Observation Tools for Debugging and Performance Analysis of Embedded Linux Applications

Carlos Prada-Rojas*, Frederic Riss, Xavier Raynaud, Serge De Paoli, Miguel Santana
FirstName.LastName@st.com

Abstract

Observation tools can play a very important role in embedded software debugging, allowing developers to understand and analyze the behavior of their embedded systems. These tools allow detecting, in particular, anomalies related to non-functional constraints, hard to find with a classic debugger. In this paper, we present two observation tools aiming at allowing non-functional debugging and performance evaluation of Embedded Linux kernel and applications, as well as some industrial experiences carried out with them.

1. Introduction

Debugging Embedded Linux Applications requires a different support than standard Linux applications. This requirement comes, on one hand, from the hardware environment used in such systems, (e.g. specific I/O devices, or limited memory) and, on the other hand, from different constraints to be respected by embedded software (e.g. real-time constraints to respect, or I/O devices to manage at kernel level).

Observation tools can play a very important role in embedded software debugging. They allow to observe and analyze the behavior of the embedded application [1]. These tools show the interactions among the different actors involved by the application to debug and provide statistics. Actors are user and kernel threads and processes, as well as events arising during execution, such as interrupts or signals [2].

Observation tools allow detecting in particular the origin of anomalies related to real-time constraints, such as the cause of a missed frame in a media-player application. Such kinds of anomalies are hard to find with a classic debugger.

In this paper, we first present some requirements and related work for embedded system observation. Subsequently, we briefly describe two observation tools dedicated to non-functional debug and performance evaluation of Embedded Linux kernel and applications: *KPTrace* and *VisualOProfile*. Finally, we describe some industrial experiences using these tools.

*Supported by a CIFRE STMicroelectronics industrial grant.

2. Main Aspects on Observing Embedded Linux Applications

This section describes the guidelines we have followed in our strategy for the development of observation tools for debugging and performance analysis of embedded Linux applications.

Typical tools for observing embedded Linux applications are *tracing-based* tools and *sampling-based* tools. *Tracing-based* tools allow detailed observation of how the application actors are scheduled and are interacting together. *Sampling-based* tools allow profiling. Such kinds of tools allow finding the application hotspots. A hotspot is a section of the code having a large amount of activity.

Ideally, the same observation tools should be available at each step of the embedded application development, from initial developments to the final product. Therefore, important criteria to consider are, on one side, the way the tool can be enabled or disabled, and on the other side, the intrusiveness of the tool:

- Enabling and disabling the tool on a running application without rebuilding it allows the tool to be present up to the final product.
- The intrusiveness of the tool must be small enough to be acceptable all along the software development phase. In addition, it must not change significantly between the time it is disabled or enabled in order to avoid hiding defects or changing the sequencing of the application.

3. Related Work

The following paragraphs depict several observation tools commonly used in standard Linux.

A common tool for observing standard and embedded Linux systems is *LTT* [3]. *LTT* is a tracing-based tool able to trace kernel and user functions on a single trace and with a reasonable probe effect for embedded systems. However it presents some limitations regarding its time-stamp precision and the addition of new trace-points during the execution.

These limitations are overcome by *KProbes* [4]. This tool consists on a kernel instrumentation mechanism, which provides a facility to execute a user-defined handler when an instrumentation point is hit. It can use accurate clocks based on performance counters and allows adding probes dynamically to a running kernel. However, it can only trace kernel functions. Its intrusiveness has a weak impact on execution times [5], but of course, intrusiveness depends on the probes density.

DTrace [6] provides a mechanism for dynamically adding trace-points with a zero intrusiveness ratio when trace-points are not enabled and a small impact on trace generation. Moreover, *DTrace* provides a high level language for describing observation events and a client/server architecture where kernel and user programs are servers of a set of *DTrace* clients or *probes*. This tool looks promising but the first version for Linux kernel is quite young (June 2009). For this reason it has not already been considered for our embedded Linux implementations.

The *printk* kernel function is probably the most widely used technique for tracing embedded applications. It is very simple to use, but it has several drawbacks: it is completely manual and for enabling or disabling it, the code must be rebuilt.

GProf [7] is a tool for displaying data collected during the execution of a program built with dedicated options. *GProf* is dedicated to profiling a single program and the report generation is produced when the program exits. These two points make it not very convenient for detecting hot spots during a whole embedded system execution.

OProfile [8] consists of a system daemon that collects samples at system level. It does not need any instrumentation of the application to observe. It can be enabled and disabled dynamically on a running embedded system. Profiling data are stored in the form of a database on the file system. *OProfile* introduces a low overhead in most common cases. The overhead depends on the interrupt load *OProfile* has to deal with [9].

From the above list, *KProbes* and *OProfile* are the best ones filling our criteria. Their intrusiveness is small and they can be easily controlled at runtime. The following section describes the implementations we have performed on top of these tools.

4. Observation Tools for Debugging and Performance Analysis of Embedded Linux Applications

For debugging real-time execution and performances of Embedded Linux applications, we have developed two different tools:

- *KPTrace* has been developed in order to perform *trace based debugging*. It is based on *KProbes*.
- *VisualOProfile* is an encapsulation of *OProfile*. It has been developed to perform *profile-based debugging*.

KPTrace and *VisualOProfile* are integrated in *Eclipse* [10]. In addition to profile and trace views, they provide a set of facilities allowing to easily controlling data collection and configuration on the target system.

4.1. KPTrace Overview

KPTrace is an observation and analysis tool based on trace-points. It allows instrumentation points to be dynamically added and removed from the kernel and user applications at runtime. Any kernel symbol can be traced, including interrupts, system calls, context switches, functions and their arguments. It is also possible to generate user trace-points using a *printf-style* API. Function trace-points and user trace-points are also possible in the user-space.

Every trace-point is recorded on the target system. The resulting trace is then post-processed on a host workstation and displayed in the IDE through the KPTrace Viewer. This viewer offers many features to analyze resulting traces: an advanced sequence chart viewer, flexible navigation capabilities (outline, zoom, markers, etc), powerful search and filtering mechanisms, management of large volumes of traces, execution statistics, event properties view (call-stack, for instance), etc.

As KPTrace needs to interpret the trace, trace-points have to be compliant to a given format. This format is shown in Figure 1, where:

- The `time-stamp` parameter indicates the time when the trace-point has been generated. For time mea-

surements such as tasks durations, time-stamps are required. Time-stamps can be efficiently retrieved from a timer IP when available in the system. Otherwise, a system function such as `gettimeofday()` can be used.

- The `type` parameter describes the kind of trace-point. There are more than 20 different types. Table 1 presents some of the available types.

- The `PID` parameter indicates the system identifier of process or the thread where the trace-point has occurred.

- The last parameter set is the `arguments` list. They are a set of attributes associated to the trace-point types. They describe input and/or output parameters of a function, memory addresses, etc.

```
<Time-stamp><Type><PID><Argument_0>...<Argument_n>
```

Figure 1. KPTrace Trace-point Format

Trace-point Type	Arguments description
Interrupts	interrupt enter (I) and exit (i).
Context switches	OS context switches (C).
IRQ	SoftIRQ enter (S) and exit (s).
Syscalls	syscalls enter (E) and exit (X).
User-def	user-def. trace-points enter (U) and exit (u).
...	

Table 1. KPTrace Trace-points Types and Arguments

Trace-points are stored in a circular buffer in memory. Two management strategies for storing this buffer are possible: the circular buffer can either be automatically saved when it is full, or saved on user request. The circular buffer is stored on the file system. Then, it is processed by the host workstation. Typically, the circular buffer size is $8\ MB$, which allow storing about 300 000 trace-points. Automatically dumping the circular buffer has a high intrusiveness when it occurs periodically. This drawback can be avoided by a single storage triggered by the user. Other triggers based on error detections are also studied, such as FIFO overflow detection or significant changes in the intervals of a periodic signal.

Thanks to the traced features that allow a full reconstruction of the sequence of execution at a given level of details, trace-based debugging is possible. Some examples on real cases are presented in section 5.1.

4.2. VisualOProfile Overview

VisualOProfile is a system-level profiling tool based on OProfile [8]. OProfile provides detailed statistics for the whole system, including Linux kernel and running applications. It runs transparently, in background, collecting information at a low overhead. Its purpose is to detect bottlenecks occurring in the whole embedded systems.

We have developed VisualOProfile as it provides complementary information to KPTrace. Thanks to its sampling mechanism, it provides statistics at function internal level, and allows detecting system hot spots. VisualOProfile can be used along with KPTrace, where KPTrace probes context-switches and interrupts and VisualOProfile provides additional workload information inside threads.

VisualOProfile is executed on the host workstation, and is integrated into an IDE providing an interface to control the OProfile daemon which runs on the target system. It also contains facilities to visualize profiling results. Only the OProfile sampler and report generators run on the target. The resulting report is stored on the file system. Then, VisualOProfile processes the report.

From the user point of view, VisualOProfile has the following added values:

- It does not require neither OProfile to be installed on the host workstation, nor to be administrator of the workstation.

- It supports different versions of OProfile formats, e.g. in its current implementation, it supports both OProfile versions 0.9.1 and 0.9.4.

- VisualOProfile decodes symbols of all profiled programs of the system and results can be managed graphically in a *profiler perspective*. Figure 2 gives an overview of the IDE. Several views are available, such as the flat function view of the whole system or the call graph of a given function.

Figure 2 shows in a table the time spent in all functions executed by the whole system provided by OProfile reports. This view allows to easily finding the most costly function by sorting the entries of the table. More precisely, it allows to detect hotspots at C-function level for the whole Embedded Linux software and to verify performance assumptions. Industrial experience using VisualOProfile is described in section 5.2.

Figure 2. Overview of OProfile Results

5. Industrial Experiences

Although algorithms may be functionally correct, the resulting application may not work properly because of the non-respect of real time constraints. These are non-functional bugs. Such kinds of defects are nearly impossible to detect with a conventional debugger as a breakpoint stops the execution and it is highly intrusive.

In this section we present debugging experiences carried out on a multimedia player application running on a SoC. The SoC is a STi7109 [11]. It is composed of a host processor running embedded Linux and accelerators performing audio and video processing. The accelerators are managed by a set of kernel modules executed on the host processor.

Experiences are performed on a multimedia application involving several kernel threads and a lot of interactions and interrupts. A particularity of this kind of application is the periodicity of function occurrences as it is processing frames decoding. There are some non-functional defects, visible on audio and video outputs and unexpected behaviors (e.g. audio glitches or corrupted images) that are appearing from time to time. We show how the origin of these behaviors can be easily identified by using KPTrace and VisualOProfile.

5.1. KPTrace Usage for Real-time Bugs Detection

We present in this section two examples showing how KPTrace can be used for detecting non-functional defects.

Figure 3. Execution Trace of a Video Decoding Application

Measuring frame durations: Figure 3 illustrates the execution trace of a video-decoding multi-threaded application. The application is composed of several user-threads (IDs `812`, `823`, `854`), as well as several kernel-threads (IDs `817`, `818`, `819`, `821`, `822`).

Based on user trace-points available through the "user-trace-points" API, multimedia frames can be tagged. A periodic treatment, corresponding to the decoding of a video frame, is clearly shown by this trace. Its period can be measured (`20 ms`), and it corresponds to the video rate. A possible non-functional bug is the potential drift between audio and video frame decoding. User trace-points allow detection and measurement of those potential drifts.

A potential abnormal behavior is highlighted in the zoomed rectangle of Figure 3. It can be carefully analyzed through tool features in order to check if the behavior is wrong but also the reasons behind. For instance, a masked interrupt during a couple of milliseconds, leading to a frame miss can be observed here.

Detecting defects: On the same multimedia application, we illustrate here a debugging scenario using KPTrace:

- KPTrace statistics show some abnormal values for periodic signals. In Figure 4, the interrupt `16` is a periodic signal from a timer, programmed with a `1` ms period. The measured period is `994` μs. The statistics show a maximum interval of `7321` μs, which is an abnormal value. Clicking on the maximum interval value opens the activity chart at the location where the maximum interval has occurred.
- Then, Figure 5 shows location of the problem: The kernel thread `2200` is running too long in critical section, and therefore is masking interrupts (The interrupt

16 does not occur during that period). The interrupt 16 is a timer related to video frame decoding. If suspended, the video output will display corrupted images.

- Looking at the source code of thread 2200, it appears that some processing are performed in critical section, but there is not verification on the amount of processed data. This is due to some extra-processing which is eventually performed, taking much more time than usually.
- After reworking the source code of PESTask thread and running again KPTrace, statistics do not longer display any significant differences between maximum intervals duration and average values of the interrupt 16 periodic signal. Therefore, the defect is fixed.

Time unit is us

Context	Total run time	% Total run time	Max interval	Avg interval	Fired/Time swit
Summary	37,625,526	100%	37,898,843	6,948	180,480
Tasks (non-idle)	28,546,885	75.9%	37,898,843	9,735	104,499
0	5,100,544	13.6%	382,563	1,036	6,488
Interrupts	3,784,408	10.1%	3,606,011	2,989	64,249
Interrupt 16 (periodic/oneshot timer)	2,028,938	5.4%	7,321	994	38,113
Interrupt 168 (ohci_hcd:usb2)	1,122,011	3%	51,997	1,994	19,000
Interrupt 212 (VTG)	255,891	0.7%	22,438	16,516	2,295
Interrupt 213 (VTG)	252,930	0.7%	21,454	16,523	2,294
Interrupt 137 (mailbox)	66,508	0.2%	3,606,011	13,303	1,216
Interrupt 205 (STFDMA_InterruptHar	36,120	0.1%	33,826	15,451	791
Interrupt 204 (STFDMA_InterruptHar	21,975	0.1%	2,323,057	22,561	538

Figure 4. KPTrace Statistics

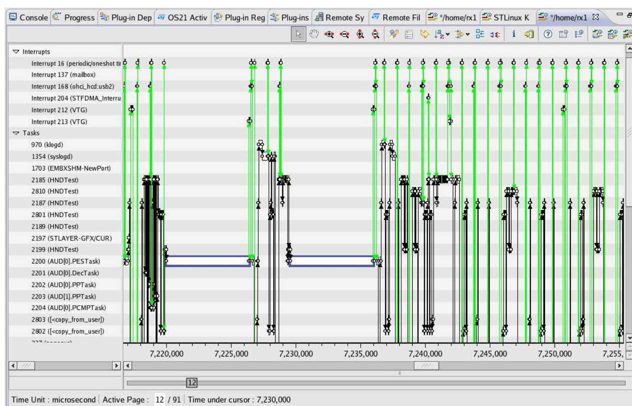

Figure 5. KPTrace Charts Displaying Interrupt Masking Issue

Intrusiveness Quantitative Values: In these examples, interrupts, context switches, a large number of functions, and user trace-points set on frame decoding are recorded. In the first example, user defined trace-points are used to measure drift between audio and video decoding. They allow easy detection of frames, and provide measures on decoding period variations for audio and video. In the second example, only interrupts and context-switches are used.

For both examples, the allocated circular buffer for trace storage is $8MB$. The duration of single trace-point duration is between 30 μs and 50 μs. The host processor core is a ST40 running at $266MHz$ processor.

5.2. VisualOProfile

In this section, we show how VisualOProfile results can be used for finding system hotspots. Hotspots depict where the real-time constraints must be respected.

Detecting Potential Defects: Figure 6 illustrates a graphical summary report of the execution of the same media-player application described above. The left side of the figure presents the list of whole system functions displayed in decreasing order according to the time spent in each function. The right part of the figure shows the graph of GetUnusedPictureBuffer() children.

We can observe that test_bit() is the most cost effective function. It is a hot spot as it uses 25.74% of CPU. This function may either be a short-duration function called very often, or a large function rarely invoked. In order to check if there is not a potential violation of real-time constraints, this function should be examined to ensure that the duration of an occurrence is always smaller than the minimum occurrence period of this function.

Intrusiveness Quantitative Values: Figure 6 shows that the intrusiveness of OProfile daemon is 1.94%.

Some inaccuracies may appear in the results due to *blind spots* (regions where no samples are collected). This is due to the way OProfile is implemented on ST40: OProfile data collection mechanism uses maskable interrupts [12]. Samples occurring in such a situation are attributed to the code immediately after the interrupts are re-enabled.

6. Conclusion and Future Work

6.1. Conclusion

In this paper we have shown that tracing-based and sampling-based observation tools give valuable support for

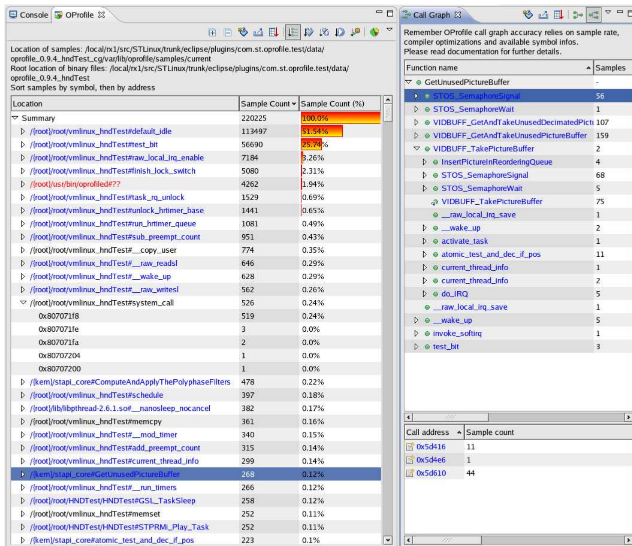

Figure 6. Graphical Profiling Report of a Video Decoder Execution

debugging embedded applications. The described experiments are performed on real embedded multimedia applications. Experiments have easily revealed the origin of defects that were difficult to find before.

These tools are used for development of Linux multimedia applications on embedded systems, involving both kernel modules and user-level software. The first one, *KPTrace*, is based on collecting an accurate trace of the execution and the second one, *VisualOProfile*, on samples of the whole system.

These tools are used together as they are providing complementary information: KPTrace provides accurate view of the system activity but no detail of what is happening inside functions, while VisualOProfile provide the remaining level of detail, allowing the detection of hot spots at system level. These tools are part of the embedded system and can be enabled or disabled at runtime.

6.2. Future work

Different investigations are currently being carried out for a better integration among the tools, as well as for supporting new kinds of platforms:

- The synchronization between VisualOProfile hot-spot statistics and KPTrace activity charts.
- The support of KPTrace on SMP architectures as the next generation of SoCs will contain SMP cores.
- The development of triggers based on automatic error-detection for saving KPTrace databases for validation purposes.
- The support of STP [13] trace ports available on new generation of SoCs in order to dump traces directly on ports rather than storing them internally.

References

[1] M. Domeika, *Software Development for Embedded Multi-core Systems*. Newnes, April 2008.

[2] J. C. de Kergommeaux and B. de Oliveira Stein, “Paje: An extensible environment for visualizing multi-threaded programs executions,” in *Euro-Par ’00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*. London, UK: Springer-Verlag, 2000.

[3] K. Yaghmour and M. Dagenais, “System administration: The linux trace toolkit,” *Linux J.*, p. 22.

[4] R. Krishnakumar, “Kernel korner: kprobes-a kernel debugger,” *Linux J.*, vol. 2005, no. 133, p. 11, 2005.

[5] “Kprobes documentation,” Website, http://www.mjmwired.net/kernel/Documentation/kprobes.txt#456.

[6] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, “Dynamic instrumentation of production systems,” in *ATEC ’04: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2004, pp. 2–2.

[7] S. L. Graham, P. B. Kessler, and M. K. McKusick, “gprof: a call graph execution profiler,” *SIGPLAN Not.*, vol. 39, no. 4, pp. 49–57, 2004.

[8] “OProfile,” Website, http://oprofile.sourceforge.net.

[9] “Oprofile performances,” Website, http://oprofile.sourceforge.net/performance.

[10] “Eclipse Project,” Website, http://www.eclipse.org.

[11] “STi7109 Datasheet,” Website, http://www.st.com/stonline/products/literature/bd/11660/sti7109.pdf.

[12] “OProfile Kernel Profiling,” Website, http://oprofile.sourceforge.net/doc/kernel-profiling.html#irq-masking.

[13] “MIPI Test and Debug Interface Framework,” White paper, http://www.mipi.org/docs/MIPI_TDWG_whitepaper_v3_2.pdf.