Software Execution Analysis

Philipp Schäfer
Chair of Computer Architecture
Institute of Computer Engineering (ZITI)
University of Heidelberg
Email: philipp.schaefer@ziti.uni-heidelberg.de

Andreas Kugel
Chair of Computer Science V
Institute of Computer Engineering (ZITI)
University of Heidelberg
Email: andreas.kugel@ziti.uni-heidelberg.de

***Abstract*—In modern times, most of our daily used appliances include embedded devices. They range from small, portable devices like MP3-Players, to stationary installations like traffic lights, and largely complex systems like a cars board computer. More complex systems use large numbers of software components that share resources and interact in complex ways. Despite this complexity, they have to address strict timing constraints and software faults are, with few exceptions, unacceptable. To provide these timing constraints, many embedded devices are driven by Real Time Operating Systems (RTOS). Developing embedded systems is a challenge. The use of hardware resources has to be maximized and performance bottleneck and errors have to be detected as early as possible. To achieve these goals it is necessary to get insight into the components interactions. This paper explores how system tracing tools can provide such insights. A short overview on currently used profiling and trace tools is given and the limitations of both debugging methods are pointed out.**

I. Introduction

To explain the importance of system trace tools, we first have to talk about the different domains of problem diagnosis as described in [1]. It is easy to track down logical errors via usual software profilers or debuggers but they will fail in several scenarios like the diagnosis of synchronization problems of multiple processes or multiple threads on multiple CPUs, or the maintenance of consistency and coherence. In general, most systems share resources with several software components that interact in such a complex way that it is impossible to track down issues with usual software debuggers. Problem diagnosis can be divided as follows:

- *Sporadic domain*: This domain includes most of the system faults. The problem source is not known and you have no information about the type of the problem. It occurs in an asynchronous and random manner.
- *Temporal domain*: The problems scope is narrowed, but it is not reproducible.
- *Logical domain*: The error can be reproduced on demand, because the exact sequence of conditions or events that trigger the error was found. At this stage, usual software profiling or debug tools can be used to track down the error.

The main goal is to reach the logical domain as fast as possible. System trace tools helps you to cope with the first two domains.
In general, system tracing addresses a large variety of tools. For example, via *printf()* a programmer can track a program's progress. System information tools like Unix *top* can monitor task creation and track resources. Code coverage and application profiling can be enabled by compiler-driven instrumentation techniques. To diagnose memory leaks and excessive memory fragmentation, a program's history of memory usage can be analyzed by memory tracing tools. At least, to get accurate timing traces to display complex interactions between multiple processes and threads, kernel-level instrumentation techniques are necessary. These techniques reveal events at an operating-system level. In this paper, the latter techniques are defined as system-tracing techniques. With all other listed tools and techniques it is very hard and, most of the time, impossible to bridge sporadic and temporal domain to the logical domain of problem diagnosis.
The most important part of a system tracing tool is that it does not have a significant impact on the systems behavior. It has to track events with fine granular timestamps which enables further diagnosis of unwanted system behavior. This is where software profiling tools like debuggers, *printf()*, memory profilers and code coverage tools retire. A program which is filled with debug code runs much slower and it is not possible to reproduce it is normal timing and behavior. It is getting even worse when these techniques are used in asynchronous or parallel software. Furthermore, a profile only contains a set of performance events and timings for the execution. The chronological order of these events is completely ignored. A trace records timestamps for every event and is extensive in time. When observing a programs behavior as a part of a whole system, the usage of system tracing tools as defined above is indispensable.
The rest of the paper is organized as follows. In section II, a more detailed view on software profiling and common used profiling tools is given. Section III defines the constraints of system tracing. Tools for different operating systems and improved graphical trace representation are presented. At last, a full custom example RTOS system is introduced. A more detailed view is given on its hardware trace module, the tracing implementation and the custom trace viewer. The paper is summarized in section IV. February 3, 2014

II. Software Profiling

As mentioned before, software profiling tools are well discovered. However, they only can help to track down issues when problem diagnosis has already reached the logical domain. This section lists two of the most common free software profiling tools: the GNU profiler [2] which is part of most Linux distributions and the Google performance tools which are used at Google and are available as free download on the

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 37.50      0.06     0.06                             ftuTransformation::xForm(...) const
 12.50      0.08     0.02  1050226     0.00     0.00  QPointF::QPointF()
  6.25      0.09     0.01  1062192     0.00     0.00  operator new(unsigned long, void*)
  6.25      0.10     0.01   410920     0.00     0.00  bool qMapLessThanKey<QChar>(...)
  6.25      0.11     0.01   153140     0.00     0.00  QMap<QChar, statisticalPlotItem*>::concrete(...)
  6.25      0.12     0.01    16080     0.00     0.00  QBitArray::setBit(int, bool)
  6.25      0.13     0.01      520     0.02     0.02  bubblePlottable::drawQuartileBox(...) const
  6.25      0.14     0.01        1    10.00    10.00  ftuGui::qt_static_metacall(...)
  3.13      0.15     0.01   137994     0.00     0.00  QBasicAtomicInt::operator!=(int) const
```

Fig. 1. GNU profiler flat profile

```
granularity: each sample hit covers 2 byte(s) for 6.25% of 0.16 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     37.5    0.06    0.00                 ftuTransformation::xForm(...) const [1]
                0.00    0.00 1839900/1839900     QVector<QwtInterval>::size() const [353]
-----------------------------------------------
                                                 <spontaneous>
[2]     20.1    0.00    0.03                 ftuCommunicate::qt_static_metacall(...) [2]
                0.00    0.03       1/1           ftuCommunicate::stopReadOut() [3]
                0.00    0.00     390/390         ftuCommunicate::socketReadyReadout() [186]
                0.00    0.00      40/40          ftuCommunicate::addCurve(QChar) [219]
                0.00    0.00       1/1           ftuCommunicate::startReadOut() [346]
                0.00    0.00     432/431610      qt_noop() [355]
-----------------------------------------------
                0.00    0.03       1/1           ftuCommunicate::qt_static_metacall(...) [2]
[3]     20.0    0.00    0.03       1         ftuCommunicate::stopReadOut() [3]
                0.00    0.03       1/1           ftuCommunicate::addDataToPlot(QByteArray*) [4]
                0.00    0.00      40/40          ftuPlotCurve::appendPoint(double, double) [70]
                0.00    0.00       1/1           statisticalPlot::updatePlot() [206]
                0.00    0.00       1/1           bubblePlot::updatePlot() [225]
                0.00    0.00      40/118         QVector<double>::last() [240]
                0.00    0.00       1/44          ftuLog::log(QString const&) [266]
                0.00    0.00      80/80          ftuPlotCurve::getYData() const [916]
```

Fig. 2. GNU profiler call graph

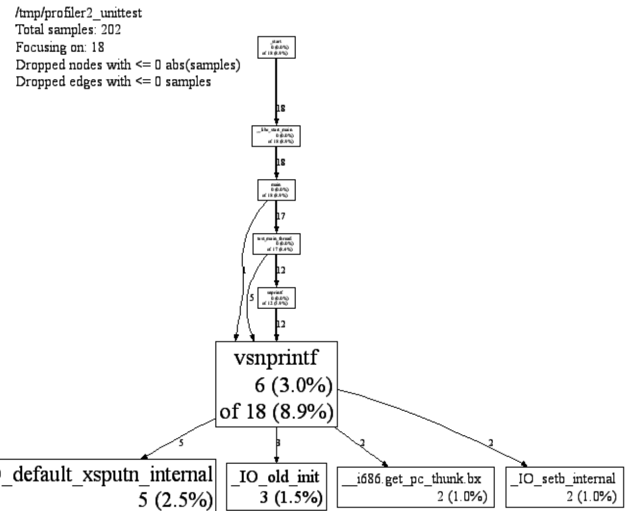

Fig. 3. Google Performance Tool (graphical view)

projects homepage [3].

A. GNU Profiler

The GNU profiler (gprof) is a commonly used tool to determine which parts of a program are taking the most of execution time. Supported languages are C/C++, Pascal and Fortran. The basic output is divided into two parts: a flattened profile contains all function calls of the program and its execution time, number of calls, time per call etc., a sample output is shown in figure 1.

The second part of the gprof output is the call graph which is a more verbose version of the flat profile. The call graph shows how much time was spent in each function and its children. With this information, you can find functions that, while they themselves may not have used much time, called other functions that did use unusual amounts of time. A sample output is depicted in figure 2.

With this sample output of gprof it is pretty easy to see that a significant amount of time is spent in *ftuTransformation::xForm(...) const*. But the call graph reveals that the function itself is not that time consuming however it is called way too often. Since gprof comes with nearly every Linux distribution, it is a good way to get an overlook of a program, but it is limited to CPU profiling and it does not provide any graphical representation.

B. Google Performance Tools

A less basic software profiler is provided by Google. Their profiling tools, called *Google performance tools* (gperf), include a heap profiler, a heap checker, a CPU profiler including a graphical representation (see figure 3) and a malloc/free implementation called thread-caching malloc (TCMalloc). TCMalloc is 6 times faster than the multi-thread malloc version of the GNU C library (glibc), called ptmalloc2, and reduces lock contention for multi-threaded programs [4]. Since it is no profiling tool, it is not further specified in this paper. It is still worth mentioning because heap checker and heap profiler are part of TCMalloc and there is no way to use the both tools separate from TCMalloc.

1) Heap Checker: The heap checker is used to detect memory leaks. In general, the heap checker dumps a memory usage profile on program start. Another one is dumped after the program exits. Both profiles can now be compared to locate leaks. There are two modes of heap leak checking:
Whole-program checking is the recommended way to use the heap checker. Since it records the stack trace for each allocation while it is active, it causes a significant increase of memory usage. Furthermore, the program is slowed down a bit. To tweak the whole-program checking, the user can decide between 4 modes. The *minimal* mode starts as late as possible in an initialization. Stricter modes are suggested because memory leaks during initialization may be ignored. The *normal* mode is made for everyday heap-checking use. The *strict* mode is like the *normal* mode but includes some extra checks. For example, if a pointer is set to NULL without freeing it, a leak message is only thrown in strict mode, not in normal. The last mode is called *draconian* and is the most precise mode. It will throw a leak message unless all memory is freed on program exit.
Partial-program checking can be used if only a specific part of the program has to be analyzed, users can create a *HeapLeapChecker* object at the beginning of the interesting code fragment. At the end of the fragment, *NoLeaks()* has to be called. This will result in similar memory dumps as on whole-program checking, but only for a specified part of the code.
The output of the heap checker can be analyzed by using the *Heap Profiler* (pprof).

2) Heap Profiler: The heap profiler can be used to locate memory leaks, locate parts of the code which do a lot of allocation or generally get an overview of what is in a programs heap at any given time. The user can control the behavior of the profiler via environment variables. It can be defined when a heap profile is dumped. For example, when a specified number of bytes has been allocated by the program or when a high-water memory usage mark increases by a specified number of

bytes. In addition, the heap profiler provides the possibility of a graphical representation of the programs heap.

3) CPU Profiler: The CPU profiler is similar to the GNU profiler but provides a graphical representation of the data. In fact, the gperf CPU profiler is used for all software profiling at Google. Running Linux 2.6+ it profiles all running threads, with Linux 2.4 only the main thread is profiled.

III. System Tracing

Software profilers are easy to use and can help a lot when application code has to be debugged and the *logical domain* of problem diagnosis has been already reached. But as the complexity of newer computer systems increases, applications often depend on a combination of several factors including scheduling, memory management, I/O subsystems, interrupts, lock contention among multiple CPUs and device drivers. As a result, bugs are often located in the *sporadic* and *temporal domain* on more complex systems. To bridge the gap between sporadic/temporal and logical domain, a tool is needed which collects information produced by instrumenting the whole system while not having a significant impact on the systems behavior and performance [5].
Summed up, tracing can be defined as a technique used to understand what is going on in a system in order to debug or monitor it. It is similar to logging since it mainly consists of recording system events. However, compared to logging, it usually records much lower-level events that occur much more frequently. Since traces typically generate thousands of events per second, tracing tools have to be optimized to handle a lot of data while having only a small impact on the system. Typically, traces can contain millions of events which results in many megabytes or gigabytes of data. However, trace analyzers and viewers are available to produce graphs, and statistics from this enormous amount of data. These tools makes it easy to get an overview on the system and spot bugs, performance issues and misbehaviors.
In this chapter, the most common tracing softwares and some of its compatible viewers are presented.

A. Linux Trace Toolkit Next Generation

The Linux Trace Toolkit (LTT) [6] can be considered as the most known and widespread tracing software. Since it is mostly superseded by its successor Linux Trace Toolkit Next Generation (LTTng) [7], this paper will concentrate on LTTng. The main differences between LTT and LTTng is the improved extensibility and a more precise time base. LTT suffers on its monolithic implementation of both the LTT tracer and its viewer. In short, the goals of LTTng can be described as having low system disturbance and architecture independence while being fully reentrant, scalable, precise, extensible, modular and easy to use [5].
LTTng user-land is divided into three main parts: *lttctl* which is a command-line application which runs in user space, the daemon *lttd* which also runs in user space, waits for trace data and writes it to disk; and a kernel part that controls kernel tracing. Figure 4 shows the general control architecture of LTTng.

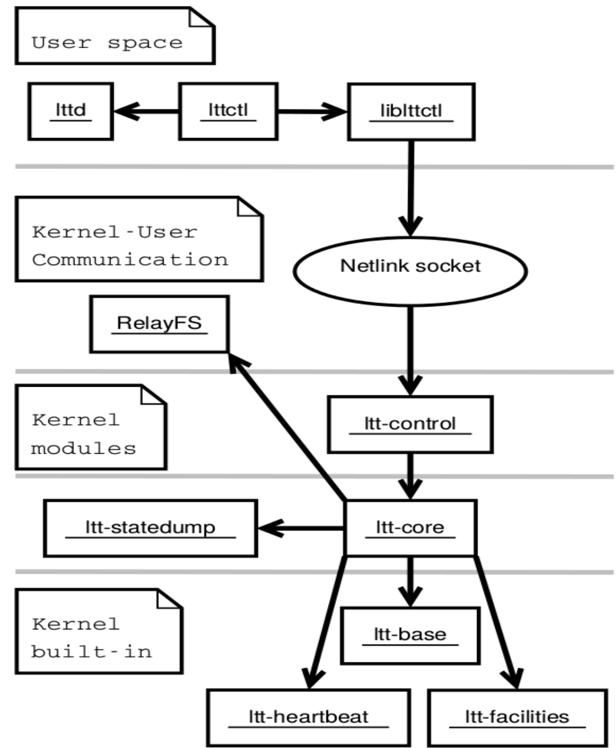

Fig. 4. LTTng control architecture [5]

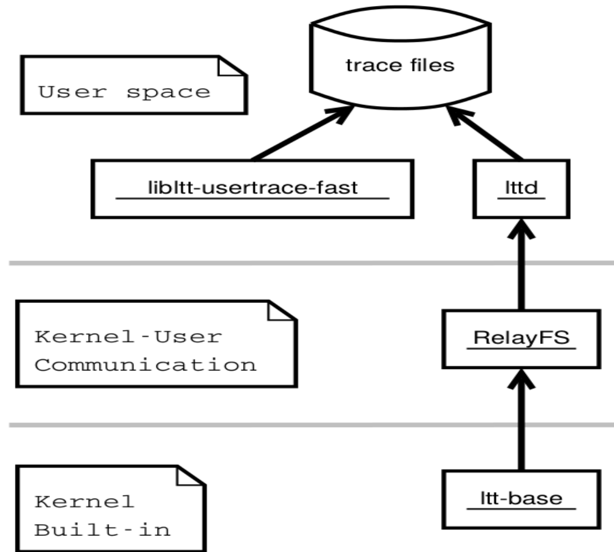

Fig. 5. LTTng data flow [5]

The main module of LTTng is called *ltt-core*. It controls sub-modules like *ltt-heartbeat* (detect cycle counter overflows), *ltt-facilities* (lists event types loaded at trace start time), *ltt-statedump* (generates events to describe kernel state) and *ltt-base* (kernel object containing symbols and data structures required by builtin instrumentation) and is responsible for a number of LTT control events. For writing data from kernel to user-space, LTTng uses RelayFS. RelayFS is a bunch of per-CPU kernel buffers that can be efficiently written into from

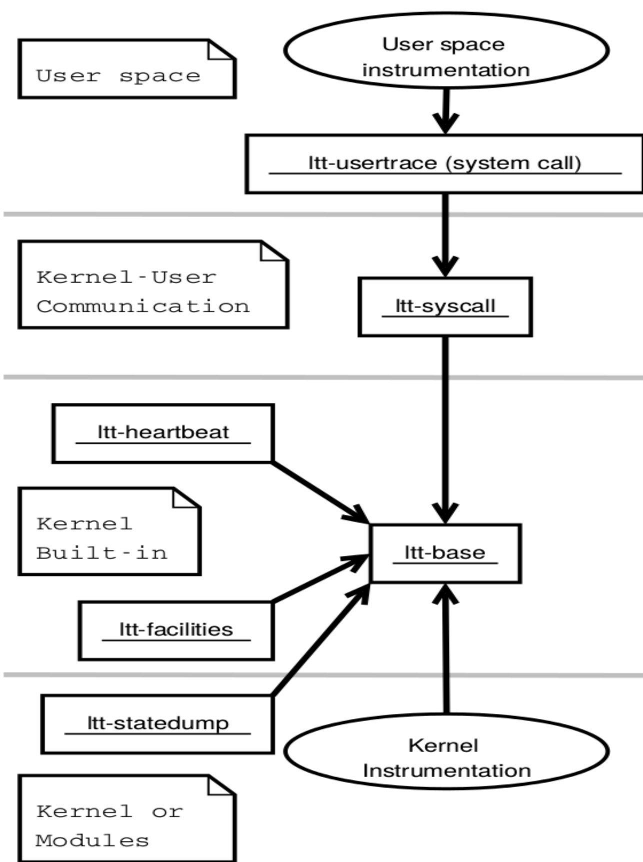

Fig. 6. LTTng tracing [5]

kernel code. These buffers are represented as files which can be mmap'ed and directly read from in user space. The purpose of this setup is to provide the simplest possible mechanism allowing potentially large amounts of data to be logged in the kernel and 'relayed' to user space. An important feature of RelayFS is the ability to write data to the CPU buffers without requiring any locks. This is achieved by an atomic *compare_and_store* on the current buffer index. Depending on the return value of *compare_and_store*, the process can write to the buffer or has to wait [8]. This enables a high scalability without much performance impact. As depicted in figure 5, data is written through *ltt-base* into RelayFS circular buffers. The user daemon *lttd* polls on RelayFS channels and writes data to disk having exclusive access to this sub-buffer. User application tracing is handled in 2 different ways. Applications with low data throughput can use a costly system call at each event call site. This requires no linking of the code against any library. Furthermore, it does not have any thread start-up performance impact. This tracing path is illustrated in figure 6. Applications with high data throughput can use the side path *libltt-usertrace-fast* which consists in a per thread companion process which writes the buffers directly to disk.

In theory, the tracing output would last out for system debugging but to oversee the enormous data, a graphical representation is needed. Some noteworthy viewers for LTTng traces are the Babeltrace viewer [9], the Eclipse viewer [10], Linux Trace Toolkit Viewer (LTTV) [11] and the Percepio Tracealyzer [12]. Where Babeltrace and Eclipse viewer are plugins, LTTV is a standalone viewer for LTT. Percepios Tracealyzer is a trace visualizer worth to go into more detail.

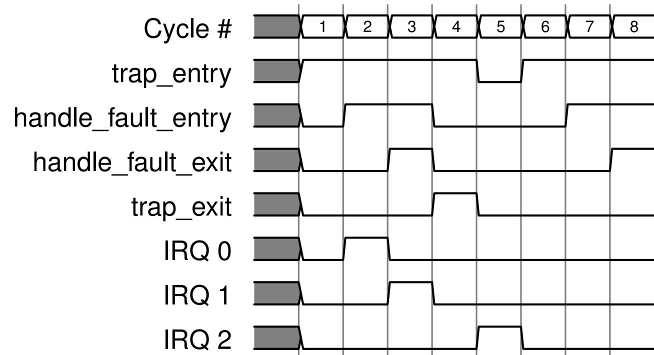

Fig. 7. Example of a waveform like trace representation

B. Percepio Tracealyzer

Tracealyzer is a trace viewer which supports several traces of different operating systems. This project was started as ABB developed a control system for industrial robots, called IRC 5 [13]. The used OS was VxWorks on an Intel-based Industry PC. The OS itself has some tracing features, i.e. registering callbacks on task-switch, task-creation and task deletion. Since recording inter-process communication events was considered as important, some code was added to the OS isolation layer. Events are stored in a single ring buffer with a fixed event size of 6 bytes (most significant two bytes are used for event code + relative time-stamp). In modes where task-switch is slower, for example on system startup, an additional extended time-stamp event is stored since the relative time-stamp does not fit in 14 bit. It uses 32 bit for the time-stamp and overrides the time-stamp field of following events. Tracealyzer is still in use by ABB Robotics for troubleshooting and performance measurement. It was extended to support more operating systems. On most systems, an own tracing library is needed to do the tracing but Tracealyzer can also display tracing output from third-party tools (like LTTng). It visualizes the VxWorks built-in tracing, Linux LTTng traces and supports RTOSes like FreeRTOS/OpenRTOS, SafeRTOS, rt-kernel and Micrium μC/OS-III [14].
Most tracing viewers have limited graphical representations. The most common way is similar to a waveform plot. All processes, threads and tasks are listed vertically and their corresponding events are drawn on a horizontal time-line. Figure 7 gives an example of a waveform visualization. Tracealyzer provides over 20 graphical views including a vertical time-line where events are shown using text labels (see figure 8). The vertical time-line is smoothly scrollable and the colored event boxes are neatly arranged. This way, a very clear graphical representation of the trace data is achieved. Another useful visualization is the communication flow graph. It shows dependencies with respect to communication and synchronization between tasks, interrupt handlers and kernel objects such as semaphores and message queues. This provides the big picture of the executed code and simplifies detection of unexpected behavior. Of course, multiple views with horizontal time-line are provided too. All of them can be shown in a common window with synchronized scrolling. These horizontal time-

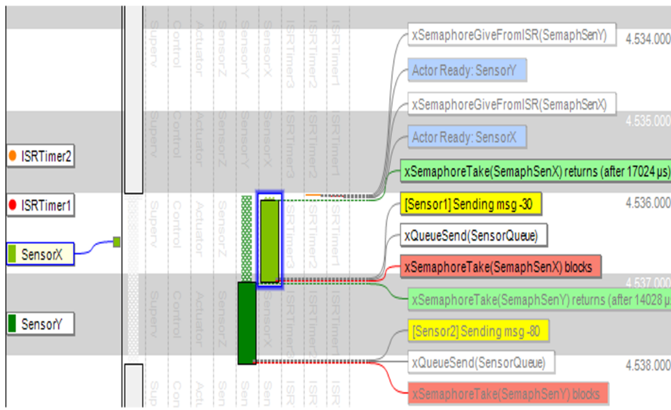

Fig. 8. Percepios Tracealyzer Mainview [12]

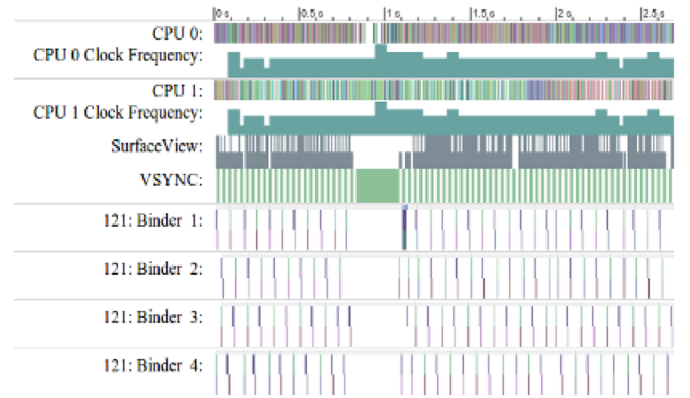

Fig. 9. Android Systrace example

line views include

- a CPU load graph which shows the amount of CPU time used by each task and interrupt handler
- a plot which visualizes the utilization of buffered kernel objects
- a user event signal plot which allows to plot any data logged as user events

Since Tracealyzer is a proprietary and non-free tool, the documentation is limited thus this paper does not describe it in more detail. Nevertheless, it is worth mentioning because of its varied graphical representations and its multi-OS support.

C. Android Systrace

Android is an OS optimized and developed to run on mobile devices. Since it is the leading product of mobile operating systems, it is worth mentioning its provided profile and trace tools. When writing applications (apps) for Android, the most common way to debug the app is the Android provided log tool logcat [15]. It is a system wide logging daemon which can be read via the Android Debug Bridge (adb) or via apps like catlog. The latter needs a rooted phone. Debugging with logcat is similar to *printf()* debugging but with the extension that segmentation faults like null pointer dereferences are logged by the system too. Logcat covers debugging of incorrect program and system behavior and segmentation faults thus most of the time, debugging with logcat suffices. If an app runs slow or uses a lot of background CPU, a simple logcat output is not enough and developers have to dive deeper. To debug apps in a more detailed way, Android ships the Dalvik Debug Monitor Server (ddms) [16]. It provides port-forwarding services, screen capture on the device, thread and heap information on the device, method profiling, logcat, process, and radio state information, incoming call and SMS spoofing, location data spoofing, and more. In general, it includes all features for useful software profiling plus tracing functionality.
To collect and review code execution data for an application and the Android system, Android provides the Systrace tool [17]. It works on all Android versions above 4.1 (Jellybean) and helps to analyze how the execution of an app fits into the larger Android environment. It tracks system and application process execution and plots it on a common timeline as shown in figure 9. Furthermore, Systrace collects data from the kernel, for example from the CPU scheduler, disk activity and threads and generates a HTML report. Systrace only works if the running kernel has tracing enabled. Since not every manufacturer enables this feature for the stock kernels of their phones, sometimes the installation of a custom kernel is necessary. Unfortunately, this is not possible on all phones because some of them ship with a locked bootloader. Since Android 4.3, developers can add custom events to the trace by calling the static class methods *Trace.beginSection(name)* and *Trace.endSection()*.

D. RTOS Tracing

As mentioned before, in RTOS systems it is indispensable to provide a functional tracing facility to detect performance bottlenecks, spot errors as early as possible and satisfy the hard timing constraints. For example, the FreeRTOS kernel which is one of the most known and used real time kernels, provides trace hook macros for embedded application data collection [18]. These macros can be used to satisfy most of the tracing features. For example, setting a digital output or an analogue output voltage to indicate which task is executing allowing logic analyzers or oscilloscopes to be used to view and record the task execution sequence and timing, or logging RTOS kernel events, task execution and timing for offline analysis.
An full custom example RTOS system is currently developed at the chair of Computer Science V at the Institute of Computer Engineering (ZITI) Heidelberg. The developed tracer is designed for an embedded RTOS system running on a Digilent Atlys™Spartan-6 FPGA Development Board and has the following requirements:

- High resolution timestamps: A statistical evaluation is not precise enough to match the hard timing constraints of real-time systems.
- Low overhead: Tracing should have no significant impact on program execution.
- Low memory consumption: Embedded systems lack on memory thus it is not possible to buffer all values for post-mortem analysis. Tracing should produce as little data as possible which enables a real-time data transfer from embedded system to host system.

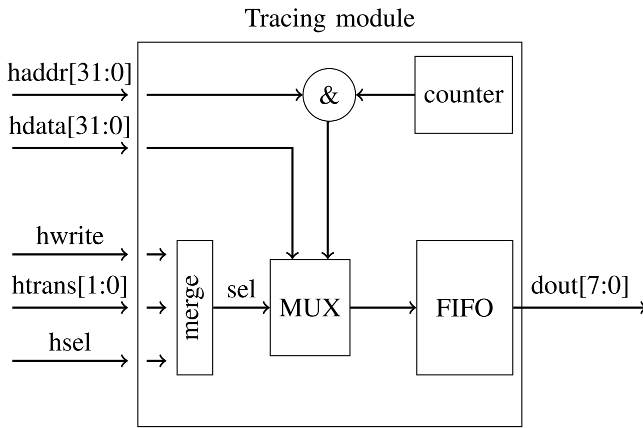

Fig. 10. Example tracing hardware module design

Address	Timestamp	
Function Address		Caller Address

Fig. 11. Bit coding for function entry/exit event

To match these requirements, additional tracing hardware is needed. The most important parts of the tracing module are plotted in figure 10. A timer clocked with CPU frequency is used to provide accurate timestamps. Events are divided into 4 different types: function entry, function exit, RTOS trace and system trace, thus 2 bit suffice to address them all. On write access, a 32 bit value is pushed to a FIFO consisting of the current timer value (30 bit) and 2 bit for the current event type. The FIFO can be read out completely asynchronously via JTAG interface to the host system. Within our hardware module, all trace informations consist of only one access on one of the 4 different event addresses.
Finally, one tracing event is coded in two 32 bit words: event type plus timestamp and the appropriate data. An example for function entry/exit events is depicted in figure 11. The function entry/exit events are generated with *gcc*. When option *-finstrument-functions* is enabled, the compiler will emit calls to *__cyg_profile_func_enter()* and *__cyg_profile_func_exit()* at the top and bottom of every function. In our system, these functions are defined to push an appropriate event to the tracing module. Developers can exclude specific functions by adding *__attribute__ ((no_instrument_function))*.
The presented tracing method is a very efficient way of logging events but it generates no human readable trace output. To survey the traces, a graphical trace viewer is in development too. It provides 3 different graphical representations. A waveform view lists all events like in figure 7 and provides the common way of event trace representation. A statistical plot concentrates on the timing of events. The time of a function is defined as the time between function entry and function exit. The plot visualizes the minimum and maximum time versus the average time of a function as shown in figure 12. At last, a bubble plot is provided which plots the number of function calls versus their total execution time. The project is still in development but the first tests look promising.

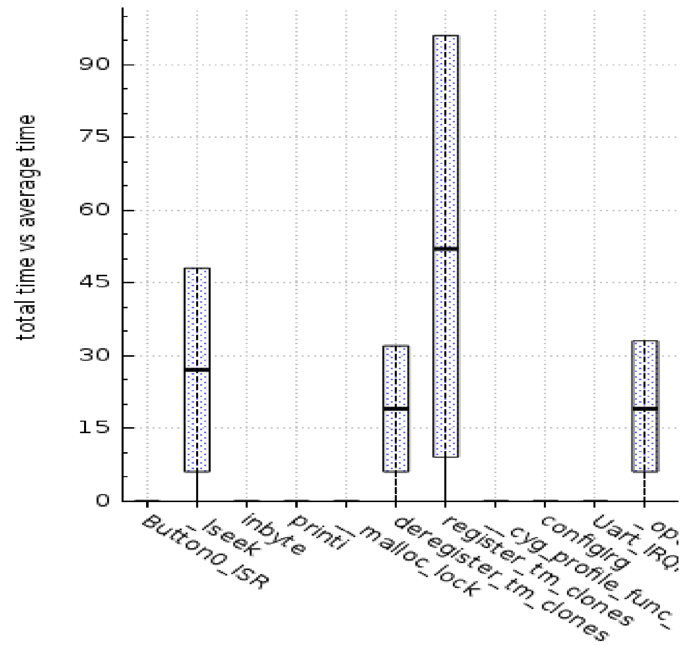

Fig. 12. FTU gui statistical plot

IV. Conclusion

This paper gave a short introduction on state-of-the art tools for both software profiling and system tracing. In addition, a full custom RTOS system tracer including its trace viewer tool was introduced. It pointed out which tool satisfies which use case and described the general usage and idea of tools for both debugging methods.
Furthermore, the main differences between software profiling and system tracing were explained. While software profiling is very helpful when problem diagnosis already reached the *logical domain*, it fails on both *sporadic* and *temporal domain*. Another disadvantage is that the chronological order of profiled events is ignored completely. Tracing tools are used to analyze how a program fits into a larger environment like the OS or a complete embedded system. They provide a strict chronological order of events by adding accurate timestamps to every event type. Tracing tools are optimized on providing a precise log (trace) of every important system event with respect to have as little impact on the systems behavior as possible. However, to accomplish these requirements, additional hardware support is needed. On systems with hard timing constraints, i.e. RTOS, providing an efficient and fully functional tracing facility is indispensable.

References

[1] T. Fletcher. *Using System Tracing Tools to Optimize Software Quality and Behavior*. Tech. rep.

[2] J. Fenlason and R. Stallman. *GNU gprof - The GNU Profiler*. 1994. URL: http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html.

[3] Google. *GooglePerformanceTools*. 2013. URL: http://code.google.com/p/gperftools/wiki/GooglePerformanceTools.

[4] Google. *TCMalloc : Thread-Caching Malloc*. 2007. URL: http://google-perftools.googlecode.com/svn/trunk/doc/tcmalloc.html.

[5] M. Desnoyers and M.R. Dagenais. "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux". In: Ottawa Linux Symposium, 2006.

[6] Opersys inc. *Linux Trace Toolkit Documentation*. 2013. URL: http://www.opersys.com/ltt/documentation.html.

[7] LTTng Project. *Linux Trace Toolkit - next generation*. 2013. URL: http://lttng.org/.

[8] T. Zanussi et al. "relayfs: An Efficient Unified Approach for Transmitting Data from Kernel to User Space". In: Ottawa Linux Symposium, 2003.

[9] EfficiOS Inc. *EfficiOS Operating System Efficiency Services and Consulting - Babeltrace*. 2013. URL: http://www.efficios.com/babeltrace.

[10] The Eclipse Foundation. *Eclipse - Linux Tools Project - LTTng Integration*. 2014. URL: http://www.eclipse.org/linuxtools/projectPages/lttng/.

[11] LTTng Project. *Linux Trace Toolkit - next generation - LTTV*. 2013. URL: http://lttng.org/lttv.

[12] Percepio AB. *Tracealyzer - Understand Troubleshoot Optimize*. 2013. URL: http://percepio.com/tz/.

[13] A. Wall J. Kraft and H. Kienle. *Trace Recording for Embedded Systems: Lessons Learned from Five Industrial Projects*. Tech. rep.

[14] Percepio AB. *Percepio Tracealyzer Datasheet*. Tech. rep.

[15] Google. *logcat*. 2013. URL: http://developer.android.com/tools/help/logcat.html.

[16] Google. *Using DDMS*. 2013. URL: http://developer.android.com/tools/debugging/ddms.html.

[17] Google. *Systrace*. 2013. URL: http://developer.android.com/tools/help/systrace.html.

[18] Real Time Engineers Ltd. *Free RTOS - Trace Hook Macros*. 2014. URL: http://www.freertos.org/rtos-trace-macros.html.