Proceedings of the 10th IASTED International Conference
SOFTWARE ENGINEERING AND APPLICATIONS
November 13–15, 2006, Dallas, TX, USA
ISBN Hardcopy: 0-88986-642-2 / CD: 0-88986-599-X

EFFECTIVE PERFORMANCE ANALYSIS AND VISUALIZATION ON EMBEDDED LINUX

Jayaraj P.B, Ranjith Gopalakrishnan[1], Vikas Jain[2]
Philips Innovation Campus
No. 1, Murphy Road, Ulsoor

Bangalore 560 008
INDIA
Ranjith.Gopalakrishnan@Philips.com, Vikas.Jain@Philips.com

ABSTRACT

This paper[1,2] presents a case study of dynamic performance analysis and bottleneck identification of a large application software stack on a Linux-based embedded platform. We use some standard techniques for performance analysis, but we also try out some techniques that are not so common for performance characterization and visualization. The methodology used for measurement collection is generic and hardware-independent (software-centric), and it is relevant for any effort that develops embedded applications on a Linux platform. Using this methodology, we have analyzed the performance of the JUICE (Joint User Interface and Control Engine) software stack developed within Philips, running on a Linux-based embedded platform. We have identified relevant metrics, detailed the tools and methods by which they can be collected (given the existing open-source tool chain) and discussed the salient aspects of performance analysis and visualization of the collected measurements. We conclude that the open-source tool chain provides good support for such measurement collection and can aid further performance analysis efforts. We also discuss the various gaps and issues in the existing tooling that we have come across in this effort.

KEY WORDS
Software Performance Engineering, Software Optimization, Software Performance Analysis, Linux, Embedded Systems

1. Introduction

More and more embedded systems and CE (consumer electronics) devices are being labeled as *software-intensive*, with the amount of software in these systems reaching several megabytes [1][2]. In [1], the fact that the amount of software in CE devices is growing at an exponential rate (quite similar to moore's law) is well brought out. It is further explained that the amount and complexity of functionality implemented in software is increasing by leaps and bounds (current high-end TVs hold about 10MB of software). This makes the realization of systems that are responsive to user inputs a considerable challenge. The fact that early performance analysis is critical in general software development was realized in the early 90's [3]. Nowadays, the scenario in the CE world is no different; the role of timely performance analysis of systems and software is crucial for the realization of responsive and robust CE devices.

Linux is readily establishing itself as one of the most important and versatile operating systems to enter the embedded computing domain. There are several examples of the usage of Linux platforms to realize products in the CE software development community, but much of these efforts don't yet address the problem of performance analysis in a systematic manner. On the other hand, hard real-time system development has performance management as a significant component [4]. Additionally, performance analysis for embedded systems in the CE (consumer electronics) has been concentrated on the hardware aspects of the platform. Hence, studies were done on low-level aspects of the system like the memory cache, bus bandwidth, bus utilization, etc. In such systems, the application software was simple, and most of the performance problems were caused in the layers below it. But with the growing complexity of software, performance bottlenecks caused by software is gaining prominence. Another traditional way of doing performance analysis involved the manual instrumentation of the code (to gather timing values, etc) and the tabulation and analysis of collected values. This is both time-consuming and error-prone, and hence, one should explore ways in which any code instrumentation can be done (as much as possible) in an automatic fashion.

In this work, we examine aspects of performance analysis of application software stacks running on Linux, and how performance bottlenecks in these can be found out – early and effectively. Also, the objective was to get insights

[1,2] This work was carried out at Philips Research India, Bangalore. Ranjith Gopalakrishnan is affiliated to NXP Semiconductors (formerly Philips Semiconductors), Bangalore. Vikas Jain is affiliated to Philips Consumer Electronics, Bangalore.

We used the JUICE (Joint User Interface and Control Engine) software stack, developed internally in Philips, for our study. JUICE is a UIMS (user interface management system) that has been designed to ease the UI (user interface) and application development for embedded systems. It implements complex UI functionalities like text scrolling, multi-layered menus, etc. Figure 1 shows a simplified architectural view of the JUICE software stack. Notice how the JUICE "talks" (via JAPI) to various applications, which, in turn talk to the middleware subsystem. In other words, JUICE is the agent by which the actual application (CD reader, CD writer, mobile phone middleware, etc) "talks" to the user. Good responsiveness by this stack is essential in providing a good end-user experience. This is a prime motivation for our selection of JUICE for this performance analysis exercise. JUICE contains around 75K lines of codes in the C language, distributed in around 500 C-style functions (250 files).

4. Performance Metrics Identified

Now, we shall discuss the several metrics that we ultimately used for the study. These metrics were converged upon after consultation with the software architects of JUICE. They are:

1. Elapsed time in function call
2. Fan-in and fan-out of the function
3. Elapsed time jitter

The total elapsed time in a function call is the amount of time elapsed between entering the function and exiting from it. This metric is also termed the latency of the function call. This is a sum of time needed to execute lines of source code that are local to the function (termed local elapsed time) and the time to execute other functions called by that particular function. A characterization of the latencies encountered in the stack is very important, as this is a good indication of the responsiveness of the software.

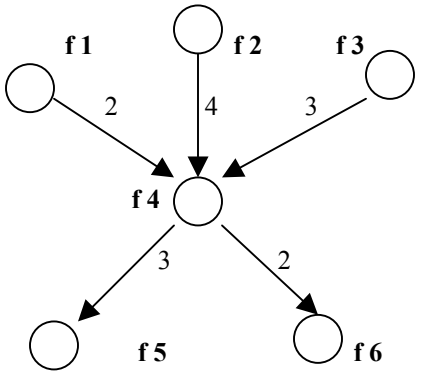

Figure 2: Explanation of the fan-in, fan-out metrics used for performance characterization

The *fan-in* of a function is the number of function calls made by other functions to that function, during the execution of the software. On the same note, *fan-out* of a function refers to the number of function calls that the function makes to other functions (excluding those in the system library, like the read() call provided by libc). In figure 2, functions *f1*, *f2* and *f3* call function *f4* two, four and three times respectively. Hence, the fan-in of function *f4* is 2+4+3 = 9. Likewise, if *f4* calls *f5* and *f6* three and two times separately, the fan-out of *f4* is 3+2 = 5.

The *elapsed time jitter* of a function call is a measure of the variability of the elapsed time in a function. This is calculated by finding the standard deviation of the elapsed time. This roughly points to the real-time nature of the function. This is because functions with real-time execution characteristics are expected to have minimal variance for their execution time (which is measured by the elapsed time parameter). Hence, the standard deviation of elapsed time (which is the elapsed time jitter) would be minimal. For example, if a function *decode_data()* has elapsed times as 100, 104, 96, 105 milliseconds for similar data input, the elapsed time jitter will be low, which may help us to conclude that the function is (more or less) real-time in nature. But, if the elapsed times vary widely as 100, 67, 134, 75 milliseconds for decoding of similar data, the elapsed time jitter would be high. This would signal the non-real time nature of this function. Thus, a system that uses this function will be relatively non real-time in nature. We think that the judicious usage of this metric can give a good indication of how deterministic the involved functions are, which may be very useful in the realization of soft real-time multimedia systems.

5. Collection Methodology

All the performance numbers that were collected were with *compiler-level instrumentation*, and did not involve any manual instrumentation. This ensures that the collection of performance numbers is simple, easy and error-free. We used the extensive instrumentation facilities provided by the GCC compiler. The compiler supports the insertion of instrumentation to collect: a) the call-graph, or a graph representing the sequence of functional calls made in the software stack, b) the latency information related to the function execution. Tools like "gprof" and "function-check" [11] were used to collate and tabulate the collected numbers. Figure 3 shows the overall methodology that we followed in the evaluation of the software. In this figure, the numbered boxes represent the stages where the architects were involved. The additional tooling that we developed was for stage 3.

Use-case of JUICE: The definition of a good and representative use-case is very important while conducting performance measurements. The use-case has to be simple (so that it is easily repeatable) and yet, it

should exercise all the relevant software modules in such a way that leads to the manifestation of the maximum number of bottlenecks. Keeping these criteria in mind, and in consultation with the architects, we identified a relevant use-case as scrolling up 30 times through a set of menus.

Experimental setup used: The target hardware to run the JUICE stack used was the AMD AU 1200 Board [12]. The target runs Linux kernel 2.6.11-rc4 (modified by MontaVista) on a MIPS core, and the measurements were collected on the IDE on-board hard-disk. The cross compiler used was mipsel–linux gcc-3.3.5-glibc-2.3.2.

Using the methodology described above, we were able to collect all the metrics listed in the previous section for every function in the stack.

Gaps identified in open-source tooling:

We shall briefly talk about the gaps or deficiencies we have identified in the tools currently available, during our efforts:

- Handling of context switches: Note that the instrumentation collects only the time of entry and exit of the function. There is a chance that in the time in-between, while executing in the body of the function, the operating system switches to execution of other functions (or other processes). The times for these executions would also be added to the latency of the function execution, which is a significant deficiency if the profiled software is heavily multithreaded.

In order to compensate for this deficiency as best as possible, we first make sure that the profiled application is the only one running on the target and the number of external hardware interrupts is as minimal as possible. Also, we run the use-case a number of times, and for each function, we take the average of all the latency times recorded.

- Overhead of instrumentation: We measured the overheads (in terms of extra time taken) due to the instrumentation introduced by the GCC compiler. It is about 4.2 microseconds per function call. This overhead is quite small, thought not insignificant. The assumption is that functions of interest have much more latencies of execution above 1-2 milliseconds.

- Remote logging: The amount of RAM memory in embedded systems is very limited. Hence, a provision by which one can transfer the collected measurements to a host machine would be very useful. Such transfers should not be too frequent, or the overhead of such transfers can affect the execution of the application.

- Absolute timing information: Currently, there is no method by which one can log the absolute times of the entry and exit into functions, and other such events. Such values would make possible integrated performance analysis over measurements from various tools.

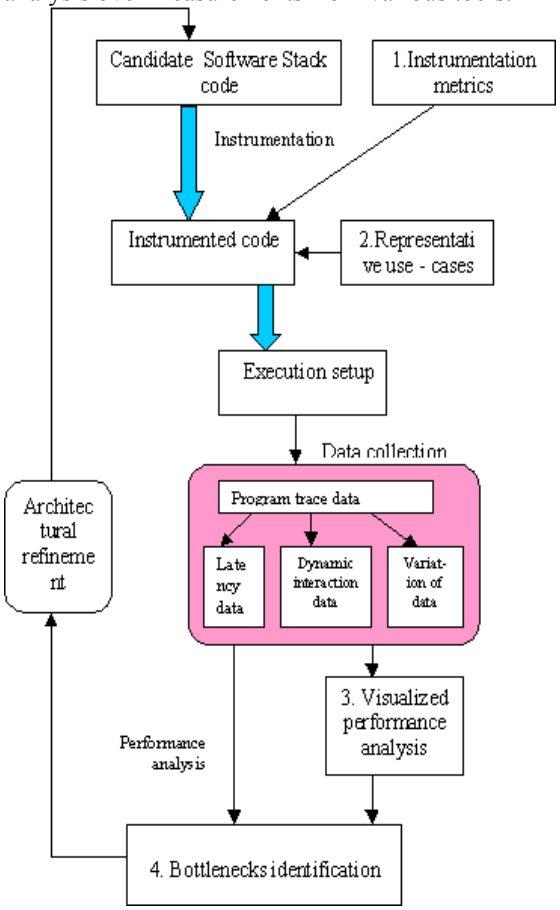

Figure 3: Major steps of the performance analysis methodology used

6. Performance Visualization and Analysis

Performance visualization is important so that one can quickly infer and identify bottlenecks from the huge amount of performance data collected. Performance visualization and analysis is essentially mining the large collection of performance measurements for interesting aspects of performance (like performance hotspots). We developed a tool (which works in tandem with the performance analysis tool *cgprof* [13]), which generates a *colored* call-graph, for performance visualization. This call graph indicates the various function calls made during the execution of the software stack. One can indicate to the tool the parameter on which the coloring of the nodes (of each function) is to be based – like the total elapsed time in the function, the fan-in of the function, etc. For example, let us consider that the fan-in of functions is the criteria chosen for the coloring of the call-graph. In this case, functions with a low fan-in will be colored green, those with a moderate fan-in will be colored yellow, functions with a high fan-in will be colored orange, and those with the highest amount of fan-in will be colored

red. This tool was very helpful in quickly assessing the parts of the architecture or code that could have lead to performance bottlenecks.

A part of the output of this tool when run on JUICE can be seen in figure 4 (coloring is based on fan-in of functions). This call-graph points to the fact that function "SliderCreate" (seen as a red, or darkest, node) is the *most used* of all functions, as it services the maximum number of function calls

Analysis of collected measurements: In section 4, the various performance metrics that were used for characterization of the software stack were outlined. Also, we have discussed the use-case that was employed to exercise the stack, during the collection of these measurements. Now, some sample inferences made from the collected data would be presented. This would give a flavor of the performance analysis possible on the collected data.

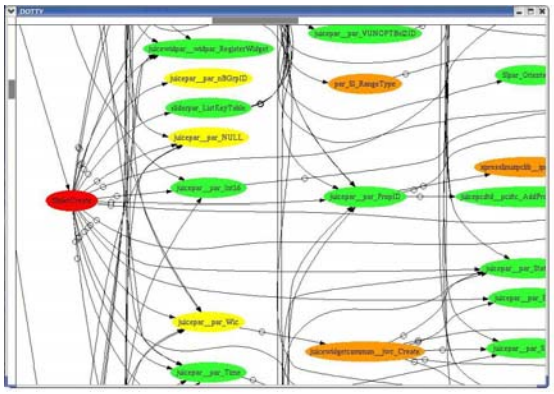

Figure 4: Screenshot from the performance visualization tool developed

In figures 5-8, we have shown a few top functions, as per some performance metrics. The values of the metric for these functions are plotted on the y-axis (this is similar for the other graphs). Such plots sometimes point us to the few functions have high values of these metrics, and these are prime candidates for performance optimization.

In figure 5, we have shown the top five functions as per the metric "total elapsed time". This metric quantifies which part of the code the platform spends most of its time (executing). From this analysis, it has been found that the "draw" routines and the "timer events" routines of JUICE take up much of the CPU time.

Analysis of the fan-in of various functions (see figure 6) found that the set of functions concerned with resource-allocation and platform-interaction are being used heavily. Hence, these are prime candidates for any performance optimization effort.

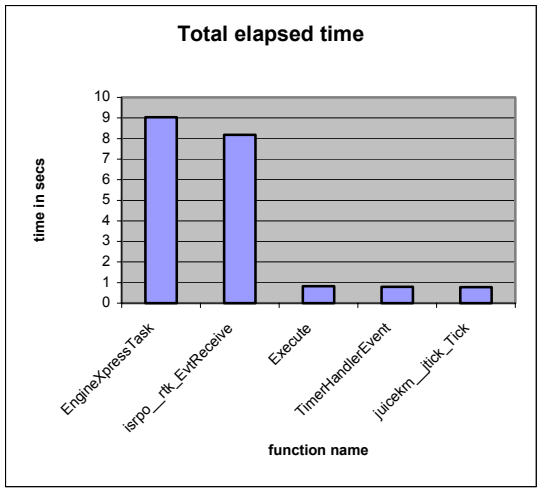

Figure 5: Functions with the highest total elapsed time

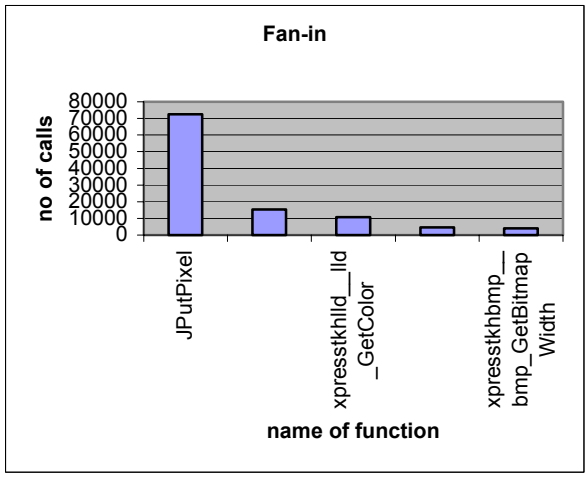

Figure 6: Top 5 heavily called functions

In Figure 7, we have depicted the top functions as per the metric: local elapsed time per call. Recall that this metric points to the most "heavy" functions. From the results that we have measured, and also in consultation with the software architects, it has been concluded that various "platform" and the "graphics" components seem to dominate here. These are parts that interact with the lower layers (hardware, etc), and hence it points to the fact that either optimization is needed in the hardware, or in the lower levels of the stack.

In Figure 8, the functions having the highest "elapsed time jitter" is shown. From this, we have inferred the (relatively) non real-time behavior of the graphics components and initialization routines. The architects confirmed our inferences and this points to the usefulness of this metric in bringing out non real-time functions.

As stated above, the inferences presented here are just to give a flavor of what can be read from the collection of measurements. But, the full set of measurements was very

useful to the architects (with a good knowledge of the functionality of each part of the code) to get some good insights into the dynamics of the stack. Also, we were able to generate color-coded call-graphs for various parts of the JUICE software stack (similar to figure 4), which were also useful for further analysis efforts.

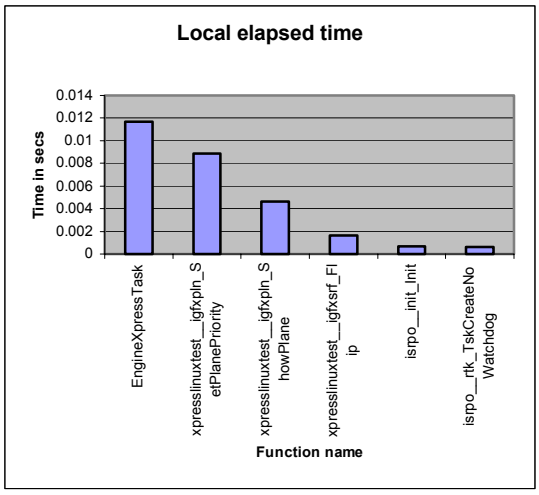

Figure 7: Functions with the maximum local elapsed time

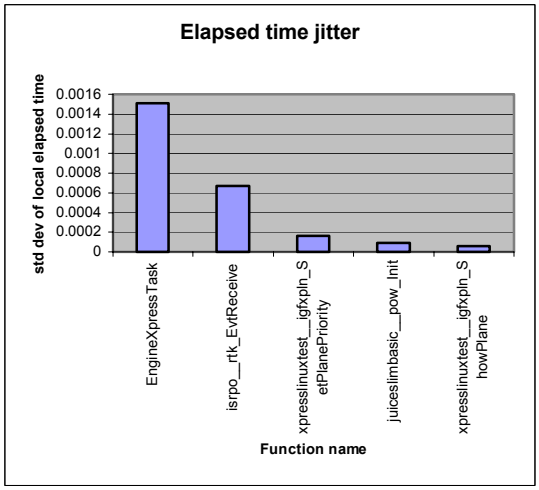

Figure 8: Functions with high elapsed time jitter

7. Conclusion and Future Work

This paper outlines a methodology for the performance analysis of a software stack running on an embedded Linux platform. In the description of the methodology, the relevant open-source tools and methods by which this may be done are outlined. Various metrics for capturing the dynamic behavior of the software are discussed. From the description of the method by which the numbers are gathered, one can observe that the overhead of our method (in terms of developer and tester effort) is minimal, and it is error-free. The study included actual measurements on an existing software stack with guidance from the concerned architects. But it was done in such a fashion as to be generic to any software stack (for which the source code is available), which is built using the GCC tool chain and which executes in a Linux environment. It is hoped that this work would help spearhead a performance-optimized proactive design paradigm for the realization of responsive CE software, based on Linux.

There are interesting ways by which our work may be extended. Right now, all the performance metrics are collected at the functional level. It is possible to extend this to the architectural level, taking support from the fact that architectural specifications would give the mapping of functions to an architectural component. This would give effective feedback for improvement of the software architecture. Also, various possibilities may be explored to overcome the deficiencies described in section 5.

References

[1] Van Ommering R., "Software Reuse in Product Populations", *IEEE Transactions on Software Engineering*, pp. 537-550, vol. 31, issue 7, july 2005.

[2] R. Bourgonjon, "The Evolution of Embedded Software in Consumer Products," *Proc. Int'l Conf. Eng. of Complex Computer Systems,* 1995.

[3] C. U. Smith, *Performance Engineering of Software Systems:* Addison Wesley, 1990.

[4] Anu Purhonen, "Performance Optimization of Embedded software Architecture – A case study", *Proc. Fourth working IEEE/IFIP Conference on Software Architecture (WICSA '04)*, 2004.

[5] Raj Jain, *The Art Of Computer Systems Performance Analysis,* Wiley John Wiley and Sons, Inc., 1991.

[6] R. Pooley, "Software Engineering and Performance: a Roadmap", *Proc. 22nd International Conference on Software Engineering,* Limerick, Ireland, 2000.

[7] P. Clements, R. Kazman, and M. Klein, *Evaluating software architecture: Method ands and Case Studies,* Addison Wesley, 2001.

[8] L. Dobrica and E. Niemela, "A survey on Software Architecture Analysis Methods", *IEEE Transactions on Software Engineering*, vol. 28, pp.638-653, 2002.

[9] Blue Mug Embedded Linux Performance Study: http://www.bluemug.com/research/linux_performance/index.shtml

[10] IBM Rational PurifyPlus for Linux & Unix: http://www.pts.com/wp2075.cfm

[11] Function Check Profiler: http://www710.univ-lyon1.fr/~yperret/fnccheck/profiler.html

[12] AMD Au1200 development board: http://www.amd.com/us-en/ConnectivitySolutions/ProductInformation/0,,50_2330_6625_12409%5E14088,00.html

[13] Cgprof - call graph tool: http://mvertes.free.fr/cgprof/cgprof.html