ibm.com/K42/papers/sc03.pdf`.
10. Prasad V, Cohen W, Eigler FC, Hunt M, Keniston J, Chen B. Locating system problems using dynamic instrumentation. *Proceedings of the Ottawa Linux Symposium*, 2005. URL `http://sourceware.org/systemtap/systemtap-ols.pdf`.
11. Nataraj A, Malony A, Shende S, Morris A. Kernel-level measurement for integrated parallel performance views: the KTAU project. 2006.
12. Oprofile: A system-wide profiling tool for Linux. `http://oprofile.sourceforge.net`.
13. Mavinakayanahalli A, Panchamukhi P, Keniston J, Keshavamurthy A, Hiramatsu M. Probing the guts of kprobes. *Proceedings of the Ottawa Linux Symposium*, 2006.
14. Desnoyers M, Dagenais M. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. *Proceedings of the Ottawa Linux Symposium*, 2006.
15. Wisniewski RW, Azimi R, Desnoyers M, Michael MM, Moreira J, Shiloach D, Soares L. Experiences understanding performance in a commercial scale-out environment. *Europar*, 2007.
16. Desnoyers M, Dagenais M. Low disturbance embedded system tracing with Linux Trace Toolkit Next Generation. *ELC (Embedded Linux Conference)*, 2006.
17. Desnoyers M, Dagenais M. OS tracing for hardware, driver and binary reverse engineering in Linux 2007; :Vol. 4, No. 1.
18. LTTng website. `http://www.lttng.org`.
19. Intel Corporation. Intel 64 and IA-32 architectures software developer's manual September 2006.
20. Wetzel J, Silha E, May C, Frey B, Furukawa J, Frazier G. *PowerPC Virtual Environment Architecture, Version 2.02*. 2005. Available: `http://www.ibm.com/developerworks/eserver/library/es-archguide-v2.html` [Viewed June 7, 2009].
21. Heinrich J. MIPS R4000 microprocessor user's manual, second edition 1994.
22. ARM. *ARMv7-A and ARMv7-R Architecture Reference Manual*. 2008.
23. High Precision Event Timers (HPET) specification. `http://www.intel.com/technology/architecture/hpetspec.htm` October 2004.

Table I. Benchmark comparison between locking primitives and added inner operations, on Intel Xeon E5405

| Locking Primitive | Sync. Only (cycles) | Sync. and Operations (cycles) | Pipeline effect (cycles) |
|---|---|---|---|
| Baseline (no locking) | 1 | 60 | 0 |
| Local CAS | 8 | 60 | -7 |
| Sync. CAS | 24 | 94 | 11 |
| IRQ save/restore | 39 | 97 | -1 |
| Spin lock/unlock | 46 | 99 | -6 |
| *seqlock* | 3 | 60 | -2 |
| Preemption disable/enable | 12 | 60 | -11 |

Synchronized CAS:

```
110:   48 89 c8                mov    %rcx,%rax
113:   f0 0f b1 0d 00 00 00    lock cmpxchg %ecx,0x0(%rip)
11a:   00
11b:   ff c2                   inc    %edx
11d:   81 fa 20 4e 00 00       cmp    $0x4e20,%edx
123:   75 eb                   jne    110
```

Local CAS:

```
1e8:   48 89 c8                mov    %rcx,%rax
1eb:   0f b1 0d 00 00 00 00    cmpxchg %ecx,0x0(%rip)
1f2:   ff c2                   inc    %edx
1f4:   81 fa 20 4e 00 00       cmp    $0x4e20,%edx
1fa:   75 ec                   jne    1e8
```

Figure 6. Assembly listings for Intel Xeon benchmarks (CAS loop content).

Interrupt restore:

```
 468:   56                      push   %rsi
 469:   9d                      popfq
 46a:   ff c0                   inc    %eax
 46c:   3d 20 4e 00 00          cmp    $0x4e20,%eax
 471:   75 f5                   jne    468
```

Interrupt save (and disable):

```
 530:   9c                      pushfq
 531:   59                      pop    %rcx
 532:   fa                      cli
 533:   ff c0                   inc    %eax
 535:   3d 20 4e 00 00          cmp    $0x4e20,%eax
 53a:   75 f4                   jne    530
```

Interrupt save/restore:

```
 600:   51                      push   %rcx
 601:   9d                      popfq
 602:   9c                      pushfq
 603:   59                      pop    %rcx
 604:   fa                      cli
 605:   ff c0                   inc    %eax
 607:   3d 20 4e 00 00          cmp    $0x4e20,%eax
 60c:   75 f2                   jne    600
```

Figure 7. Assembly listings for Intel Xeon benchmarks (Interrupt save/restore loop content).

Table II. Cycles taken to execute CAS compared to interrupt disabling

| Architecture | Speedup | CAS | | Interrupts | |
|---|---|---|---|---|---|
| | (cli + sti) / local CAS | local | sync | Enable (sti) | Disable (cli) |
| Intel Pentium 4 | 5.24 | 25 | 81 | 70 | 61 |
| AMD Athlon(tm)64 X2 | 4.60 | 6 | 24 | 12 | 11 |
| Intel Core2 | 5.37 | 8 | 24 | 21 | 22 |
| Intel Xeon E5405 | 5.25 | 8 | 24 | 20 | 22 |
| PowerPC G5 | 4.00 | 1 | 2 | 3 | 1 |
| PowerPC POWER6 | 1.77 | 9 | 17 | 14 | 2 |
| ARMv7 OMAP3[a] | 4.09 | 71 | 11 | 25 | 20 |
| Itanium 2 | 1.33 | 3 | 3 | 2 | 2 |
| UltraSPARC-IIIi [b] | 0.64 | 0.394 | 0.394 | 0.094 | 0.159 |

[a]Forced SMP configuration for test module. Missing barriers for SMP support added in these tests and reported to ARM Linux maintainers.
[b]In system bus clock cycles.

Spin lock:

```
ffffffff814d6c00 <_spin_lock>:
ffffffff814d6c00:       65 48 8b 04 25 08 b5    mov    %gs:0xb508,%rax
ffffffff814d6c07:       00 00
ffffffff814d6c09:       ff 80 44 e0 ff ff       incl   -0x1fbc(%rax)
ffffffff814d6c0f:       b8 00 01 00 00          mov    $0x100,%eax
ffffffff814d6c14:       f0 66 0f c1 07          lock xadd %ax,(%rdi)
ffffffff814d6c19:       38 e0                   cmp    %ah,%al
ffffffff814d6c1b:       74 06                   je     ffffffff814d6c23 <_spin_lock+0x23>
ffffffff814d6c1d:       f3 90                   pause
ffffffff814d6c1f:       8a 07                   mov    (%rdi),%al
ffffffff814d6c21:       eb f6                   jmp    ffffffff814d6c19 <_spin_lock+0x19>
ffffffff814d6c23:       c3                      retq
```

Spin unlock:

```
spin_unlock:
ffffffff814d6f10 <_spin_unlock>:
ffffffff814d6f10:       fe 07                   incb   (%rdi)
ffffffff814d6f12:       65 48 8b 04 25 08 b5    mov    %gs:0xb508,%rax
ffffffff814d6f19:       00 00
ffffffff814d6f1b:       ff 88 44 e0 ff ff       decl   -0x1fbc(%rax)
ffffffff814d6f21:       f6 80 38 e0 ff ff 08    testb  $0x8,-0x1fc8(%rax)
ffffffff814d6f28:       75 06                   jne    ffffffff814d6f30 <_spin_unlock+0x20>
ffffffff814d6f2a:       f3 c3                   repz retq
ffffffff814d6f2c:       0f 1f 40 00             nopl   0x0(%rax)
ffffffff814d6f30:       e9 fb e1 ff ff          jmpq   ffffffff814d5130 <preempt_schedule>
ffffffff814d6f35:       66 66 2e 0f 1f 84 00    nopw   %cs:0x0(%rax,%rax,1)
```

Benchmark loop for spin_lock()/spin_unlock():

```
 140:   48 c7 c7 00 00 00 00    mov    $0x0,%rdi
 147:   ff c3                   inc    %ebx
 149:   e8 00 00 00 00          callq  ffffffff814d6c00 <_spin_lock>
 14e:   48 c7 c7 00 00 00 00    mov    $0x0,%rdi
 155:   e8 00 00 00 00          callq  ffffffff814d6f10 <_spin_unlock>
 15a:   81 fb 20 4e 00 00       cmp    $0x4e20,%ebx
 160:   75 de                   jne    140
```

Figure 8. Assembly listings for Intel Xeon benchmarks (spin lock loop content).

Sequence read lock:

```
 330:   f3 90                  pause
 332:   89 f2                  mov    %esi,%edx
 334:   48 89 c8               mov    %rcx,%rax
 337:   a8 01                  test   $0x1,%al
 339:   75 f5                  jne    330
 33b:   39 15 00 00 00 00      cmp    %edx,0x0(%rip)
 341:   75 ef                  jne    332
 343:   ff c7                  inc    %edi
 345:   81 ff 20 4e 00 00      cmp    $0x4e20,%edi
 34b:   75 ea                  jne    337
```

Preemption disabling/enabling:

```
 3f8:   ff 43 1c               incl   0x1c(%rbx)
 3fb:   ff 4b 1c               decl   0x1c(%rbx)
 3fe:   41 f6 84 24 38 e0 ff   testb  $0x8,-0x1fc8(%r12)
 405:   ff 08
 407:   0f 85 a4 00 00 00      jne    4b1
 40d:   ff c5                  inc    %ebp
 40f:   81 fd 20 4e 00 00      cmp    $0x4e20,%ebp
 415:   75 e1                  jne    3f8 <init_module+0x3e8>
[...]
 4b1:   e8 00 00 00 00         callq  4b6 <preempt_schedule>
 4b6:   e9 52 ff ff ff         jmpq   40d
```

Figure 9. Assembly listings for Intel Xeon benchmarks (sequence lock and preemption disabling loop content).

Table III. Breakdown of cycles taken for spin lock disabling interrupts

| Architecture | Spin lock (cycles) | IRQ save/restore (cycles) | Total (cycles) |
|---|---|---|---|
| Pentium 4 | 144 | 131 | 275 |
| AMD Athlon(tm)64 X2 | 67 | 23 | 90 |
| Intel Core2 | 57 | 43 | 100 |
| Intel Xeon E5405 | 46 | 39 | 85 |
| ARMv7 OMAP3[a] | 132 | 45 | 177 |

[a] Forced SMP configuration for test module.

Table IV. Breakdown of cycles taken for using a read *seqlock* and using a synchronized `CAS`

| Architecture | *Seqlock* (cycles) | Sync `CAS` (cycles) | Total (cycles) |
|---|---|---|---|
| Pentium 4 | 4 | 81 | 85 |
| AMD Athlon(tm)64 X2 | 4 | 24 | 28 |
| Intel Core2 | 3 | 24 | 27 |
| Intel Xeon E5405 | 3 | 24 | 27 |
| ARMv7 OMAP3[a] | 73 | 71 | 144 |

[a] Forced `SMP` configuration for test module.

Table V. Breakdown of cycles taken for disabling preemption and using a local `CAS`

| Architecture | Preemption disable/enable (cycles) | Local `CAS` (cycles) | Total (cycles) |
|---|---|---|---|
| Pentium 4 | 9 | 25 | 34 |
| AMD Athlon(tm)64 X2 | 12 | 5 | 17 |
| Intel Core2 | 12 | 8 | 20 |
| Intel Xeon E5405 | 12 | 8 | 20 |
| ARMv7 OMAP3[a] | 10 | 11 | 21 |

[a] Forced `SMP` configuration for test module.

Table VI. Speedup of tracing synchronization primitives compared to disabling interrupts and spin lock

| Architecture | Spin lock disabling interrupts (speedup) | Sequence lock and `CAS` (speedup) | Preempt disabled and local `CAS` (speedup) |
|---|---|---|---|
| Pentium 4 | 1 | 3.2 | 8.1 |
| AMD Athlon(tm)64 X2 | 1 | 3.2 | 5.3 |
| Intel Core2 | 1 | 3.7 | 5.0 |
| Intel Xeon E5405 | 1 | 3.1 | 4.3 |
| ARMv7 OMAP3[a] | 1 | 1.2 | 8.4 |

[a] Forced `SMP` configuration for test module.