xmtrace - the XtratuM tracer

Thomas Hisch <t.hisch@gmail.com>
Georg Schiesser <e0307201@student.tuwien.ac.at>
Andreas Platschek <andi.platschek@gmail.com>
Opentech EDV Research GmbH
Liechtensteinstrasse 31, A-2130 Mistelbach, Austria

Abstract

A Tracer is a tool used to record the chronological sequence of events in order to analyze the functional and temporal behaviour of software while the system is running. In contrast to using a Debugger there is no need to stop at breakpoints. Instead, previously inserted tracepoints trigger writes into a buffer, including both timestamp and location of the occurrence. Tracing can be useful to debug applications while minimizing the distortion on the system's real-time behaviour. Therefore it is a well-established technique especially in the field of real-time systems.

This paper proposes a tracing tool for XtratuM based on the tracer included in RTLinux/GPL. xmtrace uses statically allocated shared memory to buffer timestamp, address and type of events. Both hard-coded system events and user-defined events are supported. xmtrace provides adequate tracepoints in the XtratuM code base for critical functions like scheduling, enabling/disabling interrupts, and acquiring/freeing spinlocks. Custom events can be added and triggered by the user via a simple interface. Events are grouped into event classes. Single events and event classes can be enabled and disabled as required, which makes the Tracer very flexible.

In this paper design and implementation of xmtrace are described, including adequate tracing points and problems of debugging real-time applications. Finally, standard usage scenarios are evaluated and comparative benchmarks interpreted.

1 Introduction

XtratuM [1, 2] is a open source nanokernel, working as a hypervisor, allowing to run multiple domains in parallel on uniprocessor systems. These domains can either be non-real-time Linux domains, or real-time domains (e.g. partikle).

Basically Xtratum virtualizes Interrupts and Timers which allows it to take over control of the whole system.

When complex real-time applications are implemented, often the single components are done by different teams. Usually, it is not too complicated to get a single real-time task running and debug the functional behaviour of this single task. However it becomes much more complicated, when you have to put the system together and run several real-time tasks in parallel. This is exactly the situation where a tracer can be of really high value. The tracer allows you to find out the sequence of events in your system, while minimizing the impact on the timing.

There are two main reasons why a tracer is an essential part of an hypervisor like XtratuM. First, to make XtratuM easier to use for more complex applications, it is necessary to provide a suitable set of tools which allow the user to debug his application. And secondly, the tracer can also be used to trace Xtratum itself, making it easier for developers to improve it and to implement new features.

The following chapter gives a short comparison of different debugging techniques (i.e. debugger and tracer) and their tasks for the debugging of real-time applicaions are pointed out.

After that the advantages and disadvantages of a tracer over a debugger are given. Finally, the implemantation of xmtrace is explained and some numbers are given to give you an idea how big the impact of using a tracer is.

2 What is tracer?

This section compares different debugging techniques like software debugger (e.g. gdb) and tracer (e.g.

LTT). At the end of this section it should be clear what a tracer is, and what you can expect of a tracer.

A tracer is a tool used to record the chronological sequence of events, in order to analyze the functional and temporal behaviour of software while the system is running.

This is achievd by collecting timestamps, location and data at predefined points in the code (so called tracepoints). The focus of the tracer definitly is on the temporal behaviour. Therefore the temporal impact to the execution of the Software has to be as low as possible. The collected data is therefore very limited (e.g. in xmtrace to one `long` value), because collecting big amounts of data takes time, and therefore increases the impact on the temporal behaviour.

An example for a very well known tracer in the Linux community is the LTT (Linux Tracer Toolkit) project [3]. It offers the possibility to acquire execution traces from Linux 2.6.9 and later and includes graphical tools to process and visualize the acquired data. The LTT also offers the possibility to trace an RTAI (Real Time Application Interface) [4] patched kernel.

In contrast to a tracer, a debugger stops execution when reaching a breakpoint. This means the program (with JTAG-debuggers even the whole system) stops to execute, allowing the user to read out the memory, watch variables and continue execution stepwise. Stepping through a program using a debugger has the big advantage, that you can collect far more data than with a tracer, and that you are able to step through the program, until a error is discovered. At this point you can investigate the system allowing you to find the cause of the failure (i.e. the fault).

This form of debugging is very valuable to find functional bugs, but as mentioned above, it completly destroys the temporal behaviour, and therefore is not practicable to find bugs which arise from temporal failures.

The prime example of a debugger in the Linux world is the GDB (The GNU Project Debugger) [5]. Special features are that GDB allows you to connect to remote targets and debug those via serial or network connections. There are also many graphical interfaces are available for the GDB, making it easier to debug your application.

There are also possibilities to connect GDB to e.g. openocd [6] which connects GDB with a JTAG debugger and allows to debug an embedded system via GDB the same way as you are used to debug a normal software application.

The problem of debugging real-time applications, is that setting a breakpoint and stopping the execution of the program, changes the timing of the application completely. For real-time applications the correctness in the time domain is as important as the correctness in the value domain.

Because of this, stopping the application does not make sense. The big advantage of a tracer compared to a debugger is therefore, that it does not require to stop the application, and only writes a small amount of data in a fast accessible memory region. This is done on predefined points in code - so called tracepoints.

3 implementation of xmtrace

xmtrace is based on the RTLinux Tracer[1] with few updates related to buffer management. It consists of loadable kernel modules and user space programs which are using the *mbuff* Shared memory device driver to communicate with kernel space. The xmtrace kernel component is split up into two modules. One module (`xmtrace.ko`) initializes the Shared memory where all the trace records and the eventinfos are stored whereas the other module is directly compiled into the core XtratuM module `xm.ko`.

The `xmtrace.ko` module allocates Shared memory for the trace structure shown in Figure 1 and Shared memory for the eventinfo structure which is used by the user space *tracer* program to map event IDs to event names. After this is done it calls a function which completes the initialization and starts the tracing process.

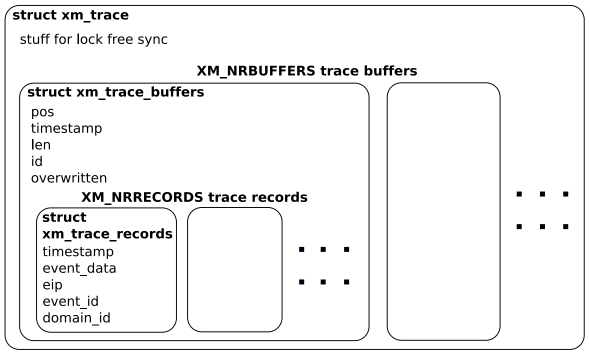

FIGURE 1: *shm layout*

Each tracepoint in the code is assigned to an Event. Some of the available events are listed in Table 1. For a list of all events see `xm_tracer.h`.

[1]written by Michael Barabanov

eflags	`XM_TRACE_HARD_CLI`
	`XM_TRACE_HARD_STI`
	`XM_TRACE_HARD_SAVE_FLAGS`,...
Interrupts	`XM_TRACE_INTERCEPT`
	`XM_TRACE_INTERCEPT_EXIT`,...
System Calls	`XM_TRACE_SYSCALL_ENTRY`
	`XM_TRACE_SYSCALL_EXIT`,...
Scheduler	`XM_TRACE_SCHED_IN`
	`XM_TRACE_SCHED_OUT`
	`XM_TRACE_SCHED_CTX_SWITCH`
User	`XM_TRACE_USER`
Domain	`XM_TRACE_DOMAIN_USER`

TABLE 1: *some trace events*

In general after `xmtrace.ko` has been loaded all events which are not masked will be traced, which means the trace records will be written into one of the `XM_NRBUFFERS` buffers shown in Figure 1. If the buffer is full with records the oldest records will be overwritten like in a circular buffer until a `XM_TRACE_FINALIZE` event occurs which is a special event that sets the `DATA_READY` bit for the current buffer and switches to the next free buffer. Buffers with a set `DATA_READY` bit can be read by the user space *tracer* program which takes care of printing all the trace-records per buffer in chronological order according to the timestamps of the trace-records.

The core tracer module which is part of `xm.ko` only consists of the `xm_do_trace` function and some other functions which can be used in XtratuM domains. `xm_do_trace` needs access to interrupt- and time-functions defined in the XtratuM core so that it will not be preempted while storing a trace record in one of the trace buffers. This is the main reason why this module is directly compiled into the XtratuM core.

A trace point is set with the `xm_trace (event-id, eventdata, eip)` or the simpler `xm_trace2 (eventid, eventdata)` function, which takes care of setting the `EIP`[2], where eventid is one of the macros in Table 1 and eventdata is a `long` value.

`xm_trace` points at `xm_do_trace` after the `xmtrace.ko` module has completed initialization. In the cleanup function of the module `xm_trace` is set that it points at a function with an empty body.

xmtrace adds some system calls to the syscall table in XtratuM shown in Figure 2 so that tracepoints can also be set and new events defined in XtratuM domains.

[2]Instruction Pointer

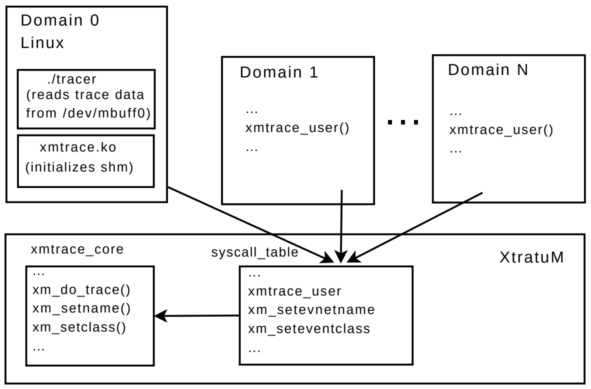

FIGURE 2: *structure of xmtrace*

4 Examples

This section presents an example that can be found in the xmtrace branch of the xtratum_devel tree in `user_tools/examples/xmtrace`. The example creates a new tracing event called xm-domain and then goes into a for-loop which is run through 100 times, every time producing a trace event.

Of Course the output of this example is too much for this paper, so from here we will work with a small piece, shown in table 2.

As you can see in table 2, the output of the tracer consists of 5 columns. The first one contains the number of the domain that produced this event (where -1 indicates that the trace event happend during xtratum core activities), the next column contains the relative time difference to the last tracing event, after that follows the name of the tracing event (e.g. scheduler in, scheduler out, hw_save_flags_and_cli,...). Then there is one long value showing the data passed to the tracer by the tracing event, followed by the EIP pointer, telling us, where in the code the tracing event was produced.

D 1	168	scheduler out	0x1	<0xce873408>
D 1	174	hw_restore_flags	0x86	<0xce873410>
D 1	3551	sycall entry	0	<0xce873f4f>
D 1	400	sycall entry	0	<0xce873fa0>
D 1	2597	sycall entry	0	<0xce873eac>
D 1	170	hw_save_flags_and_cli	0x82	<0xce872e9a>
D 1	275	hw_save_flags_and_cli	0x93	<0xce872d99>
D 1	178	user	0x13	<0xce872dc6>
D 1	275	hard sti	0	<0xce872e42>
D 1	503	sycall entry	0	<0xce873f2a>
D 1	181	hw_save_flags_and_cli	0x286	<0xce87489a>
D 1	545	hw_restore_flags	0x286	<0xce8748eb>
D 1	247	hard cli	0	<0xce872e57>
D 1	200	hw_restore_flags	0x6	<0xce872e85>
D 1	184	hw_restore_flags	0x82	<0xce872ec9>
D 1	348	xm-domain	0	<0xce872479>

TABLE 2: *example output*

The output in table 2 only shows adresses in the last columns. Of course these adresses can be resolved into symbols. To do this, we need two scripts, both located in `xtratum_devel/xmtrace`.

First a system map has to be build. This is done with the script `createmap.sh`, which by default generates a system map named `MySystem.map`. The `addr2sym` script then uses the system map to exchange the address by the name of the function plus the offset. The result of resolving the symbols in our example is shown in table 3.

D 1	168	scheduler out	0x1	proc_calc_metrics+0x38
D 1	174	hw_restore_flags	0x86	proc_calc_metrics+0x40
D 1	3551	sycall entry	0	set_timer_sys+0x4f
D 1	400	sycall entry	0	set_timer_sys+0xa0
D 1	2597	sycall entry	0	get_time_sys+0xc
D 1	170	hw_save_flags_and_cli	0x82	sync_events+0x1a
D 1	275	hw_save_flags_and_cli	0x93	sync_domain_events+0x59
D 1	178	user	0x13	sync_domain_events+0x86
D 1	275	hard sti	0	sync_domain_events+0x102
D 1	503	sycall entry	0	set_timer_sys+0x2a
D 1	181	hw_save_flags_and_cli	0x286	write_printkbuf+0x8a
D 1	545	hw_restore_flags	0x286	xmf_read+0x3b
D 1	247	hard cli	0	sync_domain_events+0x117
D 1	200	hw_restore_flags	0x6	sync_events+0x5
D 1	184	hw_restore_flags	0x82	sync_events+0x49
D 1	348	xm-domain	0	system_call_handler_asm_0x82+0x15

TABLE 3: *example output*

This example can also be modified to measure the impact of the tracepoints. We bascially have to distinguish between calls in the xtratum core and calls from the domains via the syscall.

To measure the impact you can just make two trace events directly after each other, and you get the time difference between the two timestamps which basically is the impact of the call.

On a 1.6GHz AMD Duron with 256MB of RAM this time span was 163-177ns for trace events in the core and 312-731ns for calls of the xmtrace_user syscall. These are just rough numbers to give an idea of the range how long one tracepoint takes.

5 Conclusion and Future Work

With xmtrace, XtratuM got an important feature for core developers as well as for application developers. It is a useful tool making temporal debugging a lot easier. The disadvantage is, that it is only possible to extract one long value of data, which makes it more or less useless for functional debugging, which definitly is no area of application for xmtrace. The impact of xmtrace on the temporal behaviour was kept as small as possible, allowing it to produce pretty accurate timings.

Future goals include the improvement of the userspace tools to allow better data processing and to port xmtrace to XtratuM 2, which will be released in the not too far future.

References

[1] M. Masmano, I. Ripoll, A. Crespo, Introduction to XtratuM *http://www.xtratum.org/doc/papers/xtratum_whitepaper.pdf*

[2] M. Masmano, I. Ripoll, A. Crespo, An Overview of the XtratuM nanokernel *http://www.xtratum.org/doc/papers/xtratum_overview_OSPERT2005.pdf*

[3] Karim Yaghmour and Jean-Hugues Deschênes, *Linux Trace Toolkit Reference Manual*

[4] RTAI - The Real Time Application Interface *www.rtai.org*

[5] GDB User Manual *http://sourceware.org/gdb/current/onlinedocs/gdb_toc.html*

[6] Open On-Chip Debugger *http://openfacts.berlios.de/index-en.phtml?title=Open_On-Chip_Debugger*