# High Performance Tracing Tools for Multicore Linux Hard Real-Time Systems

**Raphaël Beamonte**, **Francis Giraldeau**, **Michel Dagenais**
École Polytechnique de Montréal
C.P.6079, Station Downtown, Montréal, Québec, Canada, H3C 3A7
{raphael.beamonte,francis.giraldeau,michel.dagenais}@polymtl.ca

October 1, 2012

**Abstract**

Real-time systems have always been more difficult to monitor and debug because of the real-time constraints which rule out any tool significantly impacting the system latency and performance. Tracing is often the most reliable tool available for studying real-time systems. In recent years, the real-time behavior of Linux systems has greatly improved and, with proper CPU shielding on multicore systems, it is now possible to have latencies in the low microsecond range. In that context, tracers must ensure that their overhead is within that range, predictable and scales well to multiple cores.

The recently released LTTng 2.0 toolchain has been optimized for multicore performance, scalability and flexibility. We have studied its impact on the maximum latency for serving hard real-time applications in a multicore environment using CPU shielding. In order to achieve this, we used and extended the real time verification tools cyclictest (from the rt-tests suite), and the hwlat_detector module. These tools were first used to establish the baseline of real-time system performance and then to measure the impact added by tracing with both LTTng kernel tracing and LTTng user-space tracing (UST). This identified modifications required to the buffer switch protocol in LTTng-UST, and special care required to isolate the shielded real-time cores from the RCU interprocess synchronization routines.

This work resulted in extended tools to measure the real-time properties of multicore Linux systems, a precise characterization of the real-time impact of LTTng kernel and UST tracing tools, and improvements to LTTng, and its use of RCU, for tracing real-time systems.

It will thus be easier to assess the real-time performance of multicore Linux systems. Moreover, LTTng will become a tool of choice to study the performance and behavior of such hard real-time multicore systems, given its small and deterministic impact on the maximum latency.

## 1 Introduction

Tracing is a method to study runtime behavior of a program execution. It consists in recording timestamped events at key points of the execution. Because it can be used to measure latency, tracing is a fundamental tool for debugging and profiling real-time systems. To be suitable for real-time system instrumentation, a tracer must have low-overhead and consistent maximum latency in order to minimize execution timing changes.

The Linux Tracing Toolkit next-generation (LTTng) is a high performance tracer optimized for Linux. It supports both kernel and userspace tracing with coherent timestamps, which allow to observe system-wide execution. Earlier results for LTTng-UST show that the maximum tracepoint execution delay is 300 times the average [1]. Our goal was to assess the newer version of LTTng-UST 2.0 for use in real-time system. Our contribution consists in a methodology to measure LTTng-UST tracepoint latency characteristics in a real-time environment. We measured the latency distribution in this real-time setup and compare it to results obtained on a regular setup. We developed the Non-Preempt Test (NPT) tool, to address these specific measurement requirements. In addition, we propose modifications to LTTng-UST in order to lower maximum latency

and evaluate its effectiveness.

We present related work in section 2. We detail the test environement and the methodology in section 3. Baseline results are shown in section 4 while results obtained with our proposed improvement to LTTng-UST are discussed in section 5. Future work and the conclusion are in section 6.

# 2 Related work

This section presents the related work in the two main domains relevant for this paper, real-time systems and userspace tracing.

## 2.1 Existing Real-Time validation tools

To evaluate the real-time properties of the tracer, timing properties of the test setup must be validated. It consists in measuring latencies induced by the hardware and the operating system. We used the rt-tests suite to perform the validation. In this section, the different tools corresponding to our needs are presented.

**Hardware** Abnormal hardware latencies can occur in misconfigured hardware. To measure these, we used the hwlat_detector kernel module [2]. This module uses the `stop_machine()` kernel call to hog all of the CPUs during a specified amount of time. It then polls the CPU timestamp counter (TSC) for a configurable period and looks for the discrepancies in the TSC data. If there is any gap, it means that the polling was interrupted which, on the system configuration used, could only be an SMI. The tool hwlatdetect is a python script to simplify the use of the hwlat_detector module.

**Software** Cyclictest is a tool to verify the software real-time performance by running multiple processes on different CPUs, executing a periodic task [3]. Each task can have a different period. The priority of each process can be set to any value up to real-time. The performance is evaluated by measuring the discrepancy between the desired period and the real one.

There exists also the preempt-test tool [4]. This tool is is not part of the rt-tests suite but was analyzed before the development of the Non-Preempt Test tool presented in 4.1. It allows to verify if an higher priority task is able to preempt a lower priority one by launching threads with increasing priorities. It also measures the time it takes to preempt lower priority tasks.

## 2.2 Existing tracers

In this section, we presents characteristics of currently available tracers.

Some existing implementations of tracers rely on either blocking system calls, string formatting or achieve thread-safety by locking the shared resources for concurrent writers. For example, the logging framework `Poco::Logger` is implemented this way[1]. This category of tracer is slow and unscalable, and thus is unsuitable for use in real-time and multi-core environment.

Feather-trace [5] is a low-overhead tracer implemented with thread-safe and wait-free FIFO buffers. It uses atomic operations to achieve buffer concurrency safety. It has been used to analyze locking in the Linux kernel. However, it do not support variable event size, since the reservation mechanism is based on array indexes. Also, the timestamp source is the `gettimeofday()` system call, that provides only microseconds precision instead of nanosecond.

Paradyn modifies binary executables by inserting calls to tracepoints[6]. The instrumentation can be done at runtime or using binary rewriting in order to introduce only a low-overhead. This technique has been used to monitor malicious code. While the framework offers extensive API to modify executables, it does not include trace buffer management, event types definition or trace write mechanisms. Therefore, missing components must be implemented separately.

SystemTap is a monitoring tool for Linux[7]. It works by dynamically instrumenting the kernel using KProbes. The instrumentation is done in a special scripting language that is compiled to produce a kernel module. The analysis of the data is bundled inside the instrumentation itself and results are printed on the console at regular interval. Hence, the analysis is done in-flight and there are no facilities, as far as we know, to efficiently serialize raw events to stable storage.

LTTng-UST provides macros to add statically compiled tracepoints to a program. Produced events are consumed by an external process that writes them to disk. Contrarily to Feather-trace, it sup-

[1] Poco logging framework web site: http://pocoproject.org/

ports arbitrary event types through the Common Trace Format[2]. The overall architecture is designed to deliver extreme performance. It achieves scalability and wait-free properties for event producers by allocating per-CPU ring-buffers. In addition, control variables for the ring-buffer are updated by atomic operations instead of locking. Moreover, important tracing variables are protected by read-copy update (RCU) data structures to avoid cache-line exchanges between readers occurring with traditional read-write lock schemes[8]. A similar architecture is available at the kernel level. Since both kernel and userspace timestamps use the same clock source, events across layers can be correlated at the nanosecond scale. The rest of this paper focuses on the LTTng-UST 2.0 implementation.

Table 1 summarizes the LTTng components versions used to perform our experiments. The source code has been downloaded from the git repositories of project web site[3].

| Component | Git hash version |
|---|---|
| Userspace RCU | e1259cb1 |
| LTTng kernel modules | dae90c28 |
| LTTng tools | 26c9d55e |
| LTTng-UST | a8909ba5 |

**TABLE 1:** *Versions of LTTng components used for the experiment*

# 3 Test environment

We used the tools presented previously to validate our test setup. The system consists of an Intel® Core™ i7 CPU 920 2,67 GHz, with 6 GB of DDR3 RAM at 1 067 MHz and an Intel DX58SO motherboard. Hyperthreading was disabled as it introduces unpredictable delays withing cores by sharing resources between threads, both in terms of processing units and in terms of cache. This is something to avoid in real-time systems.

As expected, running hwlatdetect to verify the hardware latency did not find any problem; it measured no latencies for a duration of one hour. Hwlatdetect often allowed us to find unexpected latencies on particular setups in our initial studies.

The cyclictest tool was then used to verify the software latency. Even if the documentation of rt-tests specifies that cyclictest has been developed primarily to be used in a stressed environment, we were particularly interested in the verification of the idle system. Real-time and non-real-time operating systems shouldn't have big differences in this case. We performed the test on the two different kernels used in the rest of this paper, which are the debian Linux kernel 3.2.0-3-amd64 (package version 3.2.21-3), hereinafter refered as standard kernel, and the debian Linux kernel 3.2.0-3-rt-amd64 (package version 3.2.23-1), hereinafter refered as the PREEMPT_RT patched kernel. We choose to do our tests on both of these kernels to compare the performance of LTTng in a non-real-time environment versus a hard real-time one. We also expect that if LTTng is able to reach very good performance on a non-optimized system, it will most likely be able to reach it on a real-time one.

Table 2 shows the results of the cyclictest execution on these kernels, performed with command line arguments -S, to activate the standard options to test an SMP system, -p99, to set the priority to real-time, and -l100000, to get a sample for 100 000 cycles.

| | Latencies in $\mu s$ | | | | |
|---|---|---|---|---|---|
| Thread id | 0 | 1 | 2 | 3 | Kernel |
| Interval | 1000 | 1500 | 2000 | 2500 | type |
| Minimum | 1 | 1 | 1 | 1 | std |
| | 1 | 1 | 1 | 1 | rt |
| Average | 2 | 2 | 2 | 2 | std |
| | 2 | 2 | 3 | 2 | rt |
| Maximum | 16 | 15 | 25 | 753 | std |
| | 7 | 7 | 7 | 9 | rt |

**TABLE 2:** *Results of the cyclictest executions performed on our standard (std) and PREEMPT_RT patched (rt) kernels*

The results obtained shows latencies up to 25 $\mu s$ for three of the four threads for the standard kernel. The fourth shows a latency more than 30 times higher than the other threads. The results are better on the PREEMPT_RT patched kernel. The maximum latency reached is 9 $\mu s$ where it was 753 $\mu s$ on the other. We also see that the maximum of the worse thread of the PREEMPT_RT patched kernel is lower than the lowest maximum of the standard kernel (almost twice lower). The PREEMPT_RT patched kernel should be able to handle real-time problems better than the standard kernel.

[2]Common Trace Format specification repository: git://git.efficios.com/ctf.git
[3]LTTng git repositories are available on: http://git.lttng.org

# 4 Baseline results

In this part, we present the performance of LTTng in our test environment. To do so, we first introduce the Non-Preempt Test tool that was developed for this purpose and then present and discuss our latency results.

## 4.1 The Non-Preempt Test tool

One condition we wanted to test was the non preemption of an high priority process. To do so, we developed the Non-Preempt Test application, or NPT[4]. After setting an ideal environment by disabling IRQ and locking the process memory into RAM to prevent it from being swapped (with `mlockall`), the core of the application is to do loops and calculate the time gap between the start of two consecutive cycles using the `rdtsc` instruction to get the Time Stamp Counter [9] of the CPU, like the hwlat_detector module in kernel mode. In an ideal situation, this time gap will be very short – just the time to execute the few instructions in each cycle. At the end of its execution, NPT computes latencies statistics for each loops, and an histogram showing the different latencies reached and the number of times each one was reached. The NPT tool was designed to be executed in a CPU shielded environment and to be alone on its CPU. Our configuration for the CPU shielding puts all the system processes on cpu0 and NPT on cpu1, as shown in Figure 1.

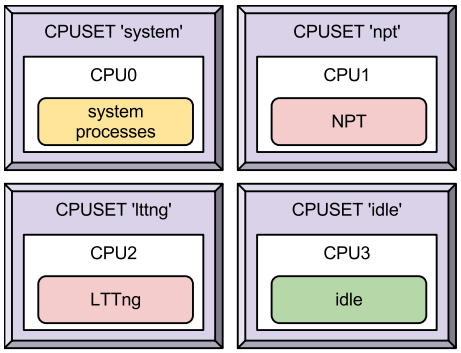


**FIGURE 1:** *The `cpusets` organisation for the running tests*

The `rdtsc` time source is a precise counter and its frequency is fixed. Even if it is not synchronized between cores, this does not affect our experiment because NPT sets is own CPU affinity (with `sched_setaffinity`) to be scheduled on the same CPU at all time. Moreover, this is reinforced by the CPU shielding. In order to reduce the effect of transient state, NPT also uses an empty loop to stress the CPU before getting its frequency as presented in [10]. It allows to remove any effect of the frequency scaling even if it is not disabled. However, the effect of the Intel® Turbo Boost Technology is not managed yet. We then discard the first five – this number is configurable – iterations of the benchmark. The study of the pipeline warm-up latency is beyong the scope of this paper.

This tool is ideal to test the performance of the kernel and UST LTTng tracers as it is easy to extend and add tracepoints in the main loop, while identifying any added latency from the tracer, as shown in Pseudocode 1. The session daemon of LTTng is put on cpu2 during the tracing tests to be CPU independent from NPT and the system processes. The session daemon spawns consumer daemons and thus they will also run on cpu2.

```
1:  i ← 0
2:  t_0 ← read rdtsc
3:  t_1 ← t_0
4:  tracepoint nptstart
5:  while i ≤ cycles_to_do do
6:      i ← i + 1
7:      duration ← (t_0 − t_1) ∗ cpuPeriod
8:      tracepoint nptloop
9:      CalculateStatistics(duration)
10:     t_1 ← t_0
11:     t_0 ← read rdtsc
12: end while
13: tracepoint nptstop
```

**PSEUDOCODE 1:** *Tracepoints in NPT*

## 4.2 Latency results

Figure 2 presents the histograms generated by NPT for an execution with 100 000 000 loops without tracing. As we can see, there is no latency peak.

[4]The Non-Preempt Test tool can be downloaded at *http://git.dorsal.polymtl.ca/?p=npt.git*

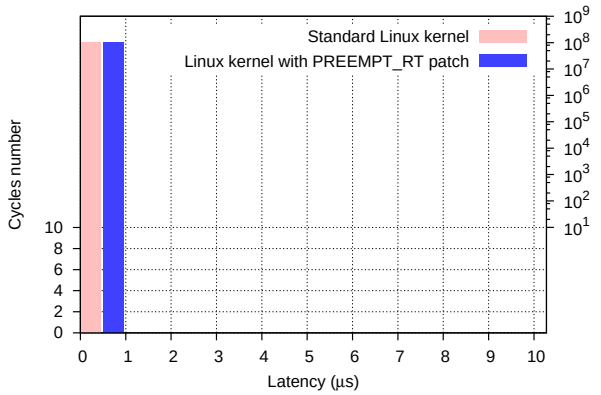


**FIGURE 2:** *Histograms generated by NPT for $10^8$ cycles on standard and PREEMPT_RT patched kernels*

Figures 4, 3 and 5 present the generated histograms for executions of NPT with 100 000 000 cycles with respectively kernel, UST, and kernel and UST tracers active.

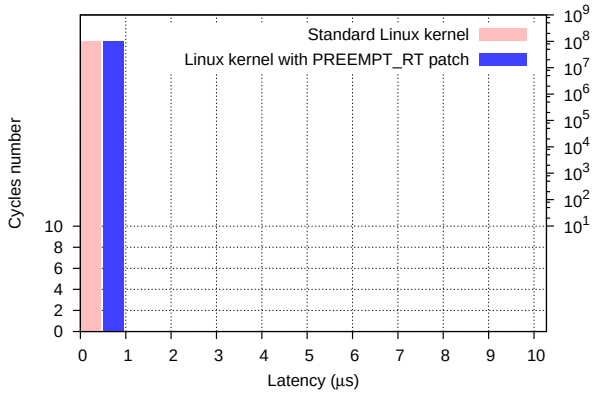


**FIGURE 3:** *Histograms generated by NPT for $10^8$ cycles on standard and PREEMPT_RT patched kernels with LTTng kernel tracing*

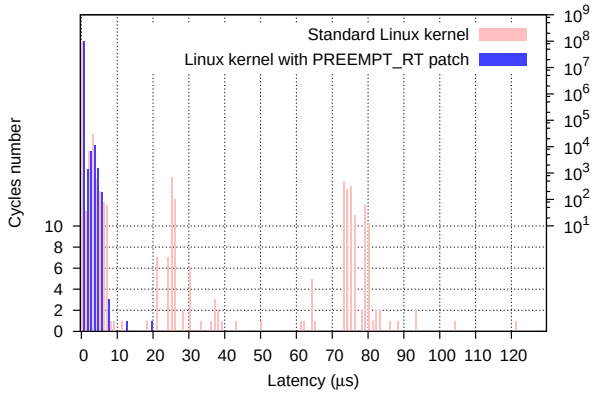


**FIGURE 4:** *Histograms generated by NPT for $10^8$ cycles on standard and PREEMPT_RT patched kernels with LTTng-UST tracing*

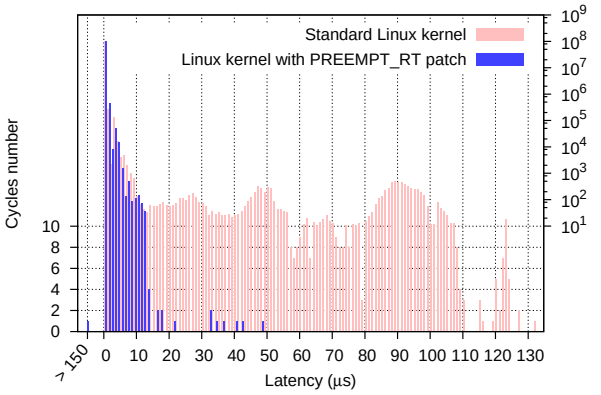


**FIGURE 5:** *Histograms generated by NPT for $10^8$ cycles on standard and PREEMPT_RT patched kernels with LTTng kernel and UST tracings*

As we can see, the usage of LTTng-UST adds many non-deterministic peaks to the execution of NPT, up to 121 $\mu s$ on the standard kernel and 19 $\mu s$ on the PREEMPT_RT patched one. On both kernels, using the kernel tracing alone doesn't seem to have any impact on the execution of NPT. Latency peaks show that the impact is more important on the UST side, maybe because there is an UST tracepoint directly added into the loop. As these peaks were also visible in the execution of NPT with both kernel and UST tracers, we used this trace to analyze the execution of NPT on cpu1. Doing so, we identified that when the UST implementation was using the control pipe of the UST consumer to inform it that the sub-buffer it was feeding was full, even if the write call was a non-blocking call, a kworker thread, which has a lower priority, took lead for a short amount of time before NPT returned back to its execution.

# 5 Reducing maximum latency

The results presented in the previous section led us to modify LTTng-UST to create a test version in which the synchronization between the application and the consumer is removed. Instead of using the kernel polling call, we changed it to active polling for the sake of this experimentation. Using active polling, the consumer will continuously try to empty the buffers and thus run at 100% of the CPU, but

with our shielded environment, it should not have any impact on the NPT execution. For its part, the application will not contact the consumer anymore to inform it of the sub-buffers state.

This LTTng-UST version is still in prototyping phase but already shows promising results at this state of development. Figures 6 and 7 show the difference of added latencies between the original and the modified version of LTTng-UST on a standard and a PREEMPT_RT patched kernel respectively.

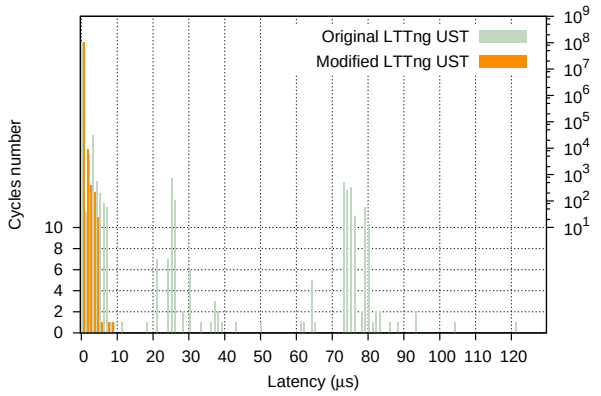


**FIGURE 6:** *Histograms generated by NPT for $10^8$ cycles on a standard kernel with original and modified LTTng-UST tracing*

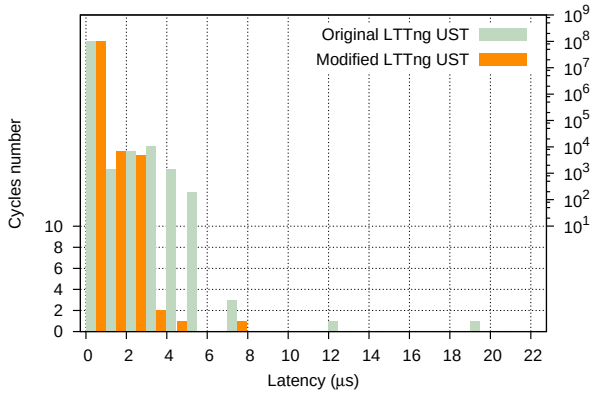


**FIGURE 7:** *Histograms generated by NPT for $10^8$ cycles on a PREEMPT_RT patched kernel with original and modified LTTng-UST tracing*

On the standard kernel, the maximum latency is lowered from 121 $\mu s$ to 8 $\mu s$, where on the PREEMPT_RT patched kernel, it is lowered from 19 $\mu s$ to 7 $\mu s$. If we compare the results of the modified LTTng-UST on both kernels in Figure 8, we can see that, contrary to the original LTTng-UST results shown in Figure 4, these are much more constant.

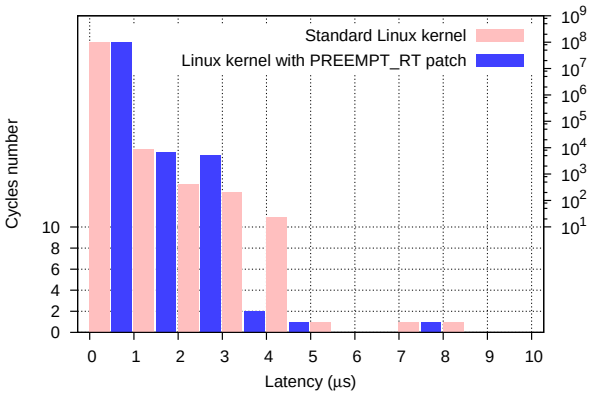


**FIGURE 8:** *Histograms generated by NPT for $10^8$ cycles on standard and PREEMPT_RT patched kernels with modified LTTng-UST tracing*

Moreover, Table 3 shows the statistics obtained from the execution of NPT for the original and modified version of LTTng for comparison purposes. We can see that even if the minimum and the mean durations are higher with our modified version, the maximum duration, the variance and the standard deviation, which are the most important values in a real-time system, are lower.

| | Latencies in *ns* | | | |
|---|---|---|---|---|
| Kernel | std | | rt | |
| LTTng | orig | mod | orig | mod |
| Minimum | 287 | 270 | 289 | 452 |
| Mean | 317 | 381 | 322 | 466 |
| Maximum | 121 744 | 8 104 | 19 837 | 7 312 |
| Variance | 74,778 | 0,321 | 1,813 | 0,438 |
| Deviation | 273 | 17,92 | 42,58 | 20,94 |

**TABLE 3:** *Statistics per cycles, in nanoseconds, generated by NPT on both standard (std) and PREEMPT_RT patched (rt) kernels for both the original (orig) and modified (mod) versions of LTTng-UST*

# 6 Conclusion and Future Work

We have presented the effects of tracing with LTTng on both standard and PREEMPT_RT patched kernels by using the Non-Preempt Test (NPT) application. We changed the way the userspace instrumented application interacts with LTTng userspace tracer (UST) to reduce and improve the determinism of the added latency. Our results are promising and show that we are in the right direction as the

maximum latencies are within 8 $\mu s$ for the standard kernel and 7 $\mu s$ for the PREEMPT_RT patched one, but we believe there is still room for improvement.

We believe that LTTng has great potential in tracing real-time systems. Therefore, we are viewing the real-time work described in this paper as the beginning of a large project in which collaborations and contributions are welcome. We intend to finish the implementation of our modified LTTng-UST version, to pursue our investigations to find if we can lower the LTTng latency, and then integrate our changes upstream. The latest version of NPT can be obtained from *http://git.dorsal.polymtl.ca/?=npt.git*.

# 7 Acknowledgments

The authors are grateful to Yannick Brosseau, Matthew Khouzam, Mathieu Desnoyers and Geneviève Bastien for the reviews, useful comments, and help to work on the project. This research is supported by OPAL-RT, CAEm the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Consortium for Research and Innovation in Aerospace in Québec (CRIAQ).

# References

[1] M. Desnoyers and M.R. Dagenais. *The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux.* In Linux Symposium, Ottawa, Ontario, Canada, June 2006.

[2] *Documentation/hwlat_detector.txt*, in the PREEMPT_RT patch. Available: *http://kernel.org/pub/linux/kernel/projects/rt/.*

[3] RT Wiki. Cyclictest [online]. Available: *https://rt.wiki.kernel.org/index.php/Cyclictest* (consulted on Oct., 1st 2012). May 2012.

[4] RT Wiki. Preemption Test [online]. Available: *https://rt.wiki.kernel.org/index.php/Preemption_Test* (consulted on Oct., 1st 2012). May 2012.

[5] B. Brandenburg and J. Anderson. *Feather-Trace: A lightweight event tracing toolkit.* In Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, pp. 19-28, 2007.

[6] A.R. Bernat and B.P. Miller. *Anywhere, any-time binary instrumentation.* In Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, pp. 9-16, 2011.

[7] F.C. Eigler and R. Hat. *Problem solving with systemtap.* In Proceedings of the Ottawa Linux Symposium, 2006.

[8] M. Desnoyers, P.E. McKenney, A.S. Stern, M.R. Dagenais and J. Walpole. *User-Level Implementations of Read-Copy-Update.* In Parallel and Distributed Systems, IEEE Transactions on, vol. 23, num. 2, pp. 375382, 2012.

[9] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, December 2009, 253669-033US

[10] Aby Thankashan. High Performance Time Measurement in Linux [online]. Available: *http://aufather.wordpress.com/2010/09/08/high-performance-time-measuremen-in-linux/* (consulted on Oct., 1st 2012). Sept. 2010.