Highly-Scalable Wait-Free Buffering Scheme for Multi-Core System Tracing

Mathieu Desnoyers
École Polytechnique de Montréal
Dept. of Computer and Software Eng.
P.O. Box 6079, Station Centre-Ville,
Montréal, Québec, Canada, H3C 3A4
mathieu.desnoyers@polymtl.ca

Michel Dagenais
École Polytechnique de Montréal
Dept. of Computer and Software Eng.
P.O. Box 6079, Station Centre-Ville,
Montréal, Québec, Canada, H3C 3A4
michel.dagenais@polymtl.ca

Dominique Toupin
Ericsson
Process Methods and Tools
Datalinjen 4 (Hus K)
Linköping, 581 12, Sweden
dominique.toupin@ericsson.com

Abstract

Diagnostic of complex problems involving the interaction between several applications and the operating systems in a distributed multi-core system with tracing requires to extract system events without disturbing its execution. Recording an event must therefore take negligible time (e.g. less than the duration of simple system call) and should not change the ordering of events (non locking).

As a result, a highly-scalable, low-overhead, lock-free trace buffering scheme was designed and proposed for the `LTTng` *tracer. It uses local compare-and-exchange operations for synchronization and resorts to* `RCU` *(Read-Copy Update) data structures for atomically updating control data. The cost associated with tracer execution has been benchmarked for different types of execution loads, on systems scaling up to 8 cores. The performance of the new trace buffering system has been measured and compared to other tracing systems such as DTrace, SystemTap and K42.*

1. Introduction

Performance monitoring of multiprocessor high-performance computers deployed as production systems (e.g. Google platform), requires tools to report what is being executed on the system. 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 information volume.

Allowing wide instrumentation coverage of the kernel code can prove to be 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 to be very low-overhead and not disturb 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` [3] tracer (available at: `http://www.lttng.org`) has been developed with these two principal goals in mind: provide good instrumentation coverage and minimize observer effect on the traced system.

This paper presents a state of the art of the existing tracing solutions along with their strengts and weaknesses in Section 2, synchronization primitives used in the buffering scheme in Section 3, a tracer architecture overview in Section 4, experimental results in Section 6 and concludes with Section 7.

2. State of the Art

In this section, we will present the state-of-the-art open source tracers. For each of these, their target usage scenarios will be presented along with the requirements imposed. Finally, we will study in detail the tracer in K42, which is the closest to LTTng requirements, explaining where LTTng brings new contributions.

DTrace [1], 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 a sequence lock to protect the clock source usage from concurrent modifications.

SystemTAP [6] provides scriptable probes which can be connected on top of Markers, Tracepoints or Kprobes [5]. 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, since 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 it shared much of SystemTAP's heavy synchronization.

Ftrace, started in 2009 by Ingo Molnar, aims primarily at kernel tracing suited for kernel developer's needs. It primarily lets 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 a binary buffer data extraction which aims at providing efficient data output. It is currently based on per-cpu busy-spinning locks and interrupt disabling to protect the tracer against concurrent execution contexts. It is currently evolving to a lockless buffering scheme.

The work on LTTng presented in this paper started back in 2006 to reach its current level of testing and safety verification.

The K42 [4] 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 simply named "trace", which was an element integrated to the kernel design per se. The systems targeted by K42 and use of lockless buffering algorithms with atomic operations are similar to LTTng.

On the design aspect, a major difference between this research-oriented tracer and LTTng is that the latter aims at being deployed 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 user-space threads to the processor on which the information is collected. Although it 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 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 formal verification performed by modeling the LTTng algorithms and using the Spin model-checker increases the level of confidence that such corner-cases are correctly handled.

3. Synchronization Primitives

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

[1]OS: Operating System
[2]NMI: Non-Maskable Interrupt

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.

4. Tracer Architecture Overview

Pursuing the objective to allow multiple analysis to be performed on a single trace collected, enabling in-depth analysis of hard to reproduce bugs, we extract trace data from the kernel.

In order to answer the various industry requirements, the architecture depicted in Figure 1 is proposed. Grey rectangles represent major phases of tracing. Within these rectangles, ellipses represent the tracing phases, linked with arrows showing the trace data flow direction. Between the tracing and post-processing phases, a dotted *Input/Output* arrow represents extraction of trace data through `I/O` mechanisms: disk, network, serial port, etc.

The *tracing* phases are performed on the traced system, using the processor, memory bandwidth and `I/O` resources required to extract the data out of the kernel. Initially, *instrumentation* is inserted into the operating system kernel. When the kernel executes and reaches an instrumentation site, it verifies if the tracing site is activated, and calls the attached probes. These probes perform all synchronization required to write events into the trace buffers.

Writing to circular memory buffers without live trace data extraction is called *flight recorder* tracing. This is one tracing mode available. The other mode consists in extracting the trace data while tracing is active. This latter phase is named *buffered data extraction*. Events are gathered in memory buffers to ensure that costly `I/O` operations are not used by the probe execution. The `I/O` phase is performed by specialized threads. It can be either done live while the trace is being recorded, or after tracing activity is over. In the latter case, only the very last buffers written will be available for analysis.

Extracting large amounts of data, albeit having a small impact on system performances, involves applying very strict implementation constraints. We deal with this at the design-level by minimizing the amount of trace data extraction synchronization required between processors and execution contexts. A zero-copy approach, involving no trace data copy between memory locations through the tracing phases, ensures efficient use of memory bandwidth.

Event reordering at *post-processing* is made possible by gathering a time-stamp value from the traced processor, written by the *probe* at each event header. The time-stamp is typically the value of a time-source synchronized across

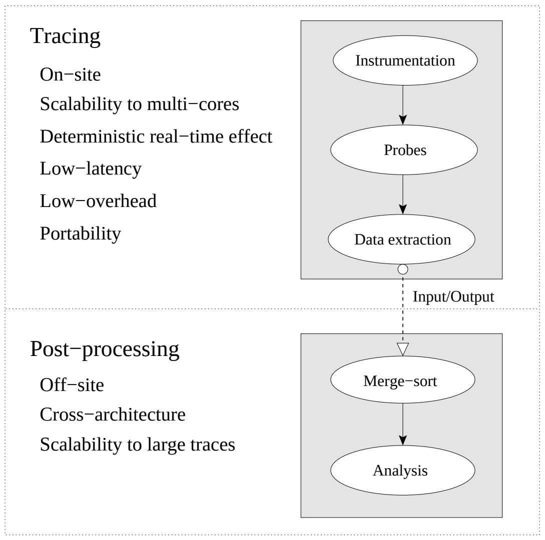

Figure 1. Tracing Phases

processors. When provided by the architecture, a cycle count register synchronized across processors can be used as time-source. This allows a posteriori reordering of events based on their time-stamps.

The *post-processing* phase can be performed either in the same environment as the traced kernel or on a completely different computer architecture. It may not be assumed that the traced and post-processing machines are the same architecture. Thus, trace data extracted must be readable by the post-processor. We propose self-described binary traces, written by the traced kernel in its native binary format, to extract compact trace data efficiently and portably.

5. Atomic Buffering Scheme

On SMP (*Symmetric Multiprocessing*) systems, some instructions are designed to update data structures in one single indivisible step. Those are called atomic operations. To properly implement the semantic carried by these low-level primitives, 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, but a special lock prefix is required before these instructions to synchronize multiprocessor access. However, to diminish performance overhead of the tracer fast-path, we remove memory barriers and use atomic operations only synchronized with respect to the local processor due to their lower overhead than those synchronized across cores. They are the only instructions allowed to modify the per-CPU data, to ensure reentrancy with `NMI` context.

The main restriction that must be observed when using such operations is to disable preemption around all access to these variables, to ensure threads are not migrated from

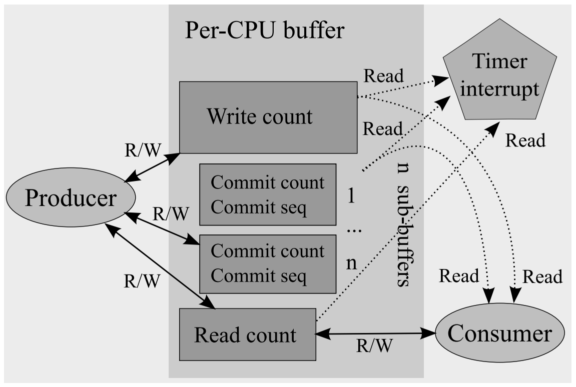

Figure 2. Producer-consumer synchronization.

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 two atomic instructions required are the `CAS` (*Compare-And-Swap*) and a simple atomic increment. Figure 2 shows the data structures being modified by those *local atomic operations*. Each per-CPU buffer has a control structure which contains the *write count*, the *read count*, and an array of *commit counts* and *commit seq* counters[3]. The counters *commit count* keep track of the amount of data committed in a sub-buffer using a lightweight increment instruction. The *commit seq* counters are updated with a concurrency-aware synchronization primitive each time a sub-buffer is filled.

A local `CAS` 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 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 `CAS` 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 if the sub-buffer size would not be 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. Holding a kernel mutex is not allowed when returning to user-space.

The consumer, `lttd`, uses two system calls, *poll()* and *ioctl()*, to control the interaction with the memory buffers, and *splice()* as a mean to extract the buffers to disk or to the 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()*.

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 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 reference to the sub-buffer about to be read is atomically exchanged 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.

Given that sub-buffer management data structures are aligned on 4 or 8-bytes multiples, we can use the lowest bit of the sub-buffer pointer to encode whether it 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.

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

- At initial time-stamp counters synchronization, done at boot-time by the operating system. This heavy synchronization, if not done by the `BIOS` (*Basic Input/Output System*), requires full control of the system.

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

- When the producer finishes writing to a sub-buffer, making it available for reading by a thread running on an arbitrary CPU. This involves using the proper memory barriers ensuring that all written 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.

6. Experimental Results

6.1. Atomic Operations

Let's first focus on performance testing of the `CAS` operation. Table 1 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.

The speedup obtained by using a `RCU` approach rather than the sequence lock ranges between 1.2 and 2.53 depending on the architectures, as presented in Table 2. 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 `NMI`s. The read sequence lock would deadlock if an `NMI` nests over the write lock.

6.2. Latency Impact

We consider the latency impact of the tracer by performing a comparative study of network response-time benchmarks in the presence and absence of tracing. We choose to measure network latency impact to characterise the tracer because it is a typical application where latency impact must be kept low. Web servers and domain name servers, which must answer queries quickly, are a good example of this application class.

We determine the tracer impact on the average network response time of a computer by measuring the packet round-trip time of 100000 *ping* echo requests. This test involves two hosts, one initiating the request and the second answering to it. The round-trip time consists in the time it takes for the packet to be generated by `ping`, sent to the network card through the operating system, sent over the network, received by the second host's operating system kernel, and sent back to the originating host through a similar route.

The repetitive nature of the test might show lower latencies than standard production systems due to the high cache locality of the workload. Hence, to make this test more representative of a real-life operating system, a workload is executed in the background, precisely to trash the processor caches and branch prediction. The chosen workload is a cache-hot Linux kernel build spread across all the machine cores.

The first latency test realized is performed in a setup minimizing the network effect where both the sender and the receiver are on the same computer, using the local host loopback interface. An 8-core Intel Xeon, clocked at 2.0 GHz is used. The number of events recorded per packet is identified by manually inspecting the recorded trace. The 95 % confidence interval for the difference between the two means found in Table 3, is $[8.88, 9.12]$ μs, which means that flight recorder tracing of 27 events adds a latency overhead on local host communication between 8.88 and 9.12 μs, with a 95 % certainty. This corresponds to an **added latency between 328 and 338 ns per event**, which is about 666 cycles. It is higher than the overhead measured with micro-benchmarks, which is 119 ns per events for this architecture. The difference between these latency results and the micro-benchmarks measurements can be attributed to processor pipeline, branch prediction and cache effects, which are higher in the latency test due to lower temporal and spacial locality than a tight loop calling the tracer.

Test	Events / round-trip	avg. (μs)	std.dev. (μs)
No tracing	–	40.0	12.8
Flight recorder tracing	27	49.0	14.3

Table 3. Tracer latency overhead for a ping round-trip. Local host, Linux 2.6.30.9, 100000 requests sample, at 2 ms interval.

Similar results, presented in Table 4 are obtained by sending *ping* echo requests from a remote host over a 100Mbps network. The number of events generated on the traced receiver side for each echo request is 7. The 95 % confidence interval for the difference between these two

Table 1. 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	4.09	71	11	25	20
Itanium 2	1.33	3	3	2	2
UltraSPARC-IIIi	0.64	0.394	0.394	0.094	0.159

Table 2. 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	1	1.2	8.4

means is $[1.56, 2.85]$ μs. Therefore, with 7 events per request, the added latency impact is between 223 and 407 ns per event, which is consistent with the measurements from the local host `ping` test. The confidence interval of network testing is much larger that the local host test due to an higher standard deviation on the measurements.

Test	Events / round-trip	avg. (μs)	std.dev. (μs)
No tracing	–	256.10	73.3
Flight recorder tracing	7	258.31	73.3

Table 4. Tracer latency overhead for a ping round-trip. 100Mbps network, tracing receiver host only, Linux 2.6.30.9, 100000 requests sample, at 2 ms interval.

Hence, the analysis of these measurements allows us to affirm that the 95 % confidence interval of the tracer latency impact on a busy system is between 328 and 338 ns per event on the Intel Xeon E5405.

6.3 Throughput

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 throughputs when running mainline Linux kernel, instrumented kernel and traced kernel.

This set of benchmarks, presented in Table 5, shows that tracing has very little impact on the overall performance under network load on a 100 Mbps network card. 8 `tbench` client threads are executed for a 120s warm up and 600s test execution. Trace data generated in flight recorder mode reaches 0.9 GB for a 1.33 MB/s trace data throughput. Data gathered in normal tracing to disk reaches 1.1 GB. 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 and 10 GB network cards can generate higher throughput, and given the 100Mbps link was the bottleneck of the previous `tbench` test, Table 6 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 worse-case scenario in terms of generated throughput kernel-wise. This workload consists in running 8 client threads and 8 server

Test	Tbench Throughput (MB/s)	Overhead (%)	Trace Throughput ($*10^3$ events/s)
Mainline Linux kernel	12.45	0	–
Dormant instrumentation	12.56	0	–
Overwrite (flight recorder)	12.49	0	104
Normal tracing to disk	12.44	0	107

Table 5. `tbench` **client network throughput tracing overhead.**

Test	Tbench Throughput (MB/s)	Overhead (%)	Trace Throughput ($*10^3$ events/s)
Mainline Linux kernel	2036.4	0	–
Dormant instrumentation	2047.1	-1	–
Overwrite (flight recorder)	1474.0	28	9768
Normal tracing to disk	–	–	–

Table 6. `tbench` **localhost client/server throughput tracing overhead.**

threads.

The kernel instrumentation, when compiled-in but not enabled, actually accelerates the kernel. It can be attributed to modification of instruction and data cache layout. Flight recorder tracing stores 92 GB of trace data to memory, which represents a trace throughput of 130.9 MB/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.

6.4. Scalability

To characterize the tracer overhead when the number of CPU increases, we need to study a scalable workload where tracing overhead is significant. The localhost `tbench` test exhibits these characteristics. Figure 3 presents the impact of *flight recorder* tracing on the `tbench` localhost workload on the same setup used for Table 6. 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.

6.5. Comparison

Benchmarks performed on `DTrace` [1], 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,

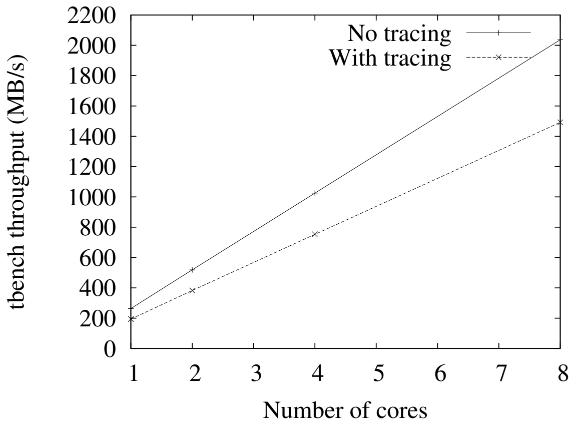

Figure 3. Impact of tracing overhead on localhost `tbench` **workload scalability.**

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 MB/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 [2]. This means `LTTng` can be used to trace workloads and diagnose problems outside of `DTrace` reach.

7. Conclusion

Overall, the `LTTng` kernel tracer presented in this paper presents 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 and scales linearly when the number of cores increases. 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 it demonstrates could be useful to other parts of the kernel, especially drivers. Using this scheme could accelerate buffer synchronization significantly and diminish interrupt latency.

References

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

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

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

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

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

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