21st IMEKO TC4 International Symposium and
19th International Workshop on ADC Modelling and Testing
Understanding the World through Electrical and Electronic Measurement
Budapest, Hungary, September 7-9, 2016

Benchmark Tool for the Characterization of the Real-Time Performance of Linux on System on a Chip Platforms for Measurement Systems

Dávid Vincze, Tamás Kovácsházy

Budapest University of Technology and Economics
Department of Measurement and Information Systems
Budapest, Hungary
18bvinda@gmail.com, khazy@mit.bme.hu

***Abstract* – The complexity of measurement systems grows rapidly. Such complex systems cannot be constructed without the high-level software abstraction provided by embedded operating systems that make hardware handling and communication services much easier to implement. Today, Linux is the dominant high-end operating system in the embedded world; however, its scheduling performance is not much investigated using measurements in sufficient detail. The paper introduces such study in which Linux's scheduling performance is investigated for various programming models (ranging from a simple application written in JavaScript or in Bash shell script to the real-time scheduling classes in Linux) and for two System on a Chip (SoC) platforms for specified kernel versions. The results of this study present valuable information for application developers in their current form. However, doing the measurements is time consuming and error prone in its current form because it must be conducted for large number of programming languages and models, system configurations (SoC type, kernel version, etc.), and for various disturbing system loads. Therefore, the primary outcome of this research is a detailed workflow of such evaluation, based on which we plan to implement an automated measurement system.**

***Keywords* – Measurement System, Distributed Measurement Systems, Heterogeneous Computing, Real-Time Linux, Scheduling Performance, Automated Measurement System**

I. INTRODUCTION

Complexity of measurement systems grows continuously. State of the art measurement systems are distributed today, and the complexity of nodes of the distributed measurement system increase in complexity also. Nodes are heterogeneous multiprocessing systems today employing general purpose CPUs, DSP processors, Graphic Processors (capable doing general purpose computing also), FPGAs, and dedicated real-time execution units (for handling time critical tasks). Most of the cases we use System on a Chip (SoC) platforms (e.g., TI Sitara, NXP/Freescale i.MX, etc.) for such applications nowadays, as they are low cost, simple, high-performance solutions providing all of the required hardware in one chip hopefully with full software support for mainstream operating systems. Having good software support is essential, as measurement systems tend to use embedded operating systems because the growing hardware complexity and the user requirements (rich GUI, handling standard services ranging from Pen Drive handling for data transfer to WEB access to the instrument) cannot be implemented with low level software. However, using embedded operating systems brings in its own problems, the abstraction provided by the operating system; which allows relatively easy development of complex services, causes serious time domain performance problems as the abstract interfaces increase SW complexity and dependencies are hard to understand in such complex software. Practically, we cannot run our programs when we want, but the systems schedules them when it considers it to be scheduled. Of course, real-time operating systems allow programs to be scheduled according to user requirements; however, even these solutions have limits in the time domain, and their performance strongly depend on the whole software also (for example, on other real-time tasks).

Therefore, it is essential to investigate the scheduling performance of operating systems in some benchmark applications. Unfortunately, these investigations are complex as there are large number of factors influencing the results, such main factors are:

- SoC type and clock frequency,
- Operating systems and actual kernel version, and its build (the software toolchain used during the compiling and building of the OS), including the operating system's Application Binary Interface (ABI) as in some cases it is also a selectable option,
- Used applications and their software environment (interpreter in case of scripts or compiler and linker in case of programs available in machine code),
- Other application (disturbance) running in parallel with the software under investigation and the system load caused by them (CPU load, memory load, or I/O load specific to some subsystems such as communication or data acquisition).

II. SCHEDULING IN LINUX

Linux kernels (version 2.2.x or later) implement a complex scheduler supporting scheduling classes. Standard user and system processes are scheduled in the SCHED_OTHER class by a Linux version specific priority based scheduler. The SCHED_RR, SCHED_FIFO, and SCHED_DEADLINE (since kernel 3.14) scheduling classes are for real-time tasks providing some level of control for the applications how they are scheduled. The SCHED_OTHER class uses a Linux kernel version dependent scheduler, which is currently the Completely Fair Scheduler (CFS). CFS appeared in kernel version 2.6.23 (released 9 October 2007) and used currently (kernel version 4.5.2 released 20 April 2016) with minor modifications.

The scheduler does not define the run-time of tasks alone, the programming language, programming model/architecture, and the used operating system services (i.e., when and for what reasons the task waits) influence the scheduler also. Currently, Linux let application programmers to write a programs for the same specification using large number of ways. The following approaches are typical:

- Shell scripting (e.g. bash),
- Scripting languages (e.g. Javascript or Python),
- C/C++ code as a standard user space process,
- C/C++ code using real-time user space threads in the process,
- Kernel programming,
- Kernel programming using real-time kernel threads.

In addition, if I/O access is involved, it is quite common that Linux provides multiple ways of communicating with the peripheral, for example, an I/O port on a SoC device can be accessed using standard device I/O (accesses through a device file from user space, or memory mapped I/O (mmap-ed access), or low level kernel I/O (register level access). It is also same for timers quite common in embedded code, a given time delay can be implemented using the scheduler's virtual timers by passive waiting (sleeplocks), special hardware timers, or by active waiting (spinlock, spinning) with varying performance and properties.

III. BENCHMARKING TIME DOMAIN PERFORMANCE OF LINUX

The performance analyses of complex computer based systems using SoCs and high-end operating systems is a complicated area of research due to the diverse systems, applications and requirements [1]. It is especially true for time domain analyses for real-time systems as here the results should be measured in the physical domain (regarding sensors or actuators) to prove that the performance metric determined during analyses are really relevant to and physically measurable in the application area, e.g., if the real-time performance is evaluated by the operating system itself using software it may be biased or totally wrong, while if it is measurable outside of the system, for example, using an oscilloscope, it is very hard to question the results.

The topic of the real-time performance of Linux has very limited scientific level publications. The real-time properties of Linux are investigated in [2]; however, this paper is from 2002, and since then Linux has been practically rewritten. Paper [3] analyses the I/O scheduler primarily, and even this paper had been written way before the release of the current scheduler of Linux (2007), so the results are not relevant to new systems. New results, such as [4-6] are for large scale IT infrastructure related applications, so they are not applicable for embedded systems, not only due to the different application but also due to the drastically different hardware (high performance many CPU server computers versus heterogeneous embedded SoC). Paper [7] shows how real-time add-on CPUs on SoCs can improve real-time performance to alternative Linux implementations, but does not try to build a generic picture about performance. Furthermore, all available publications including blog pages, discussion forum posts, etc. use different methodologies, tools, presentation format, etc., so their scientific and practical value is hard to asses and compare. Furthermore, as previously stated, the results should be measured by external measurement tools and not reported by the operating system, so standard benchmark tools are not an option (even if some exists for real-time Linux). Therefore, we decided to follow a different route for real-time performance analysis with the following constraint:

- We decided to implement the benchmark application ourselves using multiple programming tools and programming models to show application developers their possibilities while implementing Linux applications,

- We decided to select a simple but relevant application for embedded systems that also easy to evaluate with external tools (such as oscilloscopes),
- We selected an application which not only relevant to Linux, but that can be executed efficiently also on the real-time execution units of modern SoCs,
- We started evaluating performance by human labour to understand the process, and after getting the initial results we are going to automate the measurement process.

The selected and implemented benchmark application is the "bitbanging" application, i.e., let us assume that an I/O pin of the SoC must be changed/toggled (set from High to Low or from Low to High) with a given timing to implement some low level protocol in software (I2C, SPI, or proprietary protocols, etc.). Then the question arises how precisely this problem can be solved in software, or for a certain precision what kind of software must be written to fulfil the requirements? This or similar applications are very common in embedded systems, and as the state of the I/O pin is accessible outside of the SoC, the quality of timing can be measured using external tools. As the first step, the benchmark was even further simplified as we decided just to generate a square wave with given frequency and 50% duty cycle using an I/O pin from software. As timing performance can be explicitly analysed using this benchmark, it is ideal from our point of view, i.e., if the frequency or duty cycle is not the expected, we can be sure this is due to scheduling issues.

IV. EXECUTION OF MEASUREMENT

The benchmark is implemented using the following programming languages and programming models:

- Simple shell (bash) script accessing the I/O pin through device files, named “Bash script” in Figures,
- Javascript program accessing the I/O using the built in functions of the platform, named “JavaScript” in Figures,
- User space C program using the libsoc library to access the I/O pins, named “C (libsoc)” in Figures.
- User space C program using memory mapped I/O (mmap) to access the I/O pins, named “C (mmap)” in Figures,
- User space C program using memory mapped I/O plus SCHED_FIFO, named “C (sched)” in Figures,
- Kernel module written in C accessing the I/O directly, named “Kernel module” in Figures,
- Kernel module using HRTIMER accessing the I/O directly, named “Kernel module (hrt) in Figures,
- Kernel module using real-time kernel scheduling accessing the I/O directly, named “Kernel module (RT)” in Figures,
- Real-time execution unit of the SoC is also programmed and compared, named “PRU” for the TI Soc and Cortex-M4 for the NXO SoC in Figures.

The last benchmark program does not directly evaluates the performance of Linux, but gives a good baseline how a dedicated processor could handle the task. Here we must mention that new embedded system on a chips tend to have special real-time processor cores to reduce the timing requirements against Linux. For example, TI uses Programmable Real-Time Units (PRU) on their SoCs in their older designs exclusively, while other manufacturers add standard Cortex-M cores to their SoCs (like the i.MX 6SoloX SoC from NXP or the new TI AM5xxx line, which includes both PRUs and Cortex-M cores).

We tested the benchmarks with the BeagleBone White utilizing an AM3359 SoC and a UDOO NEO utilizing an NXP (formerly Freescale) i.MX 6SoloX SoC. As we have already mentioned, the AM3359 SoC has two PRUs, while the i.MX 6SoloX SoC has one 200MHz Cortex-M4 available for real-time tasks. However, the software support for these devices also influence the achievable performance, and here one can see substantial difference between the two platforms:

- The PRU of the AM3359 SoC is properly supported by modern Linux kernels and TI development systems. Access to the PRU from the kernel is low-level but very efficient.
- The relatively new i.MX 6SoloX SoC lacks polished software support for the Cortex-M4 core. We have been able to use the high-level Arduino-based software environment only. Lower-level direct access approaches to the Cortex-M4 core are not well-documented and we have not been able to make them work. All of our attempts with these approaches failed due to various bugs in tools or in the kernel, or just due to the lack of proper documentation.

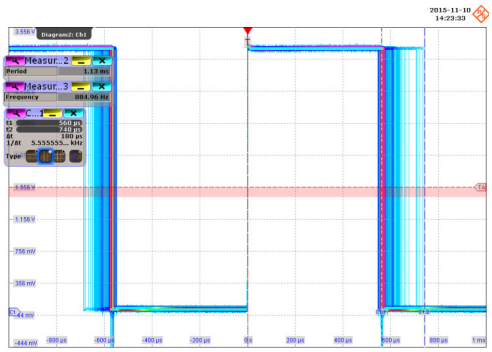

Fig. 1. Typical Output of oscilloscope during timing measurement

The evaluation of the results are done using a Rohde-Schwarz RTO1014 oscilloscope using the virtual phosphor mode with infinite hold time, i.e. , we run the benchmark for specified amount of time, and timing limits are read from the oscilloscope screen. See Fig. 1 for typical screens presented by the oscilloscope during such measurements.

In all cases (for every benchmark implementation and for hardware) we have executed the benchmark on a system:

- In idle state running only natural background tasks,
- In a heavy multitasking state running our benchmarks parallel with Sysbench generating very high (practically 100%) CPU load but without I/O load,
- In a mixed I/O and multitasking state running our benchmarks parallel with netperf generating a full speed UDP stream, this load is special because it generates a lot of interrupts (influencing the scheduler of the OS).

Currently, measurements are executed and data is collected from the oscilloscope screen manually then entered into an Excel workbook. Then the workbook is exported and loaded by Python scripts utilizing SciPy (https://www.scipy.org) to do mathematical function and visualize the data.

V. BENCHMARK RESULTS

Running the 9 benchmark implementations (from bash scripts to the real-time execution unit based ones) on the two platforms (BeagleBone White and UDOO NEO) for the three load situations (no load, CPU load and I/O load) and for the prescribed time periods (from hundreds of milliseconds to some microseconds in several steps) requires hundreds of measurements and produces large amount of data, which is very hard to analyse or present in a concise form. Here we attempt to show some results with some specific and interesting features. We decided not to do the benchmark for various Linux kernel versions to keep the required time for the research controlled. The BeagleBone White was tested with a standard Debian using a 3.8 Linux kernel, while the UDOO NEO was tested with the manufacturer supplied UDOObuntu image using a 3.14 Linux kernel.

First we present the results produced for the Beaglebone White and the UDOO NEO with Sysbench loads as these are the worst-case performance values likely to be observed in applications with low network traffic, for example. Fig. 2 shows the results of the BeagleBone while Fig 3. is for the UDOO NEO, where the figures do not include data, the data cannot be measured due to some system constraints or the errors are over 100%.

Based on the figures we can state that:

- It is clearly visible that script based solutions are the worst performers, however, while shell scripts are marginally better for the BeagleBone,

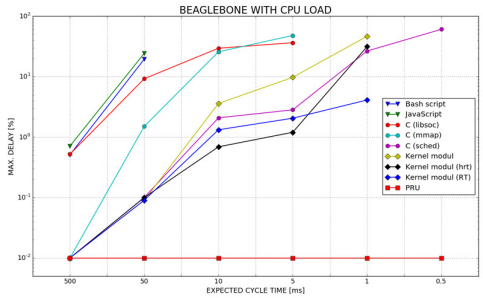

Fig. 2. Maximum of the relative time period error (delay) for the Beaglebone White with CPU load

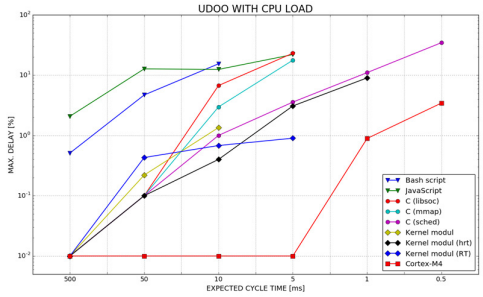

Fig. 3. Maximum of the relative time period error (delay) for the UDOO NEO with CPU load

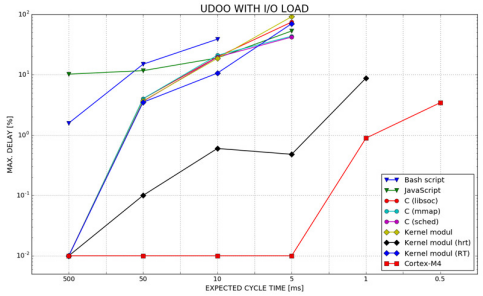

Fig. 4. Maximum of the relative time period error (delay) for the UDOO NEO with I/O (netperf) load

Javascript is substantially better for the UDOO, it is like due to the different Javascript engine on the two platforms. These software approaches are for fast prototyping of low speed applications where deadlines are soft and they are in the range of hundreds of milliseconds.

- C programming drastically increases real-time performance on both platforms, and memory mapped I/O is always a better, lower overhead solution than the libsoc (device file) bases I/O access. Memory mapping of I/O is a very effective way of allowing access to registers from user space on platforms that support it. Unfortunately, here we must note also that

memory mapped I/O is a very dangerous solution from the user space, as writing the registers without the control of the operating system kernel (drivers) can lead to unexpected consequences. Actually, one must considered memory mapped I/O in user-space as shared memory (actually, it is that on the low level), which may be accessed by multiple parallel running processes in a concurrent way, i.e., mutual exclusion must be provided in this case.

- However, by evaluating the results as a whole, Linux cannot precisely schedule programs under the sub-millisecond level. We can trust in it in the 10 ms range, where errors of timing are under 1% for real-time scheduled user space programs and kernel modules, and for the HRTIMER scheduled kernel modules, which is tend to be the best for sub-millisecond applications.
- Out of the two platforms the UDOO NEO is better if it works (it has a smaller operational envelope in the tests) and it is limited by the high-level Arduino based software environment in tests for the real-time execution unit (Cortex-M4).

Fig. 4. shows the results for the UDOO NEO in case of I/O load generated by netperf configured for maximum throughput UDP traffic. The results are similar in quality for the BeagleBone White, so we decided not to show the figures for this due to paper size constraints from now. All benchmark results are substantially worse for I/O load than for the case of the CPU load, except for the benchmarks using a Kernel module with HRTIMER accessing the I/O directly and for the real-time execution unit:

- The HRTIMER bases solution works even better for some tests, which is very hard to explain. However, the I/O load values for this benchmark are quite similar to the no load benchmark results as shown in Fig. 5. Probably, Linux uses spinlocks here to wait for short periods of time, which is heavily perturbed by the heavy CPU load; however, we have not evaluated the internal Linux code to prove our assumption.
- The performance of the real-time execution units does not depend on the load of Linux. However, here we must note that the continuous communication of the main CPU (running Linux) and the real-time execution unit (running real-time code) is a necessity in all practical applications. In this case the above statement may not be valid anymore, as some form of load dependence may be observed due to the overhead of continuous communication between Linux and the real-time execution unit.

This test shows that heavy I/O traffic with lot of interrupts can cause drastic performance problems even in real-time operating systems for most of the generic SW

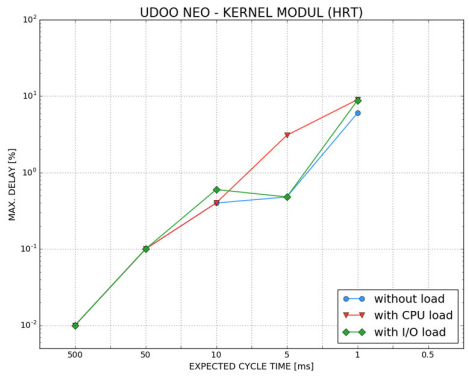

Fig. 5. Maximum of the relative time period error (delay) for the with various loads for a kernel module using the HRTIMER

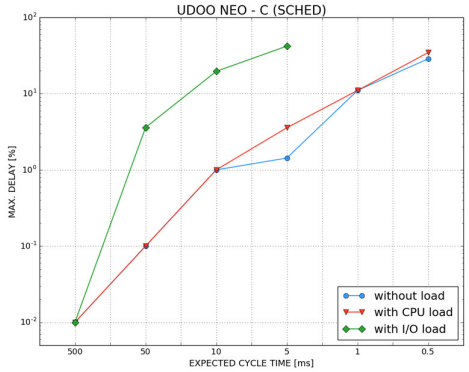

Fig. 6. Maximum of the relative time period error (delay) for the UDOO NEO with CPU load in case of Real-Time scheduled C user space program

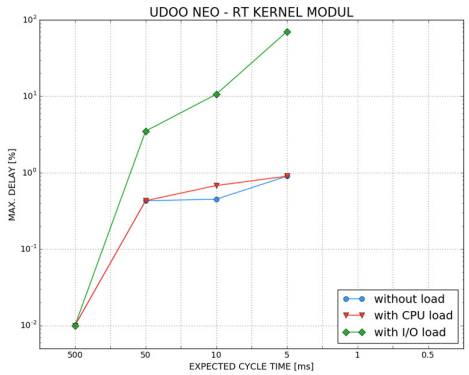

Fig. 7. Maximum of the relative time period error (delay) for the UDOO NEO with CPU load in case of Real-Time scheduled C Kernel space program

based solutions. It is true even for real-time scheduled threads in user-space or in the kernel-space, unfortunately. See Fig. 6. for the user-space and in Fig. 7. for kernel-space performance results. On the other hand, we can expect some guaranteed real-time performance from the following programming approaches:

- Specialized SW+HW support for real-time execution inside Linux, for example, in the form of the HRTIMER based approach. This solution can provide timely operation for some less strict timing requirements even in the range of 1-10 ms, which can be sufficient for some motion control applications, for example.
- And for all dedicated hardware (like the real-time execution unit) for all viable timing requirements. Unfortunately, our practical experience shows that the interoperation solutions of Linux and these real-time execution units and the software tools for such heterogeneous architectures are not mature yet for newer SoC platforms. This situation may improve in the near future (as the improvement of tools for the older TI SoC has already shown that).

VI. CONCLUSIONS

The paper presents a new performance evaluation tool and timing performance results about programming language and programming model dependent performance of the Linux under no load and high CPU or I/O load conditions, and compares these results to performance available on SoCs with special real-time execution units such as the TI Sitara or NXP i.MX 6SoloX SoC line, the results are likely to be easily generalized for other similar SoCs.

The time domain performance (real-time capabilities) provided by the real-time execution units of the tested SoC platforms shows clearly how much these solutions are valuable providing dedicated computing resources if they have proper software support. These devices can provide magnitudes better performance for simple real-time tasks with minimal system complexity increase if implemented well, while Linux can handle the high-level non real-time tasks (such as GUI, network communication, real-time system management, etc.), in which standard microcontrollers are typically quite limited. However, the use of these real-time execution units may be limited by the lack of knowledge and tool maturity in the short term.

The benchmarks, experiences, and the measurement methodology will be utilized in an automated measurement tool to be developed. SciPy has been selected as the implementation platform for the automated measurement tool as it properly fits the open-source, free software model of Linux and it is capable of all functionalities required from measured device (benchmark execution control) and measurement device (setting up measurements and collecting results) automation to plotting figures.

VII. FUTURE RESEARCH

The developed benchmark methodology is applicable to investigate the programming language and model, and SoC and load specific variation of real-time performance of Linux. However, currently it is very time consuming and error prone to run the benchmarks, and therefore, we have started to automate them. The currently developed solution will be Python based and uses the remote access capabilities of the Rohde-Schwarz RTO1014 oscilloscope for automatic measurement result collection.

VIII. ACKNOWLEDGMENT

We would like to thank Texas Instruments for their support of this work through equipment and tool donations and for udoo.org for their software support.

REFERENCES

[1] Jain, Raj. The art of computer systems performance analysis. John Wiley & Sons, 2008.

[2] Abeni, Luca, Ashvin Goel, Charles Krasic, Jim Snow, and Jonathan Walpole. "A measurement-based analysis of the real-time performance of linux." In Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE, pp. 133-142. IEEE, 2002.

[3] Pratt, Stephen, and Dominique A. Heger. "Workload dependent performance evaluation of the linux 2.6 i/o schedulers." In 2004 Linux Symposium. 2004.

[4] Xavier, Miguel G., Marcelo Veiga Neves, Fabio D. Rossi, Tiago C. Ferreto, Tobias Lange, and Cesar AF De Rose. "Performance evaluation of container-based virtualization for high performance computing environments." In Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on, pp. 233-240. IEEE, 2013.

[5] Gujarati, Arpan, Felipe Cerqueira, and Bjorn B. Brandenburg. "Analysis of the Linux Push and Pull Scheduler with Arbitrary Processor Affinities." In Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on, pp. 69-79. IEEE, 2013.

[6] Tian, Hongyun, Kun Zhang, Li Ruan, Mingfa Zhu, Limin Xiao, Xiuqiao Li, and Yuhang Liu. "Analysis and Optimization of CFS Scheduler on NUMA-Based Systems." In Emerging Technologies for Information Systems, Computing, and Management, pp. 181-189. Springer New York, 2013.

[7] Anand, Anjitha M., Balu Raveendran, Shoukath Cherukat, and Shiyas Shahab. "Using PRUSS for Real-Time Applications on Beaglebone Black." In Proceedings of the Third International Symposium on Women in Computing and Informatics, pp. 377-382. ACM, 2015.