Software Tracing of Embedded Linux Systems using LTTng and Tracealyzer

Dr. Johan Kraft, Percepio AB

Debugging embedded software can be a challenging, time-consuming and unpredictable factor in development of embedded systems. Detecting errant program execution begs the question “How did the software reach this state?” What combination of inputs and timing resulted in the error, and why?

Tracing can often provide the answer.

Tracing entails recording software behaviour during runtime, allowing for later analysis of collected trace data. Tracing is most often a development bench activity, but tracing can also be enabled for production use, continuously active to record behaviours and catch errors post-deployment. Production tracing can be an effective technique for detecting rarely-manifested errors that are therefore are difficult to reproduce in a debugger. These can include situations where the system responds more slowly than expected, gives incorrect or suboptimal output, freezes up or crashes.

Tracing can be performed either in hardware (in the processor) or in software. Hardware-based tracing generates a detailed instruction-level execution history, without disturbing the analysed system. The downside is that hardware trace requires a processor, board and debugger with explicit trace support. Thus, hardware trace support must be considered very early, already when selecting the hardware platform for the project. Moreover, most hardware trace solutions don’t allow for recording data, only control-flow, i.e. the executed code.

Software-based tracing focuses on selected events, such as operating system calls, interrupt routines and updates of important variables. This does not require any special hardware and can even be deployed in shipped products like a “black box” flight recorder used in aviation. Moreover, software trace allows for storing relevant data together with the events, such as system call parameters.

The downside of software tracing is that it uses the target system processor and RAM for storing the events. But since each event usually only takes some microseconds to store, a software-traced system still typically executes at around 99% of normal speed. Moreover, the extra processor time used by tracing can be compensated by better means for optimizing the software.

Another issue with software-based tracing is the “probe effect”, i.e., the theoretical impact on software behaviour due to the timing impact of introducing software-based tracing. However, the timing effects are small and timing effects of similar magnitude can easily be caused by common changes in the software. It is however possible to eliminate the probe effect completely by always keeping the recording active, i.e., even in the production code. This way, the trace recording becomes part of the integrated, tested system. A bonus of this approach is that traces can be saved automatically by error handling code, which can greatly facilitate post-mortem analysis.

Tracing is especially important for systems integrating an operating system. A central feature of operating systems is multi-threading - the ability to run multiple programs (threads) on a single processor core by rapidly switching amongst execution contexts. Multi-threading is very practical for
embedded software where multiple periodical activities needs to run at different rates, e.g., in a control system, or when time-critical functions needs to be activated on certain events, preempting other less urgent activities. Multi-threading, however, makes software behaviour more complex, and affords the developer less control over run-time behaviour as execution is pre-empted by the OS.

**Tracing Linux Systems using LTTng**

LTTng\(^1\) is the leading solution for software-based tracing in Linux. LTTng is open source and supported by most Linux distributions and by Yocto, the very common build system for embedded Linux. LTTng is very efficient, proven in use and supports Linux kernels from version 2.6.32. Kernels from v2.6.38 are supported without any kernel modifications.

LTTng is based on *tracepoints*, function call placeholders that are inactive by default. An inactive tracepoint has a minimal performance impact, only a few clock cycles. When LTTng is activated, it connects the tracepoints to an internal LTTng function that stores the event in a RAM buffer.

Trace data in the RAM buffer can be continuously flushed to disk or offloaded to another system over a network connection. The flushing is handled by user-space threads. Another option is to keep the trace data in RAM using a ring buffer, i.e., overwriting earlier events when the buffer becomes full. In this mode, a snapshot is saved on demand, containing the latest events.

LTTng provides two trace recorders. The *kernel tracer* records the thread scheduling, system calls, IRQs, memory management, and other kernel-level activities, utilizing existing tracepoints in the Linux kernel. The *user-space tracer* (LTTng-UST) allows for generating custom events from user-space code, i.e., by adding new tracepoints.

![Figure 1. Transparent tracing of function calls using wrapper functions and LD_PRELOAD.](image)

Although LTTng is based on software instrumentation, it does not require recompiling the target source code. The kernel already contains tracepoints at strategic locations, and by using another Linux feature it is possible to trace selected user-space function calls without modifying source code. This is done by creating a shared object file with *wrapper functions* (Figure 1) containing tracepoints. The shared object file is then specified in LD_PRELOAD when launching the application. This impacts the dynamic linking and makes the application call the wrapper functions instead of the original function. The wrapper
functions records the event using the tracepoint(s) and then calls the original function. Thus, the function wrapping is completely transparent to application code, with no need for recompilation. On the first call of a wrapper function, it looks up the address of the original function and stores it in a function pointer for use in later calls.

**Analysis of LTTng traces using Tracealyzer**

LTTng outputs the trace recordings in an open format called Common Trace Format. Since this is a binary format, a tool is required for analysis. The LTTng tool Babeltrace can convert the trace data to text files, but it is hard to “see the big picture” from vast amounts of trace data in text format. A visualization tool greatly facilitates analysis since the human brain is much better at spotting patterns in images than in text data.

*Tracealyzer* is a family of trace visualization tools developed by Percepio AB, a Swedish research spin-off company founded in 2009. Tracealyzer provides a large set of graphical perspectives to facilitate trace analysis and is available for several embedded operating systems, including Linux, VxWorks, FreeRTOS, SafeRTOS, Micrium µC/OS-III, SEGGER embOS and RTXC Quadros. *Tracealyzer for Linux* is designed to visualize LTTng trace data and supports the current LTTng v2.x as well as older versions of LTTng.

The main trace view in Tracealyzer (Figure 2) displays the execution of threads along a vertical time-line, with various events (e.g., system calls) shown using colour-coded labels. Labels can be filtered in several ways and their placement is automatically adjusted to avoid overlapping. Label background colour indicates status and type of operation, e.g., red labels show system calls that block the calling thread and green labels show where blocking system calls return to the caller. Custom application events from the user-space tracer (LTTng-UST) can be configured to appear either as service calls (e.g., malloc) or as “user events”, i.e., generic debug messages (yellow labels).

Tracealyzer is much more than a basic viewer. It understands and highlights dependencies among related events in trace data, for instance sending and receiving of a semaphore signal. This makes it easier to understand operating system behaviour, e.g., why some threads are blocked and others triggered.

An example is shown in Figure 2, where a blocking “write” call is highlighted. This call generates two LTTng events, when the call begins (entry event) and when the call returns (exit event). Since the call blocked the thread, the two events are separated by context-switches and other events. Tracealyzer understands that these events are related and highlights both events (blue outline) when one is selected. The entry event (“write(FD-1) blocks”) tells that the blocking of the calling thread “demo.out: 5133” was caused by a “write” operation on FD-1, i.e. File Descriptor 1 which is Standard Output. The thread became ready to execute almost immediately (“Actor Ready: demo.out: 5133”) but execution did not resume until 69 µs later (“write(FD-1) returns after 69 µs”).
The main view is supported by more than 20 other graphical views showing other perspectives of the trace, such as a CPU usage graph showing the total system load and each thread over time. Other views show statistics on thread timing, kernel blocking, scheduling intensity, inter-process communication and communication dependencies between threads (see Figure 3). Since a trace often contain overwhelming amounts of repeating scenarios of less interest, the many views provided by Tracealyzer gives different perspectives that makes it easier to find interesting parts, for instance when a system call fails or when a thread takes longer than normal to complete.
Application events are shown as yellow labels in the main view (user events), but can also be shown in a separate log window that provides an overview of the general behavior of the application, e.g., updates of important state variables. If numeric data is included in application logging, e.g., buffer usage, control signals or sensor inputs, this can be plotted. This can be regarded as a software logic analyzer, useful in many types of development. Moreover, the data points in the plots are linked to the main view, so by double-clicking on any data point, the main view is synchronized to display the corresponding event.

Most views are interconnected in a similar way. The main view is linked to supporting views, and these are linked to other supporting views and the main view. This makes it easier to switch between different perspectives of the trace data when analyzing a particular location.

Some embedded Linux systems employ fixed (real-time) scheduling priorities for time-critical threads, to avoid interference from by less time-critical threads. Setting the right priorities is however crucial for reliable and responsive operation. If a high-priority thread is using too much CPU time, this is shown by the CPU load graph and by response-time plots (see Figure 4). Moreover, the statistics report also provides an overview of thread priorities, CPU usage and timing statistics, which can be used to study and revise the thread priorities in general.
Tracealyzer offers several views showing thread timing properties in timeline plots, where each data point represents an instance (execution) of the thread. The Y-axis shows a specific timing property, such as execution time, response time or periodicity (time between activations). The latter is especially useful for analyzing periodical activities. If periodic thread execution is delayed at some point, it is revealed by the periodicity plot. And just like in other similar views, by double-clicking on the data point in the periodicity plot, the main trace view is synchronized to allow for analyzing the cause of the delay.

Periodic threads running at similar rates might frequently collide with respect to scheduling, i.e., they start at the same time and compete for the CPU time, even though the system might have plenty of idle time otherwise. This causes unnecessary context switching and delays the completion of all colliding threads. Such cases are “low hanging fruits” for optimization, where small changes in timing can give major performance improvements. Tracealyzer makes it easier to find such opportunities, e.g., by inspecting the “response interference” graph. This shows the response time normalized with respect the execution time. For example, if a thread takes 300 µs to complete but only used 100 µs of CPU time, the response interference is 200%. If multiple threads frequently have spikes in response interference at similar times, this is probably worth a closer look. If the execution of colliding period threads can be shifted, collisions can be reduced and performance thereby increased.

Tracealyzer is developed in Microsoft .NET, originally for Microsoft Windows, but also runs on Linux computers using Mono, an alternative Open Source .NET framework, now supported by Microsoft.
Summary

Tracing provides a powerful tool for analysing multi-threaded software systems. On Linux, tracing is enabled by LTTng, a mature and proven open source solution. Percepio’s Tracealyzer for Linux lets developers visualize LTTng trace data through multiple, interconnected graphical views. Tracealyzer makes dense and voluminous trace data more accessible to software developers, giving them greater benefit from tracing. Tracealyzer helps developers make sense of complex trace data, find bugs and tune performance, and thereby produce better software.

Dr. Johan Kraft is CEO and founder of Percepio AB, a Swedish company founded in 2009 based on his Ph.D. work in Computer Science. Dr. Kraft developed the first Tracealyzer prototype in 2004, in collaboration with ABB Robotics. Percepio AB today collaborates with several leading suppliers of Linux and RTOS platforms for embedded software.

---

1 LTTng website: http://lttng.org
2 Mono website: http://mono-project.org