Low Disturbance Embedded System Tracing with Linux Trace Toolkit Next Generation

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

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

Abstract

Embedded systems in applications that include GPS, airplanes, life support devices, video decoders and many others are often required to meet hard and soft real-time constraints. Various proprietary operating systems, like VxWorks and μC/OS-II, aim primarily at allowing programs to have an higher bound on their execution time. They can be deployed in embedded applications where response time must absolutely be met.

Due to the growing interest in the Linux operating system for embedded applications in the past years, some work has been realized to make Linux satisfy time constraints. It is currently directed towards Ingo Molnar's real-time preemption patch[1].

This paper presents how the kernel tracer "Linux Trace Toolkit Next Generation" (LTTng)[2] can be used to pinpoint time constraints problems in the Linux kernel and validate time constraints by showing the absence of failure in a given sample.

This article is based on a real life problem and illustrates the usefulness of the extensibility built into LTTng. Providing a low lock and interrupt disturbance tracing of the kernel, LTTng will be used to find the root cause of missing timer clock ticks.

[1] http://people.redhat.com/~mingo/realtime-preempt/

[2] http://ltt.polymtl.ca

1 Introduction

Embedded systems that need to satisfy soft and, especially, hard real-time constraints find interest in kernel tracers for validation of their specifications.

Some tracers are currently used for embedded Linux : Linux Trace Toolkit (LTT)[4], Linux Kernel State Tracer (LKST) and Kernel Function Trace (KFT)[1], previously known as Kernel Function Instrumentation (KFI).

KFT is a function entry/exit tracer that generates a high throughput. It does give the complete function call flow in the kernel at the cost of a high throughput generated for an approximate 27% preformance impact on the system[1], depending of the fequency at which the functions are called.

LTT and LKST instrument various key events in the kernel that gives a good insight about what is going on in various subsystems: duration of interrupts, scheduling, bottom half interrupts and more. It generates useful informa-

tion to understand how the Linux operating system works globally with a lower impact on the system than FKT. Both the LTT and LKST approaches have limitations regarding their timestamp precision. Experience has also shown that LTT does not facilitate the addition of instrumentation and analysis due to its monolithic design.

Linux Trace Toolkit Next Generation (LTTng) is a complete redesign of LTT, building on its instrumentation, reusing the RelayFS[5] kernel to user space relay and enhancing k42 lockless tracing mechanism[3]. Its primary goals are to provide precise, low disturbance, highly reentrant tracing. As a supplementary goal, it aims at simplifying extension of its instrumentation. Linux Trace Toolkit Viewer is LTTng trace analysis tool counterpart. It is designed to support large traces, offer precise analysis and be easily extensible. This new toolkit is the fruit of a redesign work that started in 2003[2].

The focus of this article is be to present how LTTng can be used in the real-time embedded field to narrow time constraints problems in the kernel and in real-time applications.

2 Problem presentation

The information that is interesting as a starting point is already generated by the standard LTTng instrumentation: we look at the periodicity of timer interrupts. An event *irq_entry* is saved each time an interrupt routine is entered. If we only consider the ones related to IRQ 0 (timer interrupt on a x86), we can obtain the following figure (Figure 1), which consists in the interval between those interrupts.

The system is set to have a 250HZ clock, a 4ms interval is thus expected. However, Figure 1 shows a clear result: some interrupts are missed at the beginning and at the end of the trace. In fact, the number of missed timer ticks goes between 2 and 3, for a timer interval reaching 15ms, which is 3.75 times higher than the normal 4ms.

If we look at the detailed event list around the missing timer IRQs at Figure 2, we see that while the NMI interrupts generated by the NMI watchdog should be fired before each timer IRQ, it is executed three times between two consecutive timer interrupts. The logical cause of this problem is having some code that disables interrupts for a long time.

3 Instrumentation

The question that arises is: which code disables interrupts for such a long period? The following steps lead to the answer.

3.1 Event description

First, we add an event description for interrupt disabling events in the locking facility[3]. See the *locking.xml* file in the *ltt-control* package at `http://ltt.polymtl.ca` for their exact XML description. Here is an overview of the described events with their associated data recorded in the trace.

- irq_save : Disable IRQs, saving state. The instruction pointer and flags are saved.
- irq_restore : Restore IRQs from previous flags. The instruction pointer and restored flags are saved.
- irq_disable : Disable IRQs. The instruction pointer is saved.

[3] A facility is a group of events, loadable dynamically.

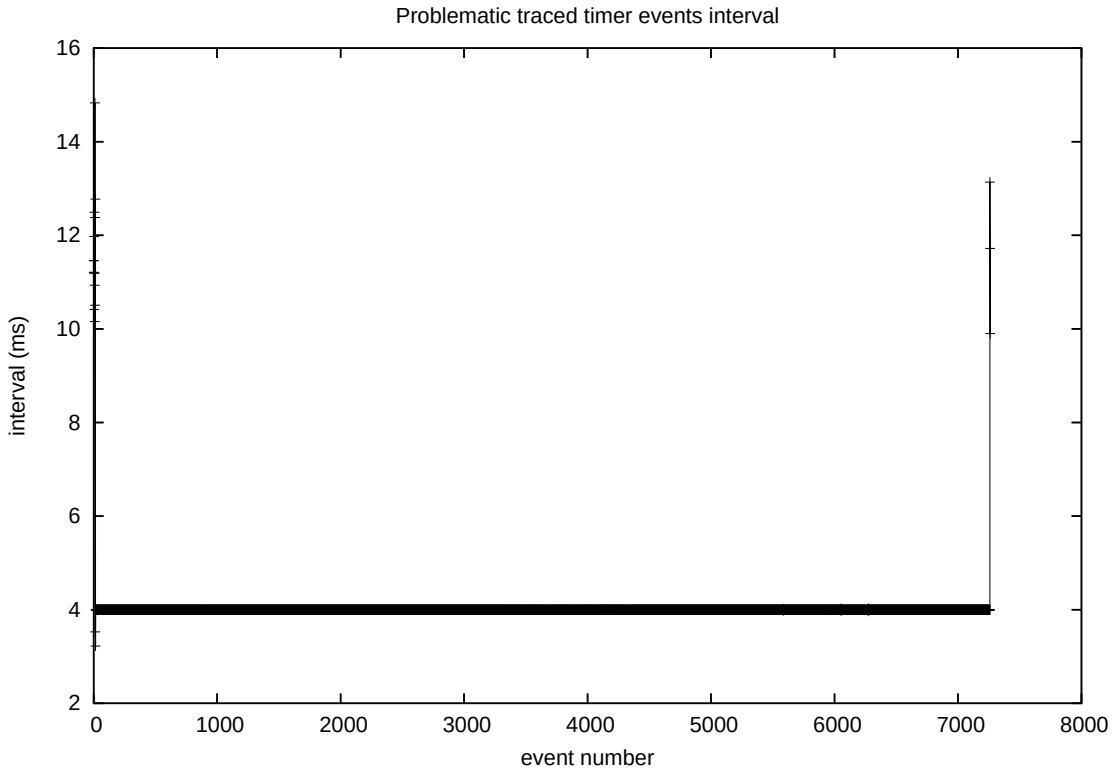

Figure 1: Problematic traced timer events interval

- irq_enable : Enable IRQs. The instruction pointer is saved.

This XML file must be placed in the *$prefix/share/ltt-control/facilities* directory of the traced system.

3.2 Addition of tracing in kernel code

The following step is to create the tracing headers from the XML event description with the *genevent* package. It is suggested to work in a temporary directory and then copy the facility "loader" c file and header in the ltt directory of the LTTng patched kernel sources. The two other headers go in *include/linux/ltt*.

The last header, *ltt-facility-locking.h*, is then ready to be included in any kernel file and have its inline functions called for logging in each active trace, as shown in Table 1.

3.3 Workaround header circular inclusion

However, the IRQ and low-level locking (spinlocks, rwlocks, seqlocks, ...) cannot themselves include the LTTng header because of header circular inclusion. Even if LTTng does not use the low-level locking primitives, it is the case for RelayFS, with which LTTng shares its data structures.

This is why, in this specific case, a workaround must be performed by creating a C file from where the inline functions will be called. Each of these functions will export a symbol that will be called at the irq_* instrumentation site.

The functions look like the code snippet in

Table 2. Once declared extern in *asm-i386/system.h*, these functions can be called like the code snippet at Table 3.

This will have no effect if the *LOCKING* facility is not selected in the kernel configuration, but will call the instrumentation function with the address of the calling function, as the block scoped label will be directly embedded in the function that calls the macro. The same is done for the irq enable, disable and restore macros.

4 Trace analysis

Once the instrumented kernel is recompiled, booted and a trace is taken, the analysis can begin. In fact, it becomes almost trivial to find the offending function at this time.

First, the output legibility is improved if we apply a filter to select the events we are interested in. Using the filter expression shown at Figure 3, the LTTV analyzer will show exactly what we want: interrupt entry for the timer, traps for the NMI watchdog and the locking facility, which we just created.

Then, we see, in the event details from Figure 4, that the interrupts are disabled at address 0xc012e30d just before the missing interrupts. They are then restored at address 0x012e503. Giving these addresses to *ksymoops -A* gives us the faulty function : `release_console_sem` (from kernel/printk.c). The erroneous code is shown in listing from Table 4.

This makes sense: some code regions at the beginning and at the end of the LTTng tracing code call printk and the test machine happens to have a serial console active.

Figure 5 shows what happens if we create a module that calls printk in a loop, therefore exaggerating the problem. We see that the timer interval is completely changed : the majority of timer ticks have a timer interval of 10ms, some are executed twice (this is how the Linux kernel acts when it detects a missing timer interrupt) and some have a timer interval over 12ms.

This is confirmed by the following test : Figure 6 shows the result of the test with serial console deactivated, which now have correct 4ms ($\pm$ interrupt latency) interval, even with a gcc and a find executed while the trace is taken.

5 Conclusion

As this article has pointed out, adding new instrumentation to LTTng is straightforward. Due to its atomic operations based algorithm, LTTng does not disturb normal kernel locking. It therefore enables it to trace not only non-maskable interrupts, but also critical code segments like the *system.h* header.

The same recipe used for interrupt disabling instrumentation could be applied without much change to spinlocks, reader-writer locks, seqlocks and others. Only preemption deactivation tracing should be addressed carefully, because LTTng disables preemption around its RCU list usage.

Some interest has been shown to port LTTng to the Real-Time Linux kernel, which would mainly consist in putting some *#ifdef* to select the RT specific version of *rcu_read_lock* instead of disabling explicitly preemption (which must be reenabled without calling the scheduler for reentrancy purpose).

Considering the extensibility of the LTTng tracer and the LTTV trace viewer, there would be benefits to port the Kernel Function Tracer (KFT) instrumentation to the LTTng tracer. This would make KFT benefit of system wide

information (like interrupt handlers instrumentation) to help interpreting function entry/exit events. Some parts of KFT, like bootup tracing, should be integrated to LTTng to reach the same level of functionality.

References

[1] Tim Bird. Learning the kernel and finding performance problems with kfi. In *CELF International Technical Conference*, 2005.

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

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

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

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

Event	Facility	Time (s)	Time (ns)	PID	Event Description
trap_entry	kernel	93	587024793	0	kernel.trap_entry: 93.587024793 (/cpu_0), 0, 0, 0x0, TRAP { trap_id = 2, address = 0xc029c2a4 }
trap_entry	kernel	93	591023837	0	kernel.trap_entry: 93.591023837 (/cpu_0), 0, 0, 0x0, TRAP { trap_id = 2, address = 0xc029c2a4 }
trap_entry	kernel	93	595022400	0	kernel.trap_entry: 93.595022400 (/cpu_0), 0, 0, 0x0, TRAP { trap_id = 2, address = 0xc029c2a4 }
irq_entry	kernel	93	595484140	0	kernel.irq_entry: 93.595484140 (/cpu_0), 0, 0, 0x0, IRQ { irq_id = 0, mode = kernel }
irq_entry	kernel	93	595492616	0	kernel.irq_entry: 93.595492616 (/cpu_0), 0, 0, 0x0, IRQ { irq_id = 4, mode = kernel }
trap_entry	kernel	93	599024128	0	kernel.trap_entry: 93.599024128 (/cpu_0), 0, 0, 0x0, TRAP { trap_id = 2, address = 0xc029c2a4 }
trap_entry	kernel	93	603021871	0	kernel.trap_entry: 93.603021871 (/cpu_0), 0, 0, 0x0, TRAP { trap_id = 2, address = 0xc029c2a4 }
irq_entry	kernel	93	606688454	0	kernel.irq_entry: 93.606688454 (/cpu_0), 0, 0, 0x0, IRQ { irq_id = 0, mode = kernel }
irq_entry	kernel	93	606695853	0	kernel.irq_entry: 93.606695853 (/cpu_0), 0, 0, 0x0, IRQ { irq_id = 4, mode = kernel }
trap_entry	kernel	93	607022569	0	kernel.trap_entry: 93.607022569 (/cpu_0), 0, 0, 0x0, TRAP { trap_id = 2, address = 0xc029c2a4 }

Figure 2: Missing timer events

Figure 3: Filter expression

Trace	Tracefile	CPUID	Event	Facility	Time (s)	Time (ns)	PID	Event Description
/tmp/trace1	/cpu	0	irq_save	locking	93	584828013	0	locking.irq_save: 93.584828013 (/cpu_0), 0, 0, 0x0, SYSCALL { EIP = 0xc012edaa, flags = 582 }
/tmp/trace1	/cpu	0	irq_restore	locking	93	584831083	0	locking.irq_restore: 93.584831083 (/cpu_0), 0, 0, 0x0, SYSCALL { EIP = 0xc012ef7b, flags = 582 }
/tmp/trace1	/cpu	0	irq_save	locking	93	584831403	0	locking.irq_save: 93.584831403 (/cpu_0), 0, 0, 0x0, SYSCALL { EIP = 0xc012e30d, flags = 642 }
/tmp/trace1	/cpu	0	trap_entry	kernel	93	587024793	0	kernel.trap_entry: 93.587024793 (/cpu_0), 0, 0, 0x0, TRAP { trap_id = 2, address = 0xc029c2a4 }
/tmp/trace1	/cpu	0	trap_entry	kernel	93	591023837	0	kernel.trap_entry: 93.591023837 (/cpu_0), 0, 0, 0x0, TRAP { trap_id = 2, address = 0xc029c2a4 }
/tmp/trace1	/cpu	0	trap_entry	kernel	93	595022400	0	kernel.trap_entry: 93.595022400 (/cpu_0), 0, 0, 0x0, TRAP { trap_id = 2, address = 0xc029c2a4 }
/tmp/trace1	/cpu	0	irq_restore	locking	93	595482585	0	locking.irq_restore: 93.595482585 (/cpu_0), 0, 0, 0x0, SYSCALL { EIP = 0xc012e503, flags = 642 }
/tmp/trace1	/cpu	0	irq_entry	kernel	93	595484140	0	kernel.irq_entry: 93.595484140 (/cpu_0), 0, 0, 0x0, IRQ { irq_id = 0, mode = kernel }
/tmp/trace1	/cpu	0	irq_save	locking	93	595489961	0	locking.irq_save: 93.595489961 (/cpu_0), 0, 0, 0x0, IRQ { EIP = 0xc0138a83, flags = 150 }
/tmp/trace1	/cpu	0	irq_restore	locking	93	595490069	0	locking.irq_restore: 93.595490069 (/cpu_0), 0, 0, 0x0, IRQ { EIP = 0xc0138ac5, flags = 150 }
/tmp/trace1	/cpu	0	irq_disable	locking	93	595490929	0	locking.irq_disable: 93.595490929 (/cpu_0), 0, 0, 0x0, IRQ { EIP = 0xc0157f68 }
/tmp/trace1	/cpu	0	irq_save	locking	93	595491659	0	locking.irq_save: 93.595491659 (/cpu_0), 0, 0, 0x0, SYSCALL { EIP = 0xc01388c1, flags = 70 }

Time Frame start: 93 s 459831403 ns end: 93 s 709831403 ns Current Time: 93 s 584831403 ns

Figure 4: Event list of new instrumentation (interrupt enabling/disabling events)

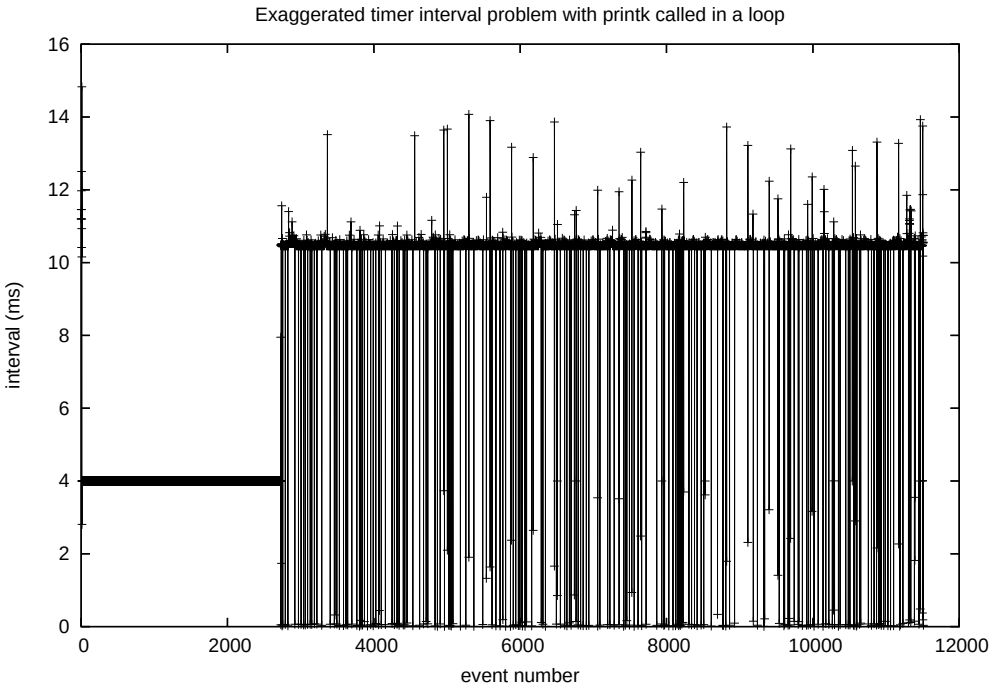

Figure 5: Printk called in a loop

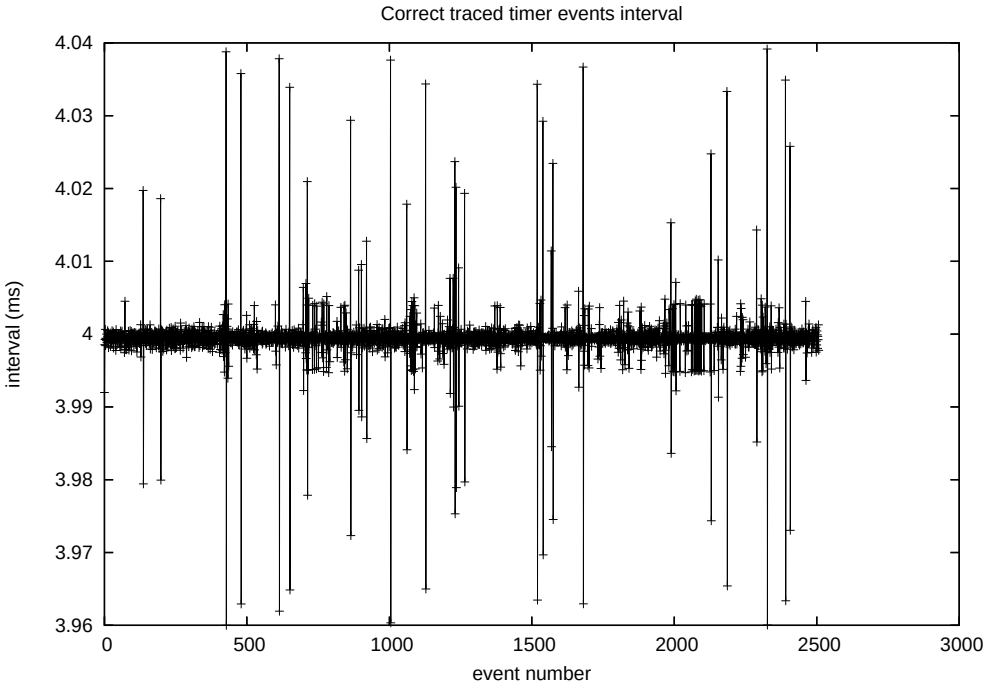

Figure 6: Correct timer interval, with disabled serial console

```
#include <linux/ltt/ltt-facility-locking.h>

[...]

void somefunction(void)
{
  void *address = 0x555;
  unsigned long flags = 0x0;

  trace_locking_irq_save(address, flags);
}
```

Table 1: Simple instrumentation invocation

```
#include <linux/ltt/ltt-facility-locking.h>

void _trace_locking_irq_save(const void * lttng_param_EIP,
    unsigned long lttng_param_flags)
{
  trace_locking_irq_save(lttng_param_EIP, lttng_param_flags);
}

EXPORT_SYMBOL(_trace_locking_irq_save);
```

Table 2: Locking specific instrumentation call site

```
/* For spinlocks etc */
#define _local_irq_save(x)  __asm__ __volatile__ \
    ("pushfl ; popl %0 ; cli":"=g" (x): /* no input */ :"memory")

#ifdef CONFIG_LTT_FACILITY_LOCKING
#define local_irq_save(x) do { \
  __label__ address;\
address: \
  _local_irq_save(x); \
  _trace_locking_irq_save(&&address,x); \
} while(0)
#else
#define local_irq_save _local_irq_save
#endif //CONFIG_LTT_FACILITY_LOCKING
```

Table 3: irq_save instrumentation

```
kernel/printk.c: release_console_sem:
[...]
  for ( ; ; ) {
    spin_lock_irqsave(&logbuf_lock, flags); <-----------
    wake_klogd |= log_start - log_end;
    if (con_start == log_end)
      break;      /* Nothing to print */
    _con_start = con_start;
    _log_end = log_end;
    con_start = log_end;    /* Flush */
    spin_unlock(&logbuf_lock);
    call_console_drivers(_con_start, _log_end);
    local_irq_restore(flags); <-----------
  }
```

Table 4: Interrupts disabled for too long in printk