Impact of environment on the execution of a real-time Linux process on a multicore platform

Thomas Beck, Frédéric Boniol, Jérôme Ermont, Luc Maillet

Impact of environment on the execution of a real-time Linux process on a multicore platform

Thomas Beck*, Frédéric Boniol†, Jérôme Ermont‡ and Luc Maillet*
*Airbus Defence and Space, Toulouse, France
Email: thomas.t.beck@airbus.com luc.maillet@airbus.com
†ONERA, Toulouse, France
Email: frederic.boniol@onera.fr
‡IRIT-ENSEEIHT, Toulouse, France
Email: jerome.ermont@toulouse-inp.fr

I. Introduction

A. Space context

In the embedded systems industry there are not environments more hostile than space. In fact, a spacecraft is alone in the void without direct human interactions. Its only way to communicate and receive orders is through antennas. In such a context, consequences of software failures can lead to the loss of the spacecraft or even worse. For example, if the AOCS (Attitude and Orbit Control System) software does not work, the solar panels will not always be facing the sun and the battery will not be able to produce enough power for the whole spacecraft. In order to reduce those risks, on-board software applications have a criticality level which represents the consequences of their failures. Therefore, a failure in a high criticality level software will result in worse consequences. In the space context, criticality level for software is defined in the ECSS (European Cooperation for Space Standardization) standards as follows:

- A Catastrophic consequences: loss of human life or environment disaster.
- B Critical consequences: loss of the spacecraft and/or the mission.
- C Major consequences: major mission degradation.
- D Minor consequences: minor mission degradation.

The development of a software with a high level of criticality is more constrained and thus more expensive. Usually, to prevent that a software failure propagates to a software with a higher level of criticality, software applications are executed on different hardware platforms. Most of the time, on-board software applications with the same level of criticality are not made by the same software team, and are separated on different hardware platforms as well. With this approach, software failures are contained by the hardware and software behavior does not affect execution of others software. Clearly this one-software on one-hardware-platform strategy has a non negligible cost considering the size, weight and power of an on-board computer.

B. Problem statement

This article aims at studying the cohabitation of two or more software applications on the same multi-core hardware platform. These two software applications are designed and developed according to the space context described in the previous section, thus each software application has a criticality level and is produced by a different developers team. Once they are executing on the same hardware platform, we want to assure two fundamental properties:

- **Execution interferences:** the perturbation made by one software on to others should not be greater than ϵ. This ϵ can be an execution time or a response time depending on the case. This epsilon is defined for each application and is part of its specification.
- **Failure propagation:** a software failure should not propagate to software with a higher criticality level. The functional failure propagation, being application-specific is not in the scope of this work.

Along with these properties we assume the following hypothesis: all software should be developed and executed on Linux. It means that Linux is used as the embedded operating system of our spacecraft. In order to ensure these two properties, our objective is to use space and time isolation of applications running on a Linux operating system. The use of Linux is motivated by its recent evolution that makes it useful for embedded systems.

C. Related work

In 2000, [1] presented core issues of IMA (integrated modular avionics) which was new at that time. In this technical report, John Rushby laid the groundwork of avionics partitioning showing that a strict isolation is a solution for resolving the IMA problematic. Many years later, solutions have been found to adapt this resolution in space avionics as shown in [2]. Hypervisors such as Xtratum and VxWorks 653 have been developed to respond to the needs of an IMA for space. Isolation proposed by this kind of hypervisor is strict, secure and with a temporal predictability. The disadvantage of those isolation properties is that it makes the whole system less modular. Scheduling is fixed , developers coding libraries are specific and not widely known and the development process is more complex. A performance comparison of these real time solutions is presented in [3].

Linux could be the perfect solution to the problems previously cited and it is used in many space mission as described in [4]. However, Linux is not a hypervisor designed to

strictly isolate and to schedule real-time software applications. Linux and its PREEMPT-RT patch is able to provide real-time features. For hard real-time software applications, a co-kernel solution named Xenomai is available. The performance differences between native Linux and Xenomai are presented in [5]. However Linux is not deterministic and its behavior can introduce latency in the system. A lot of studies measured this latency with the PREEMPT-RT patch ([6] [7] [8]). Moreover, some studies worked on the scheduling model of Linux to reduce these latencies ([9] [10] [11]).

The other aspect of using Linux in a critical real time system is to ensure an isolation of the different applications running on it. This concurrency problem is explained in [12] where they try to reduce the impact by implementing an RCU mechanism. Others studies try to address the problem from the memory point of view and thus work on the hardware memory system ([13] [14] [15]). Containerization is a powerful feature of Linux which helps when an isolation is required. [16] presents a way to modify the Linux scheduler to use container and real-time scheduling at the same time.

Linux is a powerful operating system and is useful in a lot of industries. In the space industry the interest has recently grown with the new space. The last Linux space known project was made by NASA. They sent a helicopter on Mars with an avionic partially based on Linux [17]. As explained in [18] Linux is also an academic subject to study.

Finally, our approach presented in this paper is based on the measurements like in [19]. We want to study Linux with the less customised configurations to take the more advantages of the open-source world offered by Linux.

D. Contribution of the paper

The main objective of this paper is to evaluate the robustness of Linux in a critical real-time context. This paper aims at producing tests results of the Linux behavior and its processes when using an embedded space scenario. We are studying the impact of Linux upon real time processes by varying configurations of the system (cf sections VI-B, VI-C and VI-D). We will also focus on the impacts of running Linux real-time processes upon each other (cf section VI-A).

II. Space software usecase and platform

Software applications we study exchange data between physical sensors, physical memory and the ground station. Those software can be real-time, i.e with a deadline and/or a period, depending on the requirements of the sensor they communicate with. Since all software will be executed on top of a Linux operating system, a software application is represented by a Linux process. We took one space specific use-case software to study its isolation when it is executed with others processes on the same hardware platform.

A. AOCS: attitude and orbit control system

In a satellite, the AOCS software is responsible of controlling the attitude i.e orientation and the orbit of the spacecraft. Inputs of AOCS algorithm are attitude and orbit positions coming from sensors such as Star trackers. The AOCS algorithm generates commands sent to actuators such as gyroscopes and thrusters. These commands aim at correcting the attitude and orbit of the spacecraft. In the scope of this article we study a simple AOCS algorithm doing a flyby i.e approaching a stellar object without entering any orbit. The AOCS software application implementing the algorithm is a periodic Linux process running at a 8Hz frequency. It takes its inputs from a pipe and writes its outputs in another pipe. This software consists of 16000 calculus steps. Each steps doesn't have the same behavior as the others. However, if the inputs stay the same the behavior of the same step in two different execution of the software will have the exact same behavior. The AOCS process has two children processes described below:

- A non periodic Linux process reads inputs from the input file and writes it to the input pipe.
- Another non periodic Linux process reads outputs from the output pipe and compares it to the expected outputs from the output file.

The input and output files are located on the file system. The AOCS process is thus only responsible of computation which is its true behavior in the spacecraft. By forking the AOCS process and thus creating two children processes it creates an isolation between I/Os accesses and calculus code. Moreover, memory accesses made by the I/Os processes and the AOCS process are different. Reading and writing to a file imply loading memory pages from the disk to the main memory. This operation along with every others one related to the virtual memory are handled by the operating system and are slower than accessing the main memory. In another hand, reading and writing to a pipe is faster as it doesn't need memory page management because pipes are located in the operating system area.

B. Hardware platform

Due to the hostility of the space environment, spacecraft embedded computers and electronic devices must be adapted. Those modifications have non negligible costs on satellites production. Recently, the space industry introduced COTS (commercial off-the-shelf) hardware components to be used in the next generation of satellites. Experiments presented in this paper are made on a Zynq Ultrascale+ made by Xilinx which is one of the COTS hardware platforms considered by the space industry for the next generation of satellites.

The SoC has two main processing units (application and real-time) along with a PMU (platform management unit), a CSU (configuration and security unit) and a GPU (graphical processing unit). The Zynq ultrascale+ also contains an FPGA which can be accessed by all I/Os and processing units. This SoC architecture is interesting for the space industry, the application processing unit can run COTS software (such as Linux) while the real-time processing unit handles more critical software or monitors the entire system. Moreover, FPGA is useful for handling communications between I/Os and processing units and is becoming an essential electronic component for satellites.

As described in the previous paragraph, in the context of space systems the real-time processing unit is envisaged to take measures. This unit is made of an ARM Cortex-R5 implementing the ARMv7-R architecture. It is a dual-core

with one L1 cache per core and a L2 shared cache. To ensure reliability it can be used in lock-step, i.e the two cores execute the same instructions and their outputs are compared.

Besides, the application processing unit will run Linux. This processing unit is made of an ARM Cortex-A53 processor with a frequency going up to 1.5 GHz, which implements the ARMv8-A architecture. It is a quad-core, with instruction and data L1 cache along with a shared L2 cache and an MMU (memory management unit). The L2 cache is a 1MB 16 way set-associative cache with ECC shared between the CPUs. It means that it's containing 15 625 lines of 64 bytes. The L2 cache is unified i.e it contains both data and instruction from the L1 memory system. The L1 instruction cache is a 32 KB 2 way set-associative cache with ECC independent for each CPU. It's containing 500 lines of 64 bytes. The L1 data cache is a 32 KB 4 way set-associative cache with ECC independent for each CPU. It can contain 500 lines of 64 bytes. In the scope of this paper, all software applications are executed on the processing unit.

III. Problem formalisation

Our work is focused on analyzing perturbations between software applications on the same hardware platform and respecting the properties described in the introduction of this paper. All software applications will be executed in a Linux process and processes will only contain one thread.

First of all, Linux hasn't been created to be a real-time operating system. As our software applications are connected to complex sensors they sometimes need to be executed periodically and respect a deadline. Linux developers developed real-time mechanisms for Linux. The central question of our work is to ask if real-time constraints of our use-cases can be respected with or without Linux real-time mechanisms. In this paper, we will provide some elements answering this question.

Second of all, the hardware part of our usecase is very important, as execution time can be modified by a memory response delayed by shared caches or busy busses. Interactions between software and hardware is made by two channels: systems calls and drivers. System calls allow software to use special features provided by the operating system. A complex operating system such as Linux proposes many different system calls and some of them might not be suitable for space avionics. In order to measure and then control interference caused by system calls the first step is to list system calls usable in a satellite avionic. Once the list is complete an analysis of the memory impact of each system call will be made. For example, if one software uses shared pipes they can be accessed by others software. These accesses can generate interference, thereby a space isolation of resources accessed through system calls is needed. In the scope of this paper, we consider all systems calls related to the file system, specifically the write and read system calls. In fact, using files is a useful feature for an on-board software.

The main purpose of drivers is to abstract the handling of devices. Drivers code runs in kernel mode like the operating system and can thus access the physical memory. As they are closely related to device characteristics most of the time drivers are written by the company who developed the device. This detail poses a problem in our context, the fact that a non trusted component is running inside the kernel is a clear breach of the isolation sought in this work. The last question of our work is, how to isolate shared software resources such as drivers in a Linux system. This question won't be addressed in this paper.

IV. Linux based solution

Linux is a widely used operating system. Created in the mid 90s Linux is developed by software engineers all around the world. A lot of companies invest time and money in the development of Linux. Nowadays, Linux is used by a lot of people and deployed on many different kind of hardware. It can run on desktop computers, servers, supercomputers but also on IoT and embedded devices. Most of developers learn to use Linux and its development environment which make it a standard in the computer industry. One of most powerful advantages of Linux is its reusability. The open-source philosophy allows to reuse libraries and drivers developed for Linux on different hardware. All these advantages make Linux useful in an embedded devices context. However, it is not conceived for critical applications and the development process of Linux is far from those used in real-time/critical operating system. Critical embedded industries want to be part of the Linux adventure and get all the benefits of using it in their products. Medical, aerospace, automotive, and industrial automation are considering the use of Linux for critical sections. This new approach of critical software asks a lot of questions due to the fact that qualification or certification of open-sourced Linux is certainly impossible. Thereby the embedded operating system verification paradigm has to be changed. In this section, configurable characteristics of Linux will be presented. These characteristics are appropriate for ensuring the properties described in the introduction.

A. Real-time scheduler: SCHED_DEADLINE

For Linux to be used in a critical embedded systems context it need to have real time properties. Thus, process needs to respect their deadline and period cannot be fulfilled with the default Linux scheduler. Nevertheless, Linux has multiple scheduling policies. Each Linux process can have a different scheduling policy and many of them are real-time oriented. In this article we will target the SCHED_DEADLINE policy. It allows a process to have a period and a deadline. Its implementation uses GEDF (Global Earliest Deadline First) in conjunction with CBS (Constant Bandwidth Server). To ensure an optimal scheduling, the scheduler needs to know the process runtime (cf figure 1). This runtime must be greater than its average computation time (or WCET for hard real-time). These 3 properties will be used by the scheduler to ensure the scheduling policy. CBS throttles threads attempting to over-run their runtime to guarantee non-interference between tasks.

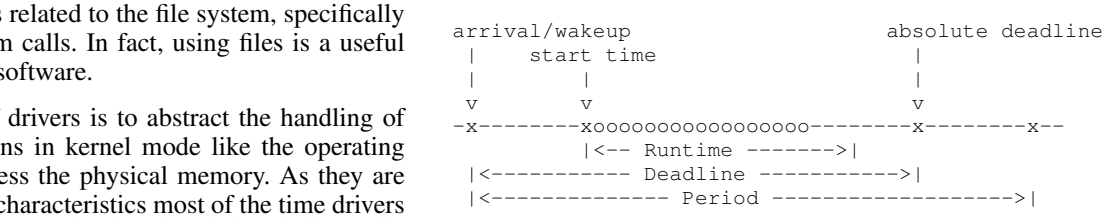

Fig. 1. Parameters of a SCHED_DEADLINE Linux process (man sched(7))

In the Linux system, SCHED_DEADLINE threads have the higher priority. In other words, if one SCHED_DEADLINE thread is runnable it will preempt threads scheduled by another policy.

B. Real-time kernel

Scheduling is not the only characteristic needed for an operating system to be considered as real-time. In Linux the PREEMPT-RT patch has been created to adapt the kernel to a real-time context. This patch is now merged into the upstream Linux stable version. As presented in the previous section, the Linux scheduling can be real-time and implement rt-throttling to ensure that no tasks hang the system. The PREEMPT-RT patch also enables the priority inheritance mechanism in the Linux scheduler. It allows a task with a low priority to take an higher priority if it is blocking a mutually exclusive resource. Thereby, the high priority task waiting for the mutually exclusive resource will be executed as soon as possible.

One important aspect of the Linux PREEMPT-RT patch is the preemption model. In the mainline kernel, plenty of the code is non preemptible which can delay the tasks execution time. To reduce the impact of this, it is possible to make the kernel in a fully preemptible mode. It means that all kernel code is preemptible (except a few critical parts). Large preemption disabled sections are split with locking constructs. Threaded interrupts handlers are forced i.e interrupts handlers run in a threaded context and new mechanisms are implemented such as rt_mutex and spinlocks which allows preemption in mutexes and raw spinlocks.

V. Methodology

A. Protocol

In this section we describe the protocol used to get the results presented in this paper. For each metric we want to retrieve two measurements. One measurement before calling the calculus function and one measurement after. Then, the difference between those two measurements is made to obtain the value of the metric during the execution of the calculus function. For example, to retrieve the number of L2 cache refill made by one execution of the AOCS calculus function, we measure the value of the L2 cache refill before calling the function and after calling the function. Then we can take the difference between those two values to get the number of L2 cache refill made during one execution of the AOCS calculus function. Note that the measurements are only taken on the AOCS process and not on the I/Os processes. It means that writing and reading from the pipes is not considered in the measurements. Also all measurements are taken on one process only which is considered the victim of the experiment. All others processes (users and kernel ones) and all kernel activities are considered the attackers of the system. In the ARM Cortex A53 used in these experiments performance counters registers are 32 bits wide. As we are running an AOCS process running at a frequency of 8Hz for 16 000 steps, the entire execution takes around 33 minutes. During such a long time, the 32 bits registers can overflowed, in this case the counts restart at zero. In most cases this is not a problem as we are taking the difference between two values. However, when the overflow happens during the execution of the calculus function, the value of the difference is not correct. To repair those values we add 2^{32} to each non valid value. The only measure taken before reading the input pipe is from the clock because one important metric to analyze is the wake up date of the process. The code of the measurements is shown in the figure 2.

```
for (i = 0; i < nbStep; i++) {
    clock_gettime(CLOCK_REALTIME, ...);
    read(pipe, ...);
    ...

    read_counter(i);
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, ...);
    Aocs_step();

    read_counter(i);
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, ...);
    clock_gettime(CLOCK_REALTIME, ...);
    ...
}
```

Fig. 2. C code of the measure point inside the AOCS process

We collected a lot of data from the performance counters, mostly on hardware memory events (bus accesses, cache accesses, cache refills, ...). In the scope of this paper, data such as context switches or core migration during the execution of an AOCS process could be interesting. Unfortunately, we weren't able to retrieve those data precisely enough. This is due to the fact that there is no way to know when a process is migrated or preempted. It is possible to get the number of the core the process is executed on, but nothing assures you that this won't change in the next moments. Thus, we aren't enable to get information about context switches or core migrations by retrieving hardware data.

In all these experiments we consider a "light" Linux distribution made with Yocto. The Linux kernel used in this paper is a 5.4 version of the Xilinx Linux kernel found in the official Xilinx Github [20]. According to Xilinx recommendations ([21]) we used the Yocto zeus version from the official Yocto repository ([22]). Using Yocto allows a complete theoretical control of what's inside the Linux operating system. In the scope of this article, each application is modelled by a Linux process and each Linux process have exactly one thread i.e threads = processes.

B. Measures impacts

Measure methods affect the experiment and therefore modify the results. Removing measure impacts is difficult and most of the time impossible. Therefore, the question is are those impacts negligible in these experiments ?

Every measures described in the scope of these experiments are taken inside a period of execution. It means that every code outside of the periodic loop isn't considered in the measures results. In particular, the loading and initialization phases of the application are not in the measures scope.

Measures presented in this article are coming from two different sources. The first one is clocks, managed by the operating system and used to measures execution time, relative

wake up time and processing time i.e time passed on the CPU between the beginning and the end of a cycle. Retrieving the value from those clocks takes approximately the same amount of time each time. The time value is encoded with two 64 bits integers, one for the nanoseconds and the other one for the seconds. Impacts of retrieving a timing value is then constant. The second source for the measures is performance counters. The activation of those counters doesn't modify the execution of the code. Retrieving the value of those counters imply reading CPU registers which aren't in the main memory. As for the clocks, the impacts of retrieving performance counters values are constant through the time. For both measures sources impacts of retrieving data are constant and don't affect the main memory i.e no memory accesses are performed. To be analyzed, those retrieved data need to be stored and made available after the experiment. The storage procedure can induce huge impacts on the measures, as the results have to be stored on the main memory. For example, the complete AOCS application has around 16 000 cycles which means that if all 6 performance counters and 2 clocks are used a storage space of $16000 * (6 + 2) * 64bits = 1MB$ will be needed. Accessing this space requires accessing memory pages which will generates memory accesses, page faults and other events related to the virtual memory which will be handled by the operating system. To reduce the effects of those memory accesses on the experiment, measures data is placed in local variable placed on the stack of the process and then copy to the memory outside of the measured section. This method will generate less impacts on the executed code but will alter a little the memory hardware state.

Finally, impacts of retrieving measures are constant and storage of the results is made outside the periodic loop. Therefore, to decide if measures impacts are negligible in this context we need to measure the constant part of the impacts. Figure 3 shows the results when no code is executed between two measures points. This chart represents the execution time induced by the measure itself which comprised between $7,5\mu s$ and $9,0\mu s$. As the WCET of the AOCS process is around 1ms we can conclude that the impact of the measurements is negligible.

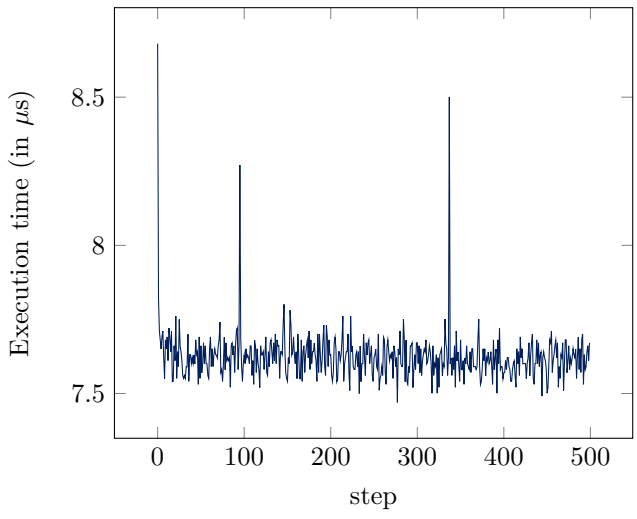

Fig. 3. Measurement of execution time with no code

VI. Experiments

Experiments presented in this paper aim at finding Linux configurations suitable for our context. Linux have a multitude of configurable points which can vary. To find suitable Linux configurations we will focus on varying those configurable points. For example, real time and isolation mechanisms such as the PREEMPT-RT patch or the Linux namespaces are configurable points which can vary from disabled to enabled.

In this section we present 4 experiments making measurements on different configurable points. The first experiment aims at comparing the use of same or different files by the same processes. The second experiment measures an impact of the Linux stock kernel on the execution time of an AOCS process. The third experiment measures a drift in the scheduler wake up date of a periodic process. The last experiment, is the same as the last two but with the PREEMPT-RT patch enabled.

A. Mono file versus multiple files

Files take an important place in our experiments. Not only they are used to store inputs and outputs of the AOCS application, they also contain instruction data. In fact, executables are files stored in the disk (SD card in our specific case). The file management is completely handled by the operating system, through multiple system calls, which try to optimize these accesses. In this context, accessing files is a source of interference from the operating system on applications but also from other applications. For example, two different applications trying to read to the same file will generate interference and the behavior will be different as if the application was alone on the system. The first experiment presented in this paper aims at exhibiting the differences between applications using the same files and applications using completely different files. This configuration could happen when using shared libraries as it is very common is the Linux development world.

In this experiment, two groups of applications were created based on the AOCS code. The first group is composed of AOCS applications using the same executable file and the same I/Os files. For the second group each executable and I/Os files are duplicated. For each group the same experiment was performed and is described as follows. The experiment begins by executing one AOCS application and measuring execution time, process time, wake-up time and performance counters related to memory such as L2 cache refill or bus accesses. The results from this first step is used for comparison and must be equivalent in both groups. The next step of the experiment is to increase the number of parallel execution. The maximum number of launched applications is 30 which represents 90 processes on the operating system (30 AOCS processes, 30 input processes and 30 output processes). In the context of this experiment, measurements are taken on only one AOCS process considered as the victim whereas all others AOCS and I/Os processes are considered attackers.

In the results of this experiment we observed that when the files are the same the performance are better on average. The worst performance is better than for the other group of processes. The figure 4 shows the average execution time for the AOCS process when others AOCS processes are executed in parallel. There is one curve for the first group and the other curve is for the other group. In these curves we can notice

aberrant points as for example, the point 3 of the mono file curve. Linux isn't deterministic and induce a lot of variability in the system, this is why we may have these points. In this paper we are focusing on the curves trends and not the particular values. Note that we did repeat those experiments a few times and these aberrant points aren't always at the same spot. Although, we do not have enough data to make statistics which could lead to removing these points.

This chart shows us that in average the group using the same files is executed approximately two times faster. Moreover, both curves can be separated in two phases. The first phase is between 1 and 4 AOCS parallel while the second phase is from 5 to 30 AOCS in parallel. In the first phase, both curves are increasing and in the second phase they seem constant. The hypothesis behind these observations is that the pivot point of 4 AOCS processes in parallel is related to the cores number of the ARM Cortex A53 the processes are executed on. In fact, many memory hardware resources are shared between the 4 cores of the processor (buses, L2 cache, memory controller, RAM,...). When 2 processes are executed in parallel, they both need to access those resources and thus hardware resources aren't always ready to respond to the cores requests. When executing more than 4 processes in parallel, cores are always occupied. If it's not by AOCS processes they will be by inputs or outputs processes. This means that there is always 4 processes running at the same time and thus memory hardware resources are busier than if there were less than 4 processes in the system. Parallel use of the hardware resources can explained this observation, but these are not the only hardware interference possible in the system.

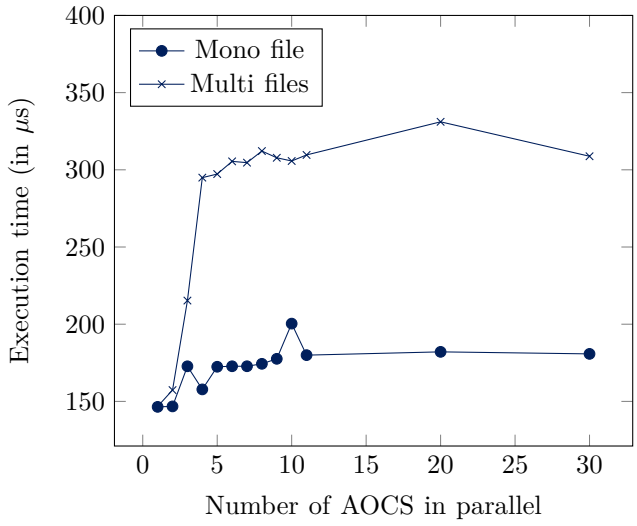

Fig. 4. Measurement of execution time in average comparison of the two processes groups

Modifying the state of a shared memory hardware device is another kind of interference. The figure 5 shows the number of L2 data cache refill i.e the number of times a cache line has to be loaded from the main memory, from the same two groups of processes. As for the previous curve, it has been obtained by executing the AOCS processes in parallel during 16 000 steps at a frequency of 8Hz. The curve is similar to the previous one, but the form of it around the pilot point is smoother. As a matter of fact, we can observe a behavior change around the point 4 but the transition between the two phase is not abrupt. In this case, we still can see an increasing between the point 4 and 8 in both curves but the slope of the curves is smaller and the shape of the curves changed from exponential to logarithmic before becoming constant. In other words the first phase of the curves (between 1 and 4) is similar to the previous curves and thus induces the shape of the execution time curves. In fact, if there are more L2 data refill the execution time of the process will increase. Then, the second phase shows a smaller rise of the curves because increasing the number of parallel processes means increasing accesses and more accesses means more L2 cache refill. The last question raised by those charts is why is the second phase constant ? At first, it does not seem logical that the impact on an AOCS process is the same when there are 10 others AOCS processes executed in parallel and when there are 30 AOCS processes executed in parallel. An hypothesis that could explain this behavior is that the cache could be filled up or at least all the ways accessed by the AOCS process are filled up with other data. In this case, each time the AOCS process is scheduled its data from the last execution has been evicted from the cache.

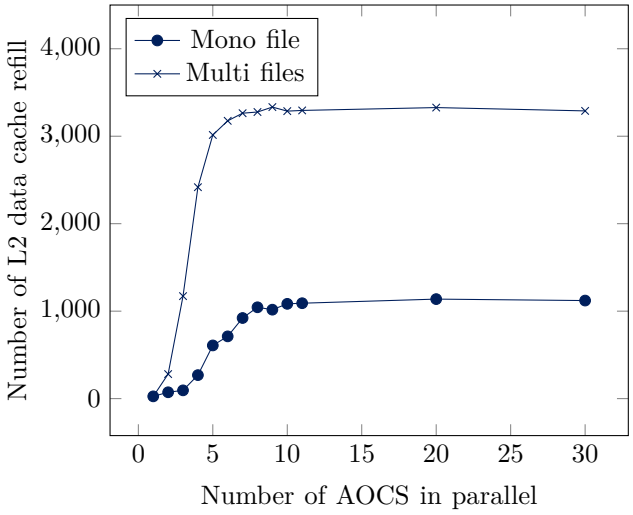

Fig. 5. Measurement of L2 data cache refill in average, comparison of the two processes groups

Now that the shape of the curves is described, let us focus on the differences between the two groups of processes. In both charts we can see a non negligible difference between the mono file curve and the multi files one. The mono file curve shows less L2 cache refill with a maximum of 1000 while the maximum of data cache refill for the multi files curve exceeds 3000. This means that when using the same files, there is a better re usability of the data which can be found in the caches. But address space of different processes in Linux are completely separated. In other words, data from a process A cannot be shared with a process B and thus process B cannot use data from the cache already loaded by process A. Then how can we explained the previous results?

When accessing a file in Linux a process uses system calls. All actions related to a file is then handled by the operating system. Even if there are 30 different processes executing in parallel with different address space, they are all taking their

data from the same file. All reads and writes are performed by the operating system from the kernel address space. Which means that all processes reading or writing to the same file can used the same cache line without needs to refill it.

To conclude, using the same files both as executable and data I/Os for parallel processes change drastically the performance we observed. We showed in this sub section that the impact of using or not the same files in parallel execution of processes is impacted by two distinct phenomenon.

B. Periodic interference peaks

During the first experiment we observed that when an AOCS process was executed alone on the system its execution time was impacted by interference. In fact, periodic peaks are present in the execution time curve as shown in the figure 6. We investigated to know if those peaks comes from the application code itself or if it is interference generated by the operating system. In this experiment, there are no difference between mono or multi files as we are only considering one AOCS process.

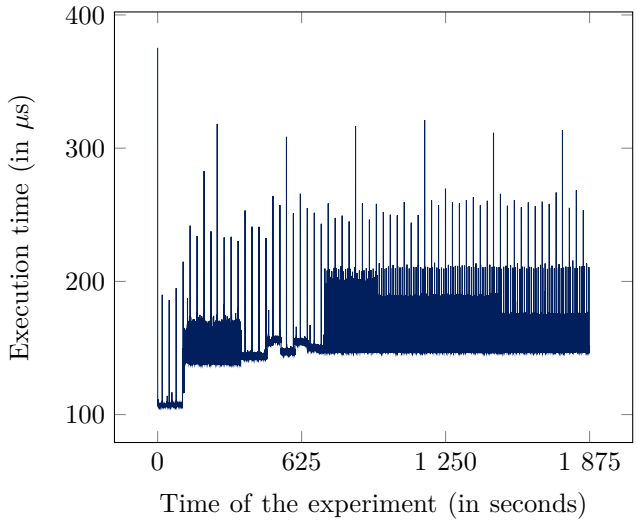

Fig. 6. Measurement of execution time for each step of an AOCS application with a period of 125 ms without any other application running in parallel

To determine the origin of those peaks two observations were made. First, the peaks don't appear at the same steps for every execution of the AOCS process. We know that the AOCS code we use doesn't have the same behavior for each step but one step will always execute the same code if its inputs are the same. In the scope of these experiments, I/Os files aren't modified. After this observation, the hypothesis that the peaks do not come from the AOCS code itself seems plausible. To assure that we had the right hypothesis we try to redo the measurements with a modified period. In this case, the period of the peaks didn't change. The figure 7 shows the results for a period of 50ms. In fact, there are 20 peaks in 625 seconds in both curves.

It is clear that the period of those peaks isn't based on the steps of the AOCS application. Therefore, we can now assure that those peaks come from the operating system. Note that these peaks also appear in the figure 2 which also confirm that the interference don't come from the AOCS algorithm.

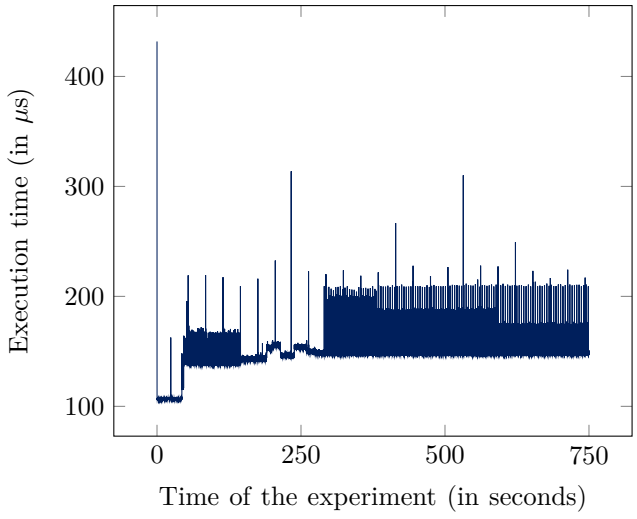

Fig. 7. Measurement of execution time for each step of an AOCS application with a period of 50ms without any other application running in parallel

One other simple hypothesis could be that those peaks come from a context switch. To answer this question an analyze of the difference between processing time and execution time can be made. The execution time counts the time spent by the processor when executing only the AOCS code. The processing time is the difference between the end date and the beginning date of the process. Therefore, if a context switch occurred during the execution of the process, the processing time will be greater than the execution time. We observed that there is no correlation between this difference and the peaks observed in the execution time chart. In fact, the processing time is not greater when there is a peak in the execution time curve. The figure 8 shows the result of the difference between processing time and execution time. Therefore, this hypothesis can be excluded because there are no peaks in this curve.

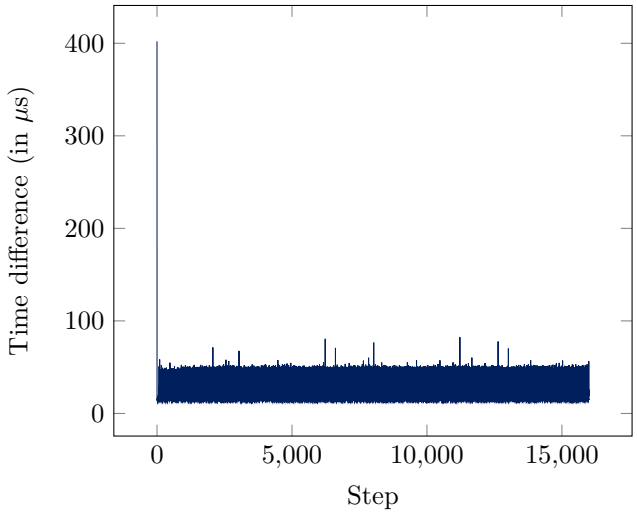

Fig. 8. Measurement of the difference between processing time and execution time.

Now the question is to find where the peaks come from. We can note that there are two types of peaks. There are smaller peaks with smaller period and bigger peaks with bigger period.

We could also note that there are exactly 9 small peaks between two big peaks. It seems that 2 periodic phenomena impact the AOCS process. There are 2 hypothesis for the second phenomenon with the bigger peaks. Either it is a particular execution of the first phenomenon occurring each 10 peaks. Or it can be from a completely different source and the 2 phenomena are synchronised for some reasons.

The measures taken from the performance counters are difficult to analyze because the AOCS algorithm doesn't have the same behavior at each steps. In particular, it doesn't make the same amount of memory accesses and doesn't access the same data in the main memory. Thereby, it is hard to know if the AOCS algorithm play a role in the value of those peaks or if they are only induced by the environment.

This questions lead to create another algorithm to help us find the source of the observed peaks. The new application is reading from an integers array stored in memory. To ensure that the compiler won't remove those unused accesses a sum of each integer of the array is performed. Moreover, this sum is repeated periodically at a frequency of 8Hz and uses the SCHED_DEADLINE policy from the Linux scheduler. We also implemented the same measurements techniques as for the AOCS process. The measures from this new algorithm give the same peaks in the execution time data. This confirmed once again that the operating system is altering the behavior of the process. What's interesting in this case is that the memory accesses and the L1 data accesses are constant at each steps. Each new refill from the L1 and L2 data cache can then be considered an interference from the operating system. As shown in the figure 9 there are also peaks in the L2 data cache refill. Those peaks occur at the same steps as the one observed in the execution time curve.

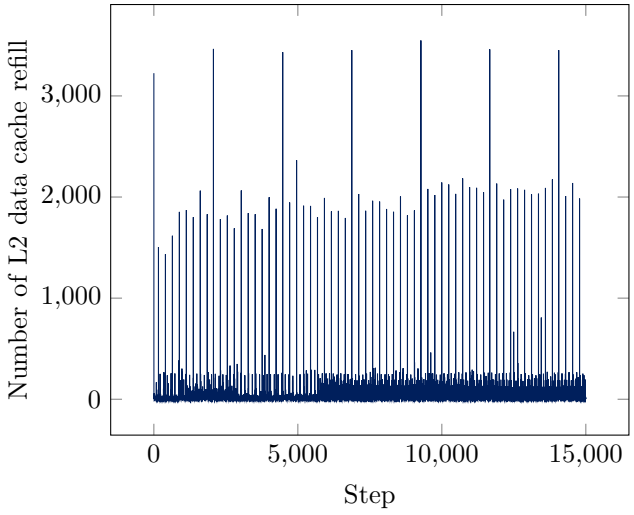

Fig. 9. Measurement of L2 data cache refill for the AOCS application

At this point the main hypothesis is that Linux or a hardware component is flushing all the memory hardware system (L1, L2 and TLB), either because it makes a lot of accesses to the memory or because it performs a hardware flush (for example because of security reasons). After this action is performed the process is woken up and have to refill all its data in the memory system which takes time and generates the peaks we observed. This hypothesis also explains why the curve which shows the difference between processing and execution time presented in the figure 8 is not constant. In fact, as the process as to access the main memory the CPU has to wait, when the CPU is waiting the real time clock is running but not the one counting the time passed in the process. Note that these peaks are observed even for very small array. It means, that even when the process has only a small amount of data in the caches, the operating system ejects it. This also confirms the hypothesis of an entire flush of the hardware memory system.

Actually kernel code responsible of this flush phenomenon hasn't yet been identified. An analysis of the behavior of the system with the Linux Ftrace tool has been made but no correlation has been found.

C. Scheduler wake-up drift

In the embedded space system the wake up date of the periodic process is an important metric to measure. If the scheduler wakes up the process a little later or a little sooner it can induce a drift which will result in a desynchronization of the process and the sensors sending input data. This desynchronization could become a problem, especially since satellites are running for years without being rebooted. We measured the wake up date of the AOCS process with the CLOCK_MONOTONIC of Linux which is a system wide clock. The chart presented in figure 10 shows the relative wake up date (in ns) at each step. The first wake up date is taken as a reference. The rest of the curve is obtained by subtracting $step * 1.25 * 10^8$ at each wake up date. It shows that after 15 000 steps the wake up date is 40 μs sooner that it should have been if the scheduler used an exact period of 125.00 ms.

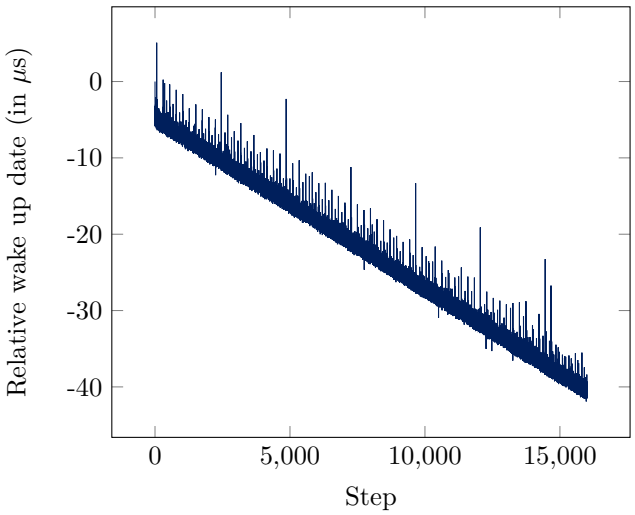

Fig. 10. Measurement of the wake up date for each step relatively to the first wake up date

The first observation made on that curve is that the scheduler seems to induce a drift in the wake up date. One hypothesis to explain this drift could be a desynchronization between the internal scheduler clock and the clock monotonic used in this experiment.

The second observation is that the peaks presented on the section B also appears on the wake up date curve. Those peaks are similar in the figure 10 and in the figure 6 even though they came from different execution. The scheduler drift seems to be linear but can be impacted by the same phenomenon than the AOCS process itself. We observe that on the same execution measures the peaks appear exactly at the same steps on both curves.

To help find what can induce this drift we look at the difference between two consecutive wake up dates. The figure 11 shows for each step the difference between the wake up date of this step and the wake up date of the step before minus $1.25 * 10^8$. In other words, a value of 1 at the step 50 means that the difference of the wake up date between the step 50 and the step 49 is 125ms + 1 μs.

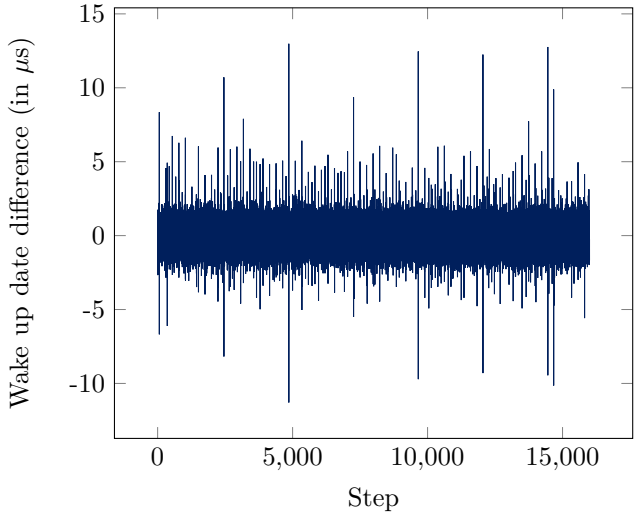

Fig. 11. Measurement of the wake up date difference for each step

In this curve we can remark that there are also peaks. As for the other curves, the peaks appear on the same steps in all different curves. This means that the two phenomenon producing those peaks both in the scheduler and the AOCS process are correlated. Moreover in the figure 11, the positive peaks seem to be correlated to negative ones. The figure 12 shows the same curve but only the first 100 steps. At the step 61 we could see a positive peak of around 8 μs. This peak is followed right after by a negative one at the step 62 of around -6.5 μs. The hypothesis behind this observation is that the scheduler try to catch up its delay at the step 61 by scheduling the next step a little earlier. But the positive delay at the step 61 is greater than the negative delay at the step 62. As a result the scheduler induces a positive delay at the step 63. By looking closely at the figure 11 we observe that this phenomenon is reproduced for each positive and negative peak. Another hypothesis could be that the time spent at executing the operating system code isn't taking into account for calculating the next period.

We can then generalise the rationale for the steps 61 and 62. Thus the scheduler induces a positive delay for each peak which should result in a positive delay in the entire measurement. However, the figure 10 shows that globally there is a negative delay.

Finally measuring wake up dates of the AOCS process showed two things. First, the phenomenon impacting the execution time presented in the last sub section also impacts the scheduling of the process. This impact induces a positive delay in the wake up dates. Second, the wake up date are globally drifting and the process at the step 15 000 is scheduled sooner than it should have been.

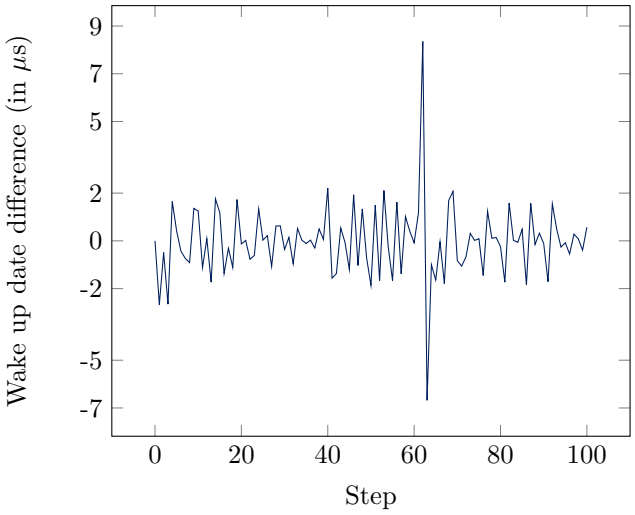

Fig. 12. Measurement of the wake up date difference for each step for the first 100 steps

As explained in the introduction of this sub section, this drift could be a problem if the system is not rebooted very often. This raises the followed question: does the phenomenon result from a desynchronisation between two internal clocks or a software behavior of the scheduler itself?

D. Impact of PREEMPT-RT

One of the first question when Linux is proposed in a critical embedded context is: is the PREEMPT-RT patch of Linux necessary ? We reproduced the 2 last experiments with PREEMPT-RT to find out if the real time patch reduces the observed phenomenon. The next paragraphs describe the obtained results for each experiment.

For the periodic peaks experiment adding PREEMPT-RT doesn't change the actual behavior. The peaks still appear in the execution time curve and are correlated to the peaks in the L2 cache refill curve. However, our observations are described as follows:

- The average execution time is the same with and without the PREEMPT-RT patch enabled. Globally, for each step the execution time isn't modified by PREEMPT-RT. We measured an average execution time of 145 393 nanoseconds without PREEMPT-RT and an average execution time of 146 429 nanoseconds with the PREEMPT-RT patch. This means that there is a difference of around one microsecond between the two average execution time.
- The bigger peaks have the same value with and without the PREEMPT-RT patch enabled.

- The smaller peaks are higher with PREEMPT-RT than without.

The first observation means that the PREEMPT-RT patch doesn't add any overhead to the execution time. However, it seems that the PREEMPT-RT patch induces a phenomena which results in a higher impact on the AOCS process. We don't have any hypothesis at this time of what causes this impact since we aren't sure of what causes the peaks in the first place.

For the scheduler wake up date drift the curves from both cases seem to have the same characteristics. In fact, the drift is still there and its slope is the same. The patch PREEMPT-RT doesn't have any impact on the wake up date drift.

Finally, in these two experiments, adding the PREEMPT-RT patch doesn't seem to modify the behavior except a small change in the heights of the smaller peaks in the execution time curve. In fact, using the SCHED_DEADLINE scheduling policy from Linux makes the AOCS process a high priority process. Without a lot of external interruptions the AOCS process isn't bothered very often. The scheduling policy is then enough for our process to be efficient.

VII. Conclusion and perspectives

A. Conclusion on the observation

The goal of this paper was to observe how the Linux kernel impacts the execution of real-time processes.

The first experiment showed us that system calls provided by Linux to access files seem to be optimised to increase the performance when multiple processes want to access the same files. In our experiment, using shared files produces less interference than using different files when reading the files even if the address space of processes are completely isolated by the operating system.

The next two experiments presented the impacts of the stock Linux kernel on the AOCS process. In the first experiment, we have periodic impacts increasing the execution time. In the second one, we found a drift in the wake up date of periodic processes. Both of these phenomena can be a problem depending on the chosen ϵ mentioned in the introduction. For instance, in the AOCS figure 6 shows that perturbations induced by Linux double the execution time. In this case, if ϵ (i.e margin associated with AOCS) is lower than the isolated execution time then this Linux-based implementation doesn't meet the requirements. However, the impacts described in those two experiments is constant and predictable through time. In fact, in all the measurements we made we found the same impact in both experiments. The execution time peaks are periodic with a constant period. For the scheduler wake up date drift, the slope is constant and around -40 microseconds for 16 000 steps ($\tilde{3}3$ minutes).

Finally, we did the same two experiments with another variation of our Linux kernel. In this case, we enabled the PREEMPT-RT patch to find if it has an impact on the two observed phenomenon. Only a small change in the value of the small peaks has been spotted but the global behavior didn't change. Thus, the PREEMPT-RT patch doesn't seem to impact the observed phenomenon.

B. Future work

For the future, the first thing we want to do is find the root cause of the two observed phenomena. In order to do that, we could add statistics tools to help understand the measured data and extract information on the phenomena. In this paper we focused on a qualitative rationale, introducing statistics will help for producing a more quantitative rationale. Another approach to find the root cause could be to use tools provided by the kernel itself such as *ftrace*. This tool will provide kernel information on what is actually executed by the kernel. Using kernel tools could also give information on core migrations and context switches.

The SCHED_DEADLINE policy used in this paper is based on the EDF algorithm. A future work could be to find a link between the scheduler wake up date drift and the formal equations used to code the SCHED_DEADLINE scheduler.

In this paper, we focused on the AOCS algorithm, in the future using other space applications will be useful to find other interference and characterize the already observed ones.

Finally, we would like to introduce containerization configuration in our kernel such as namespaces and cgroups. In fact, the goal of our work is to reduce interference between multiple applications and their environment in a Linux context. Linux provides isolation configuration and we would like to find if these tools could help seeking a certain isolation in our context.

References

[1] J. Rushby, "Partitioning in avionics architectures: Requirements, mechanisms, and assurance," SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB, Tech. Rep., 2000.

[2] J. Bredereke, "A survey of time and space partitioning for space avionics," 2017.

[3] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio, "Performance comparison of vxworks, linux, rtai, and xenomai in a hard real-time application," IEEE Transactions on Nuclear Science, vol. 55, no. 1, pp. 435–439, 2008.

[4] H. Leppinen, "Current use of linux in spacecraft flight software," IEEE Aerospace and Electronic Systems Magazine, vol. 32, no. 10, pp. 4–13, 2017.

[5] J. H. Brown and B. Martin, "How fast is fast enough? choosing between xenomai and linux for real-time applications," in proc. of the 12th Real-Time Linux Workshop (RTLWS'12), 2010, pp. 1–17.

[6] D. Bristot de Oliveira and R. Oliveira, "Timing analysis of the preempt rt linux kernel," Software: Practice and Experience, vol. 46, pp. n/a–n/a, 05 2015.

[7] D. Bristot de Oliveira, D. Casini, R. Oliveira, and T. Cucinotta, "Demystifying the real-time linux scheduling latency," 07 2020.

[8] C. Emde, "Long-term monitoring of apparent latency in preempt rt linux real-time systems," 2010.

[9] T. Cucinotta, "An efficient and scalable implementation of global edf in linux," 01 2011.

[10] D. Faggioli, F. Checconi, S. Superiore, S. Anna, M. Trimarchi, and C. Scordino, "An edf scheduling class for the linux kernel," 01 2009.

[11] P. Mckenney and D. Sarma, "Towards hard realtime response from the linux kernel on smp hardware," 01 2005.

[12] J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. Stern, "Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel," SIGPLAN Not., vol. 53, no. 2, p. 405–418, mar 2018. [Online]. Available: https://doi.org/10.1145/3296957.3177156

[13] J. Kim, P. Shin, S. Noh, D. Ham, and S. Hong, "Reducing memory interference latency of safety-critical applications via memory request throttling and linux cgroup," in 2018 31st IEEE International System-on-Chip Conference (SOCC), 2018, pp. 215–220.

[14] J. Kim, P. Shin, M. Kim, and S. Hong, "Memory-aware fair-share scheduling for improved performance isolation in the linux kernel," IEEE Access, vol. 8, pp. 98 874–98 886, 2020.

[15] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, ser. PACT '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 367–376. [Online]. Available: https://doi.org/10.1145/2370816.2370869

[16] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the linux kernel," ACM SIGBED Review, vol. 16, no. 3, pp. 33–38, 2019.

[17] H. Grip, J. Lam, D. Bayard, D. Conway, G. Singh, R. Brockers, J. Delaune, L. Matthies, C. Malpica, T. Brown, A. Jain, A. Martin, and G. Merewether, "Flight control system for nasa's mars helicopter," 01 2019.

[18] B. B. Brandenburg and J. H. Anderson, "Joint opportunities for real-time linux and real-time systems research," 2009.

[19] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A measurement-based analysis of the real-time performance of linux," in Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium, 2002, pp. 133–142.

[20] (2021) The github repository of the xilinx linux kernel. [Online]. Available: https://github.com/Xilinx/linux-xlnx/tree/xlnx_rebase_v5.4

[21] (2021) Xilinx wiki on yocto. [Online]. Available: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841883/Yocto4

[22] (2021) Yocto zeus version repository. [Online]. Available: http://layers.openembedded.org/layerindex/branch/zeus/layers/