Evaluation and analysis of Linux-based applications on multicore processor in a space environment

Thomas Beck

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut Supérieur de l'Aéronautique et de l'Espace

Présentée et soutenue par :
Thomas BECK

le mardi 17 janvier 2023

Titre :

Evaluation et analyse d'applications Linux sur processeur multicoeur en environnement spatial

---

École doctorale et discipline ou spécialité :
ED MITT : Informatique et Télécommunications

Unité de recherche :
Équipe d'accueil ISAE-ONERA MOIS

Directeur(s) de Thèse :
M. Frédéric BONIOL (directeur de thèse)
M. Jérôme ERMONT (co-directeur de thèse)
M. Luc MAILLET (co-directeur de thèse)

Jury :
M. Daniel HAGIMONT Professeur INP-ENSEEIHT, IRIT - Président
M. Frédéric BONIOL Directeur de recherche ONERA Toulouse - Directeur de thèse
M. Jérôme ERMONT Maître de conférences INP-ENSEEIHT, IRIT - Co-directeur de thèse
M. Emmanuel GROLLEAU Professeur ISAE - ENSMA - Rapporteur
M. Luc MAILLET Ingénieur Airbus Defence and Space - Co-directeur de thèse
Mme Claire MAIZA Maître de conférences Grenoble INP-Ensimag Verimag - Examinatrice
M. Laurent PAUTET Professeur Institut Polytechnique de Paris, LTCI - Examinateur
M. Frank SINGHOFF Professeur Université de Bretagne Occidentale - Rapporteur

Un étudiant n'est pas un sac que l'on remplit mais une bougie que l'on enflamme.

VLADIMIR ARNOLD

Remerciements

Tout d'abord j'aimerais remercier mon directeur de thèse, le directeur de recherche Frédéric Boniol, pour sa patience, son expertise et son soutien sans faille dans l'encadrement de mes travaux.

J'aimerais remercier ensuite mon co-directeur de thèse, le maitre de conférences Jérôme Ermont. Merci pour ton appui technique et la pertinence de tes remarques mais aussi pour ta bonne humeur qui a rythmée nos nombreuses réunions.

Un énorme merci à mon directeur de thèse du monde socio-économique, le docteur Luc Maillet, pour son soutien quotidien. Nos nombreux échanges m'ont fait grandir tant sur le plan scientifique que personnel.

Je souhaite ensuite remercier les membres du jury pour leur temps et leur engagement. Je remercie plus particuliérement, le professeur Emmanuel Grolleau et le professeur Frank Singhoff pour leur relecture approfondie du manuscrit, la qualité et pertinence de leur rapport ainsi que leurs interventions durant la soutenance. Merci aux membres du jury, la maitresse de conférences Claire Maiza, le professeur Laurent Pautet et le professeur Daniel Hagimont pour leurs questions enrichissantes durant la soutenance.

Je tiens à exprimer ma plus profonde gratitude envers Airbus Defence and Space et L'Association Nationale pour la Recherche Technologique (ANRT) pour le soutien financier sans lequel cette thèse n'aurait pu être possible. J'ai une pensée toute particulière pour Antoine Certain et Dominique Pibrac qui ont cru en moi dès le début. Un grand merci à Anthony Clerquin pour son accompagnement durant les derniers mois de ma thèse. Enfin, je ne peux oublier Franck Wartel qui a été présent depuis le commencement et qui m'a toujours encouragé.

J'aimerais enfin remercier tous mes collègues D'Airbus, de l'ONERA et de l'IRIT pour tous les moments passés ensembles. Merci à Maxime pour les paris sportifs, soirées casinos et autres projets de croisières sur la Méditérannée. Merci à PJ pour ses précieux conseils et pour son rôle d'éclaireur dans la jungle administrative des thèses CIFRE. Merci à Sylvain pour les discussions sur la compilation, le langage C et son aide précieuse dans la réalisation des expériences de cette thèse. Merci à Anthony pour son amour des Avions, sa bonne humeur, sa joie de vivre et sa passion pour l'Argentine. Enfin, merci à Valentin pour le support technique, le redémarrage de ma carte, les soirées cinéma et les nombreuses discussions de cafés.

Je remercie chaleureusement toute ma famille et en particulier ma mère, mon père et ma soeur pour leur soutien inconditionnel durant mes années de thèse.
Pour finir je remercie Jo, pour son support quotidien, ses encouragements et ses nombreux sacrifices qui m'ont permis l'obtention de cette thèse.

Abstract

The emergence of a private initiative space industry has generated a race to deploy low-cost satellites. This race translates into a strong need to reduce the manufacturing costs of a satellite, which has increased in recent years. For on-board avionics, this reduction occurs in three aspects: the use of multi-core hardware platforms, the use of COTS operating systems and the integration of different software running in parallel on the same hardware platform. At the heart of this problem, stands the notion of interference defined by the impacts on the execution time of an on-board software induced by the execution of other on-board software or the operating system. First, the objective of this thesis is to provide a model based on timed automata capable of identifying memory interferences. The second part of this thesis is devoted to an experimental approach based on the measurements of internal clocks and hardware performance counters. The objective is to measure the impact of interference on a satellite attitude and orbit control application running on a Linux distribution specially designed for on-board software. The latest experiments focus on the reduction of disturbances by a containerization mechanism present in Linux: cgroups. This mechanism limits access to shared resources for a given group of processes. Finally, these modelings and experiments made it possible to provide recommendations on the use of Linux on multi-core platforms and in a real-time space context.

Résumé Français

L'émergence d'une industrie spatiale d'initiative privée a engendré une course au déploiement de satellite low-cost. Cette course se traduit par une forte nécessité de réduction des coûts de fabrication d'un satellite qui s'est accentuée ces dernière années. Au niveau de l'avionique de bord cette réduction intervient sur trois aspects: l'utilisation de plateformes matérielles multi-coeurs, de systèmes d'exploitation grand public et l'intégration de différents logiciels s'exécutant en parallèle sur la même plateforme matérielle. Au cœur de cette problématique, se dresse la notion d'interférence définit par la perturbation du temps d'exécution d'un logiciel de bord engendrée par l'exécution d'autres logiciels de bords ou du système d'exploitation. Dans un premier temps, l'objectif de cette thèse est d'apporter une modélisation à base d'automates temporisés capable d'identifier les interférences mémoires. La deuxième partie de cette thèse se consacre à une approche expérimentale basée sur les mesures des horloges internes et des compteurs de performances matériels. L'objectif est de mesurer l'impact des interférences sur une application de contrôle d'attitude et d'orbite d'un satellite s'exécutant sur une distribution Linux spécialement conçue pour les logiciels embarqués. Les dernières expériences se focalisent sur la réduction des perturbations par un mécanisme de containérisation présent dans Linux: les cgroups. Ce mécanisme permet de limiter l'accès aux ressources partagées pour un groupe donné de processus. Finalement, ces modélisations et expériences ont permis de fournir des recommandations sur l'utilisation de Linux sur des plateformes multi-coeurs et dans un contexte spatial temps-réel.

Contents

Chapter 1

Introduction

1.1 Context

1.1.1 Constraints

The biggest particularities of a spacecraft are the constraints induced by the space environment.

First of all, once the spacecraft is launched into space, there are no easy possibilities to reach it physically. The only examples of modification of a spacecraft in orbit are very expensive. This is one of the only human-made objects that have this particularity. It means that if a problem occurs it is not possible to press a stop button inside the satellite or to move manually the solar panels. Thus, high reliability is very important in the design and development of a spacecraft.

Second of all, the spacecraft is outside of the earth's atmosphere and sometimes outside of the earth's magnetosphere. Therefore, a spacecraft is vulnerable to radiations (particles trapped in the Earth's magnetic field, particles shot into space during solar flares, and galactic cosmic rays which are high-energy protons and heavy ions from outside our solar system [1]). These radiations can be very problematic since they modify the state and behavior of electronic components. For example, computer memory can be modified by radiation and thus the data inside the memory will be corrupted or even worse, a latch-up could destroy transistors.

Another constraint is related to the thermal aspect of the spacecraft. Without any conductive fluid the thermal dissipation is completely different. It results in a much larger temperature difference between the face of the spacecraft exposed to the sun and the one plunged into darkness.

Finally, a spacecraft is subject to mechanical constraints during the launch but also in orbit due to the temperature cycle in space and its own generated maneuvers.

1.1.2 Current technologies

Most all technologies used in the space industry are specially designed or customized for space. More specifically, onboard computers are generally using radiation hardened implementation of the SPARC V8 architecture with mono-core processors. The MIL-STD-1553 or CAN bus and SpaceWire are used for communications between the different computers, sensors, and actuators. Although these technologies are used in modern satellite, they are quite old. The MIL-STD-A553 was invented by the US army at the beginning of the 70s and is limited to a 1Mbps throughput. The Spacewire technology is more recent (invented in the 90s) and has a better throughput (400Mbps maximum), even if it is used with much lower throughput than its maximum.

For the CPUs, architecture such as LEON2 and LEON3 are used. These are mono-core processors running at a frequency of tens of MHz which is very low compared to the frequencies of modern processors around 5GHz.

1.1.3 Platform and payload differences

Inside a satellite, a separation is made between platform and payload. The platform is responsible for the support of the spacecraft by providing all the services necessary for the life of the satellite. The payload is responsible for the mission of the satellite. For example, the module taking photos in an earth observation satellite is the payload while the onboard computer managing the battery, the solar panels, and the communication with the earth is part of the platform. Generally, losing the platform means losing the spacecraft while losing the payload means losing the mission. The Figure 1.1 shows the different part of the SPOT5 satellite.

1.2 Approach

The space industry entered a new era of cost reduction. On-board computers and software have a role to play in this transformation. Developing onboard systems for satellites has an enormous cost both in time and money. As explained in the section 1.1, satellite onboard software has many particularities due to space constraints and historical choices.

The first idea to reduce avionics costs is to reduce the number of on-board computers in the spacecraft. In fact, on-board computers are custom for the space environment (mostly due to the ionizing radiations in the outer space) which make them much more expensive than other computers. Moreover, minimizing the number of on-board computers, have a positive impact on the weight and power comsumption of the spacecraft. Thus there is a need to integrate multiple software on the

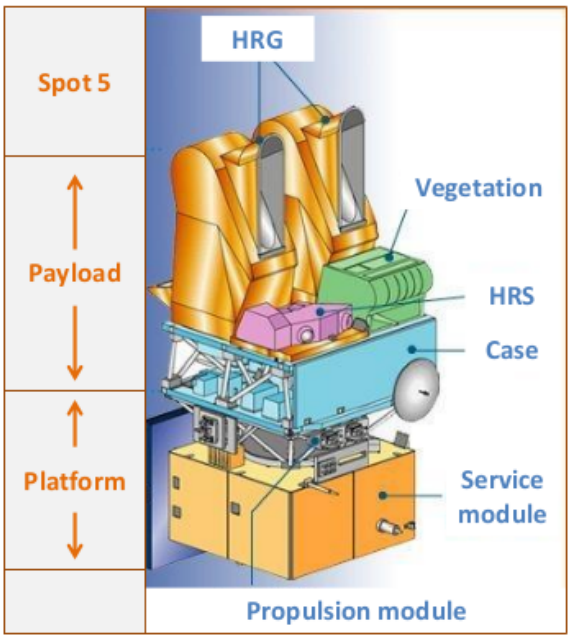

Figure 1.1: Payload and platform of the SPOT5 satellite

same hardware platform. To ensure maximum reliability, critical software is running on separate hardware platforms which reduces completely the risk of one software impacting another. The goal is to integrate software on the same platform with minimum effort. This means that software is not modified to fit with the other software running in parallel. Moreover, the operating system configuration does not depend on which software is running on top of it.

The second idea is to use non-custom computers (i.e COTS hardware). The price of a computer chip depends on the quantity manufactured. As there are just a few satellites made each year, the cost of a custom chip is extremely high. The space industry has a historical habit to use almost exclusively custom hardware. This is because, before the NewSpace era, the most important thing in a spacecraft was its reliability.

For the onboard software, this means that Zynq Ultrascale+ or RISCV platforms will be preferred to older hardware platforms. One change that comes with the use of COTS for onboard computers is that the number of cores will increase. For predictability and reliability reasons, mono-core was preferred for spacecraft onboard computers. But this is now incompatible with the COTS platform since mono-core disappeared from the market. Moreover, the new applications running in space need more computing power. Thus, the space industry has to use multi-core in order to benefit from the scaling factor COTS bring.

The last idea is to use common operating system such as Linux. Linux is known by most of the developpers and it has many available software libraries. Compared to an RTOS or a baremetal environment, it is easier and faster to develop a software running on Linux. In critical industries, such as space one, certification is often used to ensure confidence in the reliability and safety of a product. Like all elements in a critical product, the software has to be certified before being used in satellites. The European Space Agency (ESA) has its standards defined by the European Cooperation for Space Standardization (ECSS) in collaboration with the European space industry and several space agencies. Obtaining the certification for an operating system is very long and depends on the size of the operating system code. Usually, an embedded operating system is small and real-time oriented which means that there are not too many lines of code. The certification is obtained for a specific version of this operating system and each time a change is made in the code, it has to be certified. This is why, when an operating system is chosen it is not updated very often.

The main problem with this approach is that the operating system (like all the others elements in the spacecraft) is sometimes specially designed for the space industry. Developing software running on a custom operating system requires trained developers which increases the cost of the software and limits its reusability. Moreover, sustaining an operating system development is expensive and it is necessary to keep the operating system development team active, at least for correcting problematic bugs that can be found. This leads to the idea of using a more common operating system in spacecraft or at least a common programming model (such as POSIX compliance).

These changes have an impact on the design and development of the on-board computers and software.

1.3 Manuscript organization

This manuscript is organized in two parts. The first part focuses on the context in which this thesis takes place.

The second part is divided in four chapters. The first chapter proposes a modelling, made with timed automata, to identify and count memory interference. The second chapter evaluates the interference that impacts a on-board software through a Linux distribution. The third chapter presents the evaluation of an isolation mechanism available in Linux: cgroups. Finally, the last chapter provides analyses and recommandations based of the previously presented work.

List of Publications

1. Thomas Beck, Frédéric Boniol, Jérôme Ermont, and Luc Maillet. Impact of environment on the execution of a real-time linux process on a multicore platform. In *Proceeding of the 11th European Congress on Embedded Real Time Software and Systems*, 2022

2. Thomas Beck, Frédéric Boniol, Jérôme Ermont, Luc Maillet, and Franck Wartel. An automata-based method for interference analysis in multi-core processors. In *15th Junior Researcher Workshop on Real-Time Computing 2022*, page 7, 2022

3. Thomas Beck, Frédéric Boniol, Jérôme Ermont, and Luc Maillet. Mutual perturbations of fprime applications in a space context. *ETR 20*, page 29, 2021

Part I

Context

Chapter 2

Background

This section aims at providing a contextual background of this thesis' problem. First, it describes the actual architecture of onboard software in satellites. Then, an overview of different technical subjects that are necessary to understand the context is presented. Finally this chapter ends with a presentation of the problem statement of this thesis.

2.1 Computer architecture

This section is focused on describing the basic components of a computer architecture that are referred to in the scope of this thesis. This chapter describes architecture components based on the modified Harvard architecture since it is the standard one in the embedded context. This architecture is derived from the Harvard architecture which implements a separation between the instructions and the data both in storage and pathways. However, in the modified Harvard architecture the instruction memory can be accessed as data.

2.1.1 CPU description

The CPU is a central component of a computer since it is responsible for executing the instructions. It is composed of several elements such as:

- **Arithmetic Logic Unit (ALU):** is responsible for the integer arithmetic and bitwise logic operations.
- **Memory Management Unit (MMU):** is responsible for the translation between virtual and physical addresses. The MMU uses a page table for each process that is stored in the main memory and also a Translation Lookaside Buffer (TLB) which is a cache keeping the last used translations.

- **Memory Protection Unit (MPU):** is a hardware component that allows blocking access to memory area to specific processes.
- **Hardware Performance Counter:** allows counting hardware events such as memory access, bus access, cache refill, ... These performance counters are used to closely monitor the hardware behavior.

Modern CPUs are using instruction pipelining to parallelize the instructions execution. A CPU pipeline is composed of different steps (taking the classic RISC pipeline as an example):

- Instruction Fetch (IF): at this stage, the instruction is fetched from memory, reading the instruction's address on the program counter (PC) register, and is passed to the next stage. It usually takes one clock cycle if the instruction is in the cache.
- Instruction Decode (ID): at this stage, the instruction is decoded in order to be executed at the next stage. It is used to prepare the pipeline for the next instruction.
- Execute (EX): this stage is where the calculations happen. The ALU is used to perform the operation (if one is needed).
- Memory Access (MEM): at this stage, the data memory access is done.
- Write back (WB): Finally, this stage is doing the write back onto the registers.

As the accesses to data and instruction are separated in that type of architecture, it is possible to parallelize all the stages to increase the number of instructions per minute. For example, instruction A can be at the stage ID while instruction B is at the IF stage, and so on. This can complexify the architecture, especially when two stages are accessing the same registers during the same cycle, but modern processors have mechanisms to avoid that. An inconvenient is that the execution time of an instruction depends on the instructions before and after.

This thesis will principally focus on hardware counter but understanding how a CPU usually works is necessary to understand the rest of this manuscript.

2.1.2 Memory

Most of the time the main memory is made of Random-Access Memory (RAM) but usually in embedded context it is made of flash memory. The memory is divided into two parts, the first one is closer (in terms of access time) to the CPU but is volatile while the second one is much farther from the CPU and is non-volatile which means that the stored data does not disappear when the memory is not powered up. The non-volatile memory is most of the time hard disks or flash memory. For embedded systems, flash memory, EEPROM and PROM are preferred since it is more resistant

to shocks. The volatile main memory is accessed by addresses. Data is divided into single adressable unit with a constant size in bits which depends on the hardware architecture.

2.1.3 Caches

Caches are memory used to store data closer to the CPU pipeline to decrease the access time. Each time an access is made to the memory it goes to the caches first. Then a check is made to verify if the requested data is available in the cache. If the value is present it is a cache hit and the data is transferred to the CPU. If the value is not in the cache it is a cache miss and the memory access is propagated to the next memory component in the path (either another cache or the main memory). After a miss, the value retrieved from the memory is written in the caches (depending on the allocation policy) where the miss occurred so that future accesses to the same data can benefit from the value being present in the cache and thus be faster. The cache relies on the principle of locality which says that once an address is accessed it is likely to be accessed again in the near future.

Each type of cache has its own eviction algorithm which is responsible for finding which line to evict when a new line is retrieved from the memory. The simplest eviction algorithm is the LRU (Least Recently Used) which evicts the least recently used data. One important configuration for the cache is associativity, it describes how many positions a data can be in a cache. For example, in a fully associative cache, data can be placed anywhere. On contrary, in a direct mapped cache, data can be placed in only one specific line in the cache, which means that if the line is already filled it has to be evicted (there is no eviction algorithm in a direct mapped cache). Most of the time, caches are N-way associative meaning that data have N lines where they can be placed. This type of cache combines the advantages of the two other types of cache. It is relatively fast to check whether the data is in the cache or not because there are only N lines possible. At the same time, there is more than one line to store data from a specific address which decrease the probability of getting a cache miss.

Currently, there are no shared caches in space on-board computers but there will be in new space platforms. This thesis focuses on shared caches and how it impacts the system. In fact, sharing a cache can will have negative effects on the execution of the application running on the hardware platform.

2.2 Operating system

An operating system is a software application that is responsible for the management of the hardware, the software resources, and the services proposed to applications running on it. There are a lot of operating systems and the most used in embedded systems are VxWorks, QNX or Integrity OS. This section aims at presenting operating system fundamentals that will be necessary to understand the rest of this thesis. However, this section does not describe an operating system in particular but the concepts that are used to implement one.

2.2.1 Process management

One of the most important roles of an operating system is to manage processes. In this thesis, the notion of process will be used rather than task or thread. It has been chosen because it corresponds to the definition used in Linux. A process is an instance of a computer program running in the operating system. A process should not be confused with a computer program or application which represents the code written in a specific format (ELF, Mach-O, or EXE for example). This means that it is possible to create multiple processes with the same code. While it is the same code and thus the same algorithm that will be running, the operating system will not consider it the same object since multiple processes will be created.

To manage the process operations the operating system needs to maintain consistent pieces of information during the life cycle of the process. The first useful information is an identifier often called PID for Process IDentifier which allows the operating system to identify each process. This identifier is unique and once it has been used by one process it cannot usually be reallocated to a new process until the next reboot of the operating system.

The operating system also keeps the context of the process containing:

- The program counter keeps the address of the next instruction to be executed by the process.
- The page table address stores the translation between the page address and the virtual memory address. The virtual memory functioning will be described in the section 2.2.3.
- The address of the heap is where the dynamically allocated memory is stored and the address of the stack is where the local variables for the called functions are stored.
- The CPU registers states that are kept by the operating system when the process stops executing on the core. This is because when the process will be authorized to run on a core it will need to have the same state as when it stopped running.

The operating system also keeps all information related to software resources managed by the operating system such as shared memory areas, locks resources, and open files.

The scheduling information is also important for the operating system since it will be used by the scheduler to manage which process is executed where and when. This will be described more precisely in the section 2.2.2.

Finally, the operating system also keeps other information that depends on its implementation and that can be used for accounting or internal mechanisms. All this process information is stored in a process table inside the operating system environment.

There are different types of operating systems in the way they manage the OS services or the isolation of hardware resources. The scheduling algorithm is one of the most differentiable characteristics of an operating system.

2.2.2 Scheduling

The operating system is responsible for the scheduling of the processes, meaning that it has to choose where (on which core of the processor) and when a process can be executed. The part inside the operating system that manages the scheduling is called a scheduler. The scheduler takes decisions based on information and on scheduling algorithms (several of which do exist, with different properties).

One of the most important pieces of information is the scheduling state of the process. Each implementation of an operating system has its own custom process scheduling state but the three majors are: *ready*, *waiting*, and *running*. A process is in the state *running* when it is running on a core. The state *waiting* is for processes that are blocked because they need information or wait for synchronization event. It is the case when a process is sleeping or when it waits for a blocked resource to be available for example. The last common state is *ready* and it is used for a process that is ready to be executed on a core. In this state, the process is waiting for the running process to finish.

The scheduler stores the processes references in different queues. These queues are organized based on the priority, the state, the deadline, or any other information on the processes depending on the scheduling algorithm used.

When the scheduler decides to stop the execution of a process it has to suspend the execution of a process. It will stop the current process, save its current state (CPU registers, heap, stack, virtual address space context and program counter), restore the state of the new process, and execute it.

In this thesis different scheduling will be used but the objective is to evaluate the impact of these different algorithm on the system and not analyze the algorithms themselves.

2.2.3 Virtual memory

The last important aspect of an operating system is the management of virtual memory. When the developers write the code of a computer application, they do not know which memory area will be allocated for the execution of the program. Therefore, they cannot decide which memory addresses to use to store local variables. This is why processes are executed within a virtual address space.

For each process, there is a page table that stores the translation between virtual addresses and physical addresses. This page table is stored in memory and the address of the page table is stored in the process table in the operating system. The translation is made by the MMU (memory management unit) which is located in the processor between the CPU pipeline and the memory.

Generally, in the embedded context the memory is not virtual. Therefore, all the processes are using physical addresses which makes the development much complicated. This is not related to the level of technology used, many recent processors do not have an MMU while some old processors have one. Not using virtual memory makes the memory accesses more predictable and thus it is easier to respect the real-time constraints.

2.3 Virtualization

Virtualization mechanisms can be used in different contexts. In the scope of this thesis, virtualization can be used to isolate processes. Virtualization is software that virtualizes hardware or software functionalities to create a virtual computer system. This section presents different kinds of virtualization systems in the scope of this thesis.

2.3.1 Virtual machines

The first type of virtualization is also the most known and it is called a virtual machine. A virtual machine can be launched either by a running operating system or by a hypervisor. Its role is to virtualize the hardware to run software (most of the time an operating system) on this virtual hardware. One example of common usage is to run a Linux virtual machine on a Windows system. To do so, there are two possibilities: The first one is to run a normal Linux distribution without any particular changes to adapt to the virtual machine. The other one is to use a special Linux distribution that will be adapted to the virtual machine, this is called para-virtualization.

In the first case, it will be easier to implement but the system will be slower because the virtual machine has to completely virtualize the hardware. In the second

case, it will be much faster but a custom Linux distribution is required. In both cases, two software running on different virtual machines but on the same hardware do not have any clues about one another. In fact, from the process point of view, it is running alone on the machine. Virtual machines can thus be used to isolate multiple software by using different virtual machines. Note that the virtual machines still have to share the same CPU time. A process running in a virtual machine will have less cpu time available than if it was running alone on the operating system.

2.3.2 Hypervisors

Hypervisors are software designed to host virtual machines. It means that compared to an operating system they cannot execute a process. There are two types of hypervisors:

- Type 1 hypervisors are running directly on the hardware.
- Type 2 hypervisors are running on top of an operating system.

This section will focus on the first type of hypervisor because the second type is not a good fit in the context of this thesis because hypervisors are not dynamic enough. The main advantage of a type 1 hypervisor is that it can execute with a higher security level on the hardware, meaning that it can access hardware features that cannot be accessed by an operating system.

A hypervisor creates partitions and allocates time and memory to each partition. A partition contains either an operating system or a bare-metal application. To reach full isolation, a hypervisor can reset all the memory-related caches (L1, L2, L3, TLB, ...) during each partition switch. This ensures that an application in partition A cannot access data of partition B that would have stayed in the caches. However, this solution increases significantly the partition switch time and does not easily hold on multicore processors.

One drawback of hypervisors in the context of this thesis is that the scheduling is mostly static between partitions. The hypervisor scheduler has to schedule partitions and not processes, it does not have much information on the needs of the partitions. In particular, a partition that has nothing to execute at a moment will still be scheduled during its reserved time. Note that some hypervisors (such as PikeOS) allows reusing this idle time but this is not a common feature.

2.3.3 Containers

A container is another type of virtualization mechanism. The difference between a container and a virtual machine is that a container only virtualizes software layers including operating system services and not hardware.

The isolation provided by a container is not as strong as the one provided by a hypervisor since the code is running on the same hardware and with the same level of execution. First of all, a container does not have its scheduler and thus it cannot control what happens during a context switch. Second of all, the applications inside a container are executed on the same operating system. However, a container provides virtualization of operating system services and the different processes cannot interact with other processes through these services.

This thesis focuses on containerization as a virtualization mechanism in order to reduce the mutual impacts of different applications running on the same hardware platform.

2.4 Linux

Linux is a widely used operating system, created in the mid 90s, Linux is developed by software engineers all around the world. Nowadays, Linux is used by a lot of people and deployed on many different kinds of hardware. It can run on desktop computers, servers, and supercomputers but also on IoT and embedded devices.

2.4.1 Advantages

Most developers learn to use Linux and its development environment, which makes it a standard in the computer industry. One of the most powerful advantages of Linux is its re-usability. The open-source philosophy allows having a lot of libraries and drivers developed for Linux on different hardware. All these advantages make Linux useful in an embedded devices context.

2.4.2 Linux in a critical 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 systems. Critical embedded industries are interested by Linux and all the benefits of using it in their products ([5], [6] and [7]). Medical, aerospace, automotive, and industrial automation are considering the use of Linux for critical sections. This new approach to critical software raises a lot of questions because qualification or certification of an open-source operating system like Linux is very difficult considering current applicable certification guidelines. Thereby the embedded operating system verification paradigm has to be changed to ensure a certain level of confidence in Linux. One of the possibility is to make more tests rather than using safe development processes.

2.4.3 Features

This sub-section aims at presenting useful features of Linux that are important to know to understand the work of this thesis. Of course, the goal is not to describe all the functionalities, features, and mechanisms of Linux.

Systemd

Systemd is a software suite, that is not part of the Linux kernel. It is the first process to be launched during the Linux start-up and thus has the PID 1. Among its many roles, it is responsible for the init of the userspace and the management of the Linux daemons (a daemon is a process that runs in the background)

Systemd is also responsible for the configurations of the daemons and provides an interface for userspace called *systemctl*. It is also possible to analyze the bootup performance of Linux through systemd. Systemd provides a configuration interface for the contenairization mechanisms of Linux.

Some of the most known daemons managed by systemd are described as follows:

- **journald:** is responsible for event logging.
- **logind:** manages users' logins into the system.
- **networkd:** manages the configurations of the network interfaces.
- **timedated:** is used to control time-related settings.

All of these daemons are not necessarily useful in the space context, but they are activated by default in every Linux distribution.

Systemd is an important block in the behavior of Linux since it manages most of the links between user space and kernel space. Moreover, it is an interesting part of Linux in the context of this thesis, because systemd manages most of the background code. This code can be executed independently of the user space processes that are running in the system. It means that if no userspace process is created, the systemd daemons can still be executed at some point. The best example is the journald daemon which logs events when they happen. In the context of this thesis, code executed in background is problematic since it can impact the execution of the satellite processes.

Namespace

Namespace is a Linux feature that helps to ensure the isolation of different resources accessed by different processes. Processes interact not only with memory (protected by virtual address space) but also with files, hardware I/Os, shared memory areas, or mutually exclusive resources such as mutex. Namespace allows to virtually hide resources from certain processes. The different types of namespaces are described hereafter:

- **mount:** this namespace controls mount points. For example, a disk mounted by a process inside a namespace will not be visible outside of the namespace. However, at the creation of the namespace, all the mount points already present are populated to the namespace.
- **process ID:** this namespace controls the set of process IDs. Basically, two processes can have the same process ID if they are not in the same PID namespace. Process ID namespaces are nested, meaning that a process will have a PID in each process ID namespace up to the first one.
- **network:** this namespace is made to virtualize the network stack. All the network-related resources such as the set of IP addresses or routing tables are private to each namespace. Moreover, a network interface can only be present in one namespace at a time, but it can be moved from one namespace to another.
- **inter-process communication:** this namespace controls the inter-process communications such as semaphore or shared memory area. It means that two memory-shared areas can have the same identifier if they are in two different namespaces.
- **UTS:** this namespace controls the hostname and NIS domain name.
- **user ID:** this namespace controls the group IDs and user IDs and other parameters related to administrative rights in the system. This kind of namespace is useful for creating containers since it can give sudo rights to a process inside a particular namespace.
- **control group:** this namespace is used to hide the control group of the processes.
- **time:** this namespace allows using different system times.

The interprocess communication namespace isolates processes from systemV interprocess communication. It means that two processes in a different IPC namespace can use the same identifier for a shared memory area and produce distinct memory regions.

Cgroups

Cgroups is a Linux mechanism inside the kernel that assembles the processes in groups. The kernel can then control what each group can or cannot do in the system. Specifically, a cgroup can limit, monitor, and isolate the resource usage of a collection of processes. These features are described as follows:

- Limitation: limits the amount of shared resources that the group can use.
- Control: a cgroup can stop the execution of all its processes and restart it when it wants. It is also possible to block the execution of all processes at a defined checkpoint.

- Isolate: a cgroup can block access to a specific shared resource to isolate the accesses from another group.
- Accounting: it is possible to monitor the shared resource usage of a cgroup in real-time.

These features have been designed to create process containers. The shared resources that can be managed by a cgroup are CPU time, memory, disk I/O, network...

2.4.4 Real-time possibilities

Linux has never been designed to be a real-time operating system. However, the interest in using Linux in such a context has increased through time and real-time features exist in the kernel and userspace to make Linux usable in a real-time environment. This sub-section presents the real-time possibilities of Linux that are interesting in the context of this thesis.

PREEMPT-RT patch

In Linux, the PREEMPT-RT patch has been created to adapt the kernel to a real-time context. This patch is actually merging into the upstream Linux stable version.

The 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 for 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.

The PREEMPT-RT patch also enables the priority inheritance mechanism in the Linux scheduler. It allows a task with a low priority to take a 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.

The PREEMPT-RT patch is a collection of C code patches that will modify the Linux kernel code. After applying the patch it is necessary to activate the functionalities by configuring the preemption model in the Linux menuconfig before building the kernel. The preemption model menu contains 5 possible configurations (3 if the PREEMPT-RT patch is not applied), described as follows:

- **No Forced Preemption (server):** in this model, the only preemption points are system call returns and interruptions. This preemption model is designed to maximize throughput, this is why it is the one used for servers or computation

system that requires high raw processing power. Delays are usually good but cannot be guaranteed.

- **Voluntary Kernel Preemption (Desktop):** it is mostly used for desktop computers. More preemption points are added to the system in order to reduce the latency of rescheduling. The throughput is lower but the application's responsiveness is better.
- **Preemptible Kernel (Low-Latency Desktop):** all kernel code is preemptible, except the one executing in a critical section, which increases the interactivity of the system. A low-priority application can be preempted even if it is running a system call. The throughput is lower and it adds a small runtime overhead in the kernel code. It is used for desktop or embedded systems that need latency in the milliseconds' range.
- **Preemptible Kernel (Basic-RT):** this preemption model is the first one added by the PREEMPT-RT patch. To be activated, the Linux kernel had to be built with the patch changes. It resembled the "Preemptible Kernel (Low-Latency Desktop)" model with the addition of threaded interrupts. It means, that all interrupts are put into a thread and thus can be scheduled after being received by the kernel and also interrupted by other events. It is used mostly for debugging and testing purposes and is not made to be used in a production kernel.
- **Fully Preemptible Kernel (RT):** this last preemption model is the one associated with the PREEMPT-RT patch since it is the preemption model that allows the kernel to have all the functionalities of the PREEMPT-RT patch. Only a very reduced number of the code section is non-preemptible, and it is the more critical sections of the kernel (scheduler, low-level interrupts handling, and entry code). The locking primitives are replaced by their real-time relatives (spinlocks, rwlocks, etc). The priority inheritance is activated and long non-preemptible sections are broken up into multiple ones. This is the configuration needed to have real-time guarantees.

scheduling policy: SCHED_DEADLINE

For Linux to be used in a critical embedded systems context it needs to have real-time properties. Thus, processes need to respect their deadline. The period of a real-time process 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. The SCHED_DEADLINE policy 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) which has replaced the GRUB (Greedy Reclamation of Unused Bandwidth). To ensure

optimal scheduling, the scheduler needs to know the process runtime (cf figure 2.1). 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. In the Linux system, SCHED_DEADLINE threads have a higher priority. In other words, if one SCHED_DEADLINE thread is runnable it will preempt threads scheduled by another policy.

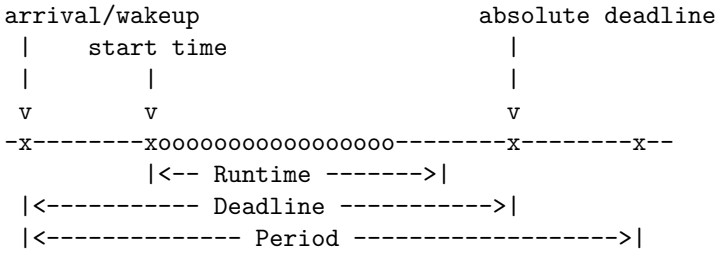

Figure 2.1: Parameters of a SCHED_DEADLINE Linux process (man sched(7))

The EDF algorithm chooses the thread with the smaller deadline to schedule it. However, the deadline that is used by the SCHED_DEADLINE policy is not the one configured by the thread developers. The SCHED_DEADLINE policy uses a "scheduling deadline" and a "remaining runtime" that are calculated by the CBS algorithm. These two parameters are initialized to zero. Then, the following comparison is computed when a task wakes up:

$$\frac{remaining\ runtime}{scheduling\ deadline - current\ time} > \frac{runtime}{period}$$

If this comparison is verified or if the "scheduling deadline" is smaller than the current time then the variables are modified:

$$scheduling\ deadline = current\ time + deadline$$

$$remaining\ runtime = runtime$$

It means that the "scheduling deadline" and "remaining runtime" are re-initialized, giving the task more time to be run. When the task is executed for an amount of time t the "remaining runtime" is decreased by t. The task is throttled by the CBS algorithm when the "remaining runtime" becomes less or equal to zero. Then the task cannot be executed until its "scheduling deadline". After that, the "scheduling deadline" and "remaining runtime" are updated as follows:

$$scheduling\ deadline = scheduling\ deadline + period$$

$$remaining\ runtime = remaining\ runtime + runtime$$

The CBS algorithm aims at ensuring that a task won't be executed in priority if it uses more time than it should have. This property is useful in a critical embedded context since it allows using dynamic scheduling while controlling the amount of CPU time allocated to each task.

scheduling policy: SCHED_RR

The SCHED_RR policy is also a real-time scheduling policy available in the Linux kernel. Unlike the SCHED_DEADLINE policy, there is no need to provide real-time parameters for the SCHED_RR policy to be working. This policy is an enhancement of the SCHED_FIFO policy which is the third real-time scheduling policy of the Linux kernel.

In SCHED_RR, tasks have a priority and preempt every other non-real-time task. A thread will be executing as long as it is not blocked, it is not preempted by a higher priority thread, or it calls the system call *sched_yield()*. When a thread becomes runnable it will be placed at the end of the list corresponding to its priority. If a thread is at the top of the list and the executed thread is being blocked, then it becomes running and stays running until all higher priority threads stay blocked. Finally, each thread has a quantum of execution time that it cannot exceeds. When the quantum of execution time is exceeded, the process returns to the end of its list.

2.4.5 Linux distribution and building system

As said before, Linux is open-source, therefore the code of Linux is available on the internet. The official Linux repository is presented in [8]. However, to be fully functional Linux has to be configured and compiled. Most of the time, Linux is provided through a distribution such as Ubuntu, Red Hat, or Debian. A lot of Linux distributions exist both for desktops and servers.

When used in an embedded systems context these Linux distributions are not the most suited. They add a lot of features that are not necessary such as a graphics environment. In order to create a custom Linux distribution that will fit perfectly with the needs of the embedded system, it is necessary to build the Linux kernel executable and add the desired functionalities. To do so, there are building systems that help customize and build Linux.

Yocto is a project that aims at providing tools and processes to build custom Linux distributions. It is composed of different parts: the OpenEmbedded build system, the build engine BitBake and the core metadata OpenEmbedded-Core. The default reference implementation is called poky and contains the build systems and basic recipes. Creating recipes is the main way of customizing the Linux build. It

allows patching the kernel code but also adding applications and configuration files in the Linux system. Using such a building system allows reducing the size of the Linux distribution as much as possible by keeping only what is strictly necessary for the final implementation context. However, most of the time it is hard to understand what is necessary for such contexts this is why the default implementation provided with Yocto is used as a customization base.

Chapter 3

State of the Art

3.1 Multi-core

Using multicore for safety-critical applications is related to the understanding of interference. In a multicore context, an interference is defined in [9] as alteration of the processor's behavior seen by software running on one core, due to activities ordered by software running on other cores. Interference analysis has received significant attention in recent years for safety-critical systems requiring certification.

3.1.1 Multicore processors in the aerospace domain

In the aeronautical domain, several projects focused on interference. As said in the introduction, the aeronautical certification authorities (EASA and FAA) have proposed [10] a certification standard, called MCP-CRI, to master interference in safety-critical MCP. To answer this standard, [9] tried to clarify the definitions of interference channels, interference sources, and interference targets, and they proposed a process to reduce the number of interferences in a given architecture. Following this initiative, another work proposed by [11] tried to adapt the MCP-CRI to COTS MCP architectures. They showed, through a particular case study (the Freescale P4080 processor), that mastering interferences highly depends on detailed information about the behavior of the hardware components, and they concluded that predicting interferences is a very challenging task because of the amount of required information. A method for isolating sources of interference is proposed in [12]. At the same time, [13] compares two methods to identify interference in an avionics context. The identification of interference path is key to the certification of multi-core processors in avionics. The PHYLOG method [14] proposes a certification framework for multicore in avionics.

3.1.2 Timing analysis

All these papers show that there is an interest in studying multicore interference in the aerospace context, but this problem is not exclusively reserved for avionics. Timing analysis for multi-core processors has received a lot of attention in recent years. C. Maiza *et al.* in [15] provide a large and complete overview of the scientific work on timing verification for multi-core processors. They classify this work into four categories:

- **Full integration:** this category considers that all information is known about the tasks executing on the multi-core. The analysis of each task encompasses the interference caused by tasks executed in parallel.
- **Temporal isolation:** this category focuses on either partial or full temporal isolation to either reduce or eliminate the interferences induced by other tasks running in parallel on the same platform.
- **Integrating interference effects into schedulability analysis:** in this category the effects of interference from the execution of other tasks in parallel are taken into account in the schedulability analysis.
- **Mapping and scheduling:** this category considers the mapping of tasks to cores and the relationship with the scheduling feasibility.

As the chapter 4 focuses on identifying and counting interference between software tasks, interference modeling falls into the first category ("full integration").

In this category, most of the work referenced by C. Maiza *et al.* assume "almost full knowledge" of the system to analyze, i.e., the software tasks and the multi-core architecture under consideration. It is the hypothesis used in most of embedded systems which is why they have a lack of flexibility. That is the case for one of the seminal papers by A. Gustavson *et al.* in [16]. They proposed to analyze by model-checking the WCET of a set of tasks running on a multi-core system, modeling the tasks and all the components of the system (including the cores, their private L1 cache, the shared L2 cache, and the memory) by UPPAAL timed automata. The same idea is followed in the chapter 4. However, as the models presented in the chapter 4 only try counting the interferences on the interconnect, the L2 cache, and the memory, it focuses on these components and considers an abstraction of the cores, the L1 cache, and the tasks. The abstraction considered in this thesis is the bus access profile of each task (defined by [17]). The belief is that it provides a simpler and more scalable way to analyze interference in multi-core processors.

[18] G. Giannopoulou *et al.* proposed a similar approach. Their objective was to compute the WCRT of a set of tasks running on a multi-core processor. They follow the same approach as [16], modeling the multi-core system as a network of timed automata. They follow an approach developed in real-time and network analysis

based on the real-time calculus method. The idea is to model the shared resources by arbitration scheme (such as TDMA, round-robin, and first-in-first-served policies), and they model the task behavior by arrival curves. This approach is based on a strong assumption: tasks as separated into three successive phases known as *acquisition*, *execution*, and *replication*. This thesis work is similar in the sense that it also considers an abstraction of the cores and the tasks. However, it seems that G. Giannopoulou *et al.* do not consider the memory hierarchy (the shared caches and the DDR banks).

3.1.3 Experimental approach

Several studies have proposed to characterize the interference in an MCP by the use of "micro-benchmarks" [19, 20]. A micro-benchmark is a small program that tries to execute only one type of request (e.g., a read or a written request to the memory), to put a constant load on a processor resource and to measure the impact of this request on the behavior of concurrent software partitions. [21] used micro-benchmarks to gather information on the complex behavior of COTS multicore architecture. More recently both [22] and [23] focused on measurement techniques based on dedicated stressing benchmarks and hardware monitor counters to characterize the architecture and the shared resources that can cause interference. Complementarily with this work, [24] and [25] have proposed a profiling approach to quantify the interferences. This approach is based on generic algorithms to mimic the memory behavior of real applications. These algorithms repeat an access sequence defined by three main parameters: (1) several reads of data from the memory hierarchy, (2) several writes of data to the memory hierarchy, and (3) a throttle rate allowing to adjust the duration of computing time between a read burst and a write burst. The author showed that this generic algorithm allows covering a wide range of memory behavior varying both in nature and in intensity. Performance counters are used in [26] to provide a measurement based probabilistic timing analysis framework. All the papers presented below worked on baremetal or low-level operating system. However, measurement can be altered by an higher level operating system. [27] proposes a practical approach to choose the best measurement method when working with multicore systems.

More recently, N. Sensfeleder *et al.* explored in [28, 29] the interference due to cache coherency management between the private L1 caches of the cores in a multi-core processor. As [16] (previously mentioned) they follow a model-checking approach. They model the coherency policy and the behavior of the caches with UPPAAL automata, and they show that it is possible to count the interferences that can occur when the software tasks and the coherency manager try to access the L1 caches simultaneously. [30] presents a method to identify the number of interference. They applied their method on an AURIX TC275 and showed that the approach is safe but also free of spurious interferences.

3.1.4 Avoiding interference

The second class of works focuses on methods to avoid interferences (classified as "temporal isolation" solution in the general survey of C. Maiza *et al.* [15]). Becker *et al.* in [31] propose a contention-free execution framework to execute automotive software applications on many-core platforms. [32] proposed a similar approach that relies both on a development workflow, and the use of an execution model defined as a set of rules to be followed by the designer and asserted through the run-time to enforce specific behaviors. Both [31] and [32] target a TDMA execution model, and use a Constraint Programming formulation to find an optimal time-triggered schedule on each core. However, if an appropriate TDMA scheduling can avoid instantaneous interferences, delayed interferences remain. A perspective of this thesis work is to extend the software partition model with time to model TDMA execution, as the ones proposed by [31, 32], and then to be able to analyze the remaining delayed interference.

The studies previously presented are focused on the multicore interference without considering the impacts of middleware. [33] proposes a model refinement and a timing analysis framework that takes into account the overhead induced by middlewares in embedded systems. Another approach is presented in [34], they deployed petri nets, temporal logic and advanced algorithms to provide a verification of middleware's behavioral properties.

3.2 Software integration

3.2.1 Scheduling

In order to reduce interference and thus the impacts of interference it is important to control the software integration on the same hardware platform. Liu and Layland [35] provides the fundamentals of deadline scheduling algorithms. These algorithms are fundamental for integrating software on the same platform since scheduling software together is necessary for the cohabitation. Years later, [36] proposed a generic scheduler with reduced latency that can work in both hard-real time and general purpose context. This scheduler works for uniprocessor and multi-core and has been easily implemented on Linux. More recently, [37] described an extension of the scheduling analysis tool Cheddar [38]. This extension uses architectural models written in AADL and provides a guide to write AADL models for multi-processors scheduling. Another way to work on the scheduling is to look at the schedulability of the system. [39] analyses the schedulability of a multicore system by proving correct a simulation interval for all schedules generated by a deterministic and memoryless scheduler. Finally, in deadline scheduling, quantum is an important metric to configure and can change the

whole behavior of the system. [40] provides a scheduler that can dynamically adapt the quantum of a multi-core application based on its behavior.

3.2.2 Partitioning avionics

In 2000, [41] 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, Jan Bredereke in [42], provide a survey of time and space partitioning for Space avionics. He began by describing the Integrated modular avionics (IMA) and its interface with the operating system ARINC 653 which are used for aircraft system [43]. [44] uses AADL to describe ARINC653 systems and simulates them. At the same time, [45] provides a methodology to enforce robust partitioning on multi-core platforms in accordance with ARINC 653. Then he describes the difference between avionics in aeronautic and in space. The main differences are: the presence of radiations and the less complex systems. The European Space Agency (ESA) had a project of adapting IMA for space avionics ([46] and [47]). The work presented in this thesis uses the IMA-SP architecture with the two layers and the possibility for partition to communicate via shared memory area.

3.2.3 Virtualization

A solution to integrate multiple software on the same hardware platform while reducing the interference is to use virtualization. [48] describes the purpose of virtualization. More recently, [49] provides a survey of virtualization solutions for space systems, applying the virtualization principle to the space context. This survey presents the virtualization architectures used in space such as Xtratum and PikeOS. 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 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 [50]. Another hypervisor that could be used in space systems is NOVA [51]. NOVA is a very light weight micro hypervisor focused on security. The reduced size of NOVA is due to the fact that most of the functionalities is handled by the user space. In the same hypervisor family there is OKL4 [52] which is a microvisor. It meets both the micro kernel and the hypervisor objectives. [53] describes Xen, a very efficient hypervisor that can host up to 100 operating system at the same time.

More recently, [54] proposed a method to ensure the execution of a critical partition running on top of a hypervisor in a multicore context. The method performance

has been verfied through measurement based experimentations. [55] presents an implementation of POSIX system calls on top of codezero hypervisor from the L4 family.

Apart from hypervisors, others solutions have been studied to integrate multiple software on the same platform. First of all, [56] presents the notion of containerization which is a virtualization mecanism that allows to virtualize services and not hardware. A container is executed and managed inside an operating system. It allows to manage the system's resources allocation. Another approach is to create partition directly in the shared cache. [57] proposes an implementation of cache coloring techniques in the Linux kernel to provides L2 cache partitioning. In the same way, [58] describes a practical approach to eliminate bank-level interference in multicore by applying the coloring technique to memory pages.

3.3 Linux

In recent years, there is a strong interest in using Linux for space applications. Leppinen proposes in [59] an overview of all spacecraft on-board avionics projects that use Linux in the last 20 years. Most of the projects presented in this paper are cubesat or very small spacecraft made by academic institutions. Planet and SpaceX are the two private companiest are also using Linux. More recently, NASA used Linux on an helicopter flying on mars [60].

3.3.1 Real-time

The first question when speaking of Linux in a critical embedded context is about its real-time capabilities. In [61] arguments are made that the real-time Linux community and the real-time systems research community should intensify their collaboration. Then it presents a list of changes and addition to PREEMPT-RT and evaluates these changes in a candidate for an EDF scheduler in Linux. Reghenzani *et al.* provide a survey on PREEMPT-RT in [62]. It shows that PREEMPT-RT can be used in hard-real time context only if the requirements are not dictated by safety-critical constraints. On the same subject, [63] applies timing analysis method used in real-time scheduling theory to the PREEMPT-RT patch added in the Linux kernel. [64] also uses models to understand the PREEMPT-RT behavior from a scheduling perspective. It also proposes a tracing method to observe the events needed by their method and compares the performance of this tracing method with cyclictest. [65] provides another tracing method to accurately measure the apparent latency in the Linux kernel with PREEMPT-RT.

Other studies are focused on the real-time behavior of the Linux kernel without the PREEMPT-RT patch. [66] is an experimental study which aims at finding

the sources of latency in the Linux kernel. They found two major sources of latency: timers resolution and non-preemptable sections. While the non-preemptable sections is a problem addressed by the PREEMPT-RT patch, the timer resolution is a predominant factor in the latency of the Linux kernel. In this article, they used accurate timers to reduce this latency and then analyzed the latency induced by the non-preemptable sections in different type of Linux. [67] discusses the possibility of reaching hard real-time response in the Linux kernel. More specifically, this paper focuses on the implementation of RCU (Read-copy-update) mecanisms. Finally, [68] compares the time performance of Linux, Linux with PREEMPT-RT and Linux with Xenomai. They conclude that for hard-real time applications it is recommended to use Xenomai since it has better performance in their analysis. In the same way, [50] compares the performance of different real-time operating systems with Linux.

3.3.2 Software integration on Linux platforms

In order to integrate hard real-time applications on Linux it is necessary to look at its scheduler. [69] presents one of the first implementation of an EDF enhancement for the Linux scheduler. This implementation provides a temporal isolation of the different tasks. This paper is the first version of the SCHED_DEADLINE policy. Years later an enhancement of the new SCHED_DEADLINE policy is presented in [70]. This implementation uses another data structure to take faster scheduling decisions. The paper also provides a comparison of the performance of this new implementation with the others real-time scheduling policy of the Linux kernel and proves that it is very close to the performance of the SCHED_FIFO policy. More recently, Lelli *et al.* give a feedback on the design and implementation of SCHED_DEADLINE in [71]. They also showed that the deadline shceduling policy provides temporal isolation and reduces the number of deadline misses compared to the other Linux scheduling policies. SCHED_DEADLINE is at the heart of the experiments presented in chapters 5 and 6.

A lot of studies are working on the partitioning in Linux. In [72], Obermaiser *et al.* evaluates the temporal and spatial isolation of Linux real-time with RTAI. They provides modifications in the API and the name service to increase the performance. More recently, [73] focuses on the concurrency of applications on Linux when running on multi-core processors. [74] and [75] written by Kim *et al.* are interested in the interference directly in the memory. They propose a memory throttling mecanism using Linux cgroups and a scheduling algorithm that will take int account the memory behavior. At the same time, the containerization approach is applied to Linux. [76] implements a container based real-time scheduling algorithm to use in Linux. This algorithm provides a two-level scheduling paradigm using cgroups.

The last question raised by the real-time Linux research community is related to

safety. In fact, Linux is not design to be used in a safety-critical context. [77] provides two methods to test the safety of Linux systems using a statistical approach. Then, it uses these methods on an autonomous emergency braking system.

Part II

Contribution

This part of the manuscript develops the contribution of the thesis. There are 3 major contributions described in the next three chapters and an analysis synthesis along with methodological recommendations derived from the contribution, presented in the last chapter of this part. Before going into the details, it is important to gather the different hypotheses and goals of these contributions.

This thesis focuses on the execution of multiple software applications on the same multicore platform. The number of cores is assumed to be 4 and the memory architecture is composed of buses between the different components, a MMU and L1 data and instructions cache for each core, a shared L2 cache, no L3 cache, a main memory and a storage disk which could be either a hard drive, a MMC or any other type of non-volatile memory. The platform used for all the experimentations is the Zynq Ultrascale+ which is described in the chapters 5 and 6.

The multicore platform is also running an operating system along with the software applications. Linux is chosen to be the only operating system considered in this thesis. The distribution is a very light custom one made with Yocto and the version of the kernel is 5.4. Note that the version of the kernel does not have a lot of importance. It has been chosen because it was the most recent one available for Zynq Ultrascale+ and Yocto at the time of the beginning of this thesis. There is no software between the hardware and the operating system, it means that hypervisors and containerization techniques can only be used on top of Linux.

Each application is running inside a Linux process without any other threads. Also the scheduler is entirely responsible of the core allocation. It means that there is no pre configured core allocation in all the system. This is because the main objective is to evaluate the Linux solution running a set of processes without having to decide which core allocation would be the best. In other words, a total confidence is given to the Linux scheduler to find the best core allocation at any time. Note that it may seem highly unusual in an critical embedded context but it is completely assumed. These contributions aim at using a Linux distribution with the least custom configurations possible and thus allowing the solution to be used in much more situations. Let us take an example to illustrate this part. A common suggestion could be to put all the most critical processes on the same core and then give the three other cores for the other processes and the operating system. This is a good idea to reduce the impact of the least critical software on the most critical ones. However, what happens if all the software applications have the same criticality (either high or low) ? If the allocation is followed strictly, it means that 1 to 3 cores will be unused. It can even be worse because the system could not have enough CPU power to execute all the software. In this case, the allocation needs to be re thinked and a lot of tests have to

be re done because it was not considered in the first allocation.

The focus of this thesis is on the impact of the environment (Linux and other software applications) on the performance of an on-board software. It means that optimising the hardware or the functional safety are not in the scope of this thesis. In the same way, the analysis of the WCET or the optimisation of the scheduling algorithm are not presented here. To sum up, the main objectives is to deal with what already exists today and are available. Therefore, this thesis uses the Linux kernel and all its mechanisms but also space on-board software applications without trying to modify them.

Chapter 4

Interference analysis and modeling

One of the key aspects of this thesis is the use of a multi-core system. This system allows multiple software applications to run at the same time on the hardware and thus it executes more code in the same time. While it increases the performance, because more code can be executed during the same amount of time, it also leads to different complications. In fact, in a multi-core system, not all hardware resources are duplicated, meaning that the different software applications have to share some of them. This sharing situation can result in a modification of the behavior of one of the software applications. This is called interference. In this chapter, modeling is presented which aims at characterizing and counting this interference phenomenon.

4.1 Method overview

The landscape in which the method is involved is shown in Figure 4.1. Within this landscape, the scope of this chapter is highlighted by the gray box.

Let us consider a set of tasks T_i hosted by an MCP (on the left of the Figure). The objective of the method is to compute an upper bound of the extra timing penalty associated with each T_i due to interference occurring in the architecture (on the right part of the Figure). For that purpose, we compute an upper bound of the maximal interference number (noted IN_i) from which each T_i can suffer.

The approach relies on the TIPS analysis method proposed by Carle *et al.* in [17] (on the left part of Figure 4.1). This method is based on the notion of Time Interest Points (TIPS), which are load and store instructions that generate and suffer from timing interference. By means of static analysis, Carle *et al.* show that it is possible to extract a temporal segment sequence, as the one shown Figure 4.2, from the binary code of a task. Each segment is characterized by a duration (noted $d_{i,j}$ for task T_i and segment j), and the maximal number of memory requests leaving the core and sent

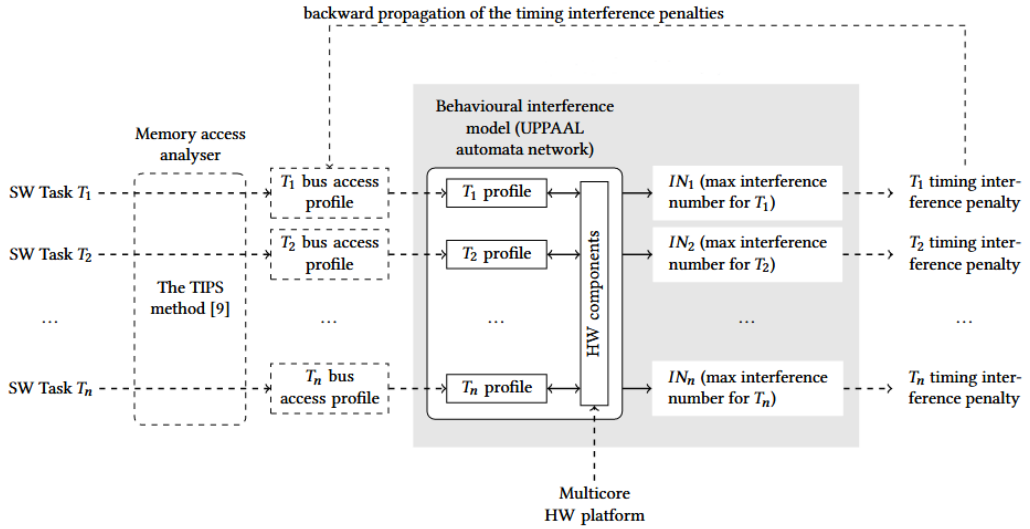

Figure 4.1: Approach overview (the gray box highlights the scope of the paper, while dashed lines are out of the scope)

to the bus (noted $\mu_{i,j}$). These requests correspond to load and store operations that are not *always hit* in the L1 cache of the core hosting the task. For instance in Figure 4.2, T_1 makes zero memory requests in segment one, and at most three requests in segment two.

Such a sequence defines a *bus access profile* of the considered task. It characterizes its worst-case memory activity that can lead to interference. As shown in Figure 4.1 these profiles are taken as inputs (one for each task), and a method is proposed to count the interference that can occur for each T_i.

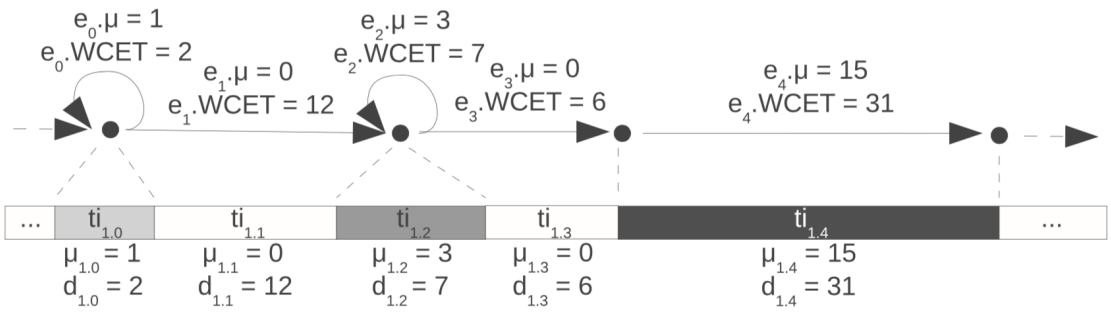

Figure 4.2: Example of bus access profile (excerpt from [17])

As shown in the gray box in Figure 4.1, the heart of the contribution consists of:

- (1) associating a UPPAAL automaton to model the bus access profile of each task;
- (2) modeling the components of the MCP by a network of automata synchronized with the automata of the bus access profiles;
- and (3) computing by model-checking an upper bound of the number of interference from which each task can suffer.

4.2 Definitions

Let us consider an archetypal MCP architecture (depicted in Figure 4.3) compliant with the assumptions of Carle *et al.* [17]. This processor is composed of (1) two cores (C_0 and C_1) owning their private cache $L1$, (2) a shared cache $L2$, (3) a DDR memory composed of one bank B, and (4) a shared bus allowing the two core blocks to address both $L2$ and the DDR. Each core hosts a task called SP_0 (on core C_0) and SP_1 (on core C_1).

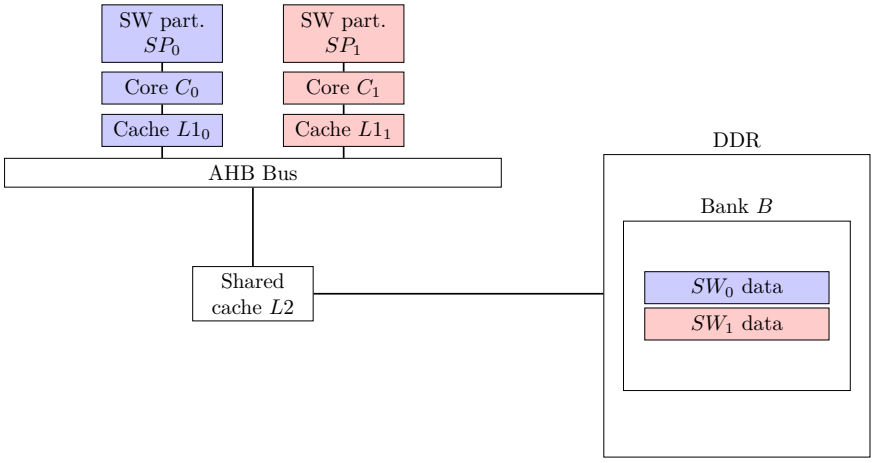

Figure 4.3: A simplified multi-core platform

For the sake of conciseness, let us suppose that SP_0 and SP_1 are characterized by the profiles shown in Figure 4.4. These profiles are periodic (period = 30), and they are divided into three segments. According to these profiles, SP_0 and SP_1 can compete to access the memory in the first segment (from 0 to 10). In this segment, each task sends at most 2 memory requests. In the next two segments, either SP_0 or SP_1 does not send any request, leaving the memory path available for the other task.

The requests of SP_0 and SP_1 follow the following path:

Figure 4.4: Bus access profile of SP_0 and SP_1

- $path(SP_0) = C_0 \rightarrow L1_0 \rightarrow AHB\ Bus \rightarrow L2 \rightarrow B$
- $path(SP_1) = C_1 \rightarrow L1_1 \rightarrow AHB\ Bus \rightarrow L2 \rightarrow B$

These two paths intersect on three components: *AHB Bus*, $L2$, and B. Requests from SP_0 and SP_1 can then suffer from interference in these three components.

Interference definitions

Requests from SP_0 and SP_1 first collide in *AHB Bus*. Whenever they arrive at the same time, only one of them can pass, the second one must wait until the bus becomes free again. The effect of the interference is a delay in which length is the duration the bus is occupied by the first request. The interference is caused by a simultaneous collision.

According to the request path of SP_0 and SP_1, a second interference could arrive in $L2$. However, the intrinsic nature of this interference is different. Let us imagine for instance that SP_0 reads data from the bank B and puts it in the $L2$ cache. As long as the data remains in the cache, each time SP_0 accesses this data its request path ends with $L2$. Let us imagine now that SP_1 reads its data from the bank B. According to the cache policy, data of SP_1 could evict data of SP_0 from $L2$. The result is the next time SP_0 would try to access its data, it should have to lengthen its request path to the memory. The effect is similar to the bus interference, SP_0 would suffer from a long delay. However, in this case, the scenario of the interference is different. There is no simultaneous collision. The two requests can occur at different times. But the request from SP_1 changes the internal state of $L2$. This change is the cause of the interference suffered by SP_0. In other words, SP_1 provokes a delayed interference on SP_0.

As shown in this example, in order to analyze the interferences that can occur, it is necessary to identify the types of HW components. We consider two types of components: transport and storage component.

Definition 1 (Transport component) *A transport component is characterized by an internal state which only depends on the presence or absence of a request using it. If a request is using the component, then it is "occupied". Otherwise, it is "free".*

In the example Figure 4.3, the *AHB Bus* is a transport component.

Definition 2 (Storage component) *A storage component is a component whose internal state depends on previous requests (including the current one if any).*

Examples of "storage components" include caches and memory banks. As explained above, the content of a cache depends on the previous memory requests that used it, and it has a direct effect on the length of the next memory request paths.

Following the distinction, we now refine the notion of interference.

Definition 3 *An instantaneous interference occurs whenever at least two requests sent by two different tasks collide on the same transport component.*

Definition 4 *A delayed interference occurs whenever a request r sent by a task T uses a storage component whose internal state has been made non-compliant with T by another task T'.*

Considering again example Figure 4.3, as explained below an instantaneous interference can occur in *AHB Bus* and a delayed interference can occur in $L2$. In the same way, a second delayed interference can occur in B. Indeed, a memory bank is composed of memory space and a row buffer acting as a local cache. The row buffer contains the last block of accessed data. Hence, requests to data in the row buffer (one talks about "row hit" requests) are faster than requests to data not in the row ("row miss" requests). When a "row miss" occurs, the requested block has to be fetched in the row buffer, making the request time longer. As in cache, the content of the row buffer depends on previous requests. Thus, memory banks are storage components that can cause delayed interference.

4.3 Modeling

4.3.1 Brief recall on timed automata

This interference description and analysis rely on timed automata [78] and on the UPPAAL framework [79]. Let us recall briefly the concepts of UPPAAL that will be used in the following sections. A more complete description of UPPAAL timed automata can be found in [79]. UPPAAL locations (i.e., states of the automaton) can have two special characteristics: committed location, and location invariant.

- Committed location: The location is left as soon as possible without any time passing. A committed location is marked with a "C". Committed locations are useful for modeling a sequence of instantaneous actions.

- Location invariant: A location can be characterized by a Boolean formula called *invariant*. The location can be occupied only if its *invariant* is true. As soon as the *invariant* becomes false, the location has to be left. Location invariants are useful to force transitions to be triggered.

Besides, transitions are also defined by different features:

- **Guards** is a Boolean condition that must be true to trigger the transition. In UPPAAL, guards are colored in light green.
- **Synchronization** allows triggering transitions when an event occurs. Let e be an event. The term $e!$ denotes an emission of e, while $e?$ denotes a reception of e. Two transactions in two parallel automata labeled respectively by $e!$ and $e?$ are synchronized, meaning that they must be triggered simultaneously. Synchronization actions are colored in cyan in UPPAAL.
- **Update** allows changing variables values when a transition is triggered. Update actions are colored in dark blue in UPPAAL.

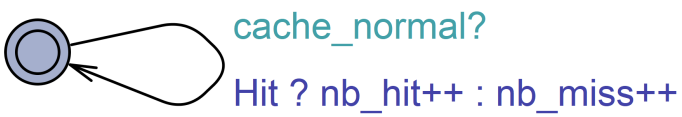

Figure 4.5: A simple UPPAAL automaton to count cache hit

As a first example, let us consider the automaton in the figure 4.5. This automaton is composed of one location and one transition. The only location is also the incoming one as shown by the inner circle in the location representation. The transition is labeled by two actions:

- cache_normal? : this synchronization transition is taken when the event *cache_normal* is sent by another transition.
- Hit ? nb_hit++ : nb_miss++ denotes an update action performed each time the transaction is triggered: if *Hit* is true, then the variable *nb_hit* is incremented, otherwise the variable *nb_miss* is incremented.

The second example presented in the figure 4.6 is a little bit more complicated. It is an automaton that counts an upper bound of the number of interferences from which a task can suffer in cache L2. This automaton is parameterized by the value of the upper bound to check (max). It has two locations and three transitions. The initial location is the one with a double circle. Transitions characteristics are described below.

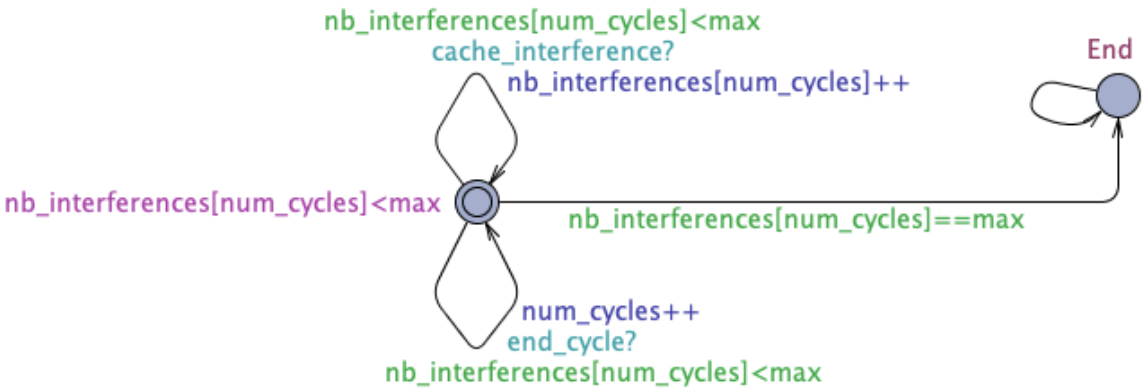

Figure 4.6: UPPAAL automaton to count interferences

- As in the previous example, cache_interference? is a synchronization action. The transition is taken each time the event *cache_interference* is sent by another automaton in the system (the automaton dedicated to the cache). , and each time the identifier of the current working task is equal to the *num_task*. The update action nb_interferences[num_cycles]++ is then performed, which increases the value of nb_interferences for the current segment (num_cycles).
- nb_interferences[num_cycles] == max is the guard of the second transition. This transition can be taken as soon as this formula becomes true. Then the location *End* is reached and the automaton stop counting the interference number.
- nb_interferences[num_cycles] ¡ max is the invariant of the incoming location. It forces the transition to be taken when *nb_interferences[num_cycles]* becomes higher than *max*.

In this example, the *max* value is an upper bound of the number of interferences if and only if the *End* location can never be reached.

4.3.2 Modeling approach

To model the interferences from which tasks can suffer, two classes of models are defined: automata for HW components (cf. section 4.3.3), and automata for tasks (cf. section 4.3.4). The whole system is then built by instantiating and synchronizing these automata. A high-level view of the interactions between the automata is represented in the figure 4.11. Software automata are horizontal to describe the path from core to memory (left to right) for given memory access. HW automata are vertical and represent the timeline of memory accesses potentially belonging to different tasks.

If the automaton modeling the HW components (the vertical part) is generic, the ones modeling the tasks (the horizontal lines) depend on the multi-core processor

under consideration. They should be generated from a high-level model of the configuration of the processor (including software allocation). This issue (i.e., how to generate the automata modeling of the tasks) is out of the scope of this thesis.

4.3.3 HW components

The role of the HW component automata is to model the answers of the component to requests sent by the tasks. It determines if a request creates interference in the component and notifies the software automaton of this result. As said in section 4.2, we consider two types of HW components: transport and storage components.

Transport component

According to the definition 1, a transport component is modeled by the automaton in the figure 4.7. It is composed of three locations (*Free*, *Occupied*, and *Check*), and three internal data (*waiting_queue*, *nb_elmt*, and *bus_state*):

- *waiting_queue* is an internal list containing the identifiers of the tasks waiting for the component.
- *nb_elmt* is the number of tasks currently waiting for the component (including the task currently using it).

The three locations of this automaton are:

- *Free*: This location is the initial one. The component is free and waits for a request. When a request arrives the location changes to *Occupied*, and the function *update_bus* is called. This function updates the internal variables of the automaton to let it know which task is calling it.
- *Occupied*: The component is already occupied by a request. Once the request is completed the next location is *Check* if the internal waiting queue is not empty ($nb\_elmt \neq 0$) and *Free* if the queue is empty ($nb\_elmt = 0$). All outgoing transitions of the *Occupied* location are waiting for the synchronization event *end_req*. This event is sent by a task automaton when it releases the component after its request has been completed by the memory.
- *Check*: This location's purpose is to switch the task whose request is handled by the component. It is a transient location reached when the current task occupying the component is releasing it and when another one is waiting for the component. The role of location is to trigger the transition returning to *Occupied* to process the next request. Once again the *update_bus* function is called when returning to *Occupied* to change the internal variables of the component.

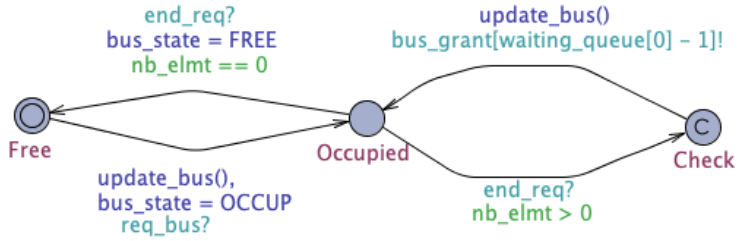

Figure 4.7: Automaton of a transport component

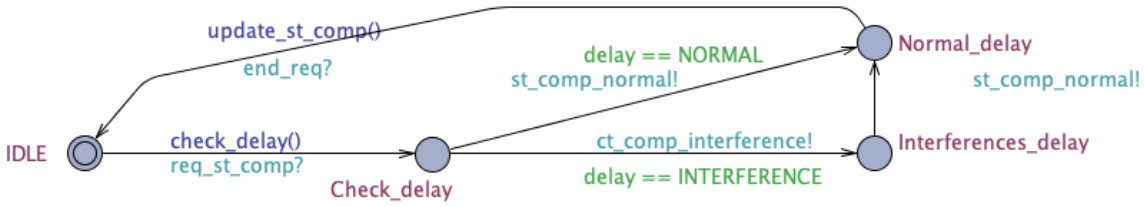

Figure 4.8: Automaton of a storage component

Storage component automata

Storage component generates delayed interferences, meaning that a task can interfere with another one by modifying the state of the component. Storage components are modeled by the automaton in the figure 4.8. The idea is to determine if the component reacts within a favorable delay (the normal case), or conversely within an unfavorable one (the interference case) when receiving a request. This notion of favorable delay is related to the task asking for the component. The automaton is composed of four locations:

- *IDLE* models the state where the component does not handle any request. It is waiting for a memory access request. When receiving it, it reaches *Check_delay* and it calls the function *check_delay* which determines whether there is an interference or not (i.e., whether the component is in an internal favorable state or not). *check_delay* compares the state of the component with the state of the com-

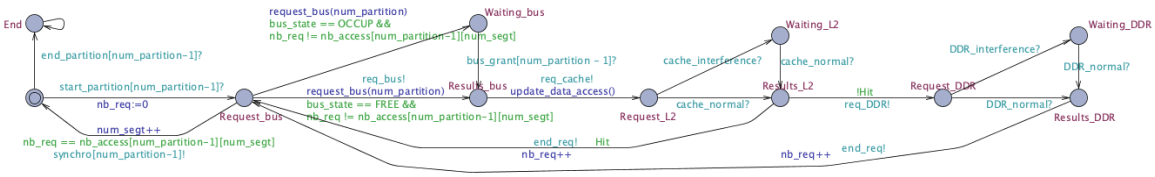

Figure 4.9: Automaton for task SP_0 of example Figure 4.3

ponent if the task is alone. Further explanations of this function are presented in the section 4.4.

- *Check_delay* is a transient location. Its role is to reach *Normal_delay* or *Interference_delay* according to the value of *delay*.
- *Normal_delay* models the normal response delay of request. Once the request is completed the component returns to *IDLE* and waits for another request. Before returning to the *IDLE*, the automaton is waiting for an occurrence of *end_req* sent by the task automaton processing the current request. When taking the transition to *IDLE*, the *update_st_comp* function is called to update the internal state variables of the component.
- *Interferences_delay*: When interference occurs the response delay is extended. This location represents the added delay induced by the interference. Thereby the next location has to be *Normal_delay* because interference is represented by an extra time added to the response delay.

Note that in this modeling it is about "delays", but they are never quantified. It is only about a "favorable" state to a task resulting in a NORMAL delay, and an "unfavorable" state resulting in an INTERFERENCE one. Thus, this chapter only considers a logical abstraction of time. The belief is that such an abstraction is scalable and it is sufficient for interference analysis.

4.3.4 Task automaton

A task automaton models the bus access profile of the task and the path of the requests through the HW components of the processor. The figure 4.9 gives the automaton of the task SP_0. It is composed of three parts. The first (locations *Request_bus*, *Result_bus*, and *Wainting_bus*) models the answer of the bus to the request. The second part (locations *Request_L2*, *Result_L2*, and *Wainting_L2*) models the answer of the L2 cache. And the last part models the answer of the DDR memory.

The initial location of the automaton is an *Idle* location. It waits for a start (or an end) event from the scheduler (that is, the automaton implementing the sequence of the segment of the task profile). When receiving a start event, the local counter *nb_req* is set to zero. And the automaton reaches *Request_bus*. The three parts follow then the same pattern composed of three locations:

- *Request_cmp* (where *cmp* is *bus*, *L2*, or *DDR*) models the state in which the task has sent the request to the component *cmp* and is waiting to know if the request generates an interference. If there is interference the next location is *Waiting_cmp*. If there isn't an interference the next location is *Results_cmp*.
- *Waiting_cmp* models the extra delay the task encounters due to the interference.

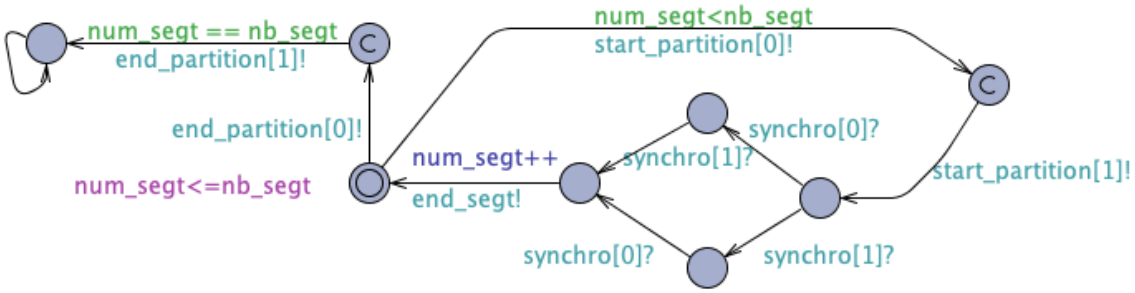

Figure 4.10: Scheduler automaton

- *Results_cmp* models the case in which the request is normally completed (with or without interference). The next location is then the *Request_cmp* of the next component or *Idle* if the component is a storage component (L2 or DDR) and if the requested data is present in the component. When the outgoing transition of this location is taken, the task automaton sends an event to the component automaton to notify it of the end of the current request.

4.3.5 Scheduler automaton

The last automaton models the sequences of segments of the task profiles. For instance, the profiles of SP_0 and SP_1 shown in the Figure 4.4 are modeled by the automaton in the figure 4.10. From the initial location, the Scheduler starts the first segment of SP_0 and SP_1. Then it waits for an occurrence of *synchro* sent by SP_0 and SP_1 at the end of their segment. When receiving them, the scheduler notifies the end of the first segment and starts the second one. And so on until the last segment (when $num\_segt == nb\_segt$).

4.4 Interference analysis

To determine when interference occurs each component automaton is instrumented with an algorithm to decide whether or not the task accessing the component is suffering from interference. Those algorithms depend on the type of component under consideration.

Transport component. To count interferences in transport components is very simple: there is an interference *iff* the component is occupied and another request is waiting for it.

Storage component. For storage components, the decision algorithm is more complex. To determine if access from task X is affected by access from task Y, it is not

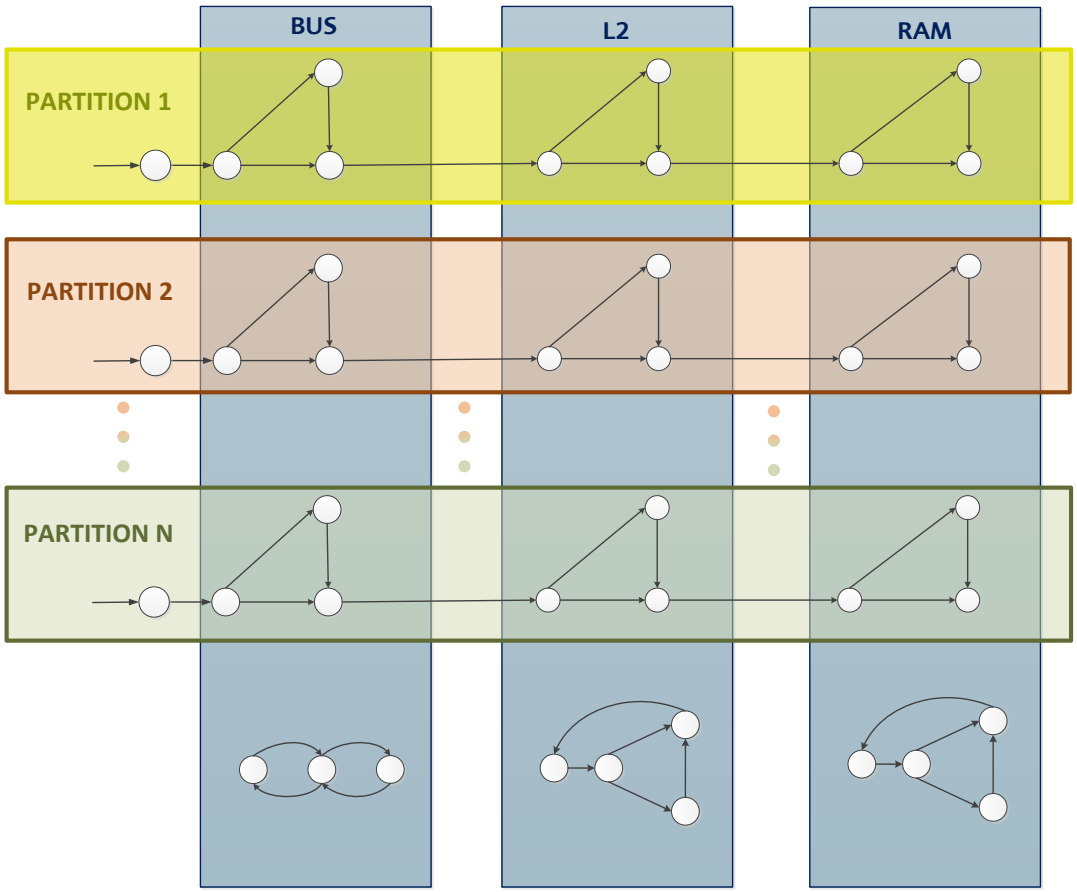

Figure 4.11: Representation of partition automata and components automata in the same schema

sufficient to maintain the actual state of the component but also the state of the component for each task as if the task were alone. Such a "private" view of the component is only paying attention to accesses from the task it is associated with. It means that access from task X to the storage component affects both the "actual" view and its "private" view of the component, but not the "private" view associated with task Y. Thereby, for each access to a storage component, it is possible to determine if there is interference or not by comparing the private view of the task with the actual view of the storage component. If there is a difference between those two views it means that interference occurred. In the example of an L2 direct mapped cache with 16.384 lines of 32 bytes, this algorithm works on: (1) an array *cache_array* containing 16 384 lists of 32 bytes representing the current state of the L2 cache ; (2) one similar array *task_i_array* for each task i representing the state of the cache if the task were alone. Let us consider two tasks. Imagine that task 1 requests data stored in line 250. *cache_array[250]* and *task_1_array[250]* are updated. Then task 2 accesses the same line, *cache_array[250]* and *part2_array[250]* are modified. When task 1 accesses again line 250, *task_1_array[250] part2_array[250]* are different meaning that task 1 suffers from interference.

Interference analysis by model-checking. To compute an upper bound of the number of interference experienced by each task, in each component for each segment, the UPPAAL model-checker is used. For instance, let us note *nb_interf_L2[0,*SP_0*]* the number of interference computed for the first segment of SP_0 in cache L2. UPPAAL shows that $\forall\Box\neg(nb\_interferences\_L2[0,SP_0] > 3)$, that is, is not possible that *nb_interf_L2_SP0* is greater than 3. This computation takes less than one second (UPPAAL version 4.1.24 running on a 3,2 GHz Apple M1 Pro with 32 Go DDR5).

4.5 Limitations of the models

The goal of the models presented above is to calculate interference induced by two software applications on one another. This modeling aims at reducing, controlling, and monitoring the impacts on a software application. The overall objective is to be able to know exactly how a software application behaves when it is executing on a Linux system with other software applications in parallel. The models describe two types of interference that happen in two different components. To do so, the hardware components have been modeled with automata and analysis has been made with UPPAAL. However, the software applications used in these models do not take into account the operating system.

The operating system is responsible for a lot of things in the execution of software applications. It handles the virtual memory, the processes management, the processes scheduling, and provides many features through system calls. In the models presented

in this chapter, the software applications do not use any system calls only basic instructions. Also, there are no context switches, i.e the time needed to pass from one software application to another is zero. Clearly, this is a huge approximation of what really happens in practice. Thus, to make the models more realistic, the operating system needs to be taken into account.

The first method available to add the operating system is to consider the operating system as any other software application. This means that the operating system has a task automaton and executes instructions as every other task automaton. This would require knowing the entire code of the operating system and when this code is executed. It might be conceivable for a small operating system but for one as large as Linux it is not feasible. Moreover, with this method, adding the scheduler to the model seems to be complex. This is because the scheduler is part of the operating system but also has a role in the automaton model. It means that part of the code executed by the operating system belongs to the scheduler subsystem, however, this code cannot be executed by the scheduler and is also modeled by an automaton. In fact, the scheduler has an impact on the system since, as with any other piece of code, it is executed on the hardware and thus uses the hardware components.

The second method available to add the operating system to the modeling would be to create a new type of automaton. This automaton would recreate the behavior of the operating system including the scheduling and the system calls. However, creating such an automaton means that it is mandatory to know every line of code in the operating system to be able to interpret its impact on the hardware resources. While it might seem plausible to make this happens with a script (not so easily), it is unlikely that the interference analysis will scale up as much as needed to analyze an automaton that is based on Linux.

Another limitation of this approach is the use of time. Time is an important factor in the analysis of hardware behavior especially in a real-time context. For example, it is particularly hard to count the instantaneous interference where there is no time in the model. In fact, the instantenous nature of the interference itself depends on time. Adding the time in those models would require at least two things.

First of all, integrating time in the models means to have an accurate model of the hardware response time and execution time. This is crucial to have a better modelization of the entire system. However, developed such a model require a lot of work and has to be repeated for each different hardware platform. In the context of this thesis it has been chosen not to develop such kind of model because it was too far from the main objective of the thesis and would take too much time.

Second of all, an important part of the system that uses time is the scheduler. The scheduler decides when a processus will be executed and in some cases uses time as a constrained resource. However, modelling the acurate behavior of the scheduler

is quite complex since it impacts the memory system but also the other applications on the system.

Finally, enhancing the models to remove the limitations presented in this section would have taken too much time and effort to get a non-general result that might not be applicable in real cases. This lead to re-think the problem and finding another way to add the impacts induced by the operating system and take time into account. Rather than trying to model Linux with an automaton, it is possible to focus on what is really interesting in the scope of this thesis: the impacts of Linux on a space onboard software that has a specific profile. The following chapters are focusing on measuring the impacts of this operating system environment and how it can be reduced by using specific mechanisms.

Chapter 5

Experimentation on Linux

This chapter focuses on Linux experiments and aims at understanding Linux behavior through measures analysis. As seen in the previous chapter, operating system modeling is not simple, and finding an automaton describing perfectly all the execution paths inside an operating system feels impossible. The approach of experiments presented below is to consider Linux as an observable system rather than a fully described one. It means that the behavior of Linux will be understood using observation methods. This approach is quite usual in experimental science such as physics, biology, or meteorology for example and this thesis's experiments are inspired from thsi approach. In fact, scientists do not have a technical reference manual for an atom or a molecule, their only way to understand such things is to observe, measure and experiment.

In the case of this work, there are plenty of technical manuals explaining how Linux works. It could be argued that building a Linux automaton from the information contained in these manuals is possible. However, there are several roadblocks to such a seemingly straightforward approach, which are exposed hereafter.

Linux complexity

Even if a lot of documentation exists for Linux, no document is exhaustive to explain the precise behavior of Linux. In fact, the only way to fully understand and predict what Linux will do at a certain point in time is to read the code of Linux. As explained in the section 2.4, the Linux code is enormous and it will be hard to reduce it in a simple automaton. Theoretically, the number of paths in the Linux code is finite, as a matter of fact, Linux has a finite number of lines of code. However, Linux is too big to be a candidate for static analysis.

Linux' behavior non reproducibility

Second of all, the execution of Linux depends on different external and internal events. For example, hardware and software interruptions completely modify the execution of the operating system. In the case of embedded systems when hardware interruptions are reduced (no mouse, no keyboard, no non-deterministic network protocols,...), we could argue that if we control the launch time of each application running on the system, then the behavior system should be reproducible. Besides, due to the number of parameters controlling the execution of such a system, it would be almost impossible to reproduce the same exact execution environment. It is equivalent to trying to reproduce two exact same plants growing. In theory, you could adjust the environment to have the exact same seeds with the exact same sun exposition and the same fertilizer, but in practice there will always be small differences.

Moreover, the execution of a complex system such as Linux depends on a large number of variables, and being able to reproduce the same execution in the exact same environment is not possible. For example, in the space environment, the hardware can be altered by SEU (single event upset) which is unpredictable. The unpredictability of SEU makes the whole system, and thus Linux, unpredictable. Therefore, we cannot assure that two execution of the same Linux will be identical from the timing perspective.

Finally, we could say that Linux' behavior is reproducible theoretically but in practice, the number of variables that need to be precisely controlled to make it reproducibl is too huge to be possible.

Experiments approach

In the experiments presented in this chapter, Linux is considered in most cases as a black-box object that has an observable impact on the behavior of known quantities (application processes). It is similar to the way scientists observe the black body in outer space. In a few cases, it will be needed to look inside the kernel with trace tools for example. In order to realize those experiments, the measurement method needs to be defined. In the next sections, the use case used for all the Linux experiments will be described along with the description of the hardware platform. After that, the methodology will be explained with a discussion on the impact of the measures. Then, the results of three different experiments are presented.

5.1 Usecase

In the context of this thesis, software applications exchange data between physical sensors, physical memory, and the ground station. Those software applications 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 applications will be executed on top of a Linux operating system, a software application is represented by a Linux process. The use case presented here is one space-specific software to study its isolation when it is executed with other processes on the same hardware platform.

5.1.1 AOCS: attitude and orbit control system

In a satellite, the AOCS software is responsible for controlling the attitude i.e orientation and the orbit of the spacecraft. Inputs of the AOCS algorithm are spatial angular coordinates, speeds, and accelerations coming from sensors such as Star trackers and Inertial Measurement Units. The AOCS determines a position in space and kinetic vectors, computes corrections to stay on the expected trajectory and attitude plan, and generates commands to actuators to realize those corrections. The figure 5.1 shows the relation between the AOCS software and the sensors and actuators. The AOCS is an infinite program that will run indefinitely by repeating a cycle of a specific period.

Figure 5.1: AOCS functioning

All the experiments presented below have been made with 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 an 8Hz frequency. It takes its inputs from a pipe and writes its outputs in another pipe. This software consists of 16000 calculus steps. It means that the infinite execution of the AOCS process has been reduced to fit into a finite time. Each step 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 be the same.

The AOCS process has two children processes described below (see figure 5.2):

- A non-periodic Linux process reads inputs from the input file and writes them to the input pipe.

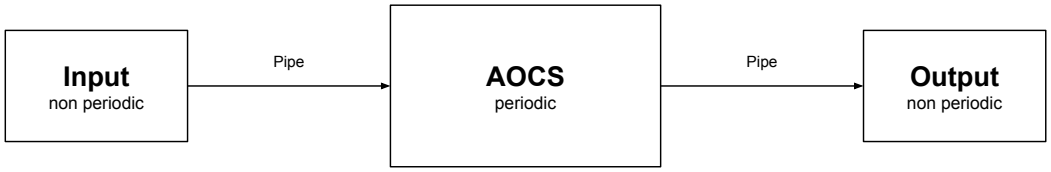

Figure 5.2: AOCS process functioning

- Another non-periodic Linux process reads outputs from the output pipe and compares them 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 for computation which is representative of the actual behavior of the spacecraft. The selected code is actual autocoded AOCS extracts from an operational project. By forking the AOCS process and thus creating two children processes it creates 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 other one related to the virtual memory is handled by the operating system and is slower than accessing the main memory. On the other hand, reading and writing to a pipe are faster as it doesn't need memory page management because pipes are located in the operating system area.

AOCS has been chosen as a use case because it has interesting properties. First of all, it is representative of an application used in the space environment since all spacecraft have AOCS software. Second of all, its execution is very complete, it has a calculus part but also other parts that need to access the memory both in reading and writing mode. Finally, it is a periodic application with very different steps, meaning that each period will execute something different from the previous one. The complex relation between all the AOCS steps adds diversity to the software that allows identification of the interference with different profiles. Even though, the I/Os used in the scope of these experiments are representative but not accurate of final implementation. In a real-world application, data would come from sensors and actuators directly and it is not a known fact that pipes would be used. Actually, the preferred method of exchange between I/O handling processes and computation processes is through memory access and synchronisation.

5.1.2 Hardware platform

Due to the hostility of the space environment, spacecraft's embedded computers and electronic devices must be hardened. Those modifications have non-negligible costs

on satellite production. Recently, the space industry introduced COTS (commercial off-the-shelf) hardware components to be used in the next generation of satellites. Experiments presented here 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 schematic diagram of the Zynq Ultrascale + architecture is showed in the figure 5.3.

The selected 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 embeds an FPGA which can be accessed by all I/Os and processing units. 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 to implement computationnaly intensive data handling, such as image or RF processing, and is becoming an essential electronic component for satellites. Additionally, compared to an ASIC, FPGA can be customized more easily to match mission specificities and pave the road towards more modularity, especially considering in-flight reprogrammation, potentially partial.

As described in the previous paragraph, in the context of space systems the real-time processing unit is envisaged to act as a trusted monitoring unit. 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 an 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 an instruction and data L1 caches along with a shared L2 cache and an MMU (memory management unit). The L2 cache is a 1MB (1024 KiB) 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 experiments of this thesis, all software applications are executed on the application processing unit.

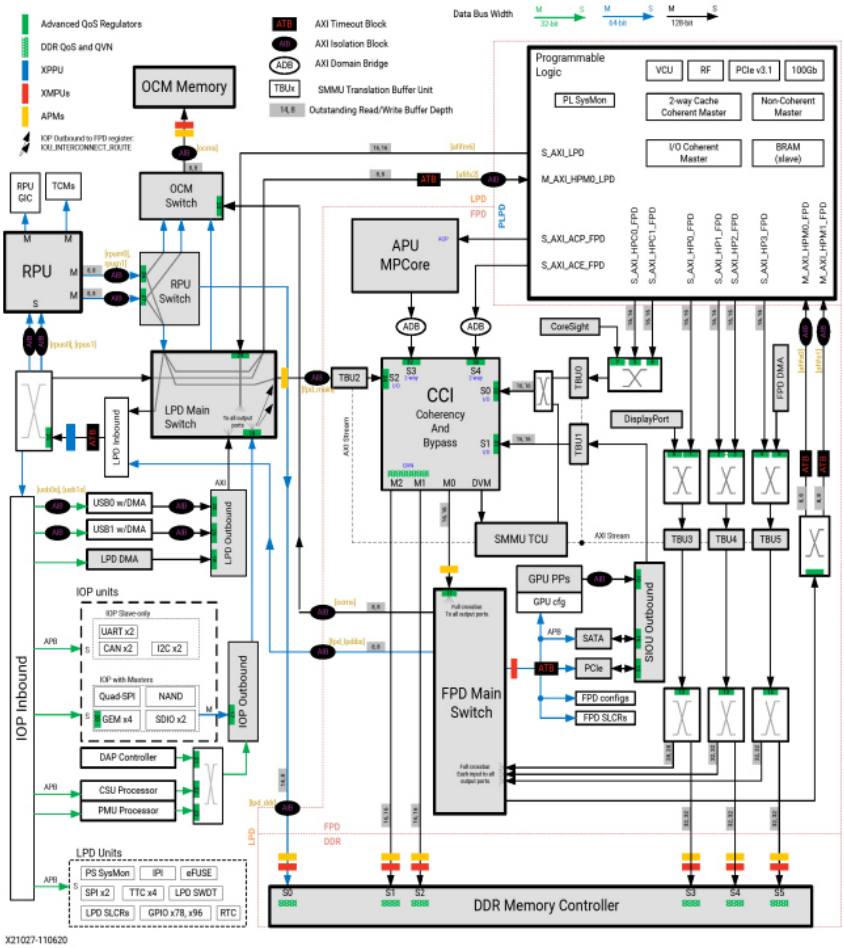

Figure 5.3: Architecture of the Zynq Ultrascale +

5.2 Methodology

5.2.1 Protocol

This section presents the protocol used to get the results presented in this chapter. 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 refills 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 refills made during

```
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, ...);
    ...
}
```

Figure 5.4: C code of the measuring point inside the AOCS process

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 are not directly considered in the measurements. Also, all measurements are taken on one process only, which is considered the victim of the experiment. All other processes (users and kernel ones) and all kernel activities are considered the attackers of the system. In other words, the victims/attackers scheme means that the experiments aim at measuring the impact of the attackers on the execution of the victims. In the ARM Cortex A53 used in these experiments, performance counters registers are 32 bits wide. As the AOCS process is running at a frequency of 8Hz for 16 000 steps, the entire execution takes around 33 minutes. During such a long time, the 32-bit registers can overflow, in this case, the counts restart at zero. In most cases, this is not a problem as we are taking the difference between the 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 2^{32} is added 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 5.4.

A lot of data are collected from the performance counters, mostly on hardware memory events (bus accesses, cache accesses, cache refills, ...). In these experiments, data such as context switches or core migration during the execution of an AOCS process could be interesting. Unfortunately, these data are only present in the next chapter's experiments. 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 identifier of the core the

process is executed on, but nothing assures you that this won't change in the next moments. But it is possible to have the number of migrations of a particular process.

All these experiments are running on a "light" Linux distribution made with Yocto. The Linux kernel used is a 5.4 version of the Xilinx Linux kernel found on the official Xilinx Github [80]. According to Xilinx recommendations ([81]), the Yocto Zeus version from the official Yocto repository ([82]) has been used. Using Yocto allows complete theoretical control of what's inside the Linux operating system. In these experiments, each application is modeled by a Linux process and each Linux process has exactly one thread i.e threads = processes.

5.2.2 Measures impacts

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

Every measure described in the scope of these experiments is taken inside a period of execution. It means that every code outside of the periodic loop isn't considered in the measurement results. In particular, the loading and initialization phases of the application are not in the scope of the measures. This is due to the fact that the Linux system isn't considered to be in a particular state at the beginning of the experiments.

The measures presented here are coming from two different sources. The first one is clocks, managed by the operating system and used to measure 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. The impact 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 that aren't in the main memory. As for the clocks, the impacts of retrieving performance counters values are constant throughout 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 storage space of $16000 * (6 + 2) * 64bits = 1MB$ will be needed. Accessing this space requires accessing memory pages which will generate 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, measured data are placed in local variables placed on the stack of the process and then copied to the memory outside of the measured section. This method will generate fewer impacts on the executed code but will alter a little the memory hardware state.

Finally, the 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 5.5 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. Note that this WCET is an observed one, retrieved through simple measurements. It is used to have an approximation of the real WCET.

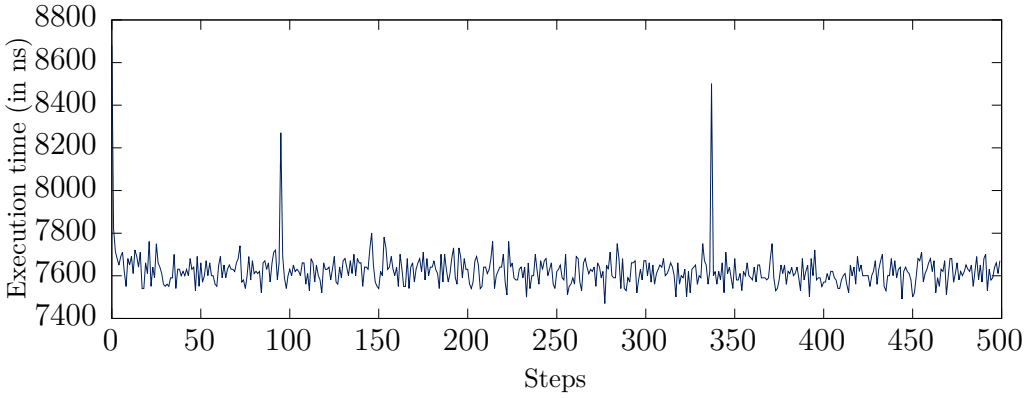

Figure 5.5: Measurement of execution time with no code

5.3 Experiments

The experiments presented here aim at finding Linux configurations suitable for the space context presented in the first part. Linux has 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 that can vary from disabled to enabled. All these experiments were made concurrently, which means that the results of the first experiment were not known at the time the second experiment was conducted.

This section presents 4 experiments making measurements on different configurable points. The first experiment aims at comparing the use of the same or different files by the same processes. The second experiment measures the 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.

5.3.1 Mono file versus multiple files

Files take an important place in these experiments. Not only they are used to store inputs and outputs of the AOCS application, but they also contain instruction data. In fact, executables are files stored in the disk (SD card in this 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 aims at exhibiting the differences between applications using the same files and applications using completely different files. Using different files means using files with the exact same data but with a different name. This configuration could happen when using shared libraries as it is very common in 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 are 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 other AOCS and I/Os processes are considered attackers.

The figure 5.6 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 for the other group. The results of this experiment showed that when the files are the same the performance is better on average. The worst performance is better than for the other group of processes. In these curves, we can notice aberrant

points, for example, point 3 of the mono file curve. Linux has not a reproducible behavior and induces a lot of variability in the system, this is why we may have these points as discussed in the introduction of this chapter. In this experiment, the focus is on the curve trends and not on the particular values. Thus, the non-reproducibility of Linux is not really a problem here.

These experiments were repeated a few times and these aberrant points aren't always at the same spot, although, not enough data have been retrieved to make statistics that could lead to removing these points by smoothing the curve.

Finally, this chart shows that on average the group using the same files is executed approximately two times faster. Moreover, both curves can be separated into 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 are 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 explain this observation, but these are not the only hardware interference possible in the system. However, this is a hypothesis and was not verified by any further experimentation.

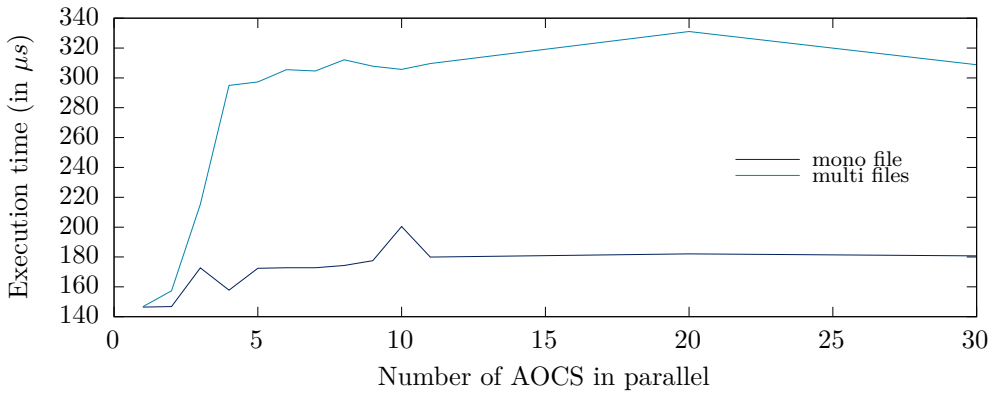

Figure 5.6: Measurement of execution time in average comparison of the two processes groups

Modifying the state of a shared memory hardware device is another kind of interference. The figure 5.7 shows the number of L2 data cache refills 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 pivot point is smoother. As a matter of fact, we can observe a behavior change around point 4 but the transition between the two phases is not abrupt. In this case, we still can see an increase between points 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 refills the execution time of the process will increase. Then, the second phase shows a smaller rise in 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. A 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.

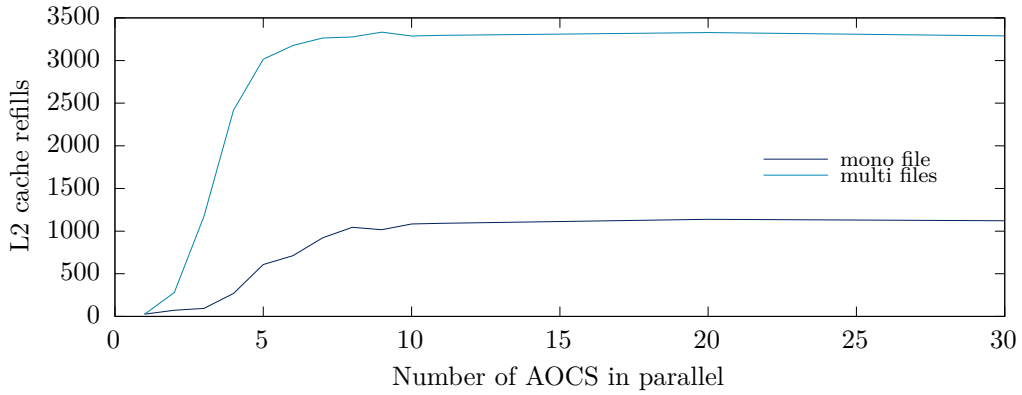

Figure 5.7: Measurement of L2 data cache refill in average, comparison of the two processes groups

Now that the shape of the curves is explained, let us focus on the differences between the two groups of processes. In both charts, there is 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 better reusability of the data which can be found in the caches. But the address space of different processes in Linux is completely separated. In other words, data from process A cannot be shared with process B, and thus process B cannot use data from the cache already loaded by process A. This raises the question about the explanation of the previous results.

When accessing a file in Linux a process uses system calls. All actions related to a file are then handled by the operating system. Even if there are 30 different processes executed in parallel with different address spaces, 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 use the same cache line without needing to refill it because L2 cache is physically addressed.

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

5.3.2 Periodic interference peaks

During the first experiment, an observation was made 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 5.8. In this section, an investigation is made to know if those peaks come from the application code itself or if it is interference generated by the operating system. In this experiment, there is no difference between mono or multi files as we are only considering one AOCS process.

Observation of the phenomenon

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. It is known that the AOCS code 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. For this reason, the hypothesis that the peaks do not come from the AOCS code itself seems plausible. One other simple hypothesis could be that those peaks come from a context switch.

To find out which hypothesis is true, the measurements were done a second time with a modified period. In this case, the period of the peaks didn't change. The

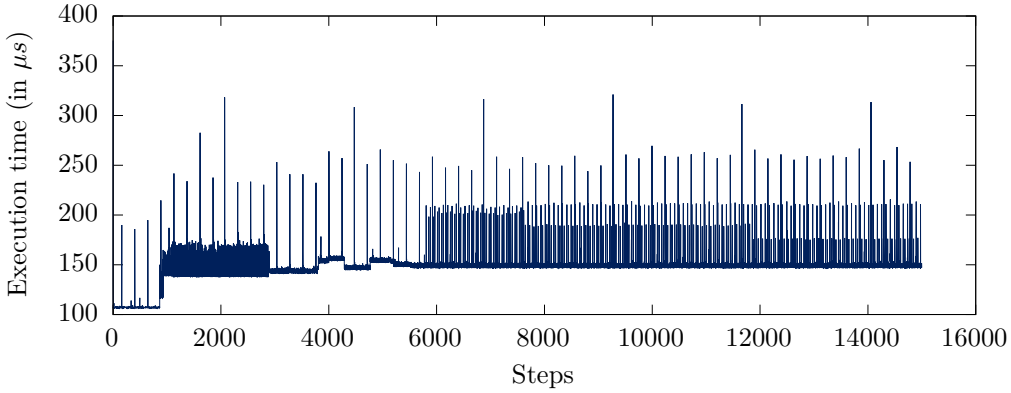

Figure 5.8: Measurement of execution time for each step of an AOCS application with a period of 125 ms without any other application running in parallel

figure 5.9 shows the results for a period of 50ms. In fact, there are 20 peaks in 625 seconds in both curves.

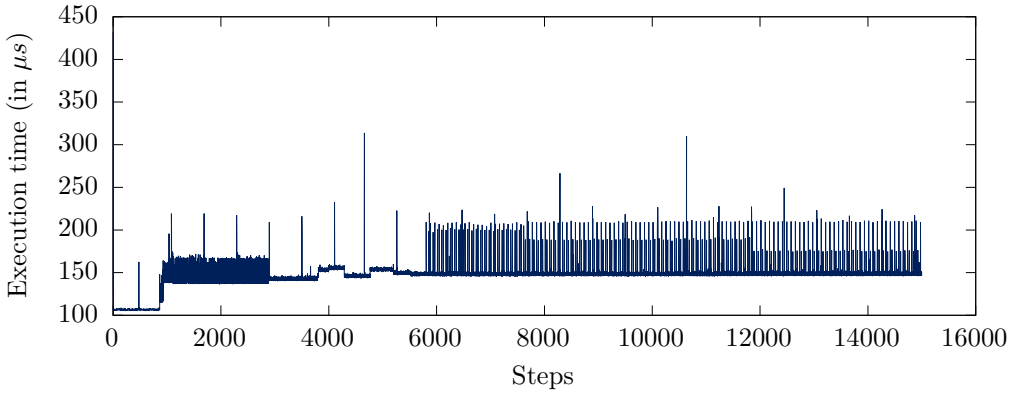

Figure 5.9: Measurement of execution time for each step of an AOCS application with a period of 50ms without any other application running in parallel

It is clear that the period of those peaks isn't based on the steps of the AOCS application. Therefore, these peaks come from the operating system. Note that these peaks also appear in the figure 5.4 which also confirms that the interference doesn't come from the AOCS algorithm.

Now let us focus on the second hypothesis. 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. However, 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 5.10 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.

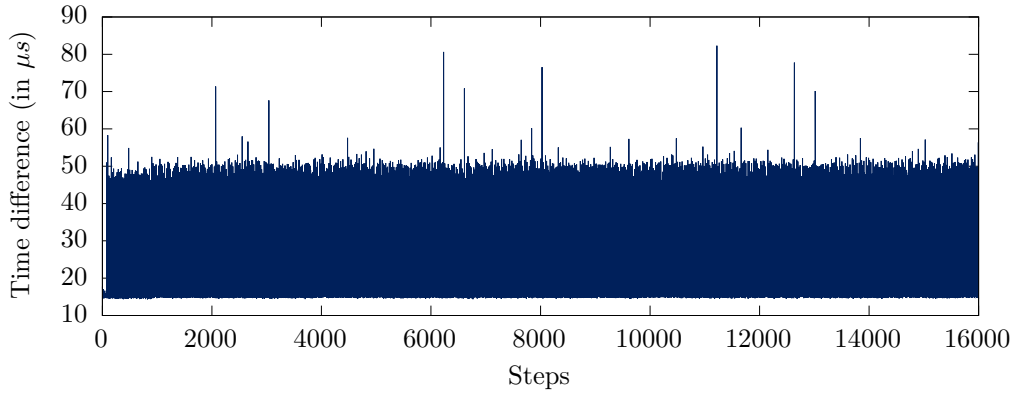

Figure 5.10: Measurement of the difference between processing time and execution time.

Finding the source

Now the question is to find where the peaks come from. We can note in the figure 5.8 that there are two types of peaks. There are smaller peaks with smaller periods and bigger peaks with bigger periods. We could also note that there are exactly 9 small peaks between two big peaks. It seems that two periodic phenomena impact the AOCS process. There are 2 hypotheses for the second phenomenon with the bigger peaks. Either it is a particular execution of the first phenomenon occurring every 10 peaks. Or it can be from a completely different source and the 2 phenomena are synchronized for some reason.

The measures taken from the performance counters are difficult to analyze because the AOCS algorithm doesn't have the same behavior at each step. 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 plays a role in the value of those peaks or if they are only induced by the environment.

Answering these questions leads to creating another algorithm to help at finding 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. The same measurement techniques are implemented, as for the AOCS process. The measures from this new algorithm give the same peaks in the execution time data. The two types of peaks appear with the same intensity and period. This means that changing the executed code did not modify the occurrence and intensity of the observed impact. The only thing that stays the same between these two experiments is the Linux system on which the processes are executed. This confirmed once again that the operating system is altering the behavior of the processes. What's interesting in this case is that the memory accesses and the L1 data accesses are constant at each step. Each new refill from the L1 and L2 data cache can then be considered interference from the operating system. As shown in the figure 5.11 there are also peaks in the L2 data cache refill. Those peaks occur at the same steps as the ones observed in the execution time curve.

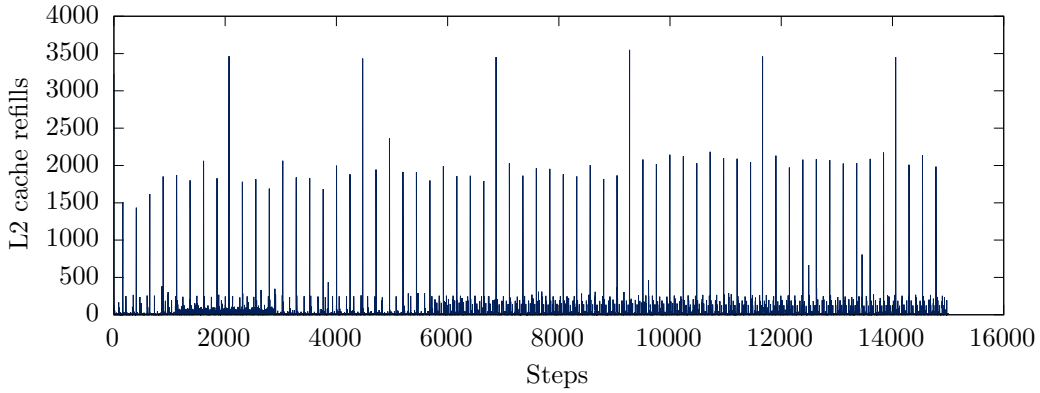

Figure 5.11: Measurement of L2 data cache refills for the AOCS application

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 awakened and has 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 5.10 is not constant. In fact, as the process has 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 a 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.

The best way to identify the source of these peaks is to know exactly what is executed before, during, and after each step where the peaks occurred. To do that, kernel trace tools can be used. A kernel trace tool allows getting information about events occurring in the kernel code such as context switches, system calls, or interruptions (hardware and software). That information is time stamped which makes it easier to analyze them. When the data are retrieved they can be displayed which makes them more readable for a human. For this particular experiment, LTTng ([83]) has been used to get the information from the kernel and Trace Compass ([84]) to display it. To reduce the impact of the trace on the experiment, only the context switches have been retrieved. This allows knowing exactly which processes were running at a specific time and also the duration of those processes. By knowing the time when a peak occurred, it is then possible to find information about the others processes around the particular execution of the AOCS process.

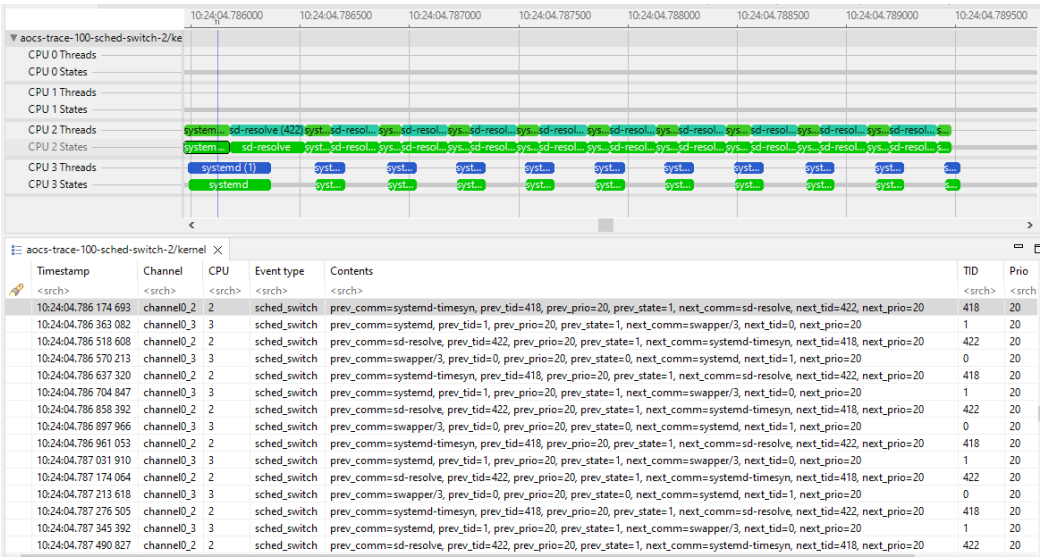

Figure 5.12: Screenshot of Trace compass before a peak occurs

The figure 5.12 shows the result of an LTTng trace done during a short execution of the AOCS process (around 100 steps). The period presented in the screenshot is right before the execution of the AOCS process. It appears that more than one process is executed during this period of time. In fact, this is due to a particular Linux systemd service called timesyncd. This service is responsible for the synchronization of the system clock. To do so, it uses the network to get information from the NTP protocol. The details of the timesyncd behavior won't be explained here, but what's important is that it needs to send ethernet packets to the network interface. This is why other services are shown on the screenshot of the figure 5.12. The first other

service is resolved which is another Linux systemd service. This service is responsible for the name resolution for local applications. It is used by local processes to transform website names into IP addresses. The last service shown in the figure 5.12 is networkd, which is once again another Linux systemd service. This service manages the network configurations of the Linux system. Again, it is not important here to understand what these services are doing exactly on the system. The only useful information is to know what is their impact on the system to be able to confirm that these services are the source of the observed peaks. The first answer would be that the awakening of these services always occurred before a peak shows up. The second answer is that during all of the 100 steps of AOCS executions used in this trace, these services are the biggest processes in execution time to occur. Finally, the services discovered by looking at the trace are all related to the network. In fact, the SoC is connected to a local network through the ethernet socket. However, it means that ethernet packets have to be handled and potentially be stored in the memory at some point. Thus, ethernet packets may have been loaded onto the shared l2 cache during this period. This would confirm that the peak present in the AOCS execution curve is indeed induced by a peak in the l2 cache refill curve.

Verification of the hypothesis

The last sub-section presented a strong hypothesis on the source of the observed peaks. But for this hypothesis to become an answer it needs to be verified. The best way to verify this hypothesis is to shut down the timesyncd service from the Linux system and to re-do the experiment. The figure 5.13 shows the result of this experiment and it is clear that one type of peak has disappeared. Removing the timesyncd service deleted the high-frequency peaks but the other type of peaks remains. It seems that at least two phenomena were responsible for the peaks. Those two periodic phenomena don't have the same period but are synchronized. This proves the hypothesis described in this section that the two types of peaks come from two different things. Also, the big peaks, shown in the figure 5.12 seem smaller than in the previous experiments. This is again an argument that the two types of peaks are cumulative.

To find the second type of phenomenon which create the big peaks, the same method can be reused, i.e using LTTng with trace compass to find which process was executing before a peak. The result of this analysis is presented in the figure 5.14.

Another time synchronization service was enabled in the system, the ntpd service. Then by removing NTP from the Linux distribution used in these experiments, the peaks completely disappeared as shown in the figure 5.15. This is also the case for the L2 cache refill curve, in fact, the peaks also disappeared in this curve.

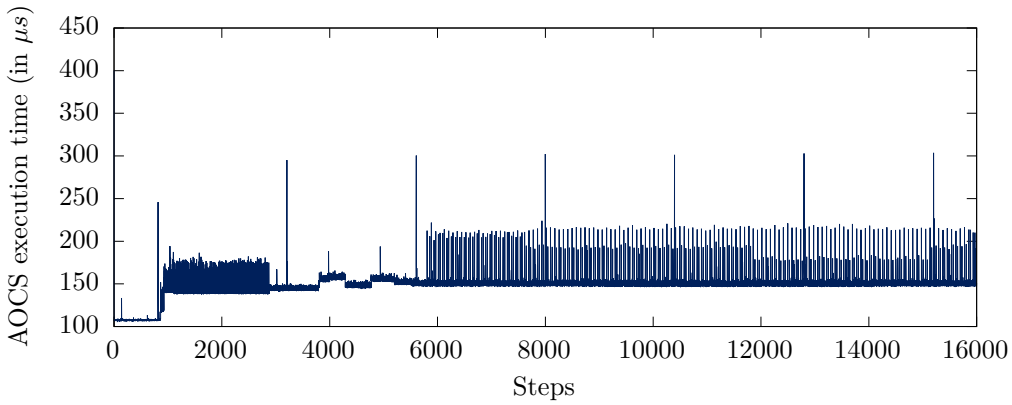

Figure 5.13: Aocs execution time after removing the timesyncd service

The new face of the AOCS execution time curve

The performance of the AOCS process seems better without those two services. In fact, these services impacted both the execution time and the number of l2 cache refills of the AOCS application. Without them, it is possible to have a better approximation of the "real" AOCS execution time curve. To do that, it is possible to take the minimum execution time at each step and plot this new set of points. The figure 5.16 shows the minimum at each step of the AOCS execution time. This minimum has been obtained by comparing 3 different execution of the AOCS process.

The curve presented in the figure 5.16 is certainly a better approximation of the true AOCS execution time curve than the one presented in the figure 5.15. However, it is still not the real AOCS execution curve, the one that could be obtained by running the AOCS alone on a bare-metal system. Obviously, this curve is theoretical, in fact, execution time only means something when it is attached to a specific platform (hardware + software platform). Considering the execution time of a theoretical algorithm doesn't make any sense without considering how it will be implemented on a real hardware platform. In the particular case of the AOCS, the algorithm does a lot of calculations based on the current position and attitude of the spacecraft. When written on paper, those calculations are only pure mathematics. In order to know what time you need to resolve those calculations, you need to know what time you need to do the simple arithmetic functions and trigonometric functions. However, in modern processors, it is a hard problem to resolve and this time isn't constant. This is due to complex mechanisms that make processors faster on average but less deterministic. But processors aren't the only culprits in this problem. As discussed before in this thesis, Linux does not have a reproducible behavior. Not only you can not know for sure how much time a system call will take to give an answer, but you

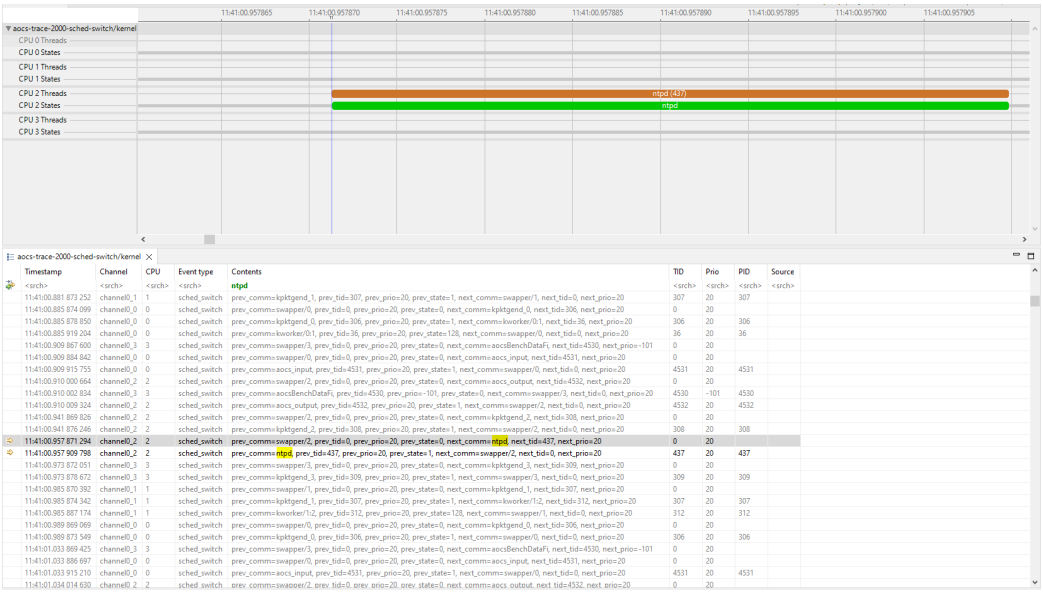

Figure 5.14: Screenshot of Trace compass before a peak occurs after removing timesyncd

can not know for sure what exactly is running at a specific time.

To understand the execution time of an algorithm it is mandatory to understand how it is implemented in the real world. Then, how is it possible to find the impact of Linux on a particular algorithm if Linux is intrinsically part of how the execution time is calculated? In other words, the best way to remove the impact of Linux on an algorithm is to not run it on Linux. In a previous paragraph, a mention was made of the possibility to compare an AOCS execution time curve with one obtained by running the AOCS algorithm on a bare-metal platform. However, this comparison would not have any sense because the implementation of the algorithm is not the same. The minimum curve presented in the figure 5.16 is the approximation of the AOCS execution time curve that will be taken for comparison in further experimentation. Taking the minimum of multiple executions allows to reduce the impact of periodic and sporadic events. However, Linux may add an overhead as it is shown in the figure 5.5, but this will not be a problem as long as it is constant over time.

5.3.3 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 between the

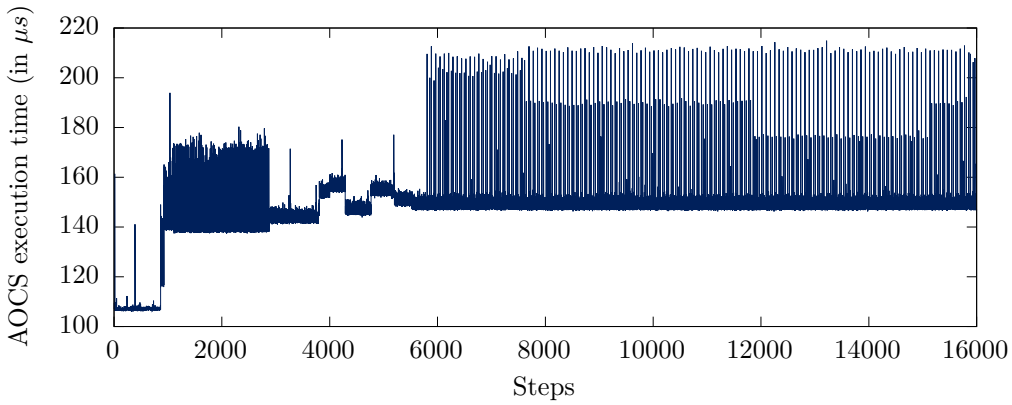

Figure 5.15: Aocs execution time after removing the ntpd service

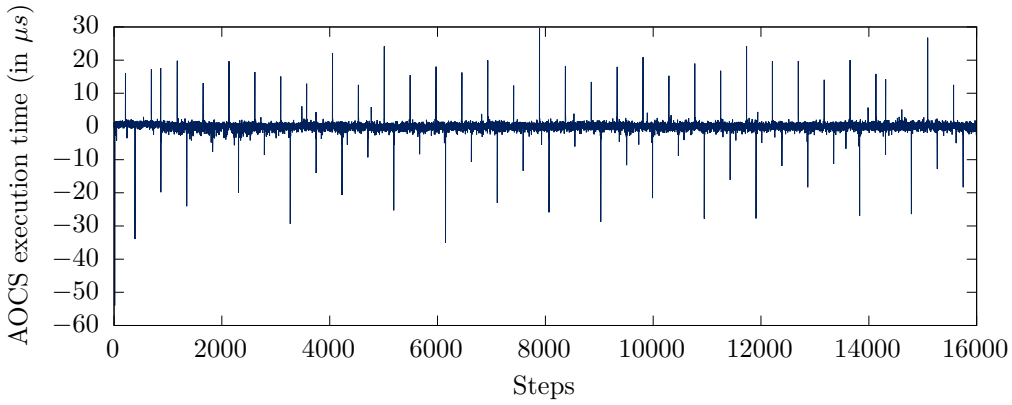

Figure 5.16: Aocs execution time minimum for each step of 3 different execution

process and the sensors sending input data. This desynchronization could become a problem, especially since satellites might be running for years without being rebooted. This experiment aims at measuring the wake-up date of the AOCS process with the CLOCK_MONOTONIC of Linux which is a system-wide clock. The chart presented in the figure 5.17 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 than it should have been if the scheduler used an exact period of 125.00 ms.

The first observation made on that curve is that the scheduler seems to induce 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 in the previous section also

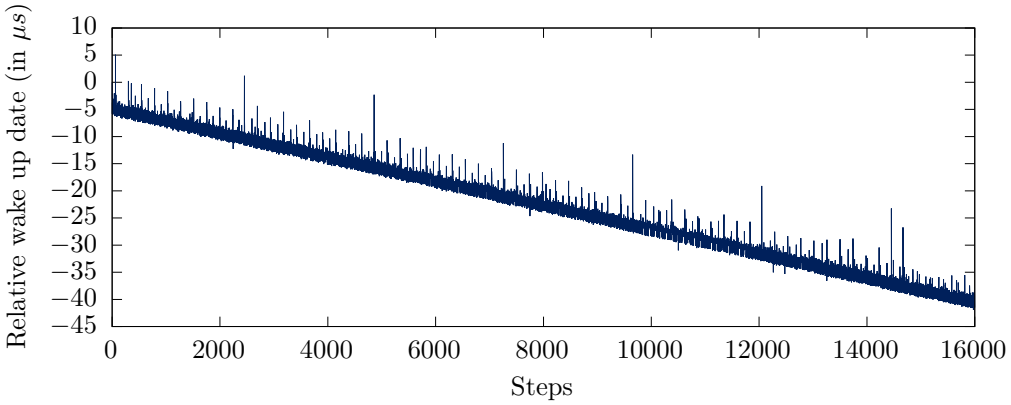

Figure 5.17: Measurement of the wake-up date for each step relative to the first wake-up date

appear on the wake-up date curve. Those peaks are similar in the figure 5.17 and in the figure 5.8 even though they came from different execution. The scheduler drift seems to be linear but can be impacted by the same phenomenon as the AOCS process itself. An observation has been made 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 it is possible to look at the difference between two consecutive wake-up dates. The figure 5.18 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 step 50 means that the difference in the wake-up date between step 50 and step 49 is 125ms + 1 μs.

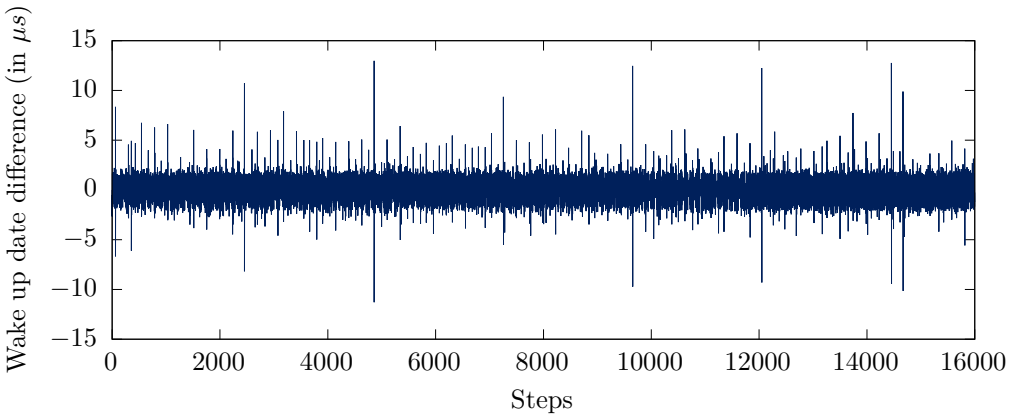

Figure 5.18: Measurement of the wake-up date difference for each step

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

phenomena producing those peaks both in the scheduler and the AOCS process are correlated. Moreover in the figure 5.18, the positive peaks seem to be correlated with negative ones. The figure 5.19 shows the same curve but only the first 100 steps. At step 61 we could see a positive peak of around 8 μs. This peak is followed right after by a negative one at step 62 of around -6.5 μs. The hypothesis behind this observation is that the scheduler tries to catch up with its delay at step 61 by scheduling the next step a little earlier. But the positive delay in step 61 is greater than the negative delay in step 62. As a result, the scheduler induces a positive delay at step 63. By looking closely at the figure 5.18 we observe that this phenomenon is reproduced for each positive and negative peak. Another hypothesis could be that the time spent executing the operating system code isn't taken into account for calculating the next period.

The rationale can then be generalized for 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 5.17 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 is globally drifting and the process at step 15 000 is scheduled sooner than it should have been.

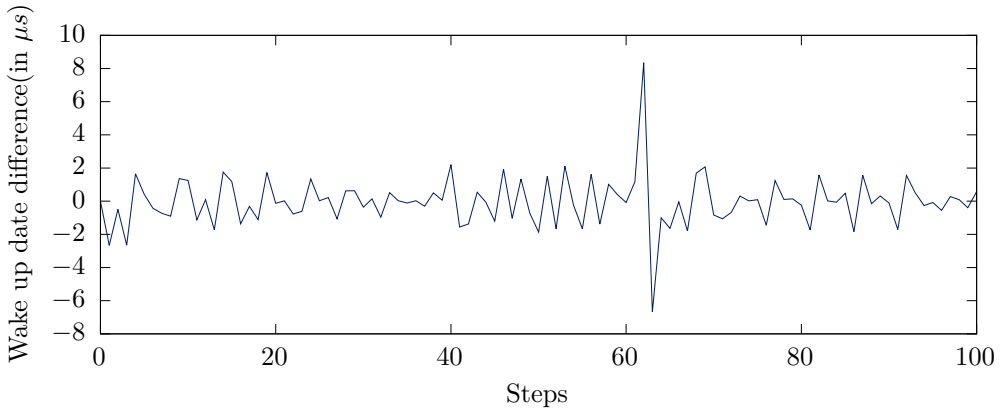

Figure 5.19: Measurement of the wake-up date difference for each step for the first 100 steps

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 question of whether the phenomenon results from a desynchronization between two internal clocks or a software behavior or the scheduler itself.

Impact of removing timesyncd

The previous subsection showed that there is a drift in the wake-up date of the AOCS process. Time in Linux can come from different sources. It is then hard to understand if the drift measured came from the desynchronization of two hardware clocks or the desynchronization between the scheduler and one of the clocks. It has also been shown that the peaks presented in the last section also appeared on the wake-up drift curve. However, these peaks came from the time synchronization mechanisms of Linux. It raises the question of the impact of removing these mechanisms on the wake-up drift curve.

As shown in the figure 5.20 and 5.21, the curves without the time synchronization mechanisms are difficult to analyze. In fact, it seems that the measure is oscillating between different values and thus there is no drift anymore. To confirm this hypothesis it is possible to zoom into the curve. The figure 5.22 shows the first hundred steps of the wake-up drift curve. The scheduling delay is oscillating between 1 ms and 3.5 ms and is never negative. This means that not only the process is always scheduled later than it should be but the scheduling delay is not constant. Removing the time synchronization mechanisms changed the behavior of the scheduling policy in some way. It went from creating a drift in the wake-up date of the process to an oscillating delay. This observation might be an explanation of why these services are enabled by default on a lightweight embedded Linux system. However, these impacts were impossible to predict without having a deep understanding of how the Linux kernel works. As the number of kernel features and services is pretty big and each one of those components is really complex, it is making the measurement approach a better way to understand rapidly how the Linux configuration impacts the whole system behavior.

Another observation that can be made on the curves presented in the figure 5.22 and 5.23 is a quantitative one. In fact, in the previous paragraph, the observed drift was creating a shift of around -40μs. The order of magnitude of these new curves is way past the 40μs, in fact, the figure 5.22 shows that the delay is oscillating between 1 ms and 3.5 ms. This is almost 100 times bigger than the 40μs of the previous curve. This difference can be explained by a loss of precision in the scheduling algorithm resulting from the removal of the time synchronization services. The figure 5.23 shows that the difference between two consecutive wake-up dates is either 128ms or 124ms. More precisely, if we don't take into account the first point, there is a difference of 128 ms every 4 steps. The other steps have a difference in wake-up date of 124 ms. It seems that the scheduler is trying to catch up on the delay by scheduling the process sooner than it should have been.

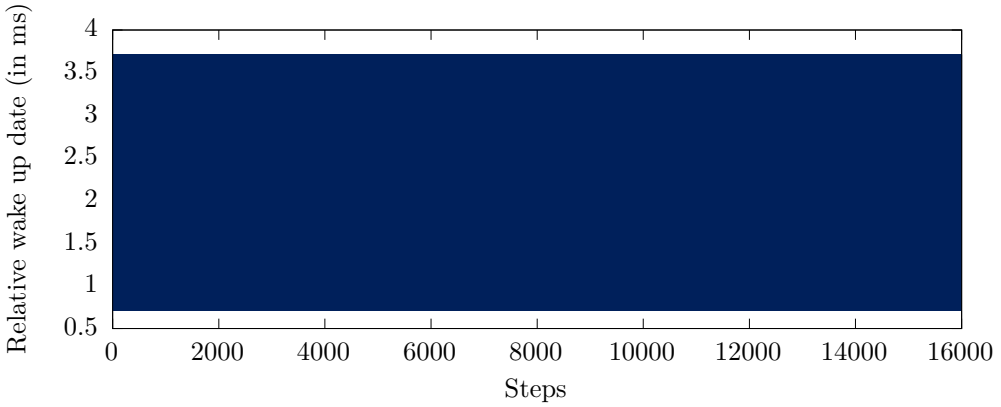

Figure 5.20: Measurement of the wake-up date for each step relative to the first wake up date after removing the time synchronization services

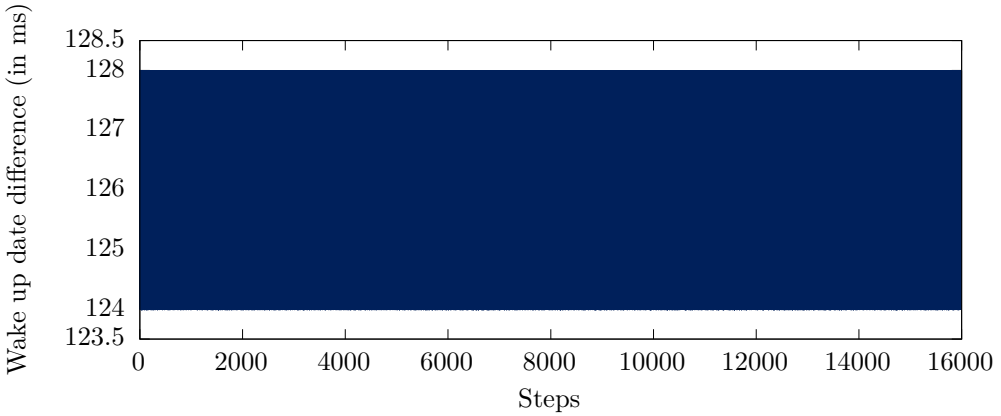

Figure 5.21: Measurement of the wake-up date difference for each step after removing the time synchronization services

5.3.4 Impact of PREEMPT-RT

One of the first questions, when Linux is proposed in a critical embedded context, is about the necessity to use the PREEMPT-RT patch. The 2 last experiments have been reproduced 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, observations are described as follows:

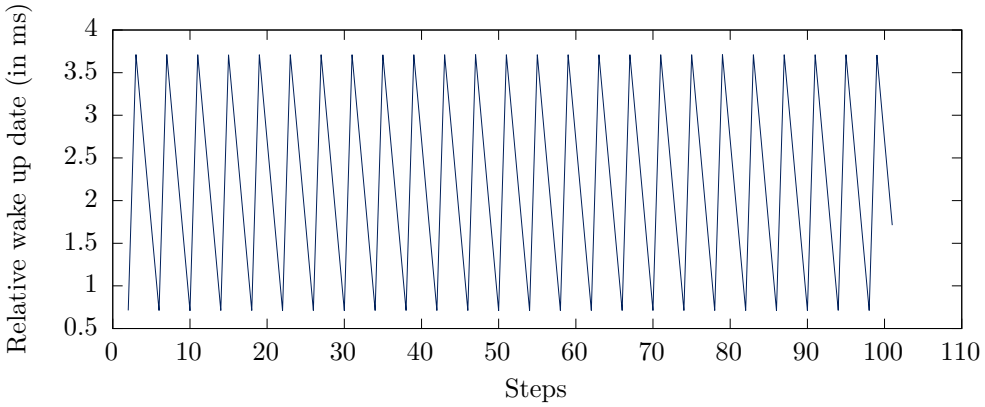

Figure 5.22: Measurement of the wake-up date for the first 100 steps after removing the time synchronization services

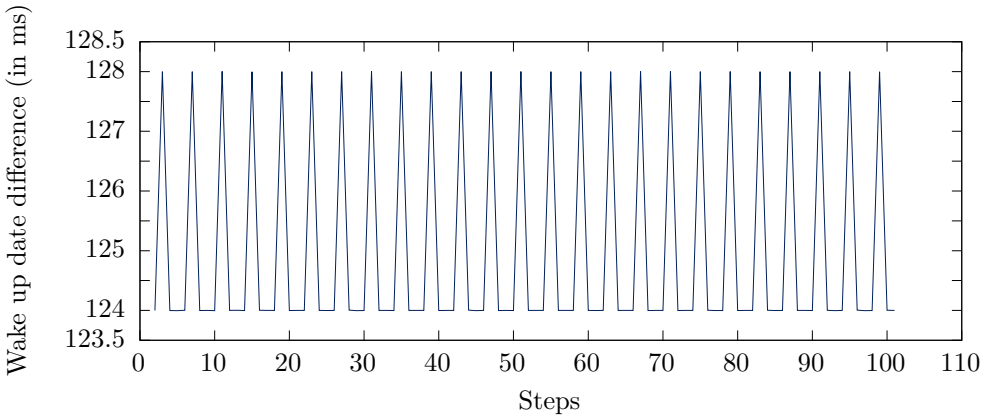

Figure 5.23: Measurement of the wake-up date difference for the first 100 steps after removing the time synchronization services

- 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. The measurements show 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 times.
- 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

phenomenon that results in a higher impact on the AOCS process.

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 for 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.

5.4 Difficulties encountered during the experiments

By looking at all the charts presented in this chapter, it may seem easy to retrieve performance counter measurements through Linux. However, during these experiments, some difficulties have arisen. This section describes the encountered difficulties and explains why they are in fact difficulties.

5.4.1 Measuring hardware performance through Linux

Linux is at the heart of these experiments and more generally the work of this thesis. But Linux is not designed to measure hardware performance in user space. Few tools exist to interact with the performance counters and it is hard to configure them.

In the early stage of the creation of these experiments, just after the light distribution of Linux was configured, it was expected to use a tool called PAPI ([85]), a tool library made by the University of Tennessee. It relies on a Linux tool called perf which is responsible for giving performance data to userspace. However, the configuration of these tools was not documented for the Xilinx custom kernel used in these experiments. It is then impossible to know if the tool is incompatible with the Xilinx kernel or not. After a few tries, it was decided to stop using these tools and to configure the accesses to performance counters with assembly code. This solution is not easier but is focused on accessing the performance counters, while the other tools needed a lot more configurations because they access a lot of other performance data which was not interesting in this context. Finally, after creating a simple Linux driver to allow access to the performance counters through userspace, it was possible to read the performance counters and configure them with a few lines of assembly code. This configuration took some time and was not intuitive.

5.4.2 Performance counters reliability

The performance counters give a lot of information on what really happens in the hardware. In the scope of this thesis, it can be really useful to understand the interference occurring in different contexts. However, there are some problems regarding the reliability of the performance counters retrieved data.

First of all, the performance counters have a maximum value that depends on the size of the register. Once the counter reaches this value it starts counting again from zero. If this occurs during the execution of one AOCS step it will create inconsistent value. These values can then be modified after the experiment to remove the inconsistency. This way it does not have an impact on the experiment. Second of all, most of the performance counters are associated with one core which means that in case of a core migration the returned value will also be inconsistent. Finally, due to the unpredictability of some hardware phenomena, the value returned by the performance counters might not always be 100% accurate. For example, memory accesses might not be counted by the performance counter if it is registered after the performance counter register is read. This means that it is hard to analyze the results given by the performance counters.

Chapter 6

Cgroups experimentation

The previous chapter was focused on the impact of the system as a whole on an AOCS software application. This chapter, however, aims at studying a specific Linux mechanism to isolate processes running on the same platform at the same time. The Linux mechanism used in the experiments presented in this chapter is called: cgroups. More specifically, it is the CPU time limitation feature that is used here.

6.1 Problem statement

In a satellite, the AOCS algorithm is critical, meaning that without it the spacecraft behavior is uncontrolled. Thus, it must be running regardless of what is happening in the other cores and the operating system. A process can be impacted by its environment in different ways:

- By modifying its direct behavior (access to its private data, I/Os modification)
- By blocking its access to resources which modifies its dynamic behavior and may modify its functional behavior.

This chapter focuses on the second aspect and particularly on the modification of the dynamic behavior of the AOCS process.

During its execution, a process will use different hardware and software resources. The CPU and the memory are absolutely necessary to execute a process. This chapter focuses on those two hardware resources. The goal of the experiments presented below is to find out if the cgroups can be used to reduce the impact of other processes using the CPU time and the memory during the execution of the AOCS process. The expected behavior is that the CPU time limiting mechanism brought by the Cgroups will stabilize the behavior of the whole system while preventing processes to deprive the AOCS process from accessing the CPU.

6.2 Cgroups

As explained in the section 2.4, cgroups provide control of the resources accessed by a collection of processes. In particular, cgroups can be used to limit the usage of a certain resource. This mechanism can be used to improve the isolation of the AOCS process in the system. It is not in complete isolation as the one provided by virtualization which makes the system more dynamic. All the processes on the system are using the same hardware resources and CPU cores which means that they can still impact each other through interference. Using cgroups to limit the usage of specific resources should theoretically reduce these impacts by stopping the processes that try to use more resources than they can access. This chapter focuses on one specific aspect of the cgroups, the limit in the usage of the CPU time. This aspect has been selected because stopping an applciation from using a CPU is a good solution for reducing memory interference. In fact, the less an application is running the less it is impacting the other applications.

6.3 Scheduling policies

In the experimentation presented in this chapter, two scheduling policies will be compared. The first one is SCHED_DEADLINE based on the global EDF algorithm. This policy is described with more details in the section 2.4.4. The second policy, which is also described in the section 2.4.4, is SCHED_RR. These two different policies are very different. One is using deadline and period to chose which process to run while the other is running all the processes in a predefined order for a specified quantum of time based on priorities.

SCHED_DEADLINE has been chosen because it allows running a set of periodic processes without having to find the best scheduling order in advance. It is particularly useful in a context of different applications developed by different teams and integrated on the same platform. The goal of the experiments presented in this chapter is to evaluate the impacts of cgroups on the system and the memory interference. Unfortunately, it is not possible to use the cgroups mechanism with the SCHED_DEADLINE policy in the native Linux kernel. It is why an academic patch is used to enhance the functionalities of the Linux scheduler. This patch adds a two-level scheduler to the Linux kernel. The first level schedules the cgroup with SCHED_DEADLINE while the processes inside the groups are scheduled with SCHED_RR.

In order to have the best understanding of the cgroups impacts, 4 experiments will be made. The first one is using SCHED_DEADLINE without cgroups. The

second one is using SCHED_RR without cgroups. The third one uses SCHED_RR with two cgroups. The last one uses the academic patch with two cgroups. Using these four different experiments allows to compare the impact of the cgroups in different scheduling contexts. In fact, to understand the consequences of the academic patch it would not have been enough to compare the results to only one of the other three experiments, since the academic patch uses cgroups, SCHED_DEADLINE and SCHED_RR.

6.4 Use cases and benchmarks

In this chapter, the experiments are implementing a victims/attackers scheme. This means that the impact induced by the attacker's processes will be measured in the victim's processes. All of these processes are running periodically on the same hardware platform as the ones presented in the previous chapter. The operating system used is the same Linux distribution as the one used in the previous chapter. However, the timesyncd and ntpd services have been disabled. This choice has been made to reduce the sporadic impacts of Linux on all the processes, both victims and attackers. But, as seen in the previous chapter, removing those services has an impact on the scheduler's behavior. Those impacts are considered periodic and deterministic, thus they will affect the measures but in a way that can be controlled. This matter will be discussed in the section 6.5.4. All the processes in these experiments have the same period and deadline and are synchronized to start each period at the same date. Thus, at each step of the AOCS process, every other process is also executing one step.

6.4.1 AOCS case-study

The AOCS algorithm used in the experiments presented in this chapter is the same as the one used in the experiments of the previous chapter. Nevertheless, a few changes had been made to the process that is implementing this AOCS algorithm. The goal of these changes is to make the experiments easier. In particular, it will help analyze the results.

The first change is about the period of the AOCS, which was reduced from 125ms to 10ms. Let us talk about the impact of this period's modification. The behavior of the AOCS is not modified by changing its period as long as its inputs are available and that it has enough time to execute between two steps. As AOCS outputs are not used there are no functional impacts. One step of the AOCS takes around a maximum of 0,25 ms to execute as it is shown in the curves presented in the chapter 5. This means that the scheduling period is at least 40 times bigger than the average

execution time. Also, the impacts that will be presented in the rest of this chapter won't be compared to the execution time measured in the experimentation presented in the previous chapter. Thus, not having the two same AOCS processes does not have an impact on the results that will be shown here.

Another big change is that only a small part of the AOCS algorithm is used for these experiments. Instead of running the entire 16 000 steps, only 800 steps will be executed. A portion of the AOCS curve with a reduced amplitude has been chosen to reduce the impact of the differences between the two different steps. This portion is the one between the 3000th step and the 3800th. As we can see in the Figure 5.16 the execution time has a low amplitude and it is around the average execution time of the whole curve between those two boundaries.

Both of these changes have also been made to reduce the time of one experiment. Running the new AOCS process takes only 9s which is short compared to the 33 minutes taken by the old version of this process. By reducing the experiment time it allows to considerably increase the number of times one experiment can be repeated. By doing so it is then possible to aggregate the results to use averaged data. Thus, it will reduce the variability of the curves induced by the non-determinism of Linux.

6.4.2 Benchmarks

In this chapter, the AOCS algorithm is not the only use case. Others applications are introduced to play the role of attackers. These applications called benchmarks are based on the ones presented in [86]. The goal of these benchmarks is to have a well-defined and controlled profile of the attackers' algorithms. In the experiments of this chapter, two types of benchmarks are used and are described in the following subsection.

Benchmarks Pi

The first type of benchmark is called the benchmarks Pi. These benchmarks aim at using CPU time and only CPU time. To do so, they implement a simple Pi calculus algorithm and put every variable in hardware registers to avoid making memory request each the algorithm needs to use a variable. The algorithm used is based on the infinite sum below:

$$\frac{\pi}{4} = \sum_{0}^{\infty} \frac{(-1)^k}{2k+1}$$

Thus, in the assembly code of the periodic loop of this benchmark, there are no memory accesses (neither loads nor writes). Also, this benchmark is entirely configurable, by modifying the number of steps needed in the Pi calculus it is possible to increase the execution time of the algorithm and then increase the amount of CPU time it will

be using. By precisely configuring the amount of CPU time used by the benchmarks it is possible to know the exact impact it has on the system. Like the AOCS process, the implementation of this algorithm is periodic and has a deadline. The periodic part is the loop responsible for the Pi calculus. This means that for each periodic step it will compute Pi one time. The benchmark is then configured to do 800 steps and to start at the same time as the AOCS process. Let's consider N the number of iterations in the Pi calculus, then the benchmark process will compute:

$$\sum_{k=0}^{N} \frac{(-1)^k}{2k+1}$$

The time complexity of the algorithm implementing the computation of this sum is $\mathcal{O}(N)$ which means by definition that N is linearly related to the execution time of the algorithm. In practice, the execution time of the benchmark process is not completely proportional to N. First of all, starting and initializing the process takes some time which happens once at the beginning of the execution. However, this overhead is not taken into account here because the execution time that is considered for the benchmarks pi is the execution time of one periodic step. In other words, the benchmark pi implements a loop of 800 steps and the execution time is considered between the beginning of the loop and the end. Finally, the execution time is considered linear to N.

Benchmarks Load

The second type of benchmark is called benchmarks Load. It has the same scheme as the benchmarks Pi but instead of using CPU time its goal is to fill up the L2 shared cache. It means that each memory access will be stored in a different line of cache. It is possible by implementing a simple stride in the memory accesses done by the algorithm. As for the benchmark Pi, it is completely configurable and thus it is possible to determine which percentage of the cache it will be using. The process is also periodic with a 10ms deadline and period and each periodic step fills up the cache to the same amount. As for the AOCS and the benchmark pi, it will be running for 800 steps.

There is still a big difference between those two types of benchmarks. The benchmark Pi is a special kind of benchmark since it uses only the CPU which is the only shared resource that is absolutely necessary to execute an application. In other words, it is impossible by definition to execute only loads without using CPU time. This seems obvious but it changes completely the results analysis of the two benchmarks. Increasing the number of loads will mathematically increase the amount of CPU time used by the process. It means that the resulting analysis will be related to the benchmarks load and not the loads. To clarify, it will be impossible to conclude

that only the loads had an impact on the system but it will be possible to say that the benchmarks load had one. Note that if the relation seems linear at first, it might not be in practice. Each load doesn't take the same amount of time to be completed as this execution time depends entirely on the addresses that are accessed and the activity of others processes running on the same platform.

6.5 Two-way variations experiments approach

6.5.1 Methodology

The global approach of these new experiments is the same as the previous one. The AOCS process is still considered the victim in these experiments and it is the same algorithm as in the previous chapter. The idea is to observe and analyze the behavior of the Cgroups CPU time limitation feature. A comparison between the different configurations of the Linux Cgroups will be proposed. The final goal is to describe the behavior of the Cgroups CPU time limitation feature by comparing it to other configurations of the Linux system. As the Cgroups feature put in place aims at isolating processes, the experiment is designed with a special protocol to increase little by little the impact of processes on the AOCS one. The AOCS process will thus be running along with other processes, but the configuration and behavior of those processes will vary.

In order to have a better understanding of what is actually varying in the system, the others processes running along with the AOCS process will have a specific behavior. Two types of benchmarks are used, the exact behavior of these processes is described in the previous section.

The first configuration that will vary is the number of benchmarks running in parallel with the AOCS process. This configuration has been chosen because increasing the number of processes running in parallel will likely increase the number of interference. The second configuration is the amount of shared resources used by all the benchmarks. This configuration has been chosen to target the shared resources where interference are likely to happen. Here, the considered shared resource depends on the type of benchmark. For example, with the benchmarks pi which are using CPU time only, the second variation will impact the global execution time of all benchmarks altogether. An important note here is that the two configurations are theoretically completely independent, i.e the variation of the first one doesn't imply a variation of the second. For example, as the second configuration is the amount of shared resources used by all the benchmarks, increasing the number of benchmarks doesn't modify the amount of shared resources used by the benchmarks.

The table presented in figure 6.1 shows an example of the parameter variation for

		Number of benchmarks																
		1	2	3	4	5	6	7	8	9	10	11	20	30	40	50	60	70	80
Execution time	1ms	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	2ms	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	3ms	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	5ms	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	7ms	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	8ms	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	10ms		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	15ms		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	20ms			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	25ms			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	30ms				X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Figure 6.1: Table showing the two parameters variation approach

the benchmarks Pi. Each cell with an "X" in this table represents an iteration of the 800 steps of the AOCS process. For each column and row of this table, only one parameter is varying. For example, in the row "7ms" the global execution time of all benchmarks is constant and by definition is 7 milliseconds. This means that from the AOCS process point of view, the amount of CPU time used by the attackers is 7ms, but the number of benchmarks sharing this CPU time increases. As the deadline of all periodic processes in these experiments is 10ms and the ARM cortex A53 has 4 cores, then there are $4 * 10ms - 7ms = 33ms$ available which is greater than 10ms, thus there are 10ms available to execute the AOCS code. The empty cells represent the impossible experiment configuration because a benchmark cannot be executed for more than 10ms.

It is also possible to determine the percentage of CPU time used by the attackers' processes. This percentage is equal to $7ms/40ms * 100 = 17.5\%$ and expressing the available CPU time as a percentage of total capacity provides a view of the system load, which can be used to foresee potential issues in the system execution. Firstly, the kernel code needed to run such processes also had to be executed. This code is responsible for the system calls, the scheduling but also the management of the processes resources, and configurations. There are also services running in the Linux system such as the ones described in the previous chapter. Those services are not used by the current software and are running in the background, but they might have a bigger priority than the AOCS and benchmarks processes depending on the scheduler policy used. This means that the sum of resource usage by all the processes (victim and attackers) must remain below 100. However, this overhead of Linux is hard to measure and one goal of these experiments is to get an estimate of this overhead by finding a limit in the variations of configuration parameters of the processes running

in the system.

Since the beginning of this sub-section, the experiment's approach has been described through the benchmarks Pi example. However, this approach can be extended to other attackers' configurations and it is possible to replace the "Execution time of all benchmarks" with other shared resources used by the processes. For example, the benchmarks load, described in the section 6.4 are designed to use distinct parts of the shared L2 cache. The same experiment model can then be applied. However, a property of the benchmark Pi makes the experiments and the analysis of the results easier. Indeed, increasing the execution time of one benchmark Pi does not have any effect on the other characteristics of the process. It means that when the execution time is increased it does not impact the number of memory access or any other shared resources in the system. In any other benchmarks that we can think of, increasing the number of access to a shared resource will mathematically increase the execution time.

6.5.2 Measures

The previous sub-section described the experiment approach but to completely understand how these experiments will work it is important to understand what will be measured and how the measures will be made. The measurements performed during the series of experiments that have been described in the previous section will be analyzed by comparison instead of being analyzed as absolute values.

The performance counters of the ARM Cortex A53 and the internal Linux clocks give substantial information on the processes' behavior. As for the previous experiments, a measure is made before the AOCS step and one after in order to evaluate the difference between these two values. These measures are only taken in the AOCS process. However, compared to the previous experiments, scheduling-related measures will be added. In periodic multi-core scheduling, the core allocation and the deadline misses give a lot of information on the health of the whole system. A few changes have been made in the AOCS process to make these measures possible.

All processes (AOCS, input, output, and benchmarks) are launched by a parent process. This parent process has two objectives. The first one is the configuration of the environment, creating and setting shared variables, synchronization of the start of all processes, and ensuring the good exit of the system. Once the AOCS and benchmarks processes have been launched the parent process will just wait for the AOCS process to finish. Thus, it won't have any impact on the experiment as it is just a sleeping process during the experiment time. One of the first goals of this parent process is to ensure the synchronization of the processes starts. To do so, it creates a pthread barrier and put the variable in a mmap which is shared with its children. Then the parent process will wait 1 second to ensure that every child was

waiting in the barrier and then it releases the barrier to start all the processes at the same time. Launching all the processes periodic parts at the same time was necessary to control as much as possible the initial conditions of the experiment.

The second job of this parent process is to retrieve the number of migrations and number of executions of each process during the execution of the 800 steps of the AOCS process. The number of execution gives an insight into the deadline misses even though it is not exactly the same metrics. Linux doesn't have any built-in solution to retrieve each deadline miss because it is not designed as a real-time operating system but knowing the number of executions of each benchmark process allows to know if serious deadline misses happened.

Serious deadline misses are those that cannot be caught up by the scheduler. A deadline miss is considered serious when the step cannot be executed in the next period. All the others deadline misses are non-serious. For example, if a deadline miss occurs but the process can execute the missing code during the next period it isn't a serious deadline miss since the scheduler has been able to execute the missing code. In the case of a lower number of executions than 800 for a particular benchmark process, it means that the scheduler hasn't been able to execute all the code available. At the end of the 800-step execution, the benchmark did not execute all its code. This information is important to understand the stability of the scheduling system. To retrieve this data, each benchmark increments a shared variable each time they end a step. Thus, when the AOCS has exited, the parent process kills all the benchmarks and retrieves the number of execution of each benchmark in the shared area created earlier.

To retrieve the number of migrations for each process, the parent process uses perf. It opens a file descriptor with *perf_event_open* for each process. This file descriptor is connected to a perf event that monitors the number of migrations of the related process. At the end of the execution, the parent process reads the number of migrations from all the file descriptors and writes it to a file.

Each combination of parameters of the table 6.1 is used in one experiment with 800 measures of execution time and performance counters. These measures are then repeated 50 times in order to reduce the volatility of the measures induced by Linux. The number of execution and number of migrations represent one measure for each process and 800 steps. It means that for each cell of the table6.1, there is one measure of the number of execution and migration per process.

6.5.3 Measures impact

Measuring has an impact on the system it monitors. The impacts of using the performance counters and system clock have already been discussed in the previous chapter. However, retrieving the number of execution and the number of migrations also have

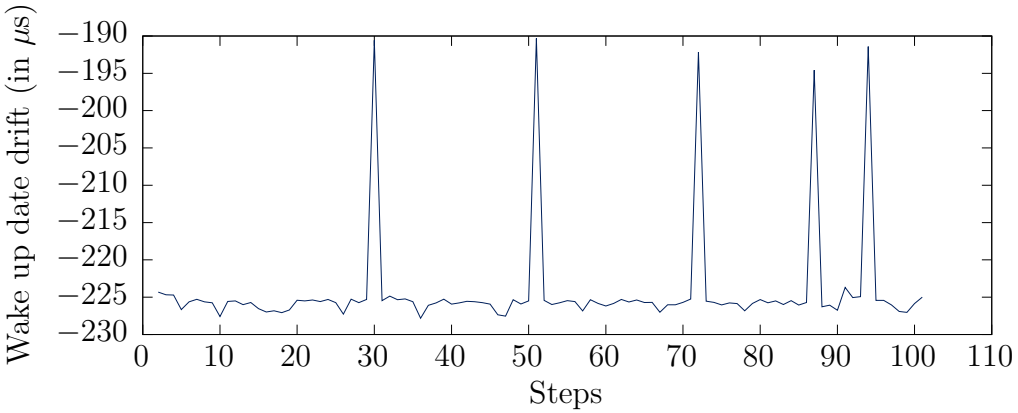

Figure 6.2: Aocs wake up drift with a 10ms period and deadline

an impact on the processes.

The number of execution is a simple counter in a shared memory area. The code surplus is a simple sum and memory access. While the sum is negligible compared to the number of instructions made in one step, the memory access to the shared area has different impacts, especially in a benchmark without any memory accesses such as the benchmark Pi. However, it is only one memory access and every process have memory accesses, which means that adding just one can be neglected. Moreover, all the processes will access the same memory shared area placed in the same line in the shared cache. This means that the impact on the cache is constant.

The impact induced by retrieving the number of migrations depends on perf. Perf is connected to the kernel scheduler code and increments the number of migrations when migration happens. This means that the impact is in the kernel code. As it is the same impact for all the processes and this impact is in the kernel code then the impact is considered to be negligible relatively to other kernel activities.

6.5.4 Impact of removing the time synchronization services

These experiments are realized without the time synchronization services which generated perturbations in the previous experiments (as discussed in section 5.3.2). It has been shown that removing those services has an impact on the scheduler behavior. As a reminder, an inaccuracy of around 2ms was measured in the AOCS process. But this was measured when the AOCS had a period and deadline of 125ms with the SCHED_DEADLINE scheduling policy. The value of this impact (2ms) increase in significance as we reduce the execution period of the process. In fact, this can have a lot more impact than on the previous experiments.

The figure 6.2 shows the wake up drift measured with the new period and deadline

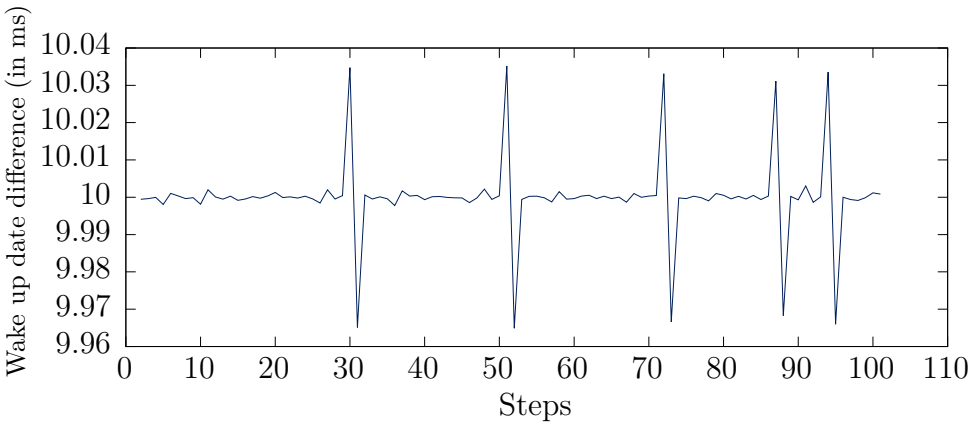

Figure 6.3: Aocs wake up date difference with a 10ms period and deadline

of the AOCS and the figure 6.3 shows the difference in time between the two wake up dates of the AOCS process. The phenomenon that has been discovered in the previous chapter is still present when the period and deadline change. However, the phenomenon has a smaller impact because of the reduced period and is less frequent than in the previous chapter experiment. Finally, this phenomenon will not be a problem for the experiments that will be presented in this chapter because its impact is very small.

6.5.5 Results

There will be 40 000 measurements to analyze and to reduce this number the statistics effect will be used. The first approach to explore the measurements is to use average values. But it is possible to calculate a mean in different ways, a choice has to be made. First, the mean will be calculated using the 800 steps of the same experiment. At the end of this process, the result is 50 tables, such as the one presented in figure 6.1, containing an average in each cell. Then, a new mean is calculated using the 50 values of the same cell. At the end, the result is a table with the same number of cells as the one describing the experiment protocol. To visualize these averaged data, charts will be used. However, 3 dimensions are needed, one for the value of the cell, one for the row of the table, and one for the column of the table.

6.6 Experimental protocol

The experiments presented in this chapter are divided into two groups. The first group will only use the benchmarks pi and the second group will only use the benchmarks loads.

For each group of experiments there will be four different experiments:

- The first experiment aims at providing a reference for comparison (kind of a baseline behavior) with the next ones. Its goal is to show the behavior of the SCHED_DEADLINE scheduling policy when the configuration of the experiment is varying. As the SCHED_DEADLINE policy is a strong candidate for using deadline processes in Linux, this experiment will also serve to provide a performance analysis of the scheduling policy.

- The second experiment uses the SCHED_RR scheduling policy and, like the previous one, this experiment is used to understand the behavior of this policy in the context of the experiment approach presented in the introduction of this chapter. This change of scheduling policy is motivated by the fact that the cgroups mechanism cannot be used with SCHED_DEADLINE. As the SCHED_RR policy is not a deadline real-time policy, the deadline and period are evaluated by the processes at the end of each period. It means that each process knows its theoretical wake-up date and calculates the amount of time it needs to sleep to end the current period. The first wake-up date is calculated by the parent process and shared with all the children through a shared memory area. The periods are created by the user code and depends on the sleep system call. But as for the SCHED_DEADLINE policy it is the kernel that manages the clocks and thus knows when to wake up each process. Even if the sleeping time is defined in the user code, the accuracy of the awakening is preserved.

- In the third experiment, the cgroups are introduced and the goal is to show if they reduce the impact of the attackers on the victim process. All the attackers are placed in the same cgroup while the AOCS victim process has its cgroup. This setup has been chosen to protect the AOCS victim process from the other attackers processes. The real-time configuration of the cgroups is based on the real-time configuration of the processes. The period is equal to 10ms and the runtime is an upper bound of the observed execution time.

- Finally, the fourth experiment introduces an academic patch of the Linux scheduler. This patch allows the use of SCHED_DEADLINE with the cgroups by implementing a two-level scheduling system.

6.7 Experiments with benchmarks pi

6.7.1 SCHED_DEADLINE experiment

Experiment description

In this experiment, the number of benchmarks and the execution time of all benchmarks will vary, as explained in a previous section. The results will be presented in 2 dimension chart with a third dimension represented by curves color. The third dimension will either be the number of benchmarks or the execution time of all benchmarks. The SCHED_DEADLINE policy needs three configuration parameters: runtime, deadline, and period. The deadline and the period are both 10ms as explained in the section 6.4, and these values stay constant with the variations of the experiments. The runtime parameter is difficult to set, and in fact it is impossible to find a value that will work for all the couples of configurations parameters. Such a runtime would be greater than 8ms because there is an experiment with 1 benchmark and a global execution time of 8ms, but 80 benchmarks with 8ms of runtime will exceed the limit of 40ms available on 4 cores in a period of 10ms. However, there is a linear relationship between the number of steps used to calculate pi and the execution time of the benchmark. This relation is described as follows:

$$execution\ time = 35 * nbIterPi$$

. As discussed in the section 6.4, the execution time of the benchmark Pi is considered to be linear, despite the fact that when the number of Pi iterations is low this might not be completely accurate. This would mean that to find the runtime of the benchmark it is possible to multiply the number of Pi iterations by 35. In this experiment, the multiplication factor chosen to get the runtime is 36. This over-approximation is to ensure a positive margin for the scheduler.

The configurations values used are described as follows:

- Number of benchmarks: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 20, 30, 40, 50, 60, 70, 80.
- Global execution time of all benchmarks (in ms): 1, 2, 3, 5, 7, 8, 10, 15, 20, 25, 30.

Note that the global execution time of all benchmarks has to be lower than 40 ms since the hardware executing the experiments has 4 cores and that the period of one step is 10ms. Theoretically, the scheduler has to schedule all the benchmarks plus the AOCS process in a time frame of 10 ms. In practice, this has been set up through barrier and timer. Using a tracer tool, such as LTTng, allows to get an insight on the behavior of the scheduler. By configuring the tool to trace only the context switch, it is possible to find this kind of information while reducing the impacts on the system

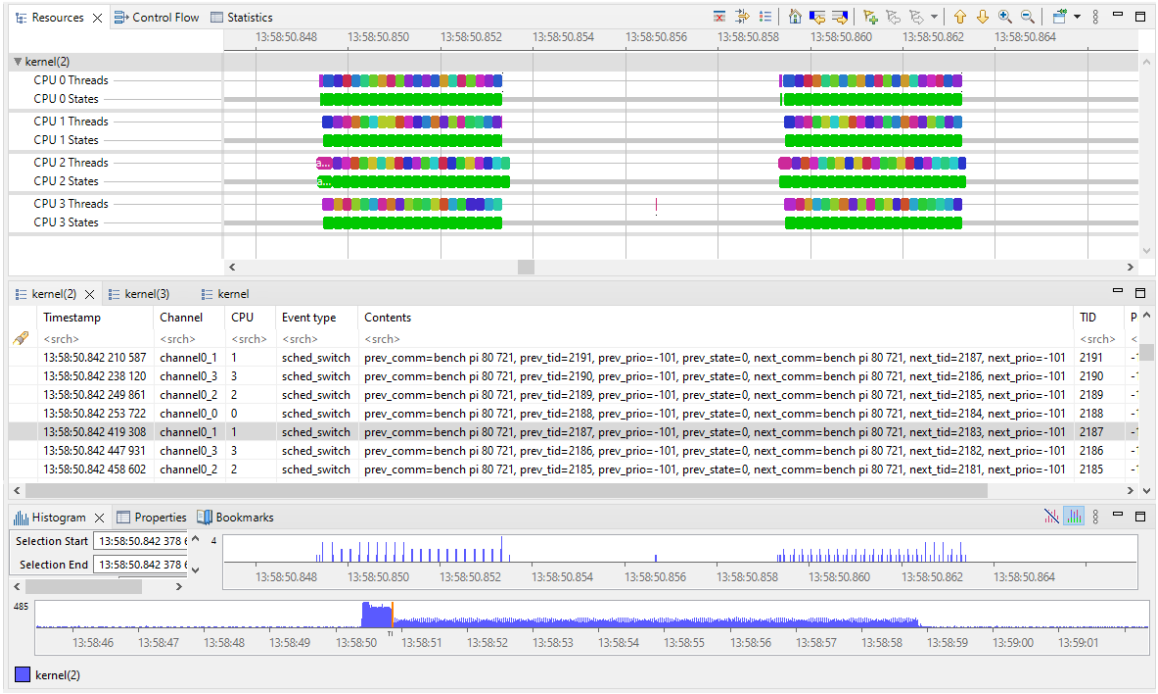

Figure 6.4: Trace of the AOCS and 80 benchmarks pi execution

as much as possible. The results of this trace is presented in the Figure 6.4. It shows two periods of the experiment execution with 80 benchmarks and a global execution time of 20ms.

In Figure 6.4 each colored rectangle represents the execution of one step of a process. There are 162 colored rectangles divided into two blocks. Each block represents a step of the experiment and then should be separated by 10ms which is verified by the trace (in the Figure 6.4). All the processes start approximately at the same time which is the desired behavior. The last thing to verify with this trace is the global execution time of all benchmarks. In this example, it has been configured to 20ms. However, it seems slightly less than that, in fact, the first block of processes should have the same size as the empty space between the two blocks as there is 40ms of CPU time to allocate during each period, and the global execution time of all benchmarks represents 20ms. This difference between the configured global execution time and the measured one can be explained: to measure the execution time of the benchmark pi, its code had to be modified. This modification of the code can impact the execution time because it adds more instructions. In the final experiment, this portion of code measuring the execution time had to be removed, because the functions used to read the system clocks need to access the memory and the benchmark pi doesn't

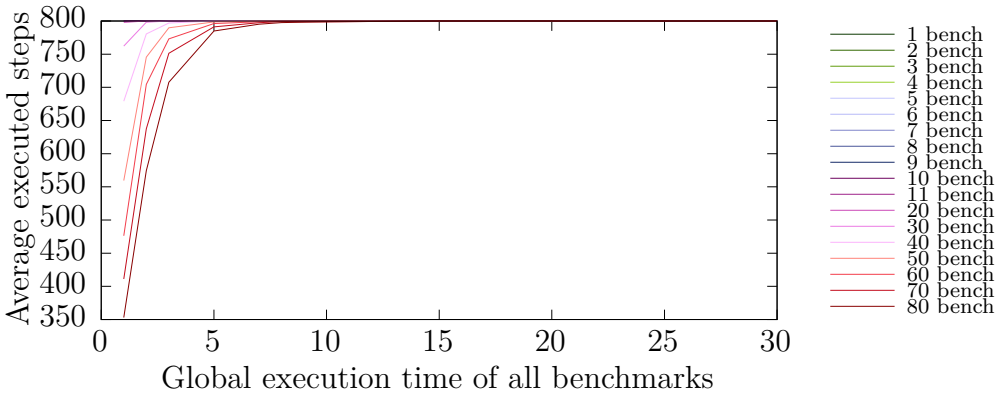

Figure 6.5: AOCS executed steps on average

access memory. Then, the actual code of the benchmark is smaller than the one used to measure its execution time. This approximation doesn't have a huge impact on the results shown below since this experiment hasn't as a goal to find a precise point in the global execution time of all benchmarks curve where the system falls apart.

Deadline misses

The first goal of this experiment is to find out if the SCHED_DEADLINE policy can schedule the processes without any deadline miss. The chart presented in the Figure 6.5 provides the view of the number of executed steps during a full run, it has been obtained by making the average of executed steps for all the processes in each experiment. In other words for each couple of configuration parameters, each process was executed a number of times between 0 and 800. During a run, each process is expected to run 800 times if the scheduler can schedule the process correctly for each activation period. Values lower than 800 indicate potential deadline misses induced by an overload of processes to execute during the 10 ms period.

The Figure 6.5 shows the number of executed steps for every configuration of this experiment. The number of benchmarks is represented by color and the global execution time of all benchmarks is represented by the x-axis.

An interesting observation is that for low values of global execution time, the number of executed steps slightly decreases. This is a completely counter-intuitive observation since the scheduler should have an easier time fitting all the processes executions in the allocated time if the global execution of all processes is lower. However, the opposite is observed. To try to understand this phenomenon, let us focus on the impact of the number of benchmarks in this chart. To see this result more clearly, it is possible to swap the parameter in the chart, meaning that the

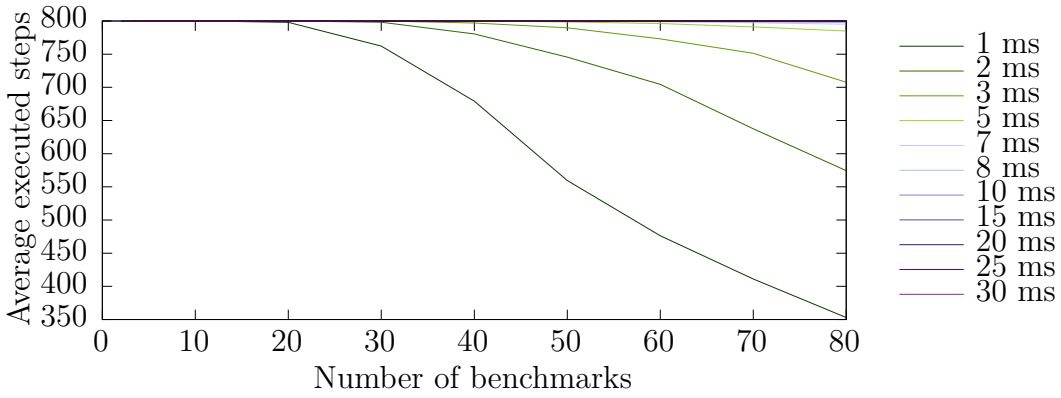

Figure 6.6: AOCS executed steps average

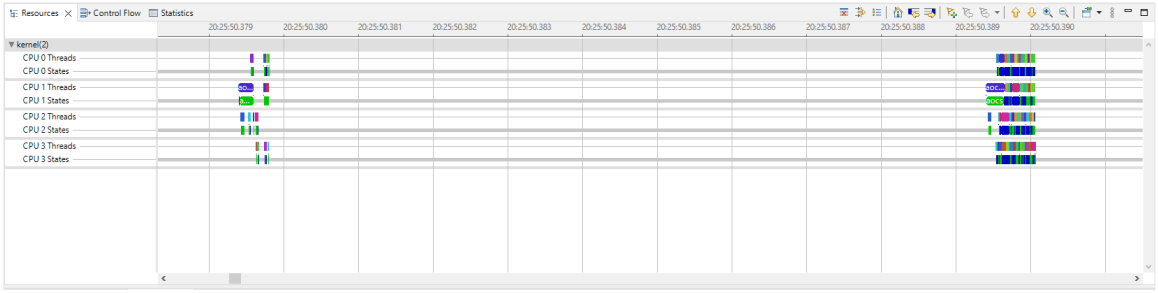

Figure 6.7: Trace of sched switches for the SCHED_DEADLINE experiment with 80 benchmarks and a global execution time of 1 ms

number of benchmarks is now represented on the x-axis and the global execution time of all benchmarks is represented by the color of the curve. The figure 6.6 is showing the result of this new configuration.

Here, the observation is that the number of executed steps drops down only for higher values of the number of benchmarks. Thus, there is a relation between the number of executed steps and both the configuration parameter values. The number of executed steps is always 800 except for low values of global execution time and high values of the number of benchmarks. More benchmarks with less global execution time means that each benchmark has a lower execution time. Therefore, when the execution time of a benchmark is low the scheduler does not schedule all the benchmarks at each period. In fact, the scheduler fails to schedule all the benchmarks in the situation where the CPU is less used. To understand this behavior it is possible to trace the scheduler switches with LTTng. The Figure 6.7 shows the trace result when putting 80 benchmarks with a global execution time of 1 ms. It shows that all 80 benchmarks are not scheduled for each period even though there are

no other activities. In other words, the scheduler could have scheduled those benchmarks but decided not to. As described in the first chapter of this manuscript, the SCHED_DEADLINE policy implements an EDF algorithm augmented with a CBS algorithm. Since there is non-used CPU time shown in the Figure 6.7, the EDF algorithm should schedule all the processes. As this is not the case, the observed behavior must come from the CBS algorithm. In fact, the CBS algorithm works by calculating a new deadline called scheduling deadline and another variable called remaining runtime. The scheduler will throttle the process if the remaining time is less or equal to zero. When the process is throttled it cannot be executed until its scheduling deadline. At this point, the process will see its scheduling deadline increased by the period and its remaining runtime increased by the runtime. This means that the process will have more time to finish its execution, however, this will delay the scheduling entirely. The problem here is not that the process will miss its deadline but that it will consume all of its allocated runtime. This is due to the fact that for small values of Pi calculus iterations the runtime approximation might not be accurate enough. In fact, even though the benchmark pi code is designed to have an execution time that is completely linear with the Pi calculus iterations, there is still a small part of the code that needs to be constant. The lower the Pi calculus iterations the higher the proportion of this constant part in the execution time of a benchmark pi step. This explains why the process uses all its allocated runtime. Finally, it seems that the SCHED_DEADLINE policy can be used in this context if the runtime is greater than the WCET of a process step, otherwise, the deadline cannot be ensured.

Impact on the execution time

Another important aspect of this experiment is the impact on the execution time of the AOCS process. The figure 6.8 shows the execution time of the AOCS process with the global execution time of all benchmarks on the x-axis and the number of benchmarks represented by the color of the curves. The first sixth of the chart shows the behavior that is described in the previous paragraph. As the number of executed steps decreased when the execution time of a benchmark is low, the impact on the AOCS induced by the benchmarks is then reduced. This is why the execution time of the AOCS is lower in this first part of the chart.

Let us focus on the other part of the chart. The first thing to say is that all the curves seem pretty flat. There are still some variations in the middle of the ”80 bench” and ”70 bench” curves but that is not specific to those curves since by redoing this experiment these variations do not appear at the same spot. This is another evidence of the non-reproducibility of Linux. The interesting thing behind the fact that these curves are flat it is that the execution time of the AOCS process is only impacted by the variations of the number of benchmarks and not by the amount of CPU time

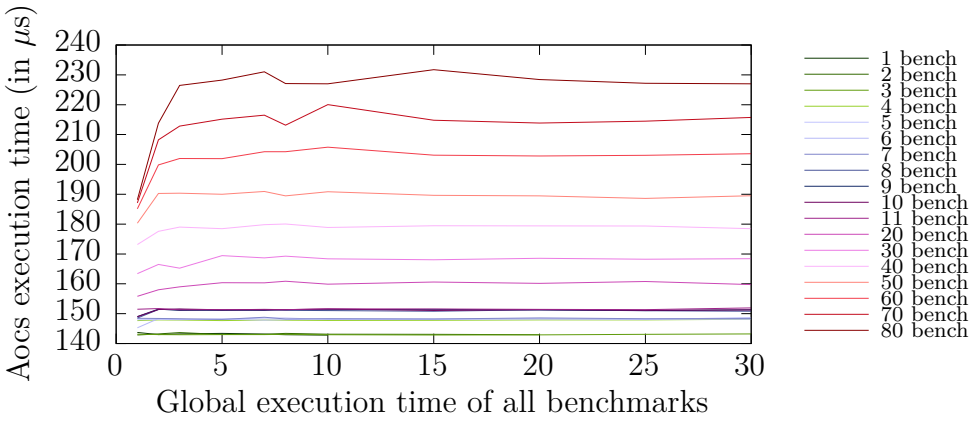

Figure 6.8: AOCS execution time average

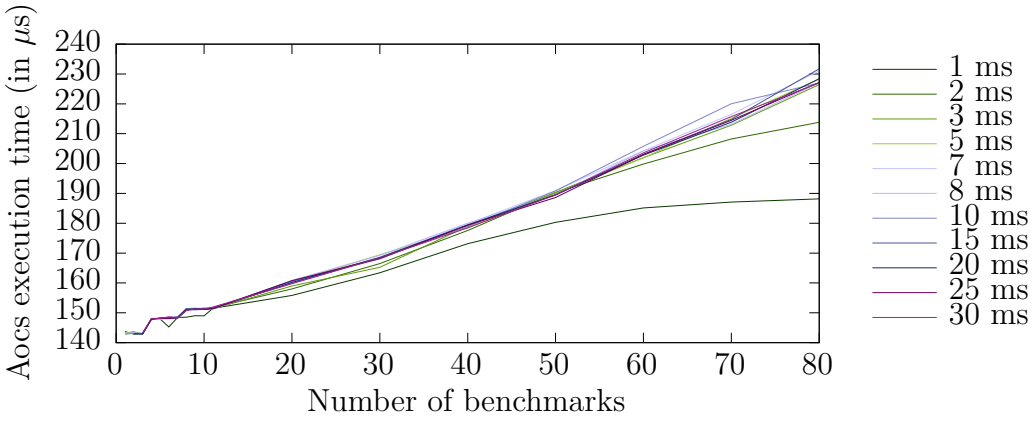

Figure 6.9: AOCS execution time average inverse

they are using. This phenomenon can be observed by switching the x-axis and the curves' color. The results are shown in the figure 6.9. Here, we can see that all the curves (except the ones representing low values of global execution time) have the same shape and values.

The other phenomenon observed in the figure 6.8 is the groups of curves that appear at the bottom of the chart. We can see three different groups of curves, the first one is composed of the "1 bench", "2 bench" and "3 bench" curves, the second one is composed of the 4 following curves, and the third one with again the 4 next curves. By considering that the AOCS process is part of all the processes executing on the Linux system, the groups are then defined by the following equation:

$$number\_of\_processes = number\_of\_benchmarks + 1$$

$$group = number\_of\_processes//4 + 1$$

where // gives the quotient in the division of the two numbers. Then, two sets of processes are in the same group if the result of this calculus is the same. Clearly, this is related to the number of cores present on the CPU. The execution time is then defined by the following:

$$exec\_time = exec\_time_0 + (group - 1) * f(number\_of\_processes)$$

where $exec\_time_0$ is the execution time of the AOCS executed alone and f is a function that depends on the number of processes, the shape of the AOCS, and the hardware platform used. This formula can be verified by looking at the Figure 6.9, indeed we can see stepped lines at the beginning of the chart. The rest of the chart would have the same shape if the number of benchmarks used in the experiments had been sampled with an interval of four instead of 10. Thus, the shape of the curves in the Figure 6.9 after the "11 benchmarks" point is not completely accurate.

Sources of impacts

Now that the impact on the AOCS process has been described, the question is to understand where this impact comes from. As seen in the chapter 4 there are two types of interference. The first type of interference called "instantaneous interference" is not the cause of the impact described. In fact, as the scheduler tries to schedule the processes as soon as possible they are often executed while other processes are running on the adjacent cores. Thus, having 2 or 4 processes running at the same time changes the number of requests to the bus at a particular instant. This is the main cause of "instantaneous interference" as described in the chapter 4. However, if these interferences had an impact on the AOCS process, there should be a difference between the curves that belong to the same group of curves, but this isn't the observation. All the curves from the same group have the same exact shape. As a consequence, the "instantaneous interference" is not responsible for the observed impact.

The other type of interference described in the chapter 4 is the "delayed interference". These interferences occur most of the time when using caches. In the chapter 5 it was clear that the L2 shared cache had a big impact on the execution time of a process on the hardware platform used. This is why the L2 cache refill has also been measured in this experiment. The results are presented in the Figure 6.10, the number of benchmarks is first represented by the color of the curves while the global execution time of all benchmarks is on the x-axis as for the other charts .

The same phenomenon for the lower values of global execution time are present in this chart. Let us consider only the part of the curves after the 5ms point. In the Figure 6.10 it is hard to distinguish the different groups of curves, but it seems that these groups had disappeared. By looking at the Figure 6.11 where the global

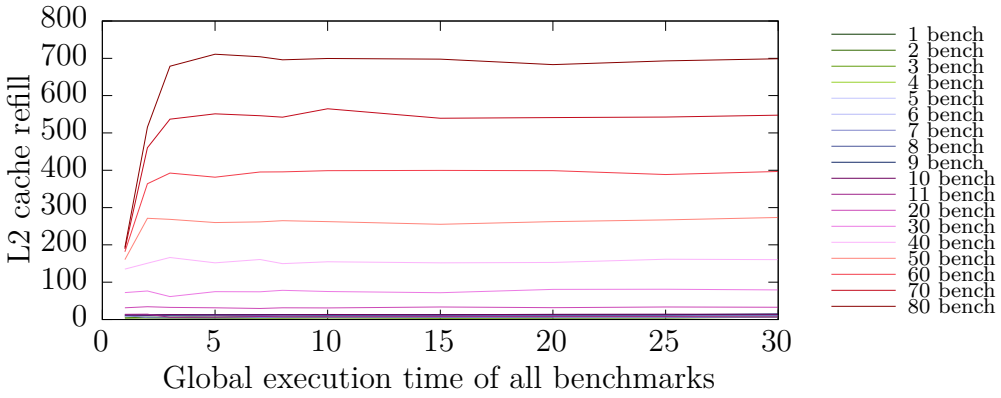

Figure 6.10: AOCS L2 refill average

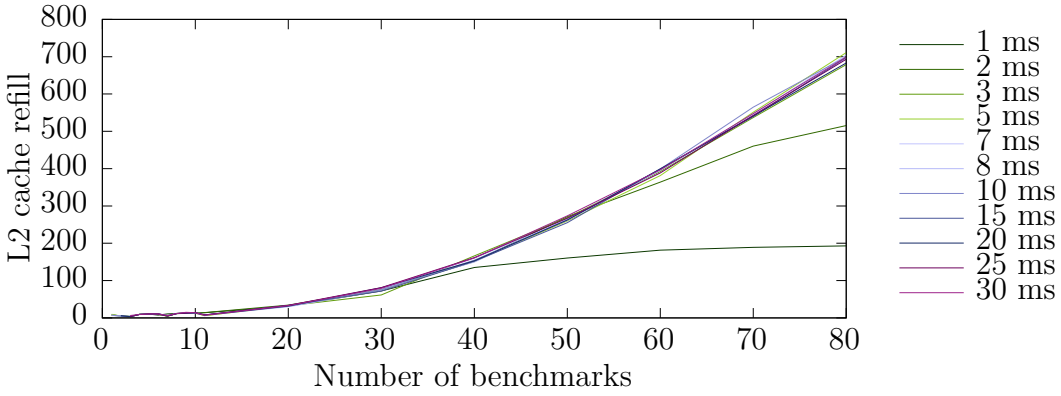

Figure 6.11: AOCS l2 refill average

execution time is represented by the color of the curves and the number of benchmarks on the x-axis we can see that the global shape of the curves is completely different from the ones in the Figure 6.9. The stepped lines have disappeared and the curves have a quadratic shape. This means that the impact seen in the execution time of the AOCS process does not only come from the interference on the L2 shared cache. To find the other interference channel that impacts the AOCS process, it is possible to look at the other performance counters. The Figure 6.12 shows the number of accesses to the L1 data cache which is not shared by the four cores. This chart shows the same groups of curves as the one in Figure 6.8. The interference induced by the L1 cache only occur when several processes are executed on the same core. Thus, the impact increases with the number of processes by core, which is what creates these groups of 4 curves.

The impact on the execution time of the AOCS process induced by the execution

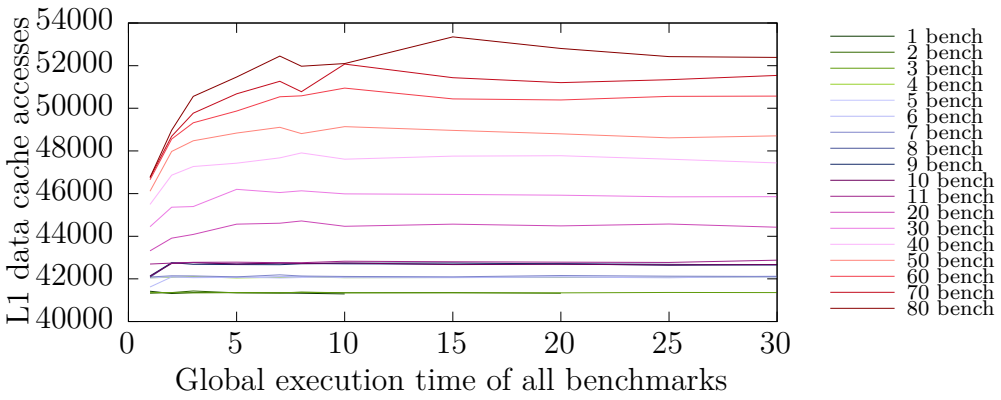

Figure 6.12: AOCS L1 data cache accesses

of the benchmarks Pi is mostly induced by the memory cache system both L1 and L2.

Core migrations

Finally, the last important metric to measure is the number of migrations because it helps to understand if the system is stable. These measures have been retrieved by taking the number of migrations that happened for each process during the 800 executed steps. Then, a mean over all the processes of the same experiment iteration was calculated. For each couple of configuration parameters of the experiment, there were 50 mean values computed which were aggregated by taking the average of these 50 points. The results shown in the charts presented in this sub-section came from this method. The Figure 6.13 shows the number of migrations with the number of benchmarks represented by the color of the curves and the global execution time of all benchmarks on the x-axis. The curves presented in this chart are divided into two parts, the first part for values of the global execution time below 15 ms is quite chaotic while the second part is more stable.

As for the other charts presented in this sub section, it is possible to swap the representation of the number of benchmarks and the global execution time of all benchmarks. This is presented in the Figure 6.14. However, this chart presents the same chaotic behavior as the one above.

Let us dig more into the data to understand where this chaos comes from. The Figure 6.15 shows the number of migrations for the AOCS and all the benchmark processes for each batch of the experiment with 10 benchmarks and a global execution time of 10ms. In this chart, most of the points are below 100 migrations, which means that most of the time a process is migrated less than 100 times during the execution

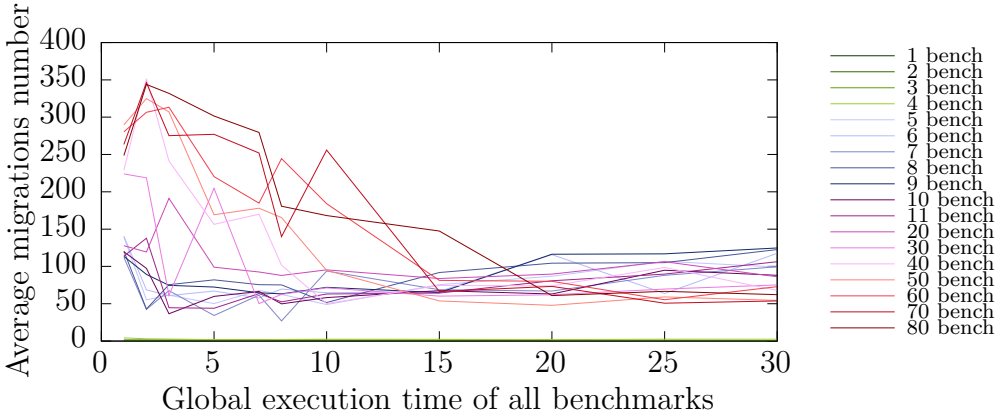

Figure 6.13: AOCS number of migrations on average

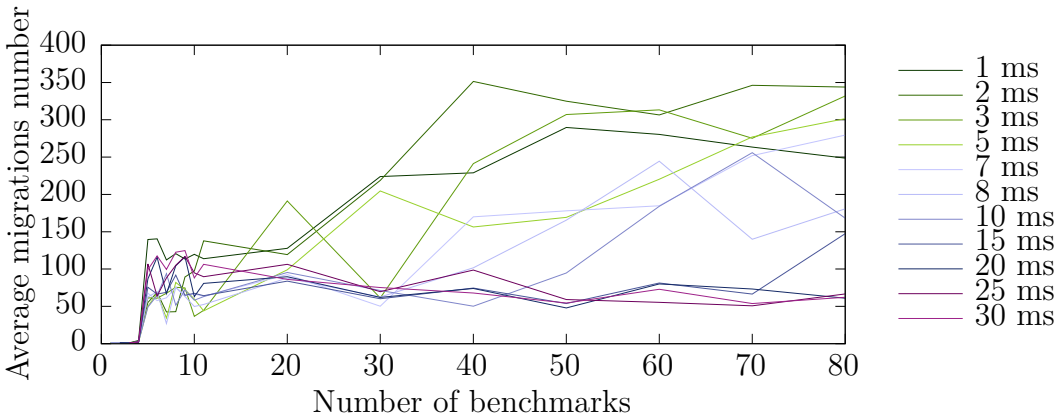

Figure 6.14: AOCS number of migrations on average

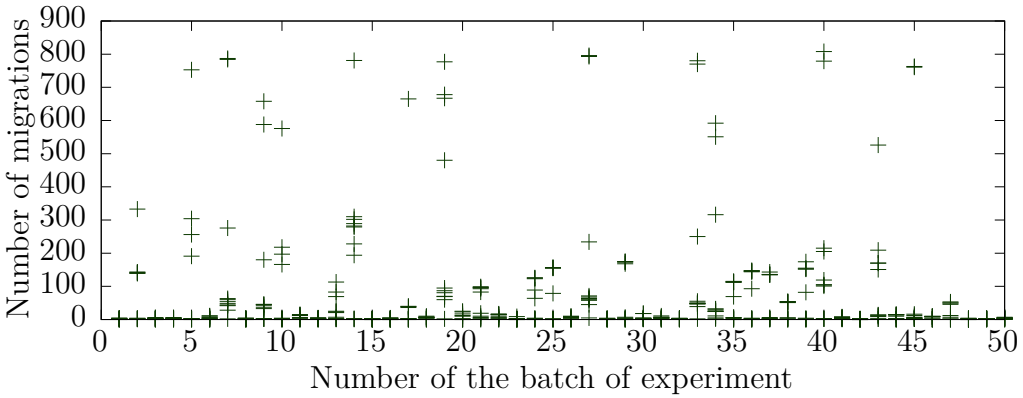

Figure 6.15: Number of migrations for aocs and benchmarks for each batch of the experiment with 10 benchmarks and a global execution time of 10ms

of 800 steps. In fact, there are 83 points that are above 100 which represents 16% of all the points in this figure. Moreover, there are only 23 points above 400 migrations which are 4% of all the points. However, this example shows that there is a lot of variability in the number of migrations during an experiment. This is, again, not a very intuitive result especially since in this configuration, 10 benchmarks for 10ms of global execution time, the system is not overloaded and the execution time of one benchmark is high enough to not fall in the previously seen effect. The hypothesis here is that the scheduler tries to schedule as soon as possible all the processes that are ready. As shown in the Figure 6.4 all the processes are grouped and executed one after the other without idle time between. In this scenario, if a process that was executed on a specific core is not ready when the core has finished its job, the scheduler has to find another process to execute on the core even if it needs to migrate one from another core.

The scheduler might take into account the characteristic of the process to make a decision regarding how to schedule it. This is why, in the last part of this section, a focus will be made on the number of migrations of the AOCS process. First, the Figure 6.16 shows the number of migrations of the AOCS process in the experiment with 10 benchmarks and 10ms of global execution time. It is interesting to observe that the number of migrations is quite low compared to what the Figure 6.15 shows. Thus, the AOCS process seems to not be considered the same way as the benchmarks processes by the scheduler. To validate this hypothesis, it is possible to plot the number of migrations of the AOCS process for all the experiments averaged through all the batches of the same experiment. The result is shown in the Figure 6.17. On average, the number of migrations of the AOCS process is contained between 0 and 4. Even though there are a lot of migrations, the AOCS process seems to avoid

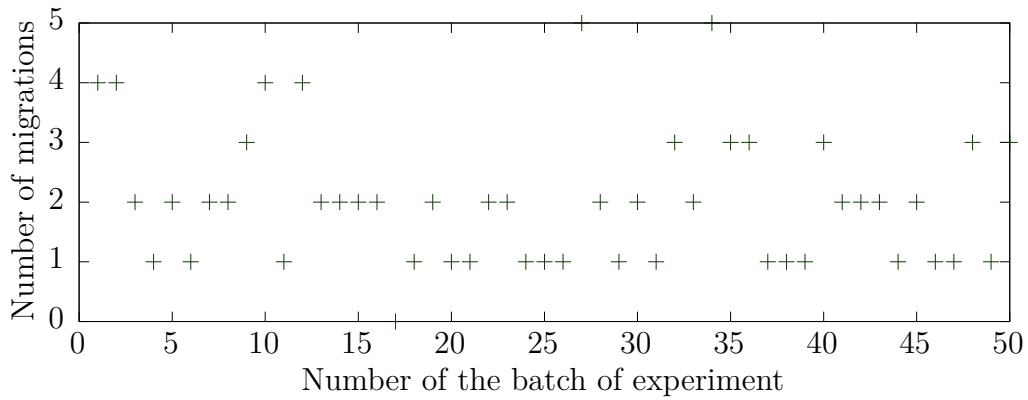

Figure 6.16: Number of migrations of the AOCS process for each batch of the experiment with 10 benchmarks and a global execution time of 10ms

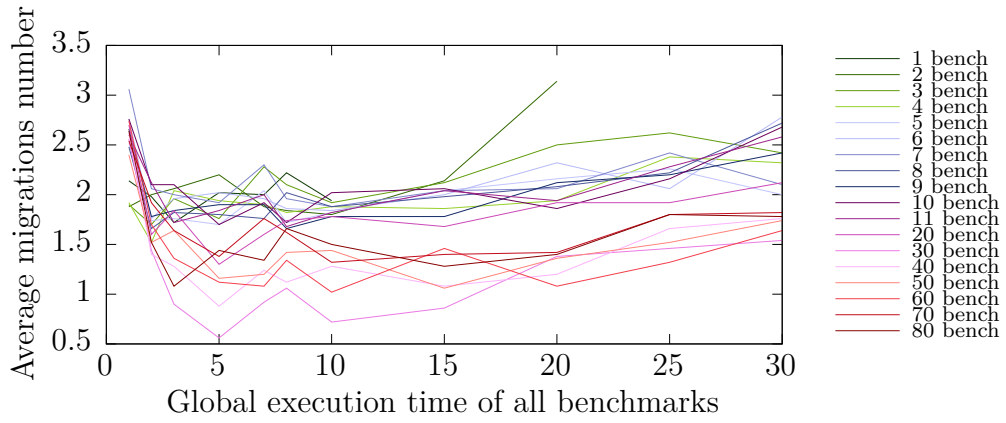

Figure 6.17: Number of migrations of the AOCS process

being selected for migration consistently. Note that this was not the goal of this experiment, the AOCS here hasn't any special configuration but this is a good result for the SCHED_DEADLINE policy since it shows that the system is more predictable. In the following experiments, the cgroups will be implemented to try to reduce these impacts.

Conclusion

To conclude this first experiment, the SCHED_DEADLINE policy is capable of scheduling real-time applications while respecting deadlines and periods. However, two remarks are important to explain.

The first one is that when a benchmark has a greater execution time than its

configured runtime the scheduler won't schedule it properly. This results in deadline misses for the benchmark processes.

The second one is the number of core migrations that occurred during the experiments. During this experiment, the benchmark processes have been migrated a lot. This behavior makes the system unstable and less predictable. However, the AOCS process did not migrate so much and this is a good point for the SCHED_DEADLINE policy even if there is no explanation on why the SCHED_DEADLINE policy chose the AOCS process to not migrate much.

Most of the impacts induced by the execution of the benchmarks come from the shared L2 cache and L1 data cache. This policy allows a simple configuration to run deadline-based processes, however, certain behavior can make the system unstable. The following sections will focus on other scheduling policies and real-time mechanisms and compare the results with this one

6.7.2 SCHED_RR experiment

Experiment description

As for the experiment presented in section 6.7.1, the number of benchmarks and the global execution time of all benchmarks will vary. The variations of these parameters will be the same as for the last experiment (section 6.7.1) with benchmarks Pi. However, even though the execution time of the benchmarks still follows the same equation (see section 6.7.1), there is no runtime to configure for the SCHED_RR policy. In fact, at the end of a benchmark step, the benchmark code computes the date of the next wake-up and the duration of its sleep. This small change modifies the behavior of the benchmarks since their code is changed. The modification is considered negligible because the added code does not depend on the number of iterations of Pi. It means that the overhead is constant during the experiment. A different overhead has to be considered when a comparison will be made between the different experiments.

As for the last benchmarks pi experiment, it is interesting to see if the scheduler behaves as the theoretical scheme of this experiment predicted. In order to answer this interrogation, it is possible to trace the scheduling switches. The Figure 6.18 shows the results of a trace for the SCHED_RR experiment with 80 benchmarks and a global execution time of 20ms. The beginning of the processes is more precise, i.e the processes start all at the same time while for the SCHED_DEADLINE experiment there is a little gap between the start of the blocks of processes as shown in the Figure 6.4.

The other properties of the trace are globally the same as the SCHED_DEADLINE ones. We can see two blocks of processes representing the 80 benchmarks scheduled

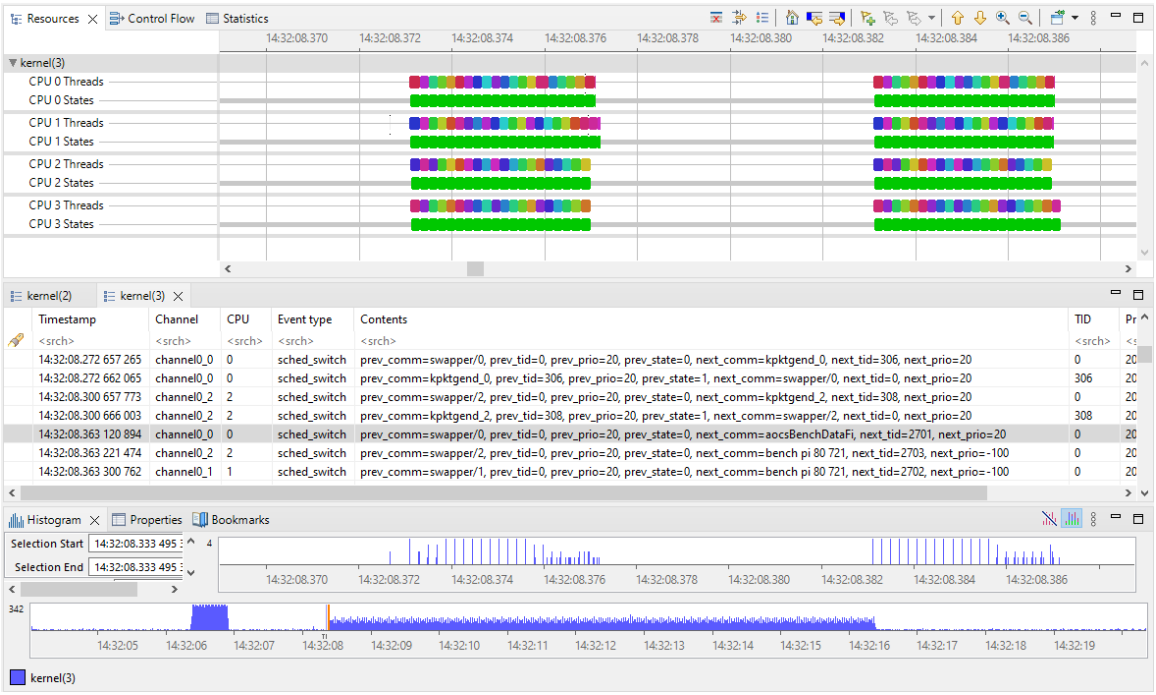

Figure 6.18: Trace of scheduling switch with the SCHED_RR policy in the experiment with 80 benchmarks and a global execution time of 20ms

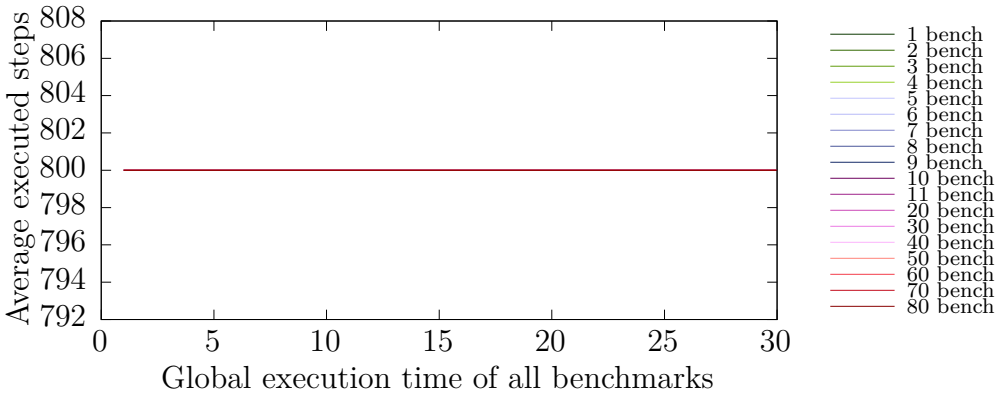

Figure 6.19: AOCS executed steps on average

for two steps. Also, the execution time of all benchmarks is still lower than expected and all the benchmarks are scheduled. The processes are all scheduled one after the other as intended in the theoretical plan. Finally, it seems that the processes are scheduled in the exact same order each time which was not the case for the previous experiment. This observation, however, would need to be confirmed from a larger sample of results.

Deadline misses

Now that the experimental setup has been described and the scheduling is verified, let us concentrate on the deadline misses. The Figure 6.19 shows the number of average executed steps for all the processes. As a reminder, for each couple of configuration parameters (numbers of benchmarks and global execution time), the number of executed steps is retrieved for each process (benchmarks + AOCS), then these values are average which results in one mean for each experiment. As these experiments are repeated 50 times another mean is calculated with the 50 values coming from the same experiment. Every curve of the chart is constant with a value of 800, which means that every process has executed all its steps.

The phenomenon observed during the first experiment has disappeared. In fact, even for low values of the execution time of one benchmark the scheduler still schedules the processes. This is a pretty big difference because the scheduler uses as much CPU time as it can (to schedule the application processes). Compared to SCHED_DEADLINE, the stability of the system has increased. This is likely a side effect of the selection of a simpler scheduling policy which is less dependent on arbitrary configuration values. Clearly, the simplification of the scheduling policy induced this result: there are no runtime, deadline, and period parameters in this policy, and

the scheduler does not have to make complex computations to know how and when it can schedule which process. More precisely, it is the runtime parameter that impacts the most the scheduling behavior. Deadline and period parameters are only here to compute priorities and end dates, thus they do not impact the system's behavior. The runtime parameter needs to make assumptions with only a priori information on the processes.

In this particular case of the SCHED_RR policy, the relation between executed steps and deadline misses might not be trivial. In fact, while it was clear that with the SCHED_DEADLINE policy the process could be throttled if it consumed more than expected, with the SCHED_RR policy there is no such mechanism. In other words, a process can miss a deadline and still execute all its steps. The process computes itself its next wake-up date and thus the amount of time it needs to sleep, but if the next wake-up date is in the past it does not wait and continues its execution. This is made possible by using the *clock_nanosleep* system call. This system call can be used with absolute time on a specific clock which means that the process will be sleeping until the clock reaches the desired date. As explained in [87], the system call returns instantly if the given date is in the past. Thus it is possible that a process misses its deadline but still manages to execute its next step right after the end of the previous one. The number of executed steps is a good metric anyway since it allows us to see if a process did not have time to execute all that it was intended to.

Impacts on the execution time

The Figure 6.20 shows the execution time of the AOCS for the different values of the number of benchmarks and the global execution time of all benchmarks. The observation made on the Figure 6.19 continues here as the first part of the chart is similar to the last part unlike the first part of the Figure 6.20. However, it seems that the execution time is a little higher for low values of global execution time and high values of the number of benchmarks. However, this impact will not be investigated further in this thesis.

The group of 4 curves observed in the previous experiment (in the section 6.7.1), have disappeared, and we can observe only one group that includes all the configurations up to 10 benchmarks (it is not explicit that the 10 benchmarks point is the threshold value because it is in fact very progressive). All the other curves are pretty distinct from each other. This observation is confirmed by the chart presented in the Figure 6.21. The stepped lines have completely disappeared and the global shape of the curve is completely quadratic. Moreover, the values of the curves are lower than in the SCHED_DEADLINE experiment.

Finally, at this point in the exploration of the SCHED_RR results, it is possible to make a comparison between the first two experiments presented in the section 6.7.1

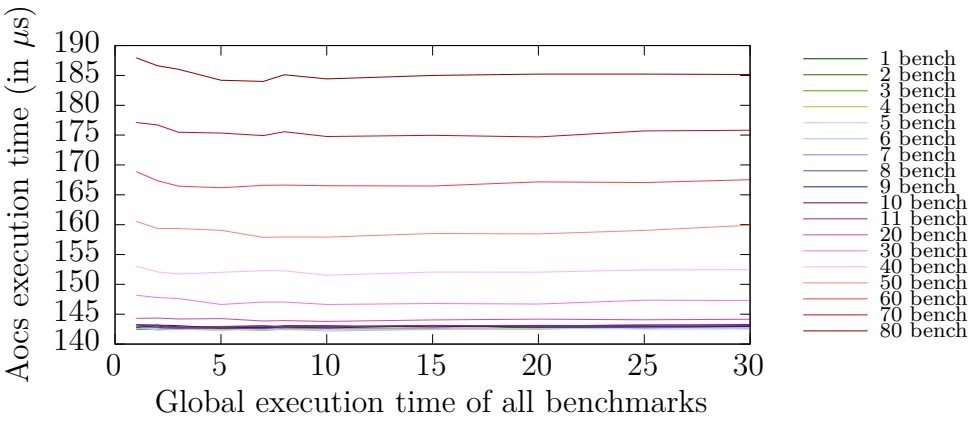

Figure 6.20: AOCS execution time on average

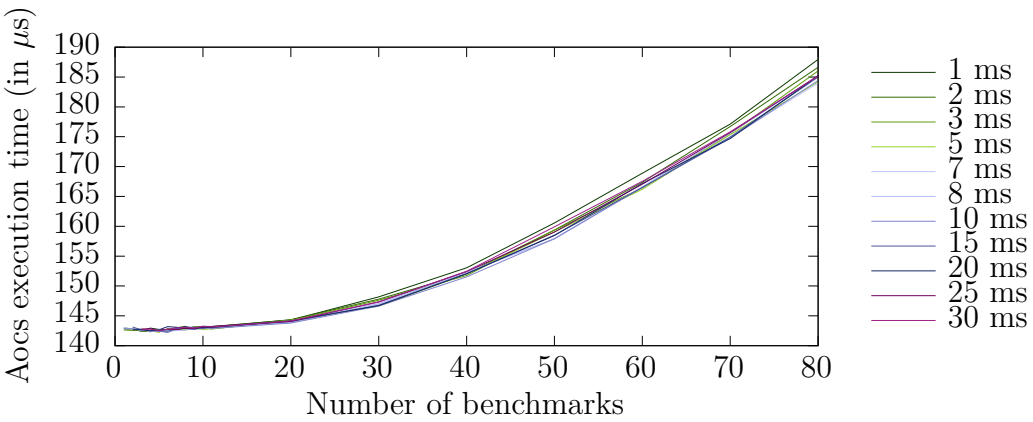

Figure 6.21: AOCS execution time on average

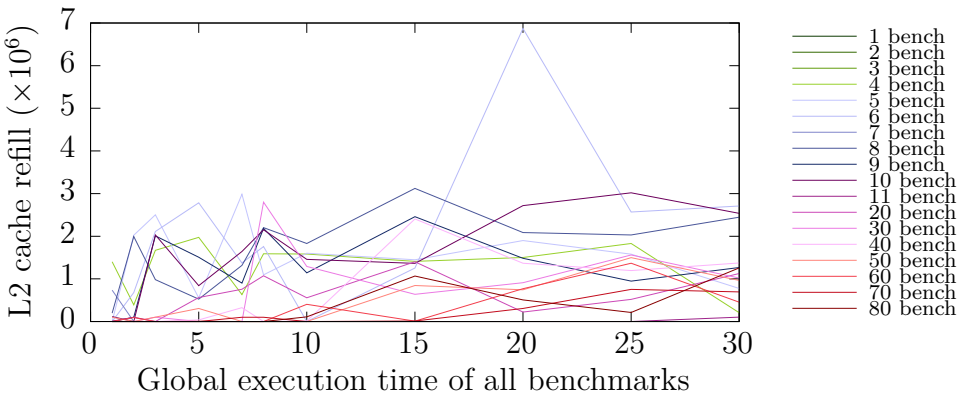

Figure 6.22: AOCS L2 refill on average

and 6.7.2. These experiments have the exact same pattern and protocol. The only difference is the scheduling policy that is used in Linux. These experiments have been designed for comparison of the scheduling behavior. What has been learned so far is that the system behavior is less dependent on configuration parameters variation and that the absence of arbitrary scheduling policy configurations parameters generates a more efficient scheduling. Also, the AOCS process execution time is less impacted by the activities of the benchmarks running in the meantime. The rest of this section will focus on finding the cause of these differences by looking at other measurements made during this experiment.

Source of impacts

Even though the impact of the execution of all the benchmark processes on the AOCS execution time is reduced compared to the SCHED_DEADLINE experiment, there is still an impact. As has been observed a lot in this thesis (chapter 5), the shared L2 cache is a major source of interference in the Xilinx Zynq Ultrascale+. The Figure 6.22 shows the number of L2 cache refill on average when the number of benchmarks and the execution time of all benchmarks vary. Clearly, the curves do not have the same shape as the ones presented in the previous experiment. The number of L2 cache refills is very high (around 10^6) which is an inconsistent value. Knowing that the execution time of an AOCS step is less than 200μs and considering that all memory accesses resulted in an L2 cache refill, it would mean that the response time of one memory access takes around 2ns. Even with the best RAM technology available today it is unrealistic.

This results from an erroneous measurement and to understand this phenomenon, once again a trace tool is used. By tracking down the exact moment where an

incoherent high value of L2 cache refill occurred, the trace shows the source of this phenomenon. The trace shows that the AOCS process is migrated during one step execution and this is what provokes these high values of L2 cache refill. It happens because the computation of L2 cache refills that is executed during a step includes an arithmetic operation on values obtained from performance counters - when the process migrates, performance counters (before and after execution) are read from different cores and are not coherent. When the process migrates it reads another performance counter register and the difference becomes incoherent. Incidentally, it indicates that the AOCS process was not migrated during the execution of a step in the last experiment since these high values did not appear.

The chart presented in the Figure 6.22 cannot be interpreted as is. As it is almost impossible to re-do the experiment until these phenomena do not happen, it has been chosen to remove the incoherent point from the data pool. The impact of removing these points on the final result is statistically insignificant. Over the 7 680 000 measurements used to get this chart, 2535 have a number of L2 cache refill that is greater than 10 000, which represents 0.03% of all the measurements. Thus, it is negligible to remove those points from the data pool. Note that the migration of the AOCS process during its execution only affects the performance counters that are not shared between cores. The other measurements such as the number of migrations, the execution time, or the number of executed steps are not impacted by this behavior.

The Figure 6.23 shows the new curves obtained after removing the non-coherent point as explained in the previous paragraph. Two observations can be made from these curves. First, the curves are constant and do not depend on the global execution time of all benchmarks. Second, the number of cache refills is slightly higher than in the last experiment. As the only thing that changed between the first experiment and this one is the scheduling, it means that the scheduling policy has a non-null impact.

The Figure 6.24 shows the same data but the x-axis and the curves' color have been switched. These curves are similar to the execution time curves presented in the Figure 6.21. Thus, the impact on the AOCS process comes directly from the contention in the L2 shared cache. However, the L1 cache does not have the same effect here as observed in the SCHED_DEADLINE experiment. The next sub-section addresses and analyzes this behavioral difference.

Core migrations

Finally, let us take a look at the core migrations during this experiment. The Figure 6.25 shows the number of migrations for all the processes depending on the number of benchmarks and the global execution time of all benchmarks. First of all, the number of migrations is much lower than in the first experiment (in the section 6.7.1). In fact,

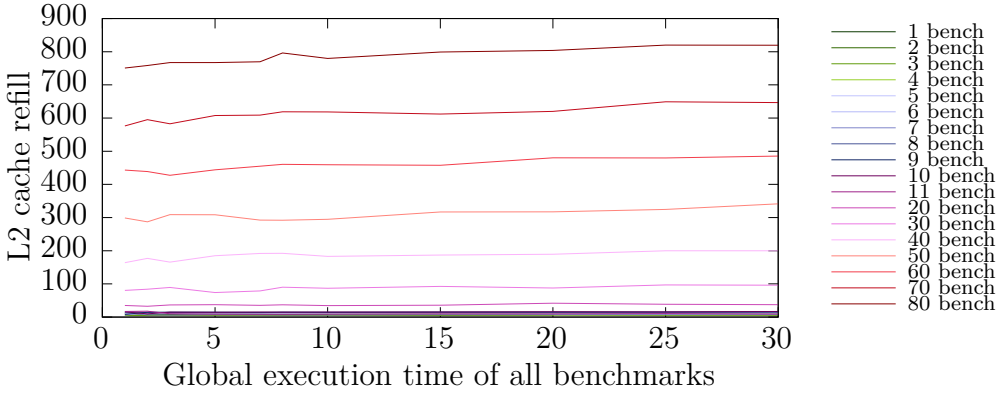

Figure 6.23: AOCS l2 refill after removing all the non-coherent points

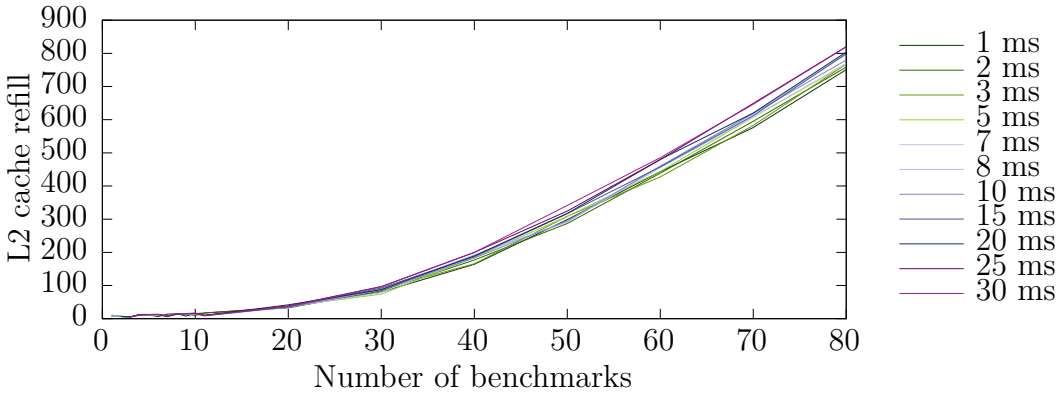

Figure 6.24: AOCS l2 refill inverse after removing all the non-coherent points

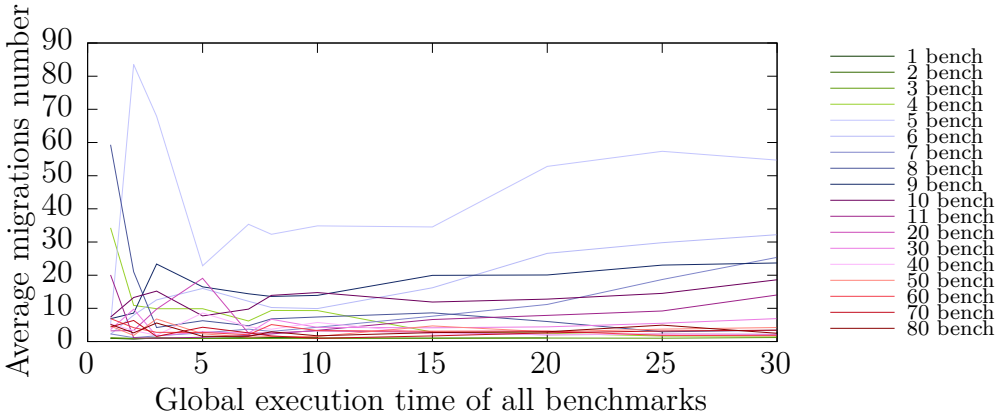

Figure 6.25: AOCS number of migrations on average

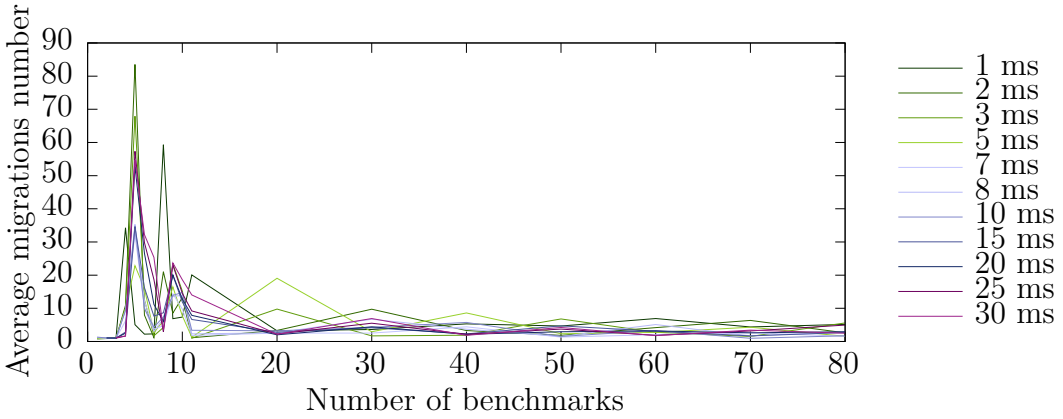

Figure 6.26: AOCS number of migrations on average

the majority of the curves are under 20 migrations. The system has better stability with this new scheduling policy. However, one single curve, the 5-benchmark curve, is still significantly above the others as confirmed by the Figure 6.26.

The scheduler seems to have more problems scheduling the system when there are 5 benchmarks. The ”9 benchmarks” curve is also globally higher than the others. More generally, the Figure 6.26 shows that the scheduler becomes more stable after 10 benchmarks before there are some higher values of the number of migrations. Up to 4 processes (i.e 3 benchmarks + AOCS) it is quite simple for the scheduler because each process can have a single core for itself. For some reason, the scheduler struggles to find more stability between 4 and 10 benchmarks regardless of the global execution time of all benchmarks.

A lower number of migrations means that the processes have a tendency to be executed on the same core which results in less refill in the L1 caches. This is why in

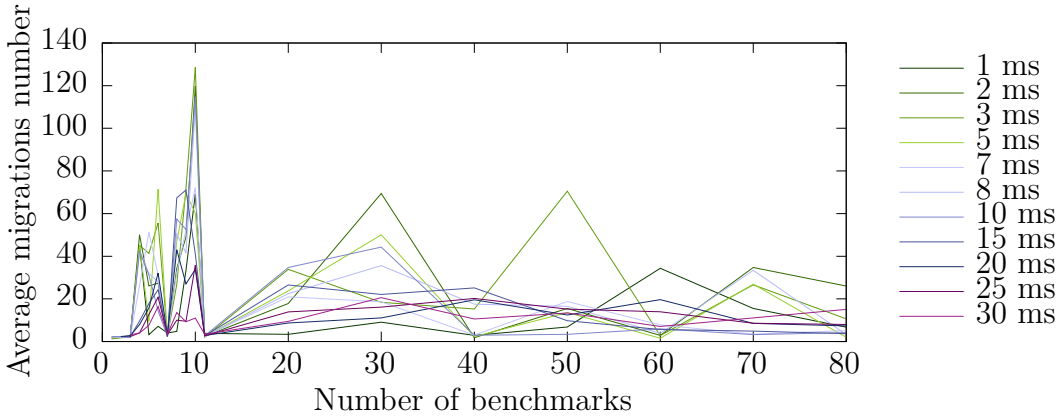

Figure 6.27: Number of migrations of the AOCS process

this experiment the impact on the execution time of the AOCS process does not have the same groups of curve behavior. In fact, the impacts from the different caches have the same shape and the stepped lines have disappeared.

Finally, the Figure 6.27 shows the number of migrations of the AOCS process. The AOCS process migrates more than the other processes on average especially when there are not too much benchmarks executing in parallel.

Conclusion

To conclude this experiment, the SCHED_RR policy is able to schedule deadline-based applications with the help of the *clock_nanosleep* system call. The impact on the execution time is almost the same as for the SCHED_DEADLINE experiment but the phenomenon that occurred with the low value of global execution time has disappeared. Also, the system is more stable regarding the core migrations which results in a reduction of the impact of the L1 data cache. However, the number of migrations of the AOCS process is greater than the other processes which are worse than the SCHED_DEADLINE experiment.

Using this scheduling policy makes the system more predictable because the processes are less likely to migrate and the impact on the execution time is reduced.

6.7.3 SCHED_RR + cgroups experiment

Experiment description

The experimentation setup is the same as the one described in the section 6.7.2. The cgroup configuration does not impact the executed code. The only change is that the parent process configures the cgroups by writing the real-time parameters in the

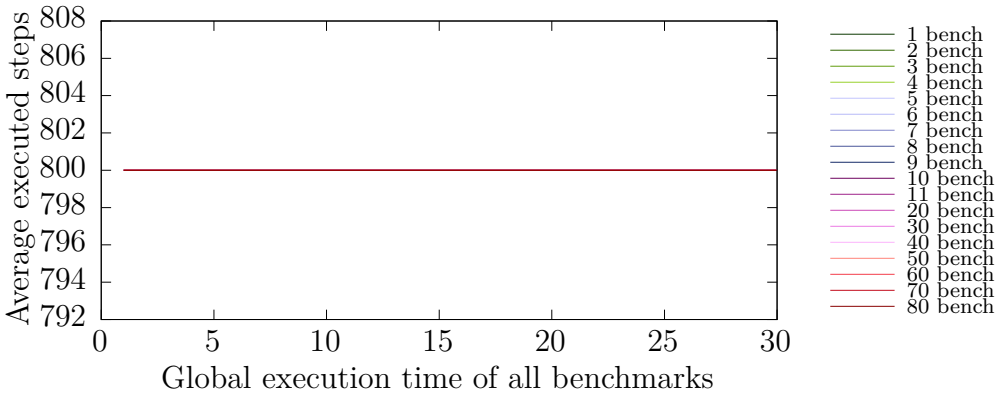

Figure 6.28: AOCS executed steps on average

configuration files. The rest of the code is unchanged. The creation and initialization of the cgroups are made from the bash terminal before launching the parent process.

In this experiment, there is one croup with all the benchmarks and another with the AOCS process. The cgroups are configured to only limit the CPU time. The period/deadline of both cgroups, represented by the parameter 'sched_rt_period_us" is set to 10ms. The runtime represented by the parameter 'sched_rt_runtime_us" is the amount of CPU time that can be used by the processes inside the cgroup. This parameter is set to 1 ms for the AOCS cgroup and 8,5 ms for the benchmarks cgroup. This means that on each core the AOCS can use 1 ms of CPU during the period of 10ms and all the benchmarks can use 8,5ms of CPU during the same period. It represents respectively 10% and 85% of CPU time on one core and the last 5% remain free to let some CPU time to the system processes.

Deadline misses

As for the others experiments, the number of deadline misses is approximated by counting the number of steps executed and subtracting from the baseline (800). The Figure 6.28 shows the number of executed steps of all the processes executed during this experiment. It seems that adding the cgroups did not change anything compared to the last experiment. The scheduler still uses the same round-robin policy. As the curves are all constant with a value of 800 it is safe to say that all the steps have been executed and that deadline misses are unlikely to happen. This means that adding the cgroups configuration does not modify the behavior of the Linux scheduler with the SCHED_RR policy.

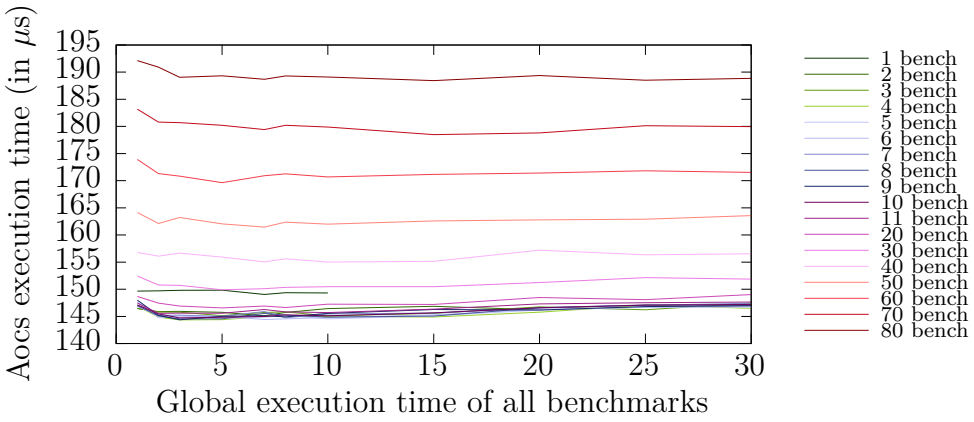

Figure 6.29: AOCS execution time on average

Impacts on the execution time

The impact on the execution time of the AOCS process follows approximately the same pattern as the previous experiment. The Figure 6.29 shows the curves of the AOCS execution time with the global execution time of all benchmarks represented by the x-axis and the number of benchmarks by the color of the curves.

First of all, the decreasing phenomenon for low values of global execution time discovered in the experiment described in the section 6.7.2 is still present. Thus the global execution time of all benchmarks has an impact but only when it is low enough. This phenomenon may appear because of the SCHED_RR policy. It is clear that it is not a statistical error but as the impact of this phenomenon is quite low it has been decided not to investigate to find the source of this phenomenon.

Second of all, there are still no groups of curves but the ”1 benchmark” curve is higher than all the curves below 30 benchmarks. This phenomenon is clearer in the Figure 6.30. We can observe a straight decreasing line at the beginning of the chart. The cgroups do not have the same behavior when there is only one benchmark executed in parallel.

Finally, the cgroups added a constant overhead in the execution time of the AOCS process. All the curves are around 2% higher than in the previous experiment. The next sub-section will focus on the source of these phenomena and the relation with hardware interference.

Source of impacts

As for the others experiments, the first hardware counter that needs to be checked is the L2 cache refill one. The value returned by this performance counter was incoherently high due to the migration of processes during the execution of one step. As

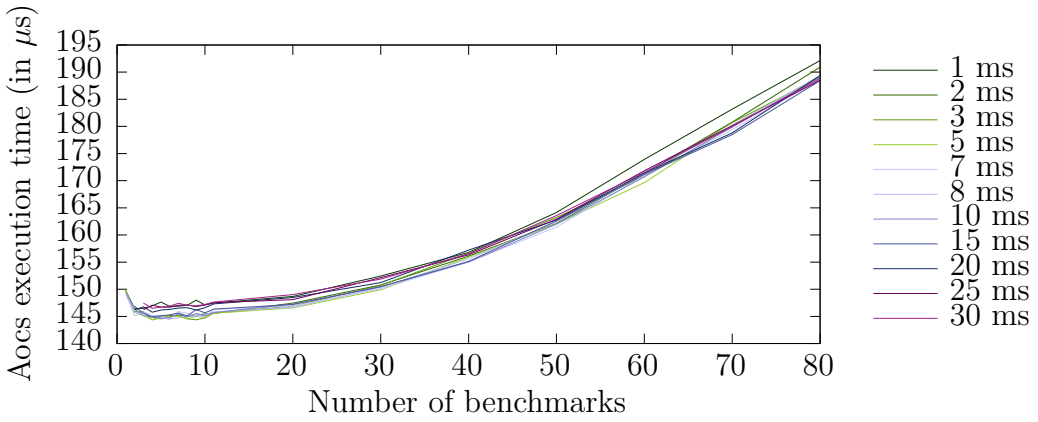

Figure 6.30: AOCS execution time on average

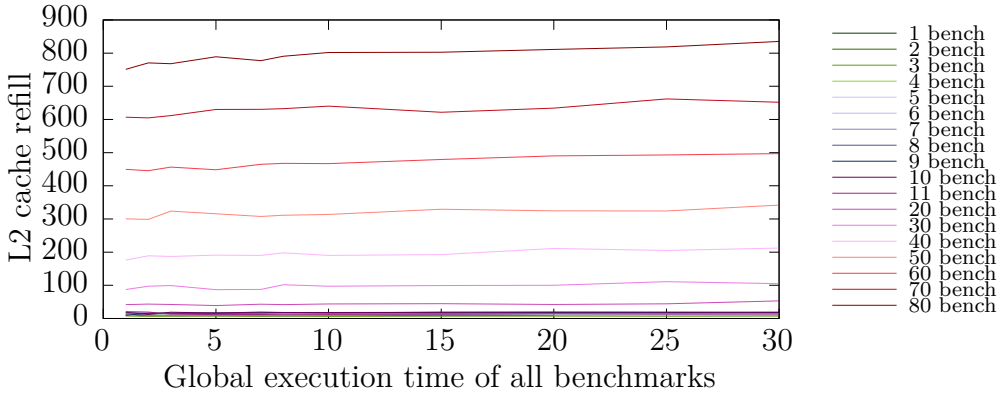

Figure 6.31: AOCS L2 refill on average

for the previous experiment presented in the section 6.7.2, the values of the measurements where the number of L2 cache refills is greater than 10 000 have been removed from the data pool. There are 2404 over 7 680 000 measurements that are above the 10 000 thresholds, which represents 0.03% of all the measurements. This is nearly the same percentage as the previous experiment.

The Figure 6.31 shows the number of L2 cache refills after the non-coherent values have been removed. The curves look exactly like the ones in the previous experiment. It means that the differences in impacts seen in the AOCS execution time curves do not come from the L2 cache interference. Thus, this leaves the L1 cache interferences.

The Figure 6.32 shows the number of L1 accesses made by the AOCS process. In this chart, two major observations can be made. The first one is that the ”1 benchmark” curve is higher than the group of curves. The second observation is that

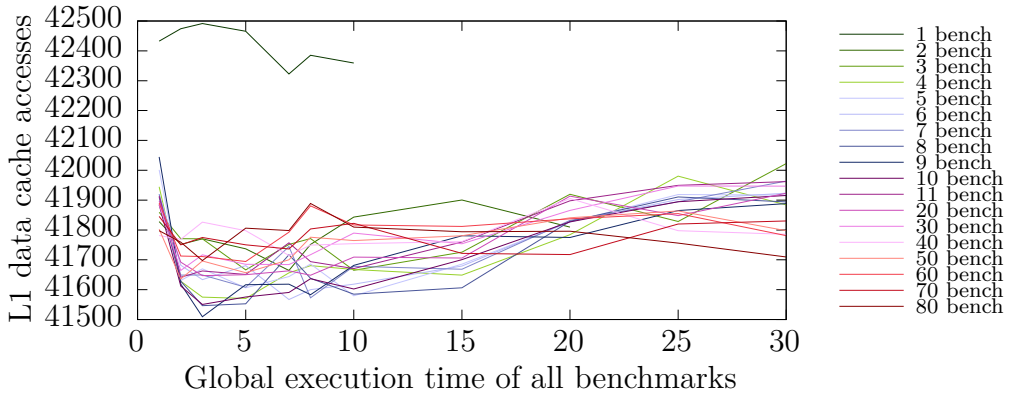

Figure 6.32: AOCS L1 accesses on average

the start of all the other curves is flatter.

The first observation explains why the impact on the execution time of the AOCS is different for 1 benchmark. A hypothesis could be that the scheduler puts both processes on the same core when the cgroups only have one process. The second observation could explain the phenomenon occurring at the start of the execution time curves. The number of L1 accesses is higher when the global execution time is lower which can induce the decreasing part of the execution time curves.

Core migrations

Finally, the last observation will focus on the core migrations. The Figure 6.33 shows the number of migrations of the AOCS process. Compared to the previous experiment presented in the section 6.7.2, the average migration number has decreased a little. However, the ”5 benchmarks” curve is still higher than the other curves as shown in the Figure 6.34.

The Figure 6.35 shows the number of migrations of the AOCS process. The maximum of these curves is lower than in the previous experiment but the average value of all curves is the same. Globally, there are not a lot of differences between this experiment and the previous one in terms of migrations.

Conclusion

Adding the cgroups to the system did not change the system's behavior significantly. Globally, the impacts are the same as without the cgroups. There is however a special behavior happening when only 1 benchmark is running in parallel with the AOCS process. The migration number is also a little bit lower than without the cgroups.

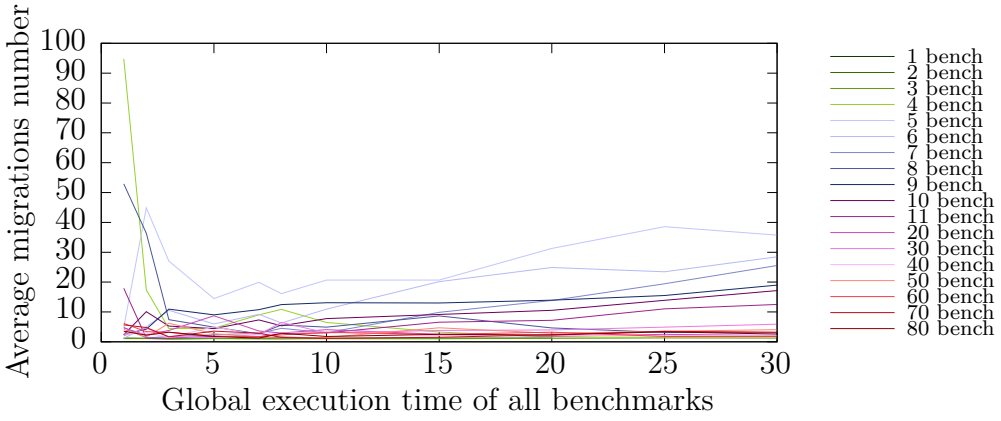

Figure 6.33: number of migrations on average

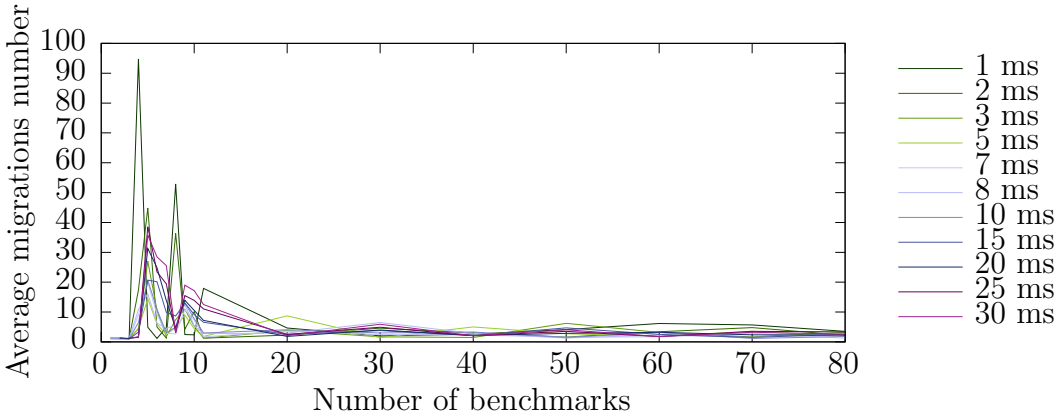

Figure 6.34: number of migrations on average

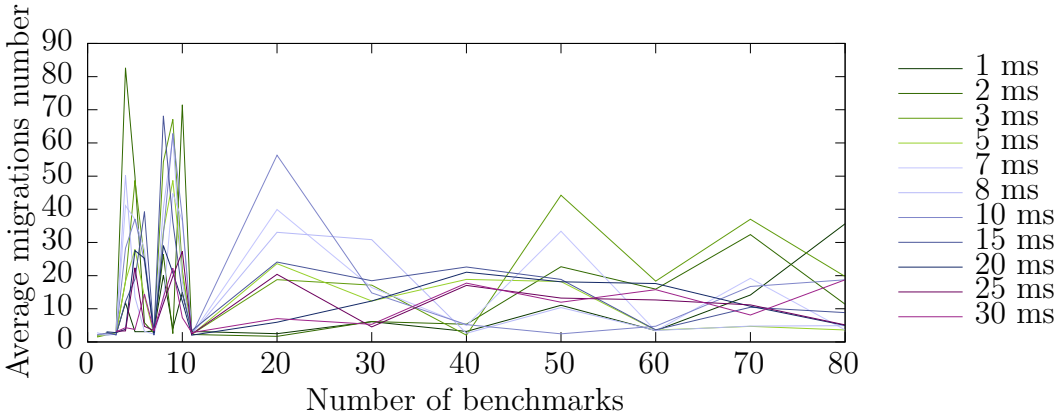

Figure 6.35: AOCS number of migrations

Adding the cgroups does not have a huge impact compared to the previous experiment and the stability of the system is the same as in the experiment with the SCHED_RR policy.

6.7.4 Academic patch experiment (SCHED_DEADLINE + cgroups)

The patch

Abeni *et al.* developed a Linux scheduler patch, presented in [76]. This patch aims at creating a two-level scheduler. The first level uses SCHED_DEADLINE and schedules the cgroups. The second layer uses SCHED_RR and schedules the processes. This patch allows using the SCHED_DEADLINE policy with the cgroups mechanism which is normally impossible.

Experiment description

This experiment uses the patch described in the previous sub-section. This allows the scheduler to operate on two different levels. The first one is to schedule the cgroups and it uses the SCHED_DEADLINE policy while the other one schedule the processes inside each cgroup with the SCHED_RR policy. The goal of this experiment is to find if using this patch decreases the interference between the victim and attackers processes.

The experiment protocol and cgroups configuration are exactly the same as the previous experiment of section 6.7.3. There is still one cgroup for the AOCS and another one for all the benchmarks. The only thing that changes is the kernel itself which is modified by the patch made by Luca Abeni *et al.* This patch has been chosen because it allows using the SCHED_DEADLINE policy with cgroups which is impossible in the classic version of the Linux kernel.

Deadline misses

The Figure 6.36 shows the number of average executed steps by all the processes. The processes execute all their 800 steps which means that deadline misses are unlikely to happen. The new scheduling policy added by the scheduler patch does not modify the impact on deadline misses.

Impacts on the execution time

The impact on the execution time is shown in the Figure 6.37. This chart is similar to the second experiment with only SCHED_RR. However, the curves are a little higher.

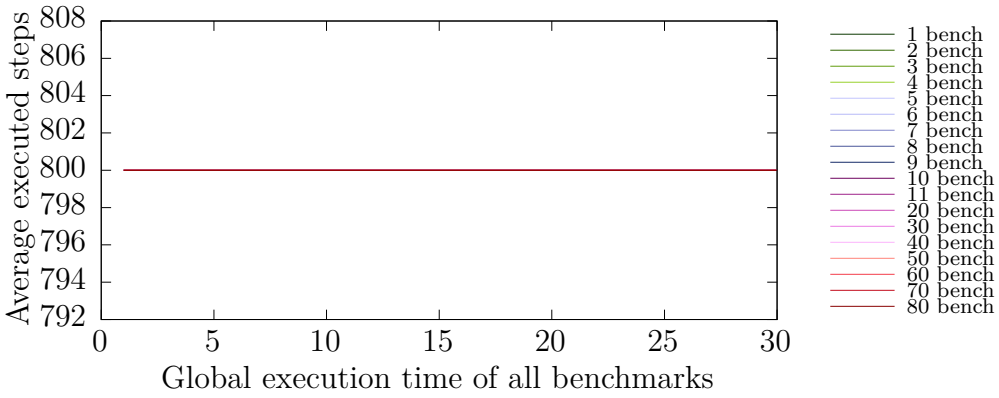

Figure 6.36: AOCS executed steps on average

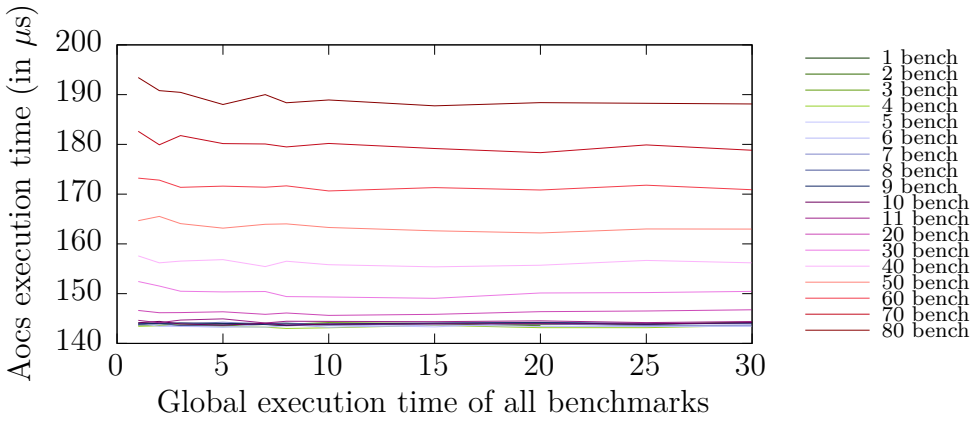

Figure 6.37: AOCS execution time on average

The phenomenon that happened with the ”1 benchmark” curve, seen in the previous experiment, has disappeared.

The Figure 6.38 shows the execution time of the AOCS process with the number of benchmarks on the x-axis and the global execution time of all benchmarks represented by the color of the curves. Globally, the shape of the curves is still the same as in the two previous experiments. These three experiments show the same impact on the execution time of the AOCS process.

Source of the impacts

As for the other three experiments (section 6.7.1, 6.7.2 and 6.7.3), the first source of impact is the L2 shared cache. The Figure 6.39 shows the l2 refill of the AOCS process. It can be seen that this time there are no incoherent points, meaning that

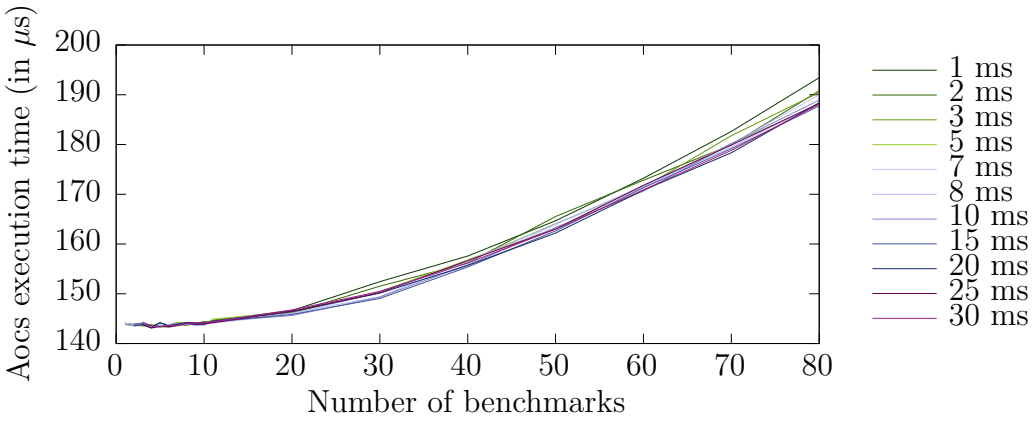

Figure 6.38: AOCS execution time on average

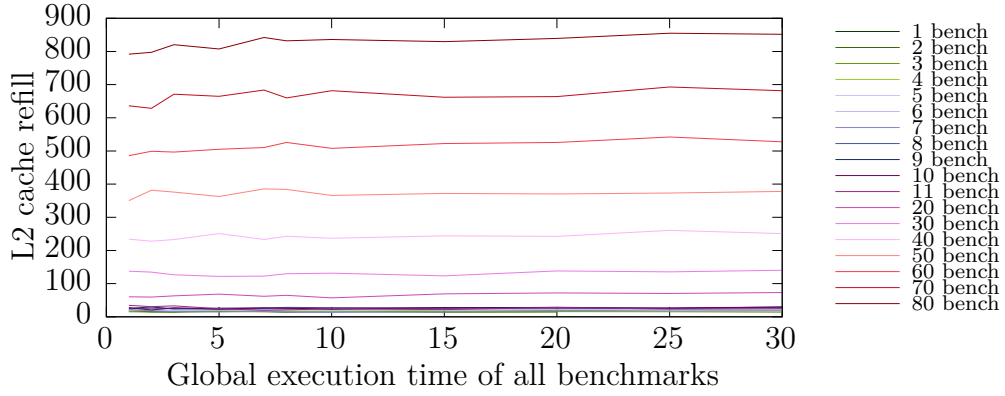

Figure 6.39: AOCS L2 refill on average

there are no migrations of the AOCS process during the execution of one step. As for the execution time chart, the curves look like the other l2 refill charts presented in the two previous experiments but are a little higher. This difference between this experiment and the two previous ones can explain the difference in the execution time curves. As the L2 refill curves are a little higher, the execution time is a little higher too.

Core migrations

As indicated in the previous paragraph, the AOCS process does not migrate during the execution of one step in this experiment. It means that even though all the previous curves presented in this section look like the curves of the previous experiments, it might not be the case for the core migrations curves.

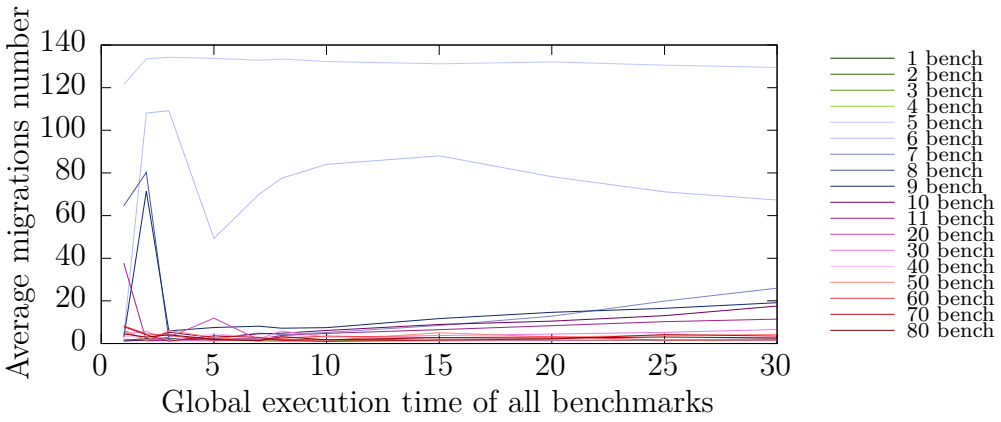

Figure 6.40: AOCS number of migrations on average

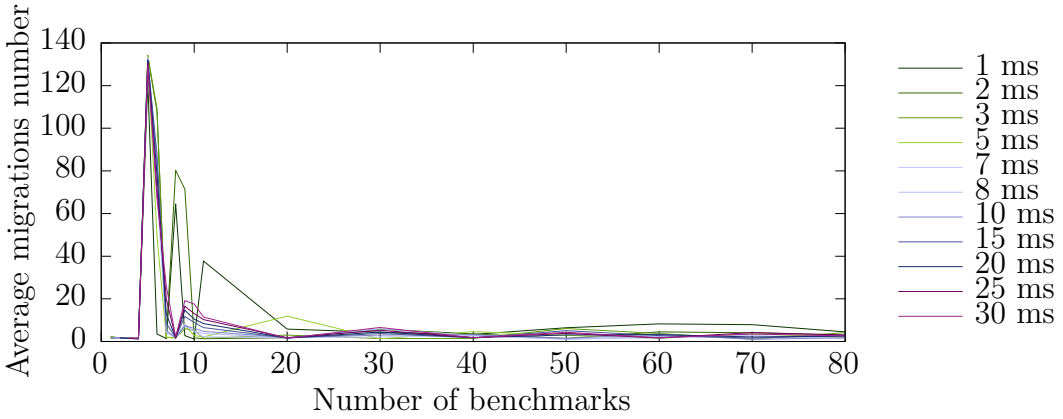

Figure 6.41: AOCS number of migrations on average

The Figure 6.40 shows the number of migrations on average for all the processes. Two observations can be made regarding these curves. First, the ”5 benchmarks” and ”6 benchmarks” curves are higher than the other curves but also higher than the corresponding curves in the chart of the two previous experiments presented in the section 6.7.2 and 6.7.3. Second, the rest of the curves are lower than the corresponding curves of the two previous experiments.

This second observation is more visible in the Figure 6.41. Adding the scheduler patch makes the system much more stable in terms of core migrations. But it also accentuates the phenomenon with the 5 and 6 benchmarks curves.

Finally, the Figure 6.42 shows the number of migrations for the AOCS process only. The curves are between 2.5 and 5 migrations which are very low compared to the other experiments. The phenomenon for the 5 and 6 benchmarks curves has disappeared and all the curves have the same values. The scheduler patch protects

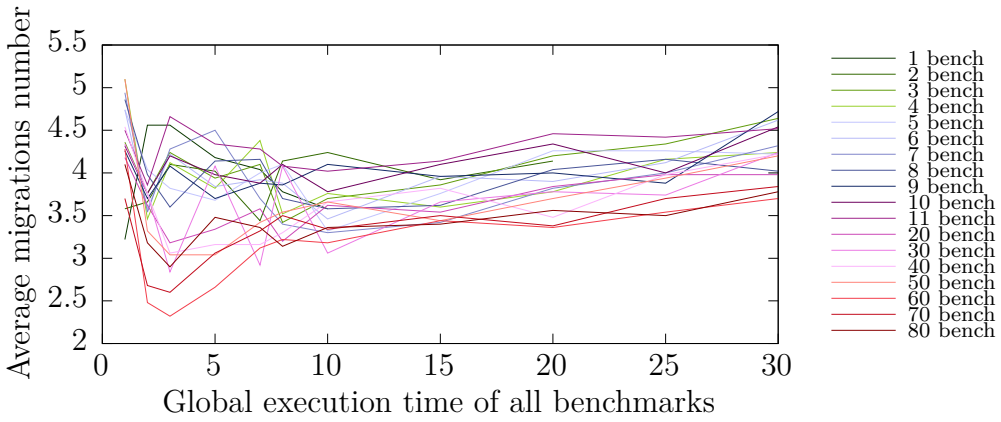

Figure 6.42: AOCS number of migrations

the AOCS process from core migrations and thus reduces the impact of the scheduler on the AOCS process.

Conclusion

To conclude this experiment, the scheduler patch made by Luca Abeni *et al.* provides great improvements. While the impact on the execution time is significantly the same, the number of migrations is much lower. Especially, the number of migrations of the AOCS process is below 5 for 800 execution which is really low. It seems that the patch combines the advantages of both the SCHED_DEADLINE policy and the SCHED_RR policy while using the cgroups mechanism to limit CPU time usage.

6.7.5 Conclusion on the benchmarks Pi experiments

To conclude the benchmarks pi experiments showed that both the SCHED_DEADLINE and SCHED_RR policies are capable of handling real-time processes. In the case of SCHED_DEADLINE, there is however a phenomenon happening when the configured runtime of the scheduling policy is lower than the actual execution time of the process. The configuration of this scheduling policy is thus more complex and might induce deadline misses for certain processes.

The impact on the execution time is almost the same between the two different scheduling policies. This difference is constant and thus does not depend on the varying parameters of the experiments. The code executed by the processes is slightly different depending on which scheduling policy is used. Moreover, the number of core migrations is completely different between in both cases and is lower with the SCHED_RR policy.

Adding the cgroups configuration did not have a big impact on the results previously discussed. However, the patch made by Luca Abeni *et al.* combined the advantages of both scheduling policies. The number of migrations is reduced both for the benchmarks and the aocs process.

6.8 Experiments with benchmarks loads

6.8.1 SCHED_DEADLINE experiment

Experiment description

In this experiment, the variant parameters in the system configuration are the number of benchmark processes and the total number of load accesses performed by all the benchmark processes combined. The attackers used in this experiment are the benchmarks load presented in the section 6.4. These benchmarks are configured to make a specific amount of memory accesses in order to fill the L2 shared cache. The experiment follows the same protocol as the one presented above.

The scheduler uses the SCHED_DEADLINE policy with a 10ms period and deadline for each process. The runtime parameter is set to 0.1ms which is an upper bound of the observed execution time of a benchmark load making 8192 memory accesses. Compared to the experiment with benchmarks pi, the runtime parameters given to the SCHED_DEADLINE policy do not change with the variation of the number of memory accesses made by the benchmarks. This is because the execution time of the benchmarks loads is not a parameter that will be varying in the experiment. The goal is to configure the scheduling policy so that it can schedule all the processes correctly. However, it is true that the variations in the number of memory accesses will induce a variation in the execution time but it is a consequence of the experiment parameter variation. As a load access is necessarily using CPU power, there is no need to isolate the CPU consumption. Therefore, the quantitative results depends on the platform performances.

The configuration values that will be used for this experiment are described as follows:

- Number of benchmarks: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 20, 30, 40, 50, 60, 70, 80
- Number of loads: 256, 512, 1024, 2048, 3072, 4096, 6144, 8192

The number of benchmarks is the same as previously. The number of loads is set up to progressively fill up the L2 cache. It means that beyond a certain number of loads the cache will be filled and each new access will not create interference with another process. This is why the maximum numbers of loads in this experiments is set at 8192. The limitation is that it is not possible to control where the memory access

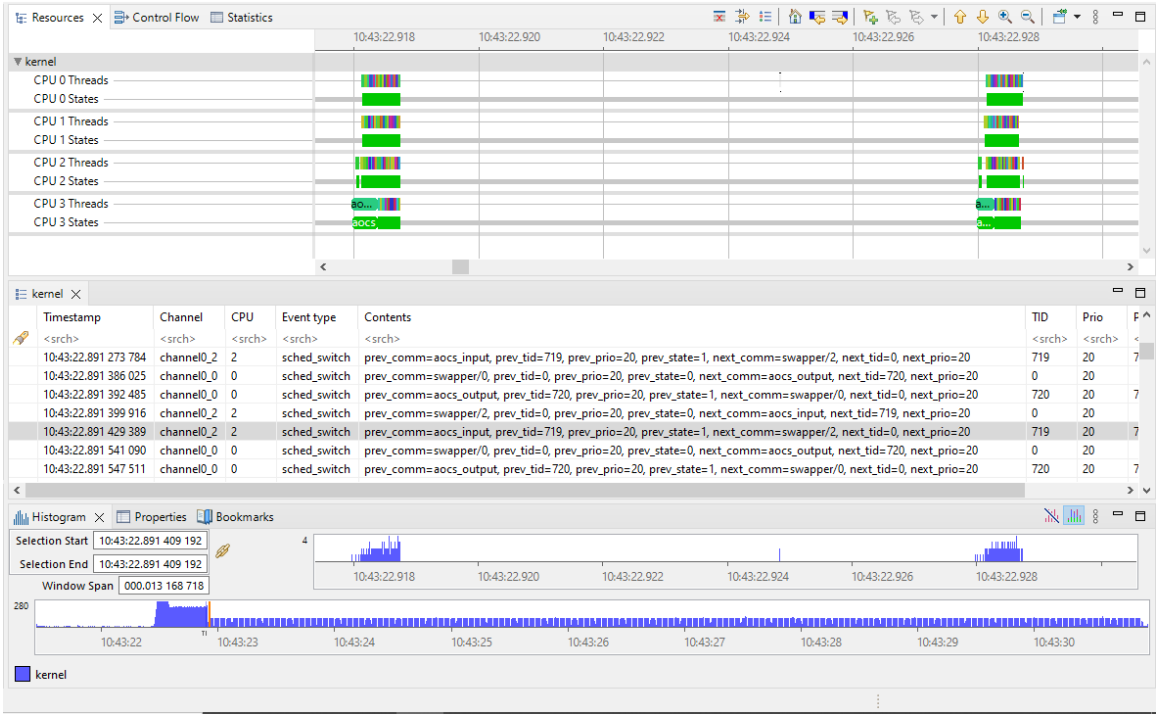

Figure 6.43: Trace of the sched switch events with 80 benchmarks loads making 8192 loads accesses all together

are going to be stored in the cache, because the physical addresses are chosen by the operating system.

The Figure 6.43 shows the kernel trace of the sched switch events when running the AOCS with 80 benchmarks loads each of them making 102 loads accesses which is a total number of loads accesses of around 8192. As for the other experiments, the scheduler manages to schedule all the processes periodically and with a scheduling scheme that is almost perfectly periodic in its internal timings. This confirms that the theoretical setup is actually what's happening in the system. Compared to the others traces shown for the previous experiments, the execution time of the benchmarks is much lower. This is due to the fact that the execution time is not a varying parameter in this experiment and that the execution time of the benchmarks load varies a little with the number of loads. The configured runtime of 0.1ms is then a good upper bound as seen in the figure 6.43.

This experiment aims at finding if the benchmarks load have different impacts on the AOCS process. Theoretically, this experiment has been designed to differentiate the impacts between attackers using only CPU time (benchmarks pi) and attackers using only memory accesses and the L2 cache (benchmarks loads). In practice, the

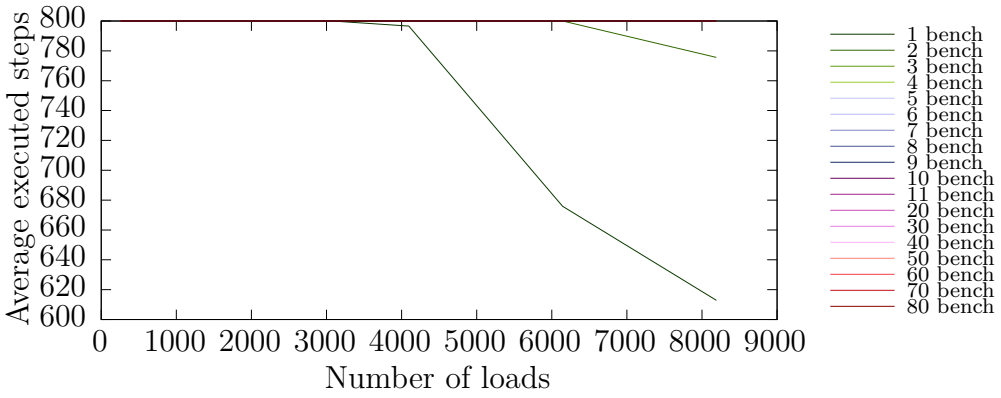

Figure 6.44: AOCS executed steps on average

second type of attacker is impossible to make since using memory accesses will automatically need to use the CPU time. Thus, a part of the benchmarks load impacts will be the same that the benchmarks pi impacts. However, the global impacts won't be exactly the same and it is this difference that is sought in the benchmarks loads experiments.

Deadline misses

Now that the experimental setup has been established, we have to determine whether there are deadline misses during execution. As in section 6.7.1, this is determined by checking the number of executed steps. The Figure 6.44 shows the number of executed steps as a function of the number of benchmarks and the number of loads made by all the benchmarks. Globally all curves are constant with a value of 800 except two of them: the "1 benchmark" and "2 benchmarks" curves. The first curve begins to decrease around 4000 loads and falls down to 610 executed steps which represents a loss of 25%. The second curve is decreasing after around 6000 loads and falls down to 780 executed steps which is a 2.5% loss. This phenomenon can be confirmed by looking at the Figure 6.45. It is only happening for a very small number of benchmarks (lower than 3) and a very high number of loads (greater than 4096 loads). It means that it occurred when the number of loads for each benchmark is very high (greater than 3000 loads by benchmarks). When the number of loads is high for a process its impact on the execution time is higher. If the execution time exceeds the runtime parameter given to the scheduling policy, then the scheduler might fail to schedule the process. This phenomenon has already been seen in the SCHED_DEADLINE experiment with benchmarks pi. Thus, it could explain the phenomenon seen in these charts.

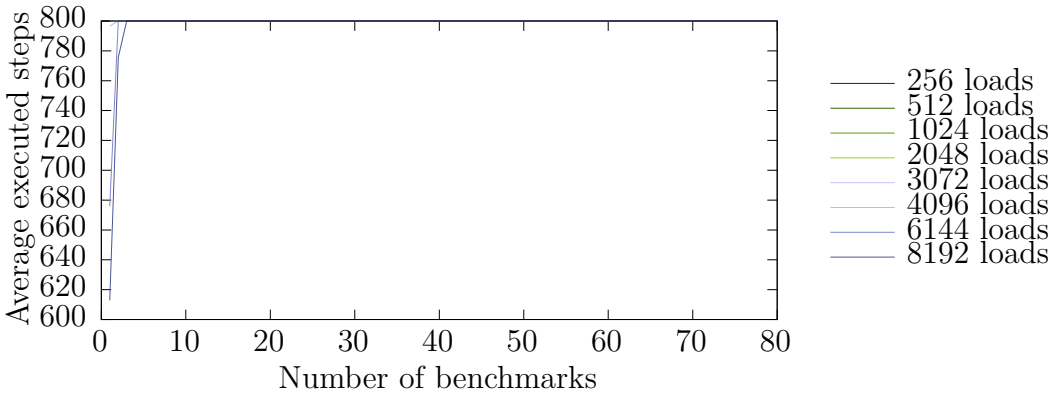

Figure 6.45: AOCS executed steps on average

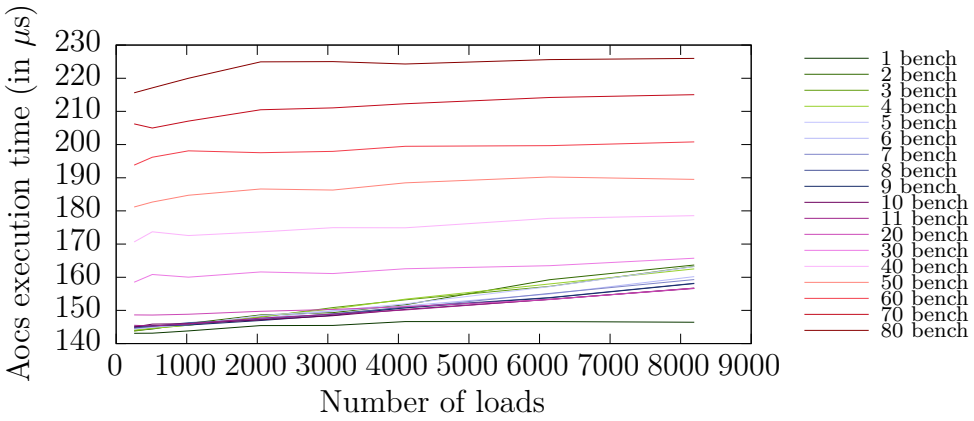

Figure 6.46: AOCS execution time on average

Impacts on the execution time

The Figure 6.46 shows the execution time curves. In this chart, there are two groups of curves: the first one is composed of the curves below 20 benchmarks end the second one of the curves above 20 benchmarks. However, there is still one curve that is alone and does not belong to any of these groups, it is the "1 benchmark" curve. In fact, as seen in the previous sub section, when there is only one benchmark, it does not execute all its steps which reduces the impact of this benchmark on the execution time of the AOCS process. All the curves are increasing but the first group of curves have a more positive slope than the other group.

This difference between the two groups of curves can be seen in the Figure 6.47. We can see a break in the curves at the 20 benchmarks points. The first part of the curves is slightly decreasing, slightly increasing or constant (except for the first point) and the second part of the curves is increasing linearly of around 10 000 ns

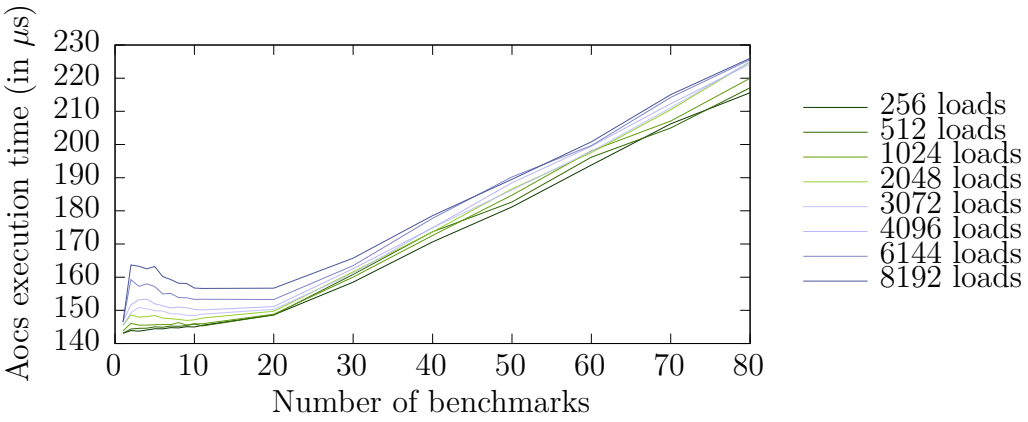

Figure 6.47: AOCS execution time on average

each 10 benchmarks. The probable explanation behind this phenomenon is that the benchmarks have more chances to fill up the cache when there are fewer benchmarks in parallel. This is because the virtual address space of different processes is completely different and the translation of these virtual addresses can overlap and thus go to the same cache lines.

As for the benchmarks pi experiments, the impacts of the number of benchmarks is higher than the impact of the used amount of the shared resource. However, with the benchmarks load, there is a non-negligible impact from the number of loads which was not the case with the global execution time of all benchmarks in the benchmarks pi experiments. This can be explained by the leading role of the L2 cache in the impact on the AOCS execution time.

Source of impacts

As for all the others experiments, the L2 shared cache have a big role in the impacts on the execution time. The Figure 6.48 shows the curves of the number of L2 cache refills. The two groups of curves that were described in the execution time curves are still present here. Moreover, the L2 cache refill curves look very similar to the execution time curves, which would confirm that the impact on the execution time mostly comes from the L2 cache contention phenomenon.

The other L2 cache refill curves presented in the Figure 6.49 also look like the execution time curve. The break that happens in the 20 benchmarks point is less steep but the global shape of the curves is the same.

Overall, it is possible to say that the impacts on the execution time, seen in the previous sub section, come mostly from the L2 shared cache. These impacts are bigger than in the benchmarks pi experiments because in this experiment the benchmarks

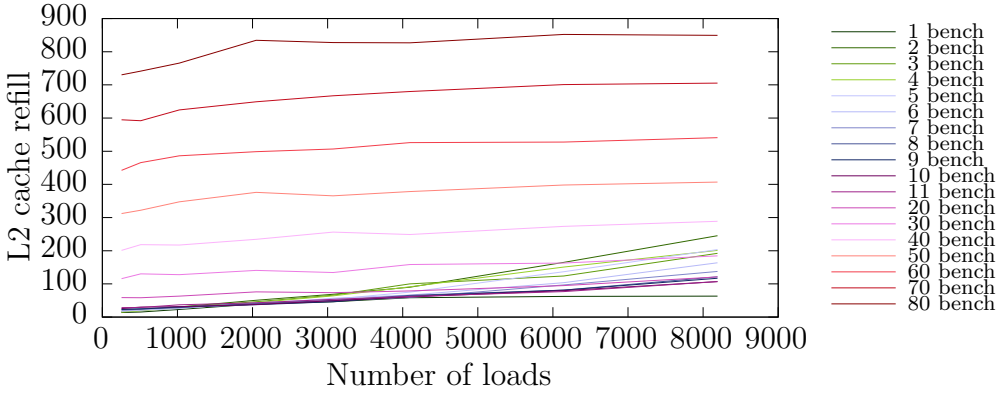

Figure 6.48: AOCS L2 refill on average

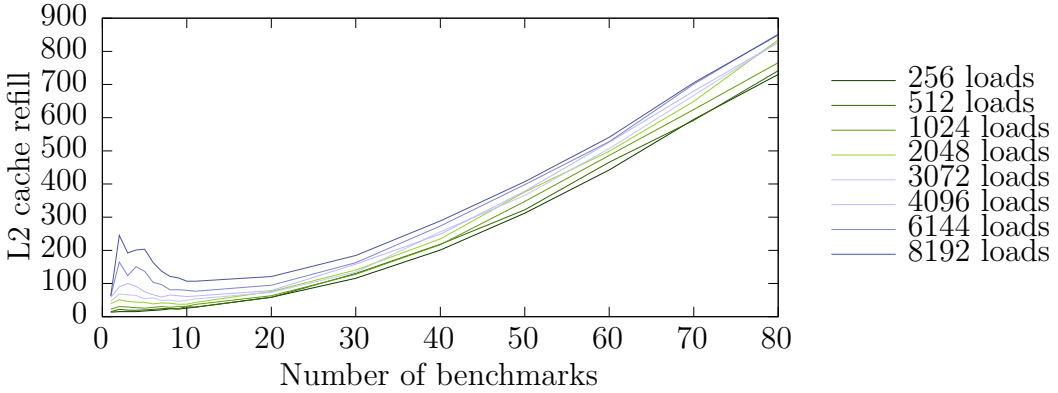

Figure 6.49: AOCS l2 refill on average

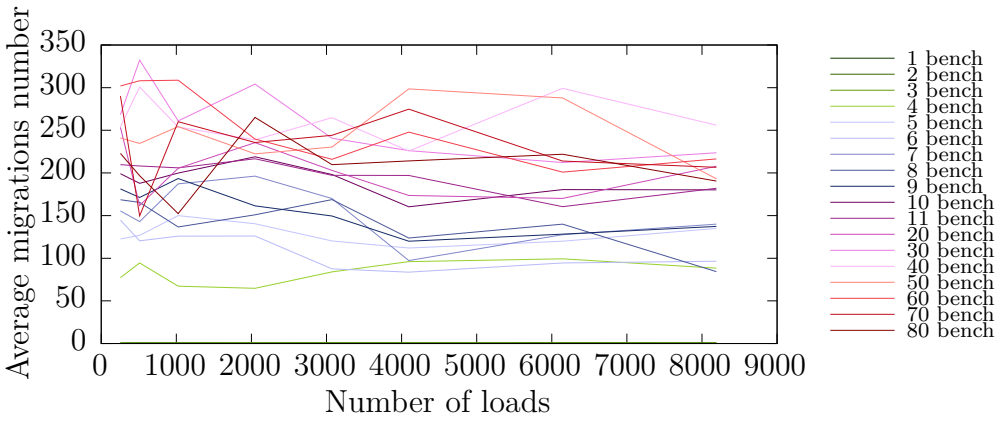

Figure 6.50: AOCS number of migrations on average

are accessing the memory which increases the usage of the L2 cache.

Core migration

The core migrations of all the processes are shown in the Figure 6.50. All the curves are high compared to the number of executed steps (800), with an average number of around 200 migrations. However, even though the curves do not have a clear shape, they are less chaotic than the core migrations curves of the SCHED_DEADLINE experiment with the benchmarks pi. On this chart it is hard to observe a clear phenomenon.

In order to observe a clearer phenomenon, let us take a look at the Figure 6.51 which also shows the number of migrations but the number of benchmarks and the number of loads have switched their representation. We can see that for low values of the number of benchmarks (lower than 5) there are not a lot of migrations (less than 100 migrations). Also when the number of benchmarks increases, the curves become separated. After around 20 benchmarks, it seems that the curves are more stabilized around the value 250.

As for the others core migrations curves, it is hard to understand the behavior of the scheduler by looking at this chart. Moreover, a more important metric is the number of migrations for the AOCS process in particular. The Figure 6.52 shows the number of migrations of the AOCS process only. While the curves are still chaotic, the average number of migrations has decreased a lot compared to the average number of migrations showed in the Figures 6.50 and 6.14. In fact, there are less than 3 migrations on average during the 800 steps of the AOCS process. This is a really low number that makes the core localization very stable from the AOCS process point of view.

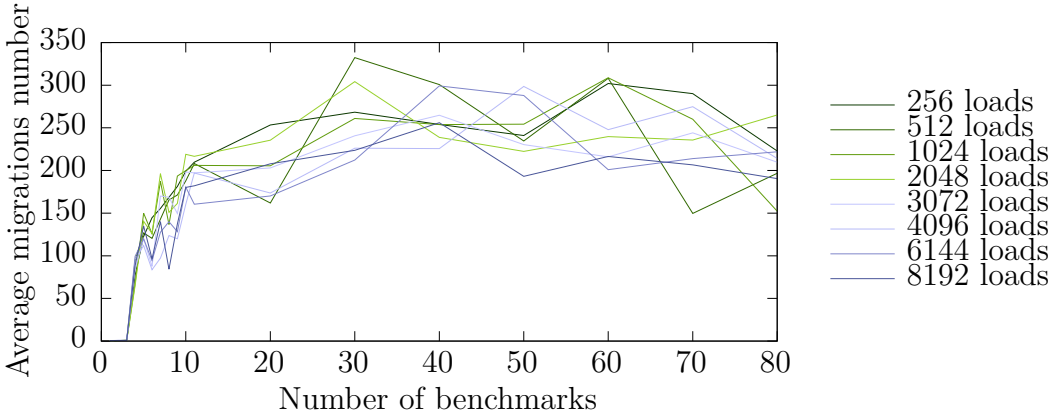

Figure 6.51: AOCS number of migrations on average

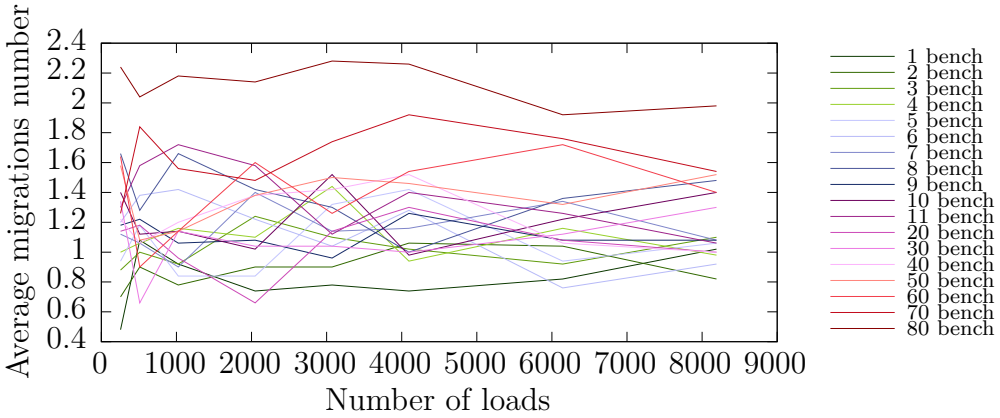

Figure 6.52: AOCS number of migrations of only AOCS

Conclusion

The impact induced by the benchmarks loads in the SCHED_DEADLINE experiment is a little bit different than with the benchmarks pi. It depends on both the number of benchmarks and the number of loads and is mostly induced by the L2 cache refill. However, the phenomenon with the number of executed steps is still present but it occurs in less cases. The behavior of SCHED_DEADLINE regarding the core migration is a bit different compared to the benchmarks pi experiments. On average, there are more migrations but the curves are more stable.

6.8.2 SCHED_RR experiment

Experiment description

This experiment is the same as the previous one but instead of the SCHED_DEADLINE policy, it uses the SCHED_RR. The number of benchmarks and the number of loads will vary the same way. The processes will be made periodic by using the *clock_nanosleep* system calls such as the SCHED_RR experiment with the benchmarks pi (section 6.7.2). As this scheduling policy has been used before it is not necessary to verify the scheduling by using a trace tool.

The main goal of this experiment is to find out if the benchmarks loads change the behavior of the SCHED_RR policy. As the nature of the benchmarks has changed, it is expected that the SCHED_RR policy will change its behavior. A load is more complex that a non-memory CPU instruction since it relies on much more hardware components. Therefore, a guess can be taken that the scheduler will struggle more to find the best scheduling solution. Like the previous benchmark loads experiment, the objective of this experiment is to find the difference between the impacts from the benchmarks pi and loads.

Deadline misses

The first observation required is to check the number of deadline misses which, as for the other experiments, will be approximated by the number of executed steps. The Figure 6.53 shows the number of executed steps and all the curves are constant with the maximum value of 800 executed steps. As with the same experiment for the benchmarks pi (section 6.7.2), this scheduling policy manages to schedule all the processes without problems.

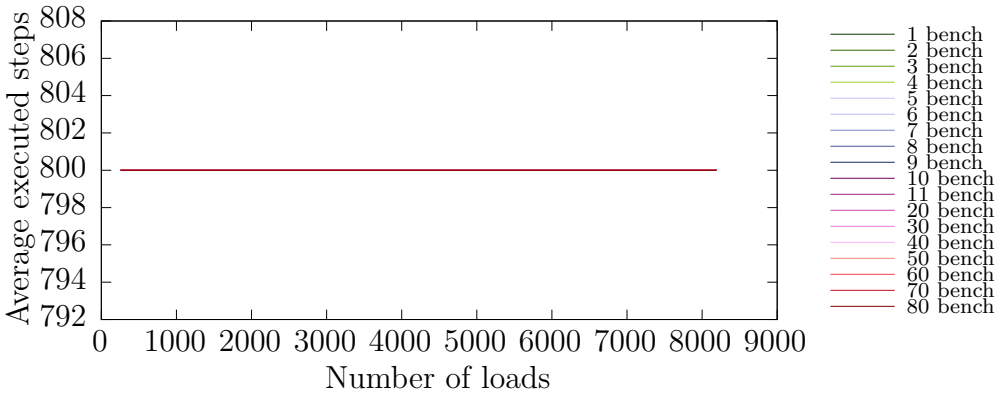

Figure 6.53: AOCS executed steps on average

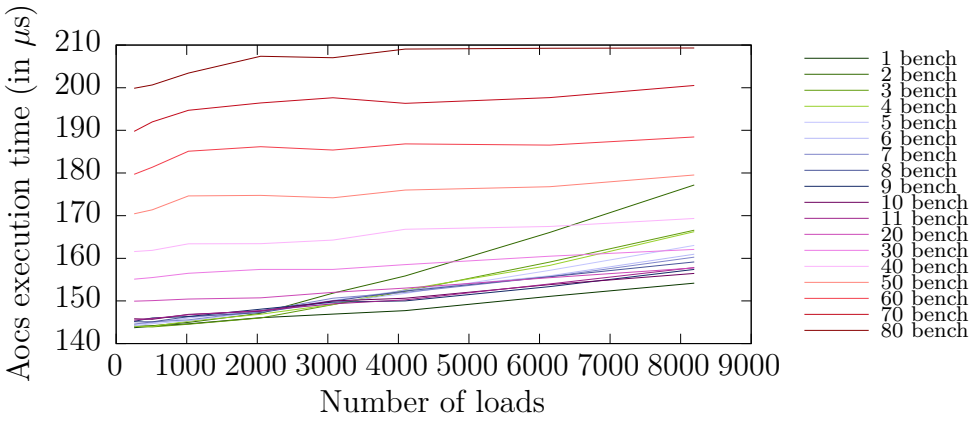

Figure 6.54: AOCS execution time on average

Impacts on the execution time

The Figure 6.54 shows the execution time of the AOCS process. The curves presented in this chart have the same shape as those in the SCHED_DEADLINE experiment. The two groups of curves are still present but the exception for the first curve has disappeared and the "1 benchmark" curve is higher than all the curves of the first group. This is related to the fact that in the SCHED_DEADLINE experiment when there is only one benchmark it is not executed for 800 steps when the number of loads increases.

The Figure 6.55 shows the execution time of the AOCS process but the number of benchmarks is represented on the x-axis. The global shape of the curves is still the same as the one in the SCHED_DEADLINE experiment. However, the break that happened around the 20 benchmarks seems to have disappeared. The slope of the curve is still the same with 10 000 ns for 10 benchmarks.

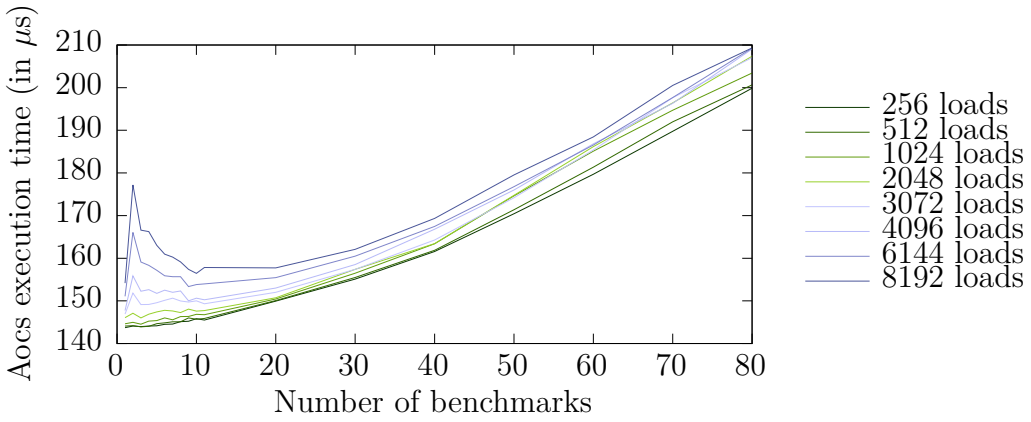

Figure 6.55: AOCS execution time on average

The phenomenon at the beginning of the chart is still happening which does not contradict the prospective explanation described in the last benchmarks load experiment presented in section 6.7.1. As for the benchmarks pi, the execution time curves of the SCHED_RR experiment are lower than the ones of the SCHED_DEADLINE experiment. The impact of changing the benchmarks type is visible mostly at the beginning of the chart but also in the difference between each curve. In fact, in the benchmarks pi experiments, the execution time only depended on the number of benchmarks while in these experiments it also depends on the number of loads.

Source of impacts

To understand the source of the impacts on the execution time, let us analyze the L2 cache refill curves. As for the previous experiments with SCHED_RR, core migrations may happen during the execution of one AOCS step. Thus, it can make the performance counters' measures unusable. This is why the curves presented in this sub-section will only take into account the measures where the L2 cache refill value is under 10 000. This threshold value of 10000 has been chosen because values under 10000 cache refills are coherent values and that when looking at the data all the non-coherent values were way higher than 10000. For example, there were no values around 11 000. Therefore, it is a perfect candidate to be a threshold value in this context. In this experiment there are 8 measures out of 5 760 000 which is completely negligible, thus these measures can be removed without any problems.

The Figure 6.56 shows the number of L2 refills of the AOCS process. It is also possible to switch the x-axis and the color of the curves. The results are presented in the Figure 6.57. In both cases, the curves look exactly like the ones in the execution time curves presented in the Figures 6.54 and 6.55. Thus the impacts on the execution

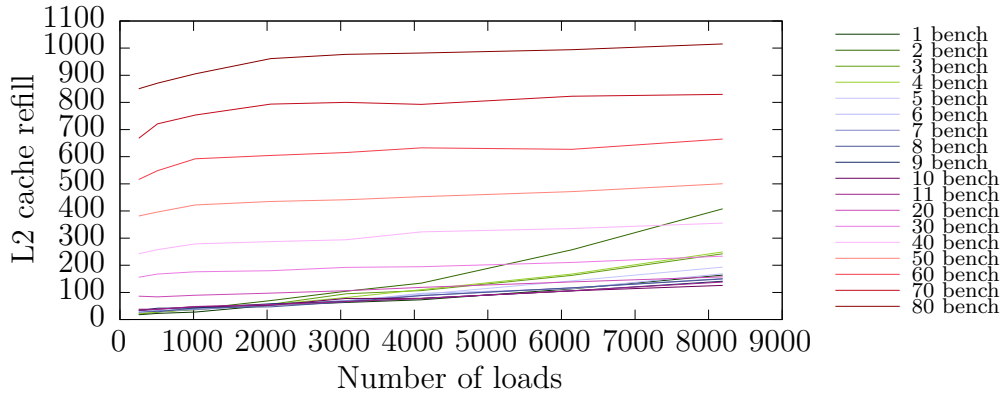

Figure 6.56: AOCS L2 refill on average

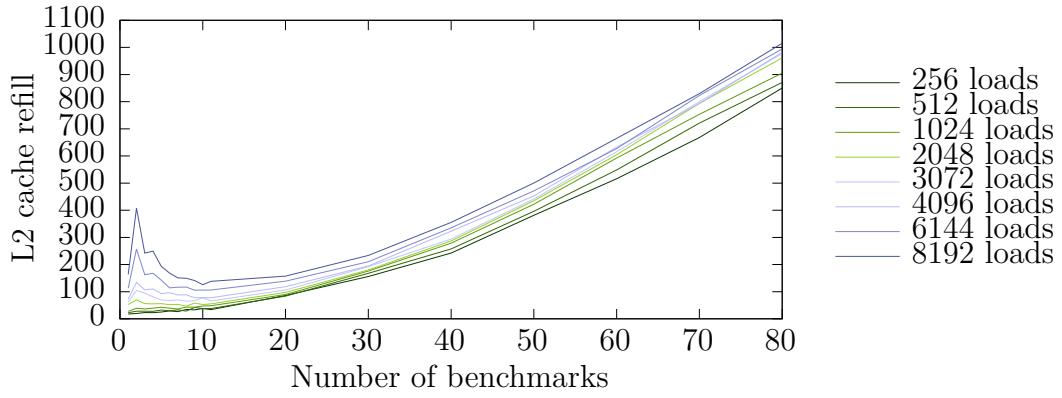

Figure 6.57: AOCS l2 refill on average

time of the AOCS process come from the L2 cache.

Core migration

The last sub-section of this experiment will focus on the core migrations. The Figure 6.58 shows the number of migrations of all processes. The curves are mostly constant with a value between 0 and 20 migrations. Compared to the SCHED_DEADLINE experiments with benchmarks load, there are much fewer migrations and the curves seem more stable. In fact, there are not a lot of points that are above 30 migrations which is not a lot for 800 steps. Compared to the same curves but with benchmarks pi, the phenomenon where two curves were above all others has disappeared. The hypothesis is that the SCHED_DEADLINE policy struggle to find an optimal scheduling for certain numbers of executed processes.

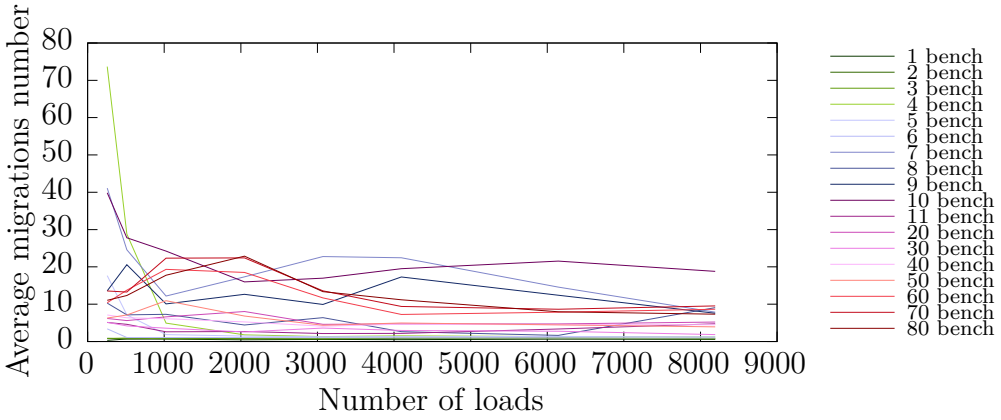

Figure 6.58: AOCS number of migrations on average

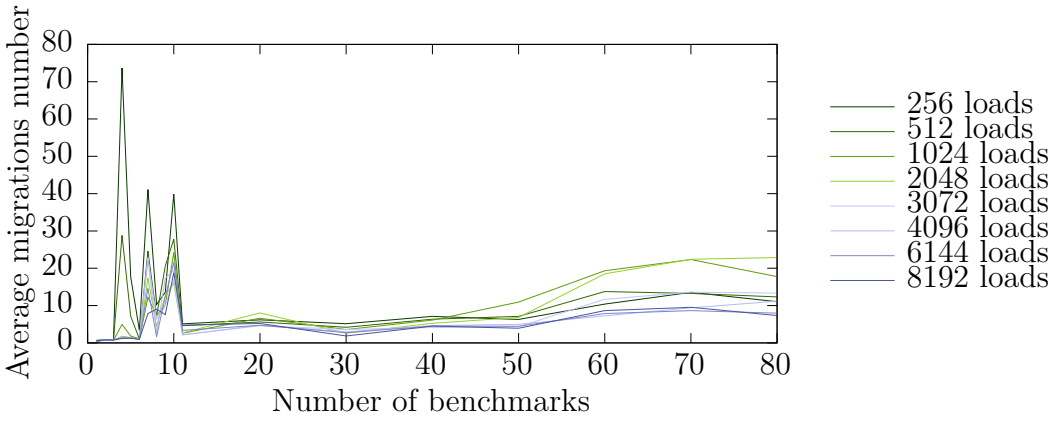

Figure 6.59: AOCS number of migrations on average

However, as shown in the Figure 6.59, there are still a lot of migrations happening when the number of benchmarks is lower than 10 but the number of migrations is lower (between 10 and 70). Moreover, at the end of the chart, we can see that some curves are increasing from below 10 migrations to between 10 and 20 migrations but this phenomenon is not big enough to be analyzed here.

Finally, let us take a look at the number of migrations of the AOCS process only. The Figure 6.60 shows this number of migrations. The number of migrations of the AOCS process only resembled the average one. However, the peak around the 10 benchmarks point is much higher for the AOCS only. Also, all the curves are a little higher for the aocs alone. The AOCS migrates more than the others processes with the SCHED_RR policy, this is the same tendency that has been demonstrated in the benchmarks pi with the SCHED_RR policy presented in the section 6.7.2. To explain this behavior, the hypothesis is that it is not the SCHED_RR policy that migrates

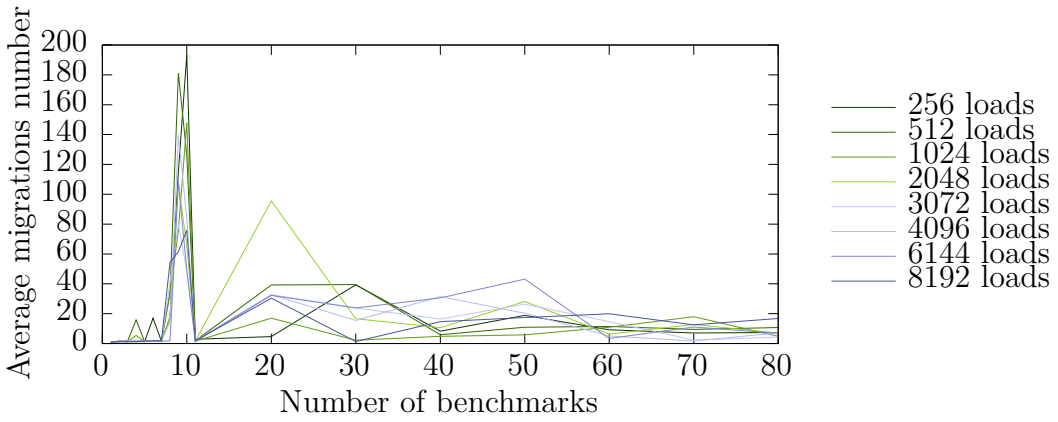

Figure 6.60: Number of migrations of the AOCS process only

more the AOCS process but the SCHED_DEADLINE one that migrates it much less because of its higher runtime. The number of migrations of the AOCS process with the SCHED_RR policy only depends on its place in the roud-robin queue since all the processes have the same priority.

Conclusion

To conclude, the number of loads impacts the L2 cache refill differently than the benchmarks pi. This has an impact on the execution time of the AOCS. In terms of migrations and number of execution, the SCHED_RR policy behaves exactly like during the benchmarks pi experiment. The differences between the SCHED_RR and SCHED_DEADLINE policy shown in this sub-section are comparable to the ones seen between the same two experiments but with the benchmarks pi presented in the sections 6.7.1 and 6.7.2.

6.8.3 SCHED_RR + cgroups experiment

Experiment description

This experiment is the same as the previous one with two cgroups configured. The cgroups are configured like in the corresponding benchmarks pi experiment (see section 6.7.3). The first cgroup is composed of the AOCS process only and the second one is composed of all the benchmarks. They both have a deadline of 10ms and a runtime of 1ms for the AOCS cgroup and 8.5ms for the benchmarks cgroup as explain in the section 6.7.3.

The main goal of this experiment is to find out if the impacts of the benchmarks loads can be reduced by the cgroups. The benchmarks load do not use the CPU

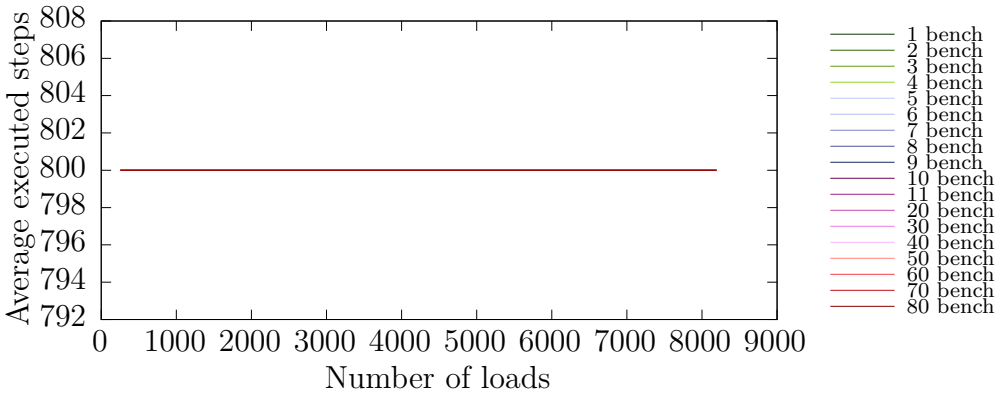

Figure 6.61: AOCS executed steps on average

time like the benchmarks pi and thus it could be argued that trying to reduce their impacts by configuring a CPU time limiting mechanism would not be the best solution. However, it has been seen before that the cgroups have an impact on the core migrations behavior of the scheduler and the core migrations have an impact on the usage of caches. Finally, this experiment aims at finding if the cgroups are useful in this particular setup.

Deadline misses

As for all the other experiments, it is interesting to first find out if there is any deadline misses. The Figure 6.61 shows the number of executed steps which is the approximation of the deadline misses used in these experiments. All the curves are constant with the value of 800 which is the maximum possible.

Thus it is possible to conclude that adding the cgroups configuration in the system did not add deadline misses. It means that all the processes respected their deadlines.

Impacts on the execution time

The Figure 6.62 shows the execution time of the AOCS process during this experiment. The curves have the exact same shape and the same values as the ones from the previous experiment (section 6.8.2). However, compared to the corresponding benchmarks pi experiment the phenomenon that appeared with the ”1 benchmark” curve is not reproduced here. This is because all the curves are higher than in the benchmarks pi experiment and thus the ”1 benchmark” curve is in the group of curves. The phenomenon did not disappear but is now hidden by the other curves.

The Figure 6.63 also confirms the similarity of the two experiments. It means that in terms of execution time, the cgroups configuration had no noticeable impact

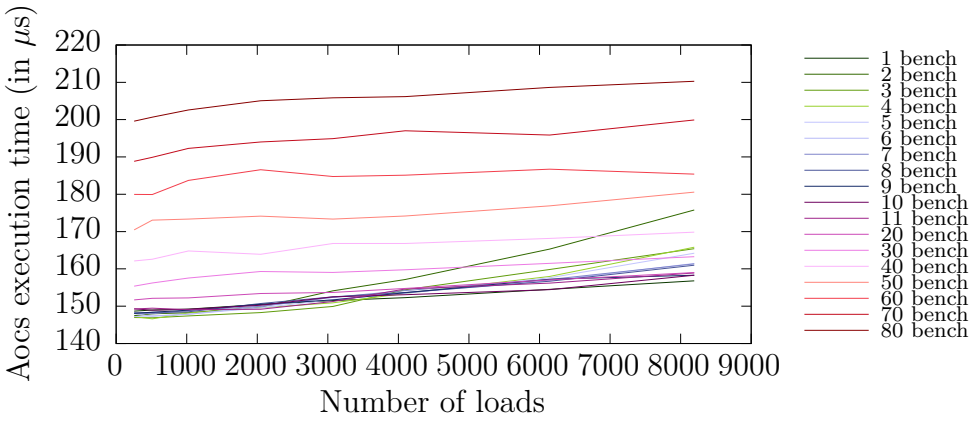

Figure 6.62: AOCS execution time on average

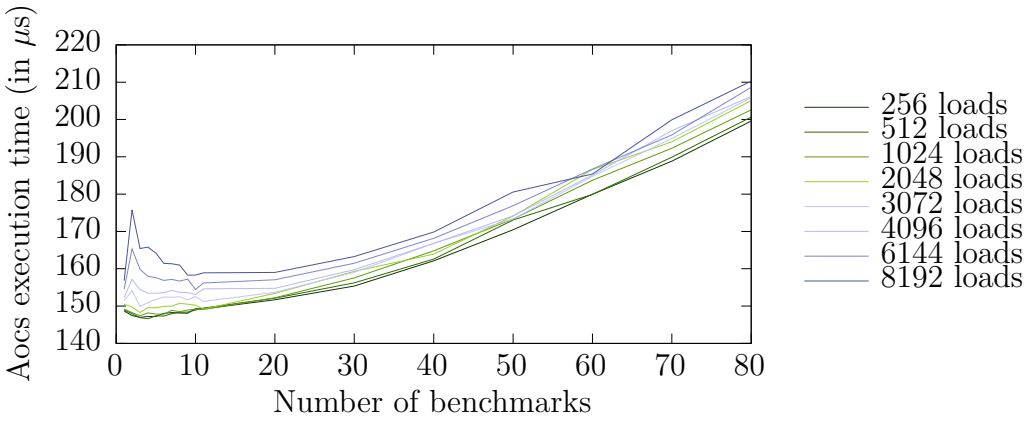

Figure 6.63: AOCS execution time on average

on those measurements.

Source of impacts

The following curves are presented in the Figures 6.64 and 6.65 show the number of L2 cache refills of the AOCS process. The aberrant points have been removed from these charts. There was 9 out of 5 760 000 non-coherent point where the measured L2 cache refill was greater than 10 000. This number is negligible and thus does not modify the analysis of the results.

Without surprise, these curves are very similar to the execution time curves but also to the L2 cache refill curves from the previous experiment. The L1 cache does not have a huge impact on the AOCS execution time. The L2 cache is still a contention point specifically with the benchmarks loads. The fact that those benchmarks are

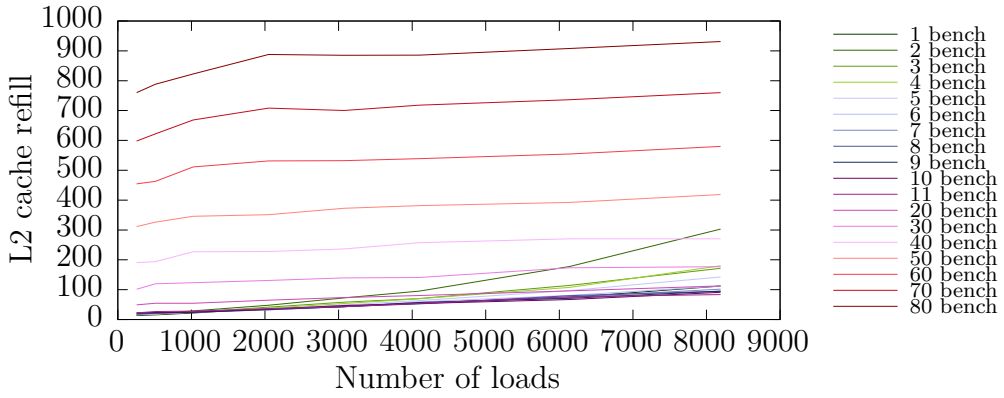

Figure 6.64: AOCS L2 refill on average

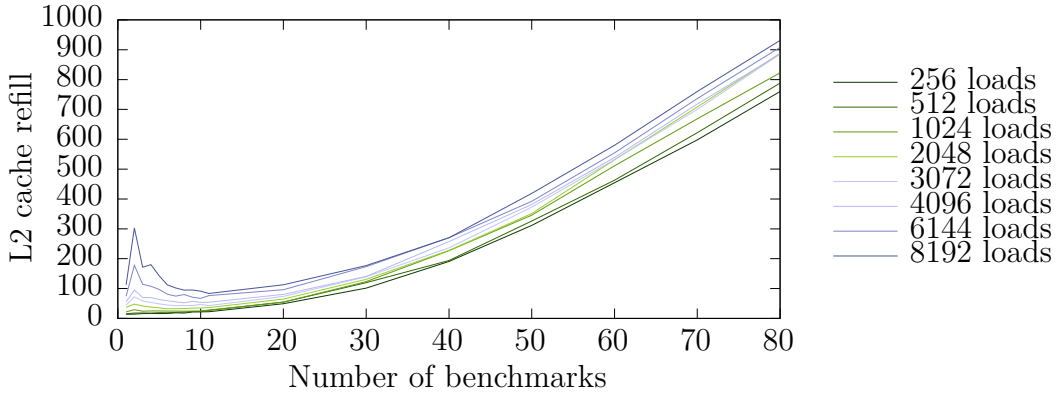

Figure 6.65: AOCS l2 refill on average

making more memory accesses makes the cgroups less able to reduce the interference. In fact, the memory will be accessed by the benchmarks no matter what the scheduler decides.

Core migration

The core migrations curves are presented in the Figure 6.66. This chart looks almost exactly like the corresponding one in the previous experiment. To confirm this feeling it is possible to look at the other migrations curves in the Figure 6.67. Here again, these curves are very similar to the previous experiments ones.

Finally, let us find out if the AOCS migrates more or less than without the cgroups. The Figure 6.68 shows the number of migrations of the AOCS process only. The curves presented in this figure have the same shape as the corresponding ones from

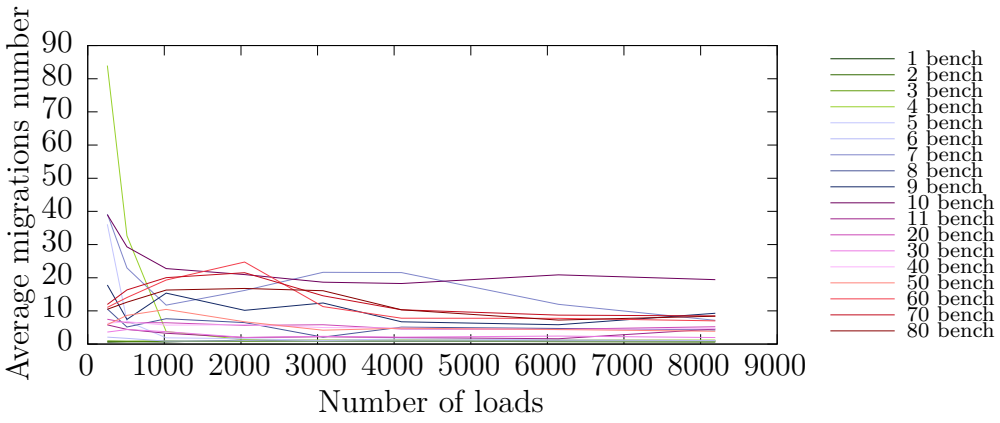

Figure 6.66: AOCS number of migrations on average

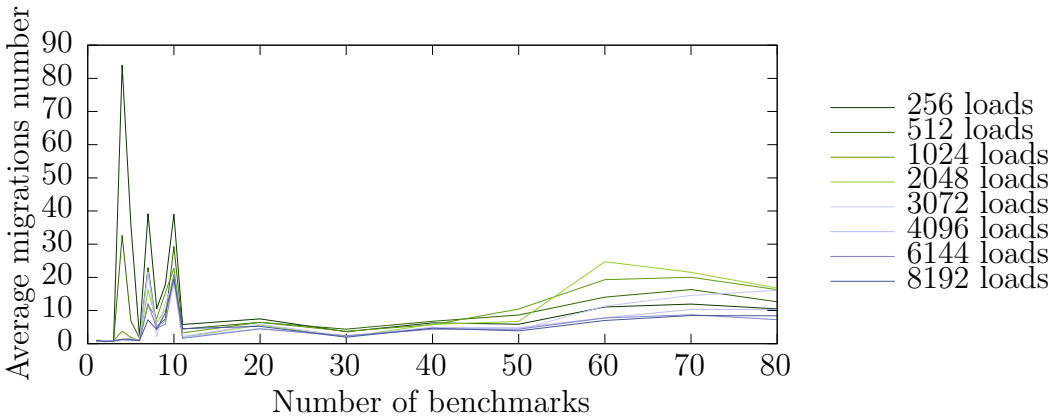

Figure 6.67: AOCS number of migrations on average

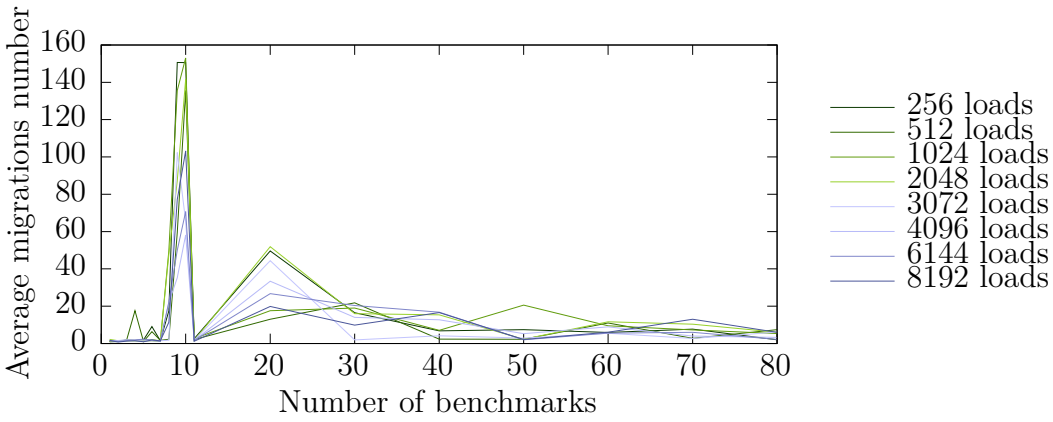

Figure 6.68: Number of migrations of the AOCS process only

the previous experiment. The peak around the "10 benchmarks" point is still here and the number of migrations is globally higher than the average one. Thus, adding the cgroups did not have any impact on the core migration of the AOCS process only.

Conclusion

To conclude this experiment, adding the cgroups did not modify the impacts significanty of the benchmarks on the AOCS process. The behavior of the whole system is globally similar as the one in the experiment without the cgroups.

6.8.4 Academic patch experiment (SCHED_DEADLINE + cgroups)

Experiment description

This experiment uses the patch made by Luca Abeni *et al.* along with the benchmarks loads. The configuration of the cgroups is the same as for the other experiments (sections 6.7.3 and 6.8.3). There is one cgroup for the AOCS process and one other for all the benchmarks with a deadline of 10ms and a runtime of 1 ms for the first cgroup and 8,5ms for the second one.

This experiment aims at finding if the two-level scheduler can reduce the interference previously seen in the L2 cache.

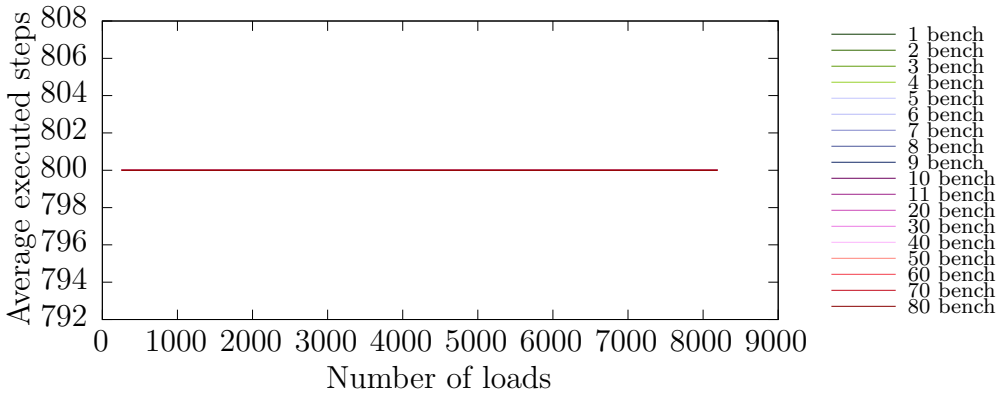

Figure 6.69: AOCS executed steps on average

Deadline misses

As with the other experiments, we first check if no deadline miss occurs. As shown in the Figure 6.69, all the processes execute 800 steps for all experiment runs, regardless of the benchmark configuration. No deadline miss occurs and the system is successfully scheduled. It means that all the processes execute all their steps and thus it is possible to conclude that the scheduling is behaving well since all the processes are executed correctly.

Impacts on the execution time

The Figure 6.70 shows the curves of the AOCS execution time. Once again with the benchmarks load the execution time curves are very similar to the ones of the SCHED_DEADLINE experiment with benchmarks loads presented in the section 6.8.1. There are still two groups of curves: one before 20 benchmarks and the other after 20 benchmarks. The shapes of the curves are the same as the ones from the previous experiments. However, the values of the curves are lower than the ones in the SCHED_DEADLINE experiment.

This tendency is also confirmed by the other execution time curves shown in the Figure 6.71. We can see the increase in the curves when the number of loads increases. Moreover, the decreasing phase at the beginning of the chart is still present and the peak of this phase is higher than in the SCHED_DEADLINE experiment.

In terms of impacts on the execution time, adding the academic patch reduced the execution time on average compared to the SCHED_DEADLINE experiment but increased it for low values of the number of benchmarks.

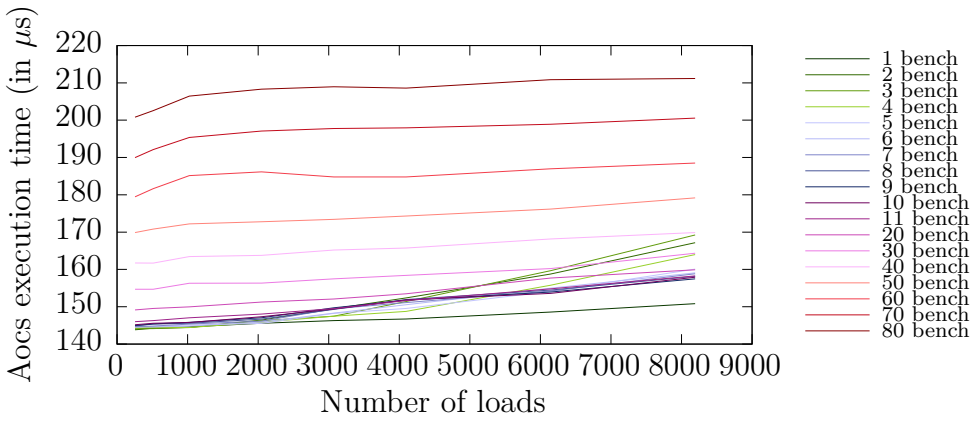

Figure 6.70: AOCS execution time on average

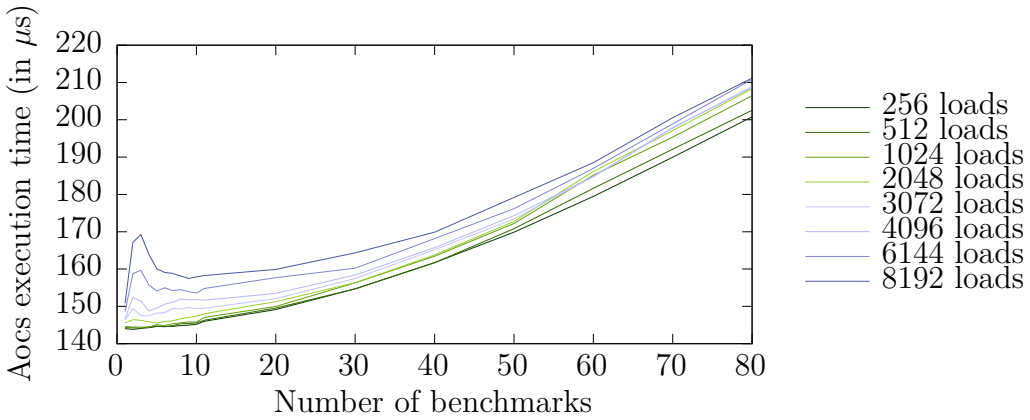

Figure 6.71: AOCS execution time on average

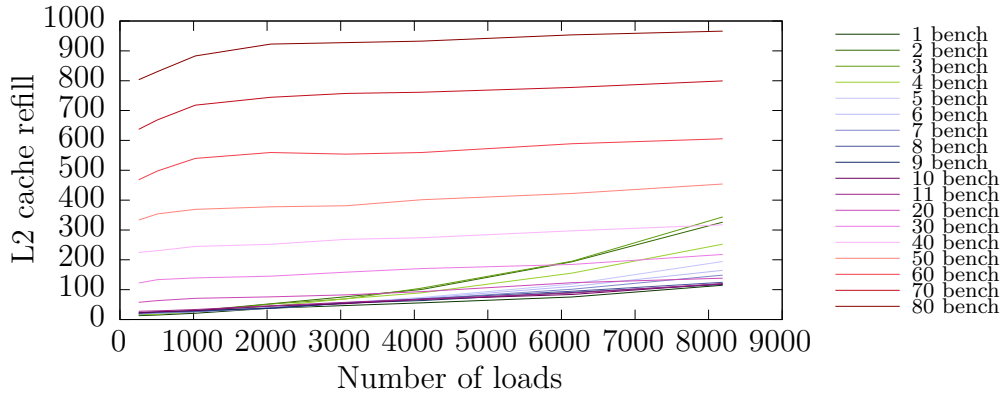

Figure 6.72: AOCS L2 refill on average

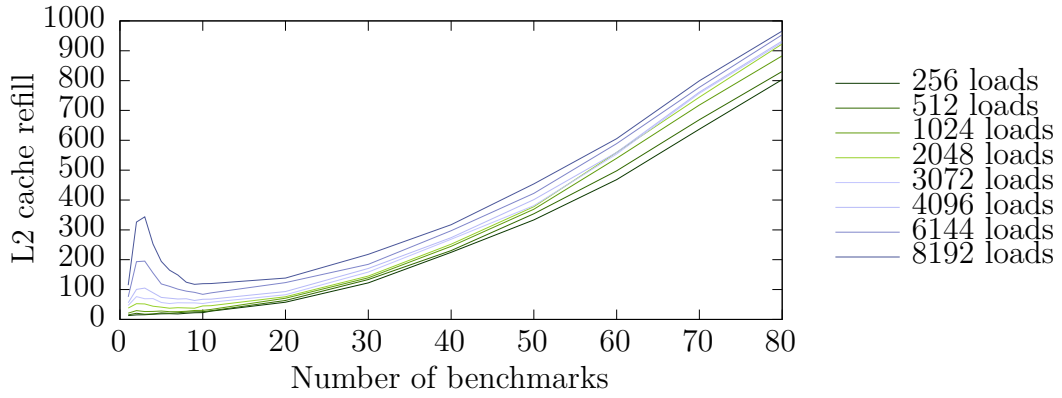

Figure 6.73: AOCS l2 refill on average

Source of impacts

The Figures 6.72 and 6.73 shows the number of cache refills of the AOCS process. These curves look like the execution time curves showing that most of the impacts on the execution time come from the refills of the L2 cache.

However, in the first group of curves, there are small differences between the L2 cache refill chart and the execution time chart. Some curves that belong to the first group of curves seem lower in the L2 cache refill chart than in the execution time chart. This can mean that there is another source of impact for the execution time which can be the L1 data cache refills.

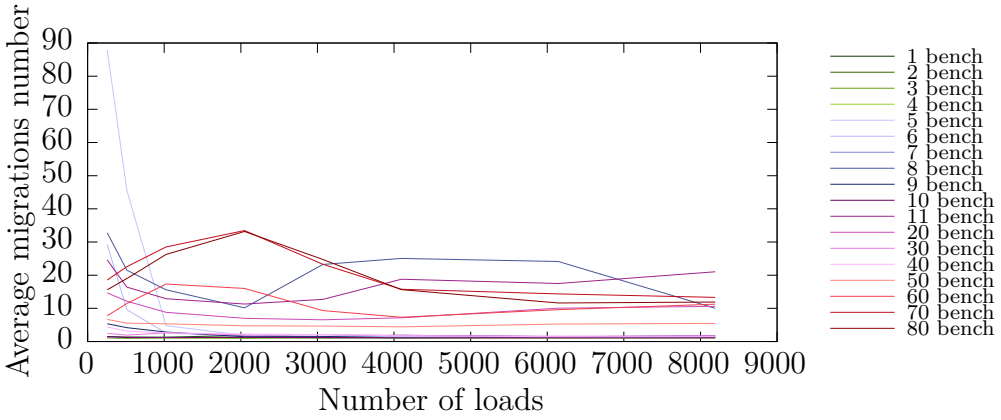

Figure 6.74: AOCS number of migrations on average

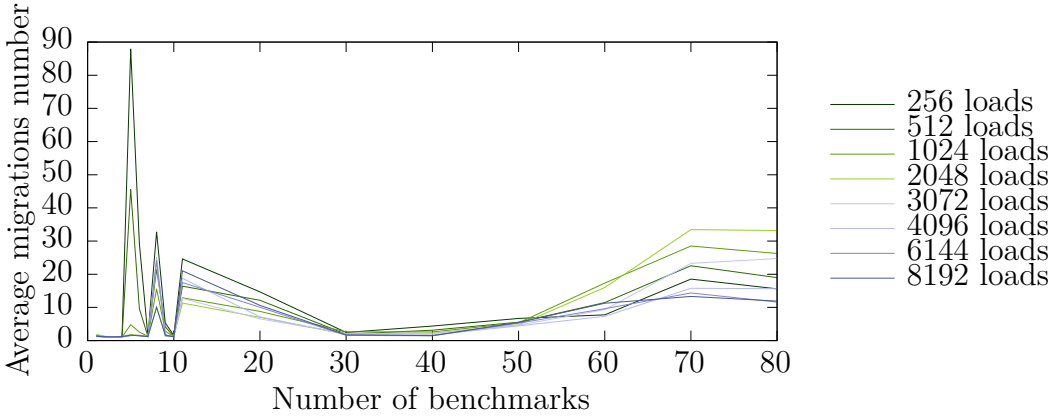

Figure 6.75: AOCS number of migrations on average

Core migration

The Figure 6.74 shows the number of migrations of all the processes during this experiment. Except for one abnormal point, the average number of migrations is very low for all the curves. The maximum is around 30 migrations which do not represent a huge amount for 800 step executions. Compared to the other benchmarks loads experiment with SCHED_RR, the average number of migrations is a bit higher but the peak points are lower. This is confirmed by the Figure 6.75 where we can see that there are fewer big peaks before the 10 benchmarks point.

In terms of the number of migrations of the AOCS process only, as shown in the Figure 6.76, there are much fewer migrations than with the others SCHED_RR experiments. The system is more stable with the AOCS process since it migrates much less which reduces the interference. The SCHED_DEADLINE policy has information on the runtime of the AOCS process and thus decides to not migrate it too much

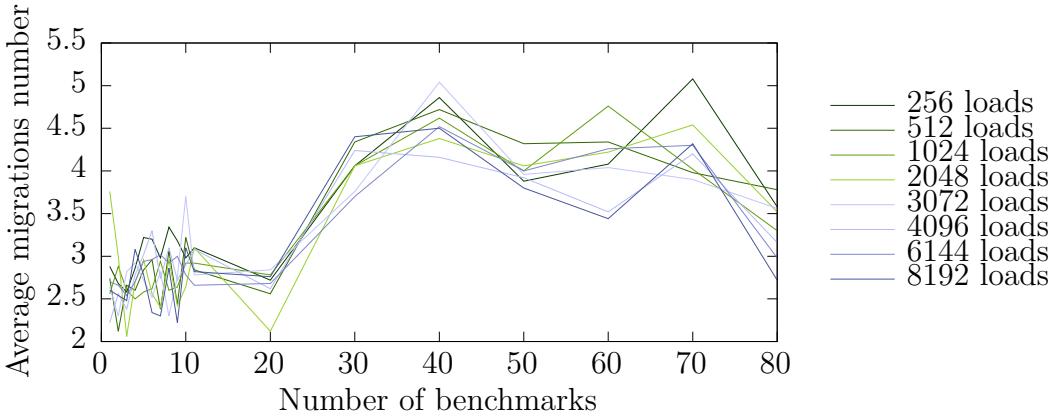

Figure 6.76: Number of migrations of the AOCS process only

since it has the higher runtime of all the processes.

Conclusion

To conclude, as for the corresponding benchmarks pi experiment, this experiment has the same impacts as the SCHED_RR experiments but the number of migrations of the AOCS is the same as the one from the SCHED_DEADLINE experiment. The academic patch combines the advantages of the SCHED_RR and the SCHED_DEADLINE policies.

6.8.5 Conclusion on the benchmarks loads experiments

To conclude on the benchmarks loads experiments, the difference between SCHED_DEADLINE and SCHED_RR experiments is very small. In fact, despite the constant overhead that is different for each scheduling policy the only aspect that changes between the two scheduling policies is the number of migrations. As described for the benchmarks pi experiments, the number of migrations is much lower on average for the SCHED_RR policy but it is the contrary for the average number of the AOCS process only.

Adding the cgroups and the academic patch did not modify the global behavior compared to the benchmarks pi experiments. The fact that the benchmarks load are both making memory accesses to the l2 cache and also using the CPU power makes the induced impacts hard to differentiate from the one presented in the benchmarks pi experiments.

6.9 Difficulties encountered during the experiments

This chapter introduced a lot of experiment results and the analysis of interference regarding the problem statement of this thesis. Although, in practice, to make these experiments run there are some difficulties that will be explained in this section.

6.9.1 Duration of the experiment

The first difficulty encountered is the duration of each experiment. Let us take the SCHED_DEADLINE experiment with benchmarks pi as an example. In this experiment, there are two varying parameters which take 11 and 18 possible values. It means that there are $11 * 18 = 198$ couples of parameters for the experiment configuration. Each of these couples represents 800 steps to be executed within a period of 10ms. It takes $800 * 10ms = 8seconds$ to execute 800 steps and 2 more seconds need to be added because the parent process waits before releasing the barrier that every child process has finished initializing. To execute every couple of configuration parameters it takes $198 * 10s = 1980s = 33$ minutes. Then this is repeated 50 times to get smoother results which leads to a duration of around 27,5 hours. Note that for the benchmarks loads experiments the duration is lower because there are only $8 * 18 = 144$ possible couples of configuration parameters instead of 198. The number of hours needed to have the results from one experiment is multiplied by the number of experiments that are presented in this chapter. In the end, it took weeks of execution to get all the results considering that some experiments were done multiple times.

This significant experiment duration time makes it hard to do some changes in the experiment protocol. Each time a change is made it is required to redo all the experiments. It is however possible to run the experiment for less time and adjust the protocol by looking at the small experiment results.

6.9.2 Academic patch behavior

In the fourth experiment with benchmarks pi and loads, a Linux kernel patch has been used. This patch was made by researchers and then was not part of the Linux ecosystem and thus was not verified by the open-source community.

First of all, the patch was not available for the 5.4 Linux version. This would be a problem considering that the previous experiments were made with this Linux version. Moreover, the Xilinx Linux kernel was only available for a version that was not compatible with the academic patch. Finally, the simplest solution was to modify the patch to make it compatible with the 5.4 version of Linux.

The patch incompatibility with the Linux version was only due to minor changes in the code. There were changes in function names or signatures but also new functions that caused modification of the line numbers.

Second of all, when using the patch with benchmarks pi a software bug appeared in the operating system. This bug occurred at the end of the 800 steps of the AOCS process when it kills all the benchmark processes. The operating system then got unresponsive, it is not possible to interact with it, the ssh connection is closed and the only way to recover the system is by shutting down the parent process of the experiment with a Ctrl+C. If nothing is done, after a while the process will terminate and the system will continue its activity. This phenomenon does not affect the experiment results since it happens after all processes are killed. The problem with this bug is that the duration time of the whole experiment is much longer. It took around 7 days non-stop to get all the results from the 50 repetitions of the experiment.

6.9.3 Reproducibility of observed phenomena

Finding the source of a phenomenon after it has been observed, usually implies to reproduce the phenomenon. When the conditions of the experiments are fully controlled, to reproduce a phenomenon is not that hard. However, in the context of the experiments presented in this chapter, the non-reproducibility of the system under observation makes the control of the events difficult. It means that to reproduce a phenomenon the experiment has to be repeated until it occurs. This increases the duration of the experiment.

Chapter 7

Analysis synthesis and methodological recommendations

This chapter is focused on synthesizing the experiments analysis and the Linux configuration. Its goal is to give recommendations on the experiments protocol and the configuration of Linux.

7.1 Results analysis and experiments recommendations

7.1.1 The importance of L2 cache refills

Almost all experiments presented in this thesis, have shown that the L2 cache is a bottleneck in the system. The observed execution time of the AOCS process depends on the L2 cache to the extent that the execution time curves and L2 cache refill curves have, most of the time, very similar shapes. By definition, when an L2 cache refill occurs the cache will transfer the memory access to the bus towards the main memory. In the memory hierarchy, the main memory is one of the farthest from the CPU, meaning that accessing it takes much more time than accessing the L2 cache. This could explain the relationship between the execution time and the L2 cache refills. There are two possible options to reduce this impact: either reduce the number of L2 cache refills or reduce the amount of time taken by access to go from the L2 cache to the main memory. Reducing the access time from the L2 cache to the main memory is a hardware problem and is not in the scope of this thesis.

However, there are solutions to reduce the number of L2 cache refills. A hardware solution would be to increase the size of the cache by either adding more lines or more ways. If more space is available in the cache the chances of having an L2 cache

refill are lower. This solution is correct only if the memory access is distributed homogeneously during the execution. If all the processes access memory addresses that are stored in the same cache line, increasing the size (or even the number of ways) of the cache won't have a satisfying effect. In the context of this thesis, this solution is not applicable because the size of the L2 cache is determined by the SoC manufacturer and creating a custom chip is out of the scope.

The second solution to reduce the number of L2 cache refills is to use cache partitioning techniques. Cache partitioning allows dividing a cache into different parts and making these parts accessible only from allowed processes (see [57] and [88]). The reduction of the number of cache refills depends on the partitioning configuration and profile of the applications. An important parameter to consider is the interference nature of the cache refills. As explained in the section 4.2 a refill is considered to be an interference only if it is induced by a different process than the one that suffers from the refill. It means that cache partitioning can reduce the number of interferences but not necessarily the number of cache refills. This is because from the process point of view the size of the cache will decrease since it will only have access to a part of the whole cache, as a consequence the number of refills will increase but the number of processes that can interfere with the process will decrease. However, the number of interferences is harder to measure than the cache refills and it will be difficult to choose the best cache partitioning configuration.

Partitioning the cache means that the profile of each application needs to be known to find the best configuration possible that will reduce the number of interference. Also, cache partitioning will be optimized when used with static scheduling. This way, the order of execution of each process is known before executing it and it is easier to find the optimal partitioning configuration.

If the profile of each application is unknown, a solution could be to use a empirical method to find the best paritioning configuration of the cache. The real challenge of this solution is that it has to be modified during the flight of the spacecraft since the applications are not known.

In the scope of this thesis, the cache partitioning mechanism could be a good candidate. However, a solution has to be found to use this mechanism with a dynamic scheduler such as the ones used by Linux. For example, creating one cache partition per core could be a solution if it is used with a scheduling policy that tends to keep processes on the same core during their execution. This solution is not ideal because migrating processes is used by scheduling policies to have more flexibility in their decisions. Another approach is to create one cache partition for every cgroup. Note that this solution could lead to have the L1 cache partitioned.

7.1.2 Impacts of network packets

The different experiments presented in this thesis showed a significant impact of the network-related code on the AOCS process. The section 5.3.2 showed the impacts of the *timesyncd* and *ntpd* services. There are two common points between these two services. The first one is that they are both systemd services (described in the section 2.4). The second one is that they need to send and receive network packets.

The figure 5.12 shows the trace of execution of the *timesyncd* service. In this trace, we see that the *sd-resolve* service is executed (often). The role of this service is to provide name resolution to local applications. This service needs to send and receive network packets to work properly. Moreover, the *timesyncd* service also needs to send and receive network packets. The processing of these packets and also the network stack used to send and receive those packets use a lot of memory. The network stack needs a lot of buffers to handle the interface between the kernel and the network physical layer. It can also be seen on the trace that the systemd process is also executed a lot of times. The impact of these executions cannot be neglected even if it is hard to measure the real impact on the memory of this process. This rationale is the same for the *ntpd* service which relies on the same network protocol as the *timesyncd* service.

Overall, using a complex network stack associated with systemd daemon management induces a huge impact on the AOCS process. These network services have been designed to handle the network operations of a desktop computer or a server connected to a dynamic network. However, an embedded system has a reduced network size and the network is not modified through time. There are no new routers of endpoints in a satellite. Implementing an optimized custom network stack should reduce the impact of network-related applications. Note that in the case of a satellites constellation the idea of discovering new endpoints can be useful but it happens in a very specific scheme and a custom protocol could be used.

7.1.3 Impacts of the CPU relative load

The chapter 6 shows experimental results of the impact of the CPU relative load. The benchmarks pi were designed to increase the CPU relative load. These experiments conclude that the CPU relative load does not impact the execution time of the AOCS process. It could seem to be a counter-intuitive result, however, it is not.

When thinking of the CPU relative load the rationale is that increasing it will increase the amount of work the system has to do. While this is true for the CPU it is not true for the operating system. In other words, a process with an execution time of 0,1ms with a period of 10ms and the same process with an execution time of 8ms have the same impact on the AOCS process as long as they are not interrupted. The

only limitation is that the operating system needs some time to execute, meaning that the process cannot be executed during 9,9ms on a period of 10ms. This is confirmed by the results presented in the chapter 6. It is a counter-intuitive result because most of the time increasing the execution time is related to an increase in resources use (memory, I/Os, caches, bus, ...) which induces more interference and thus has a higher impact on the victim process.

Software applications which do not use shared resources other than the CPU power are not very common. The vast majority of software applications use memory. However, if the processes are isolated and the shared resources accesses do not have an impact on the other processes, then increasing the execution time of a process does not affect the other processes. This means that it is possible to reduce the margin of safety of CPU power taken to ensure that every process will have time to execute no matter what impacts they are facing. Thus, therer is more CPU power available on the platform.

7.1.4 Measuring hardware counters through Linux

Hardware performance counters are designed to help understand the behavior of hardware components such as caches, memories, and buses. They can also serve as profiling software applications and thus give information on the execution of these applications. The execution behavior of an application depends on the operating system executing it. For example, a bare-metal application cannot use services and its entire code is executed directly on the hardware, thus its execution time is not the same as if it was executed on an operating system. A software application running on an operating system is executed inside an environment that provides services but also decides the scheduling. This environment impacts the behavior of the application and thus the measurements of the hardware performance counter.

In the chapters 5 and 6 the performance counters of the ARM Cortex A53 have been used to measure the behavior of Linux-based software applications. The measures provided by the hardware performance counters are impacted by the behavior of Linux. It is impossible to separate the application and the operating system since the application is modified depending on the operating system executing it. Then, it is difficult to interpret the values of the performance counters measurements. For example, when measuring a certain number of cache refills it is impossible to dissociate the number of refills made by the application code and the one made by the operating system code on behalf of the application. Accessing a shared resource, opening a file, or calling a system call are operations made by the operating system for the application. But the application developer cannot modify the number of cache refills of these operations because it depends mostly on the internal state of the operating system. Moreover, the interference generated by these operations is not the same for

each call since it is related to the internal state of the operating system.

Finally, the hardware performance counters measurements can be used to measure the differences between two execution of the same execution. But to have a better understanding of where the interference are coming from it is necessary to distinguish interferences that are due to the internal state of the operating system et the ones that are related to the application code. Modelization along with performance counters measurements can help to make this distinction.

7.2 Linux configurations

7.2.1 SCHED_DEADLINE configuration

The SCHED_DEADLINE policy is used to add a period and a deadline to processes. As explained in the section 2.4 the SCHED_DEADLINE policy needs to be configured to be functional. The policy does not guess the period and deadline that the process requires. The three parameters needed to configure this scheduling policy modify completely the behavior of the scheduler. The experiment results presented in the chapter 6 show that the configuration of this deadline policy can induce a lot of deadline misses. In particular, the "runtime" parameter has a lot of impacts on the behavior of the scheduler with the SCHED_DEADLINE policy. If this parameter is smaller than the WCET of the application it can induce deadline misses and on the contrary, if it is greater than the WCET of the application it means that the CPU time is "reserved" for nothing.

Finding the WCET of a given application is a well-known problem in the literature and it is very hard to have a precise WCET. Moreover, in the scope of this thesis, the WCET of the AOCS process not only depends on the AOCS code but also on the Linux operating system. Due to the non-reproducibility of the Linux behavior between two different executions, it seems impossible to find the precise WCET of the AOCS process. Therefore, the best runtime parameter to configure SCHED_DEADLINE properly will be necessarily greater than the WCET by taking a margin on the observed greatest execution time.

Finally, SCHED_DEADLINE is useful in a real-time periodic environment but its configuration is complex and depends on how much information on the process there is.

7.2.2 Minimal Linux image

Linux is made of millions of lines of code and provides an huge amount of possibilities thanks to its number of features. In an embedded context the philosophy is to use

just what is necessary. For Linux, it means for example that it is possible to free some storage space by removing the graphical environment which will not be used in a satellite. This means that the libraries used by applications running on top of Linux have to be known.

It might be tempting to try to find the lightest Linux distribution, however, it is not necessarily a good idea. Reducing the size of the Linux distribution means deleting features of Linux. While it is clear that the graphical environment is not needed in the space context it seems more complicated to understand which specific features are needed or not in a specific context. For example, the chapter 5 shows that some time synchronization services were necessary to ensure the reliability of the scheduler with a high period.

Finally, a lot of work is necessary to find out which services are used or not in a specific context. First, an analysis of which libraries are used by the applications needs to be done. Second, it is necessary to find out which services are useful for the applications and which ones are impacting the processes.

7.2.3 PREEMPT-RT patch for space context

The main goal of the patch PREEMPT-RT is to enhance the real-time behavior of Linux. The real-time behavior is characterized by the number of deadline misses and respect for the period.

In the chapter 5 it has been shown that the PREEMPT-RT patch does not have an impact on the measurement results. To understand why the patch does not have an impact on the execution time it is necessary to look at the chapter 6 experiments. In these experiments, the number of executed steps of the AOCS process has been measured. As explained in the chapter 6, the number of executed steps is a good approximation of the number of deadline misses. These measures showed that there are most of the time no deadline misses. The only case where deadlines are missed is due to a configuration problem of the SCHED_DEADLINE policy as has been explained in the chapter 6. Therefore, adding the PREEMPT-RT patch will not have a huge impact on the measured performance of the AOCS process. Note that the experiments presented in the chapter 5 are not designed to test the PREEMPT_RT patch. In particular, the usecase process does not have short deadline and there are not too much interruptions.

Chapter 8

Conclusion and perspectives

8.1 Conclusion on the contribution

This thesis aims at using Linux on multicore platforms in a space environment.

The chapter 4 presents an interference modelling that allows identifying and counting memory interferences of a given set of applications. Made with timed automate, this modelling provides an accurate count of memory interference and can be connected to an academic method described in [17].

The chapter 5 provides experiments on an AOCS application running on Linux on a Zynq Ultrascale+ SoC. The experiments results showed the impacts of Linux on an AOCS application. More specifically, it has been proved that using the same files for different concurrent applications has better performances than using multiple files. It is an example of a positive impact of the operating system. Conversely, systemd services have a negative impact on the AOCS application. It has been showed that this impact is characterised by the increases of L2 cache refills. But the AOCS application is not the only part to be impacted, in fact the Linux scheduler does not behave the same when certain services are disabled. It results in a drift of the processes wake up date.

The chapter 6 aims at proposing experiments on isolation mechanisms inside Linux. The goal was to evaluate an isolation mechanism with different scheduling policies using two types of attackers processes. The experiments results showed that the isolation mechanism do not have a negative impact on the victim process. Also, when used with a custom scheduling policy (presented in [76]), it provides better performances with a combination of advantages from SCHED_DEADLINE and SCHED_RR.

Finally, the chapter 7 provides a synthesis on the experiments results and methodological recommandations on the use of Linux in a space environment.

8.2 Limitations

This thesis has potential limitations.

First of all, only one usecase application is used in all the experiments. The experiments could be remade with others space applications.

Second of all, due to the non reproducibility of the Linux behavior, the number of repetitions of the experiments might be increased to get more stable results. In the chapter 6, each experiment has been repeated 50 times. However, this number is not enough to have satisfactory distribution but allows having a reasonable experiment duration.

Finally, the benchmarks loads do not exactly do what they are expected to. In fact, these benchmarks are making memory accesses but not directly L2 cache access. There is no instruction that allow to access the shared cache directly. Thus, when two different benchmarks load are making the same amount of memory accesses at different time, it does not imply that the impact on the L2 cache will be the same.

8.3 Future work

Based on the works of this thesis, this section gives potential future works.

8.3.1 Enhance the models

The modelling presented in the chapter 4 is not complete. There are two major features that should be add in the future.

First of all, there is no time in the models. By adding time it would be easier to characterize the impacts on the execution time of a process and not only counting interference. However, the number of cycles taken by a specific instruction is not constant in modern CPU architecture. Thus it is impossible to know how much time an instruction will take to be executed. It exists at least two solutions to resolve this problem. The first one is to use the worst number of cycles taken by a specific instruction. This solution will give an upper bound of the execution time of the process. The second solution is to use an average number of cycles taken by each instruction, but this solution will not be completely accurate all the time.

The second feature that can be added is the operating system. In fact, in the current version of the modelling, the operating system is not taken into account. This is due to the fact that an operating system such as Linux is too huge to be able to modelize it by reading its code. Another aspect of the problem is that the operating system have a special role on the platform. Part of it can be seen as applications running on a core alongside other processes. This is the case for the systemd services

for example. But another part of the operating system make the connection between the application level and the hardware such as system calls for example. Thus, the impact of the operating system is not just an overhead for each other process. In fact, the more applications running, the more the operating system will be executed. This will have an impact on the other applications as well.

To resolve this problem in the future, another approach can be used. Rather than making a static model of the operating system from reading his code, it could be possible to make a model that only apply impact on the other processes. In other words, the operating system would be an automaton that will impact the counts of interference of the partition automata.

The question is then to find the impacts of the operating system. To do so, a statistical approach can be applied. Using experiments such as the ones presented in the chapters 5 and 6 allows retrieving information on the impacts of the operating system on other processes. This impact can either be an average one by running different kind of applications with different usecases and context and measuring the average impact. Or it is possible to characterise it more precisely by measuring the impact in a specific context. The experiment approach used in the chapter 6 can be useful since it provides a method to explore different applications configuration. With this method, the operating system automaton will have multiple states corresponding to different usecase configuration.

8.3.2 Reproducibility of Linux behavior

As seen previously in this thesis, the reproducibility of the Linux behavior is a problem. It would be possible to work on this particular aspect of Linux to find out if it is possible to reproduce the behavior of an experiment multiple times in a row. This would require to control as precisely as possible the experiment environment. Also, an analysis of the Linux services that have a part of randomness can be done. For example, a service that will wait for a response through a network interface will decrease the reproducibility.

8.3.3 Application architecture

The application usecase presented in this thesis was taken as is without making any modification. The experiments approach of this thesis could be used to retrieve insights on the applications architecture.

Part III

Appendices

Appendix A

Résumé Long Français

A.1 Introduction

A.1.1 Contexte

Contraintes

Les contraintes de l'environnement spatial sont les plus grandes contraintes qui s'appliquent à un engin spatial. Premièrement, un satellite en orbite est inaccessible par l'homme contrairement à d'autres systèmes embarqués (comme une voiture ou un avion par exemple) ce qui nécessite de concevoir les systèmes spatiaux avec une grande fiabilité. Deuxièmement, un satellite se déplace en dehors de l'atmosphère terrestre et il est donc soumis à différents types de radiations qui peuvent altérer ou même détruire les composants électroniques. La gestion de la température à l'intérieur du satellite est aussi une contrainte forte, il y a un énorme écart de température entre la face exposé au soleil et celle à l'ombre. Les dernières contraintes sont d'ordre mécaniques et interviennent durant le lancement et ses maneuvres en orbite.

Technologies actuelles

La plupart des technologies utilisées dans le domain spatial aujourd'hui sont soit conçues, soit customisées pour ce domaine. De plus, ces technologies sont très anciennes et n'ont pas les performances de celles utilisées au sol. On peut citer par exemple le MIL-STD-1553 qui est un bus de données de l'armée Américaine des années 70 et qui est limité à un débit de 1 Mbps, ou encore le LEON2 qui est un processeur Européen et qui tourne à une fréquence de quelques dizaines de MHz.

Différences entre plateforme et charge utile

Un satellite est séparé deux parties. La plateforme est responsable du support du satellite tandis que la charge utile s'occupe de la mission. Les deux parties sont aussi importantes bien que la plateforme soit plus critique car elle peut engendrer des dégats sur l'environnement.

A.1.2 Approche

Depuis plusieurs années l'industrie spatiale pousse pour une réduction des coûts de fabrication et cette transformation passe aussi par la modification des ordinateurs et logiciels de bords. La première idée pour réduire le coût de l'avionique de bord est de réduire le nombre d'ordinateurs à bord en intégrant plusieurs logiciels sur la même plateforme matérielle. La deuxième idée est d'utilisée des processeurs du marché qui ne sont donc pas fabriqués exclusivement pour l'industrie spatiale (par exemple le Zynq Ultrascale + de Xilinx). Ces transformation induit le passage d'une architecture mono-coeur à une architecture multicoeur. La dernière possibilité de réduction de coût est l'utilisation de système d'exploitation commun comme Linux. La complexité réside dans l'adapation d'un système d'exploitation classique à un contexte spatiale, critique et temps-réels.

A.1.3 Organisation du manuscrit

Ce manuscrit est organisée en deux parties: le contexte at la contribution. La partie contribution s'articule autour de 4 chapitres: une modélisation à base d'automates temporisées, des expériences sr Linux, des expériences sur les cgroups et une synthèse d'analyse et des recommandations méthodologiques.

A.2 Contexte

A.2.1 Architecture matérielle

Cette section décrit les différents composants de bases de l'architecture d'un ordinateur moderne.

Description du processeur

L'élément centrale est le CPU qui est responsable de l'éxecution des instructions machines. Il est composé de plusieurs composants internes lui permettant de faire des calculs mais aussi d'accèder à la mémoire ou encore de surveiller le comportement

de la couche matérielle. Les CPUs modernes sont organisés en pipeline pour pouvoir paralléliser l'exécution des instructions. Cette thèse se focalise sur les compteurs de performances du processeur qui permettent de mesurer les actions du processeur.

Mémoire

Généralement, la mémoire principal d'un ordinateur est faite de RAM mais dans les systèmes embarqués on utilise plus souvent de la mémoire flash. La mémoire est divisée en unité addressable qui sont accessible via une adresse physique.

Caches

Un cache est un type de mémoire qui est plus proche du processeur en termes de temps d'accés mais qui est aussi beaucoup plus petit que la mémoire principale. Lorsque qu'un accés à la mémoire est nécessaire, le processeur vérifie dans un premier temps si cette donnée n'est pas disponible dans le cache. Sinon, il va chercher la donnée dans la mémoire et la copie aussi dans le cache pour un futur accés. Actuellement, il n'y a pas de cache partagé dans les ordinateurs à bord des satellites mais cela va changer dans les futures années. Partager un cache entre plusieurs coeurs peut avoir des effets néfastes puisque les données utilisées par un coeur peuvent être remplacées par celle d'un autre coeur.

A.2.2 Système d'exploitation

Le système d'exploitation est le logiciel qui gère la plateforme matérielle, les ressources logicielles et les services proposées aux applications.

Gestion des procesus

Le premier rôle d'un système d'exploitation est de gérer les processus (logiciel en exécution). Pour ce faire, le système d'exploitation conserve des informations sur chaque processus tout au long de leur cycle de vie. Parmis ces informations, on peut citer : l'identifiant du processus, l'adresse de l'instruction exécutée par le processus, la table des adresses de pages mémoires et l'état des registres du processeurs. De plus, le système d'exploitation est responsable de l'ordonnancement des processus, il doit donc effectuer les changements de contexte entre processus. L'algorithme d'ordonnancement d'un système d'exploitation et un facteur différenciant par rapport aux autres système d'exploitation.

Ordonnancement

Le système d'exploitation doit choisir quel processus s'exécute à quel moment, cette partie s'appelle l'odonnanceur. Un processus peut être dans plusieurs états pour l'ordonnanceur. L'état *prêt* signifie que le processus peut s'exécuter. L'état *en attente* signifie que le processus attends un événement pour pouvoir s'éxecuter. L'état *exécution* est réservé aux processus qui sont en train de s'éxecuter sur un coeur. L'ordonnaceur utilise plusieurs files pour stocker les processus et gérer leur différentes priorités.

Mémoire virtuelle

Le dernier aspect important du système d'exploitation est la mémoire virtuelle. Elle permet aux développeurs de logiciels de ne pas se soucier de l'adressage de leur code. En effet, la mémoire n'est pas infinie et plusieurs logiciels pourrait décider d'utiliser la même addresse physique. Il y a donc une table de page pour chaque processus qui définit la traduction entre une adresse virtuelle (propre au processus) et une adresse physique (propre à la mémoire). Généralement, il n'y a pas de mémoire virtuelle dans le contexte embarqué, ce qui rend le développement d'applications plus compliqué.

A.2.3 Virtualisation

La virtualisation est un mécanisme utilisé pour créer un ordinateur virtuel. Dans le contexte de cette thèse, elle peut être utilisée pour isoler différents processus.

Machines virtuelles

Le premier type de virtualisation est appelé: machine virtuelle, elle peut être exécutée par un système d'exploitation ou un hyperviseur. Le rôle d'une machine virtuelle est de virtualiser les composants matériels, de sorte que les processus s'exécutant dans cette machine virtuelle aient accés aux mêmes fonctionnalités que s'ils s'éxecutaient sur la plateforme matérielle directement. Un système d'exploitation peut s'exécuter à l'intérieur d'une machine virtuelle sans nécessairement avoir besoin de modifications.

Hyperviseurs

Un hyperviseur est un logiciel conçu pour exécuter des machines virtuelles. Il s'exécute soit au-dessus d'un système d'exploitation, soit directement sur la plateforme matérielle. Le rôle de l'hyperviseur est de créer des partitions pouvant recevoir des systèmes d'exploitation ou des logiciels bare-metal. L'inconvénient d'un hyperviseur dans le

contexte de cette thèse est que son ordonnancement est statique, ce qui nécessite une connaissance à priori du système complet.

Conteneurs

Un conteneur est un type de virtualisation qui ne virtualise que la couche logiciel doit les services du système d'exploitation. Le conteneur fournit une isolation plus faible que celle d'un hyperviseur et ne possède pas son propre ordonnanceur. Dans le contexte de cette thèse, les conteneurs sont étudiés dans une optique de réduction des impacts mutuels de différents logiciels s'exécutant sur la même plateforme matérielle.

A.2.4 Linux

Avantages

Linux est un système d'exploitation crée dans les années 90 et utilisé par la majorité des développeurs du monde. Les avantages principaux de Linux dans le contexte de cette thèse sont sa réutilisabilité et sa philosophie open-source.

Linux dans un contexte critique

Linux n'est pas conçu pour hebergé des applications critiques mais les industries critiquess s'y intéressent de plus en plus ([5], [6] et [7]). Le problème majeur concerne la qualification de Linux qui n'est aujourd'hui pas envisageable compte tenu des normes de qualifications actuelles dans ces différentes industries.

Caractéristiques

Systemd Systemd est une suite de logiciel qui ne fait pas partie du noyau Linux mais qui est responsable du lancement du premier processus à l'initialisation du système. Systemd est aussi responsable de la configuration des démons logiciels et fournit une interface utilisateur appelée systemctl. Parmis les démons les plus connus de systemd on peut citer: journald pour l'enregistrement des évènements, logind qui s'occupe de la connexion des utilisateurs au système ou encore netword qui gère les interfaces réseaux.

Namespace Les namespace sont des fonctionnalités de Linux qui permettent d'isoler certaines ressources entre différents processus. Lorsque plusieurs processus sont dans le même namespace, ils ont accès aux mêmes ressources partagés en fonction du type du namespace. Par exemple, le namespace *mount* isole les point de montage de Linux

et le namespace *network* isole les interfacs réseaux. Une ressource qui est crée à l'intérieur d'un namespace, n'est visible que par les processus de ce namespace.

Cgroups Cgroup est un mécanisme de Linux qui permet de limiter, surveiller et isoler l'accès à une ressources à un groupe de procesus. Dans le cadre de cette thèse c'est la notion de limite qui est la plus intéressante. Par exemple, un cgroup peut limiter l'usage du processeur à un groupe de process.

Possibilités temps-réels

Patch PREEMPT-RT Le patch PREEMPT-RT est un patch de Linux qui permet une adaptation de Linux à un contexte temps-réel. La plus grosse modification du patch est l'ajout de zone preemptible dans le kernel Linux. Selon le niveau d'activation du patch il y a plus moins de zone preemptible dans le kernel. Il ajoute aussi des mécanismes d'héritage de priorité et d'interruptions threadés au kernel. Ces mécanismes ont pour but de rendre le système plus disponible pour les applications temps-réel.

Police d'ordonnancement: SCHED_DEADLINE Les processus temps-réel se caractérisent par leur période et échéance, c'est pourquoi une politique d'ordonnancement appelée SCHED_DEADLINE est disponible dans le kernel Linux. Cette politique est basée sur l'algorithme GEDF et utilise aussi l'algorithme CBS pour gérer la bande passante. La configuration d'un processus SCHED_DEADLINE nécessite trois paramètre: runtime, deadline et period (voir Figure A.1).

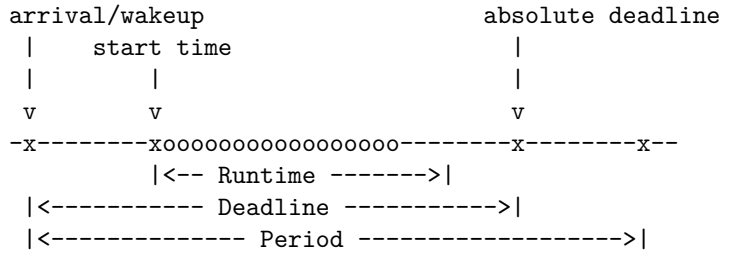

Figure A.1: Parameters of a SCHED_DEADLINE Linux process (man sched(7))

Cet algorithme d'ordonnancement éli le processus avec la plus courte écheance. L'échéance utilisée dans l'algorithme est une échéance calculée à partir du paramètre runtime et de l'état de l'exécution en cours. Le but de cette démarche est d'éviter qu'un processus prenne plus de temps que prévu à l'exécution.

Police d'ordonnancement: SCHED_RR Une autre politique d'ordonnancement temps-réel disponible dans le kernel Linux est : SCHED_RR. Cette politique est basée sur un algorithme round robin avec différentes files de priorités. De plus, chaque processus à un quantum de temps à ne pas dépasser avant d'être remis à la fin de sa file.

A.3 Etat de l'art

A.3.1 Processeurs multi-coeur

L'utilisation de processeurs multi-coeur pour des applications critiques nécessite d'étudier les phénomènes d'interférence.

Processeurs multi-coeur dans le domaine spatial

Dans le domaine aerospatial, l'utilisation de processeurs multi-coeur est régi par les normes de qualification ([10] et [9]) adaptées aux processeurs modernes dans [11]. Le projet Phylog ([13] et [14]) propose des méthodes d'identification des interférences.

Analyses temporelles

L'étude proposée par C. Maiza *et al.* ([15]) décrit les différents types de travaux sur les vérifications temporels pour des architecture multi-coeur. Ce manuscrit se place dans le contexte de connaissance à priori des architectures, décrit dans cette étude. Les travaux présentés dans [16] (approche similaire dans [18]) propose une analyse par model-checking du WCET sur multi-coeur, c'est cette même idée qui est reprise dans le chapitre 4.

Approche expérimentale

Plusieurs travaux adoptent une approche plus expérimentale du problème d'interférence multi-coeur. [19, 20, 21] utilisent des micro-benchmarks pour caractériser les interférences. [22, 23, 27] se focalisent sur les techniques de mesures et [24, 25] proposent des approches de profiling d'applications.

Réduction d'interférences

La deuxième catégorie de travaux est centrée sur les méthodes de réduction d'interférences. [31, 32] présentent des modèles sans contention pour exécuter des applications sur des multi-coeurs. [33, 34] raccrochent le problème d'interférences aux middlewares, qui joue un rôle important dans leur occurence.

A.3.2 Intégration de logiciels

Ordonnancement

La réduction du nombre d'interférences passe par l'intégration de logiciels sur une même plateforme matérielle, qui est lié à la manière dont les applications sont ordonnancées. [36] présente un ordonnanceur capable de réduire la latence pour le temps-réel dur. [37, 38] présentent un outil d'analyse static pour l'ordonnancement.

Partitionnement d'avioniques

Le partitionnemment pour les avioniques est un sujet de longue date, introduit en 2000 dans [41]. Au coeur de cette problématique se trouve la notion d'IMA (integrated modular avionics) dont l'interface est décrite dans [43]. Cette interface a été étudiée par [44, 45] pour obtenir un partitionnement robuste et pouvoir le simuler. Finalement, l'IMA a été adapté pour le domaine spatial par l'ESA ([46, 47]).

Virtualisation

La virtualisation est un bon moyen d'intégrer différentes applications tout en réduisant les interférences ([48]). Il existe beaucoup d'hypersiveurs avec chacun des caractéristiques différentes ([50, 51, 52, 53]). Mais d'autres solutions existent, comme par exemple la conteneurisation ([56]) ou encore le cache coloring ([57]).

A.3.3 Linux

L'interêt pour Linux dans le domaine spatial n'a fait qu'accroître ces dernières années ([59]). La NASA l'a même utilisé pour aller sur mars ([60]).

Temps réel

La question subsiste toujours de savoir si Linux peut tenir les contraintes temps-réel. Différents travaux se sont focalisés sur l'étude du patch PREEMPT-RT ([63, 64, 65]). [66] propose une approche expérimentale pour étudier le comportement de Linux dans un contexte temps-réel.

Intégration de logiciels sur des plateformes Linux

[69] présente l'une des premières implémentations d'une amélioration EDF pour l'ordonnanceur Linux. Plus tard, une amélioration de la nouvelle politique SCHED_DEADLINE est présentée dans [70]. Plus récemment, Lelli *et al.* donne un retour sur la conception

et l'implémentation de SCHED_DEADLINE dans [71]. SCHED_DEADLINE est au cœur des expérimentations présentées dans les chapitres 5 et 6.

De nombreuses études travaillent sur le partitionnement sous Linux. Dans [72], Obermaiser *et al.* évalue l'isolation temporelle et spatiale de Linux temps réel avec RTAI. Plus récemment, [73] se concentre sur la simultanéité des applications sous Linux lors de l'exécution sur des processeurs multicœurs. [74] et [75] s'intéressent aux interférences directement dans la mémoire. Ils proposent un mécanisme de limitation de la mémoire utilisant des cgroups Linux et un algorithme d'ordonnancement qui tiendra compte du comportement de la mémoire. Dans le même temps, [76] implémente un algorithme de planification en temps réel basé sur un conteneur à utiliser sous Linux.

A.4 Analyse et modélisation des interférences

A.4.1 Description de la méthode

Considérons un ensemble de tâches T_i hébergées par un MCP (à gauche de la figure). L'objectif de la méthode est de calculer une borne supérieure de la pénalité temporelle supplémentaire associée à chaque T_i due aux interférences se produisant dans l'architecture (sur la partie droite de la Figure 4.1). Pour cela, nous calculons une borne supérieure du nombre maximal d'interférences (noté IN_i) dont chaque T_i peut souffrir.

L'approche repose sur la méthode d'analyse TIPS proposée par Carle *et al.* dans [17] (sur la partie gauche de la Figure 4.1). Au moyen de l'analyse statique, Carle *et al.* montrent qu'il est possible d'extraire une séquence de segments temporels, comme celle montrée Figure 4.2, à partir du code binaire d'une tâche. Chaque segment est caractérisé par une durée (notée $d_{i,j}$ pour la tâche T_i et le segment j), et le nombre maximal de requêtes mémoire sortant du cœur et envoyées au bus (noté $\mu_{je,j}$). Ces requêtes correspondent à des opérations de chargement et de stockage qui ne sont pas *toujours frappées* dans le cache L1 du core hébergeant la tâche. Par exemple, dans la figure 4.2, T_1 n'effectue aucune requête mémoire dans le segment un et au plus trois requêtes dans le segment deux.

Comme le montre la case grise de la figure 4.1, le cœur de la contribution consiste en :

- (1) associant un automate UPPAAL pour modéliser le profil d'accès au bus de chaque tâche ;
- (2) modéliser les composants du MCP par un réseau d'automates synchronisés avec les automates des profils d'accès au bus;

- et (3) calculer par model-checking une borne supérieure du nombre d'interférences dont chaque tâche peut souffrir.

A.4.2 Définitions

Considérons une architecture MCP (représentée dans la Figure 4.3) conforme aux hypothèses de Carle *et al.* [17]. Ce processeur est composé de (1) deux cœurs (C_0 et C_1) possédant leur cache privé $L1$, (2) un cache partagé $L2$, (3) une mémoire DDR composée d'une banque B , et (4) un bus partagé permettant aux deux blocs centraux d'adresser à la fois le cache $L2$ et la DDR. Chaque cœur héberge une tâche appelée SP_0 (sur le cœur C_0) et SP_1 (sur le cœur C_1).

Supposons que SP_0 et SP_1 sont caractérisés par les profils représentés sur la figure 4.4. Ces profils sont périodiques (période = 30), et ils sont divisés en trois segments. Selon ces profils, SP_0 et SP_1 peuvent entrer en compétition pour accéder à la mémoire dans le premier segment (de 0 à 10). Dans ce segment, chaque tâche envoie au plus 2 requêtes mémoire. Dans les deux segments suivants, SP_0 ou SP_1 n'envoie aucune requête, laissant le chemin mémoire disponible pour l'autre tâche.

Les requêtes de SP_0 et SP_1 suivent le chemin suivant:

- $chemin(SP_0) = C_0 \rightarrow L1_0 \rightarrow Bus\ AHB \rightarrow L2 \rightarrow B$
- $chemin(SP_1) = C_1 \rightarrow L1_1 \rightarrow Bus\ AHB \rightarrow L2 \rightarrow B$

Ces deux chemins se croisent sur trois composants: *Bus AHB*, $L2$ et B. Les requêtes de SP_0 et SP_1 peuvent alors souffrir d'interférences dans ces trois composants.

Définitions d'interférences

Definition 5 (Composant de transport) *Un composant de transport est caractérisé par un état interne qui ne dépend que de la présence ou de l'absence d'une requête l'utilisant. Si une requête utilise le composant, alors il est "occupé". Sinon, il est "libre".*

Definition 6 (Composant de stockage) *Un composant de stockage est un composant dont l'état interne dépend des requêtes précédentes (y compris celle en cours le cas échéant).*

Definition 7 *Une interférence instantanée se produit chaque fois qu'au moins deux requêtes envoyées par deux tâches différentes entrent en collision sur le même composant de transport.*

Definition 8 *Une interférence retardée se produit lorsqu'une requête r émise par une tâche T utilise un composant de stockage dont l'état interne a été rendu non conforme à T par une autre tâche T'.*

A.4.3 Modélisation

Approche de modélisation

Pour modéliser les interférences dont peuvent souffrir les tâches, deux classes de modèles sont définies : les automates pour les composants matériels (cf. section 4.3.3), et les automates pour les tâches (cf. section 4.3.4). L'ensemble du système est alors construit en instanciant et en synchronisant ces automates. Une vue de haut niveau des interactions entre les automates est représentée dans la figure 4.11. Les automates logiciels sont horizontaux pour décrire le chemin du cœur à la mémoire (de gauche à droite) pour un accès mémoire donné. Les automates HW sont verticaux et représentent la chronologie des accès mémoire appartenant potentiellement à différentes tâches.

Si l'automate modélisant les composants HW (la partie verticale) est générique, ceux modélisant les tâches (les lignes horizontales) dépendent du processeur multi-cœur considéré. Ils doivent être générés à partir d'un modèle de haut niveau de la configuration du processeur (y compris l'allocation logicielle). Cette question (c'est-à-dire, comment générer la modélisation des automates des tâches) est hors de la portée de cette thèse.

Composants matériels

Le rôle des automates du composant matériel est de modéliser les réponses du composant aux requêtes émises par les tâches. Il détermine si une requête crée des interférences dans le composant et notifie l'automate logiciel de ce résultat.

Composant de transport

D'après la définition 5, un composant de transport est modélisé par l'automate de la figure 4.7. Il est composé de trois emplacements (*Free*, *Occupied* et *Check*) et de trois données internes (*waiting_queue*, *nb_elmt* et *bus_state*):

- *waiting_queue* est une liste interne contenant les identifiants des tâches en attente du composant.
- *nb_elmt* est le nombre de tâches actuellement en attente du composant (y compris la tâche qui l'utilise actuellement).

Storage component automata

Le composant de stockage génère des interférences retardées, c'est-à-dire qu'une tâche peut interférer avec une autre en modifiant l'état du composant. Les composants de stockage sont modélisés par l'automate de la figure 4.8. L'idée est de déterminer si

le composant réagit dans un délai favorable (cas normal), ou au contraire dans un délai défavorable (cas perturbateur) lors de la réception d'une requête. Cette notion de délai favorable est liée à la tâche demandant le composant. *Notez que dans cette modélisation il s'agit de "retards", mais ils ne sont jamais quantifiés. Il s'agit uniquement d'un état "favorable" à une tâche entraînant un retard NORMAL, et d'un état "défavorable" entraînant un retard INTERFERENCE. Ainsi, ce chapitre ne considère qu'une abstraction logique du temps. La conviction est qu'une telle abstraction est évolutive et qu'elle est suffisante pour l'analyse des interférences.*

Automates de taches

Un automate de tâche modélise le profil d'accès au bus de la tâche et le chemin des requêtes à travers les composants matériels du processeur. La figure 4.9 donne l'automate de la tâche SP_0. Il est composé de trois parties. La première (états *Request_bus*, *Result_bus* et *Wainting_bus*) modélise la réponse du bus à la requête. La deuxième partie (états *Request_L2*, *Result_L2* et *Wainting_L2*) modélise la réponse du cache L2. Et la dernière partie modélise la réponse de la mémoire DDR.

L'état initial de l'automate est un emplacement *Idle*. Il attend un événement de début (ou de fin) de l'ordonnanceur (c'est-à-dire l'automate implémentant la séquence du segment du profil de tâche). Lors de la réception d'un événement de démarrage, le compteur local *nb_req* est mis à zéro. Et l'automate atteint *Request_bus*

Automate de l'ordonnanceur

Le dernier automate modélise les séquences de segment des profils de tâches. Par exemple, les profils de SP_0 et SP_1 représentés sur la figure 4.4 sont modélisés par l'automate de la figure 4.10. À partir de l'état initial, l'ordonnanceur démarre le premier segment de SP_0 et SP_1. Ensuite, il attend une occurrence de *synchro* envoyée par SP_0 et SP_1 à la fin de leur segment. A leur réception, l'ordonnanceur notifie la fin du premier segment et démarre le second. Et ainsi de suite jusqu'au dernier segment (quand $num\_segt == nb\_segt$).

A.4.4 Analyse des interférences

Pour déterminer quand une interférence se produit, chaque automate de composant est instrumenté avec un algorithme pour décider si la tâche accédant au composant souffre ou non d'interférence. Ces algorithmes dépendent du type de composant considéré.

Composant de transport. Compter les interférences dans les composants de transport est très simple : il y a une interférence *si et seulement si* le composant est occupé

et une autre requête l'attend.

Composant de stockage. Pour les composants de stockage, l'algorithme de décision est plus complexe. Pour déterminer si l'accès depuis la tâche X est affecté par l'accès depuis la tâche Y, il ne suffit pas de maintenir l'état réel du composant mais aussi l'état du composant pour chaque tâche comme si la tâche était seule. Une telle vue "privée" du composant ne prête attention qu'aux accès depuis la tâche à laquelle il est associé. Cela signifie que l'accès de la tâche X au composant de stockage affecte à la fois la vue "réelle" et sa vue "privée" du composant, mais pas la vue "privée" associée à la tâche Y. Ainsi, pour chaque accès à un composant de stockage, il est possible de déterminer s'il y a interférence ou non en comparant la vue privée de la tâche avec la vue réelle du composant de stockage. S'il y a une différence entre ces deux points de vue, cela signifie qu'une interférence s'est produite. Dans l'exemple d'un cache mappé direct L2 avec 16.384 lignes de 32 octets, cet algorithme fonctionne sur : (1) un tableau *cache_array* contenant 16 384 listes de 32 octets représentant l'état courant du cache L2 ; (2) un tableau similaire *task_i_array* pour chaque tâche i représentant l'état du cache si la tâche était seule.

Analyse des interférences par model-checking. Pour calculer une borne supérieure du nombre d'interférences subies par chaque tâche, dans chaque composant pour chaque segment, le vérificateur de modèle UPPAAL est utilisé. Notons par exemple *nb_interf_L2[0,*SP_0*]* le nombre d'interférences calculé pour le premier segment de SP_0 dans le cache L2. UPPAAL montre que $\forall\Box\neg(nb\_interferences\_L2[0,SP_0] > 3)$, c'est-à-dire qu'il n'est pas possible que *nb_interf_L2_SP0* soit supérieur à 3. Ce calcul prend moins d'une seconde (UPPAAL version 4.1.24 fonctionnant sur un Apple M1 Pro 3,2 GHz avec 32 Go de DDR5).

A.5 Expériences sur Linux

A.5.1 Cas d'usage

SCAO: système de contrôle d'attitude et d'orbite

Dans un satellite, le logiciel SCAO est chargé de contrôler l'attitude, c'est-à-dire l'orientation et l'orbite de l'engin spatial. Les entrées de l'algorithme SCAO sont les coordonnées angulaires spatiales, les vitesses et les accélérations provenant de capteurs. L'AOCS est un programme infini qui fonctionnera indéfiniment en répétant un cycle d'une période spécifique.

L'application logicielle SCAO implémentant l'algorithme est un processus Linux périodique fonctionnant à une fréquence de 8Hz. Il prend ses entrées d'un tube et écrit ses sorties dans un autre tube. Ce logiciel se compose de 16000 étapes de calcul.

Le processus AOCS a deux processus enfants décrits ci-dessous (voir figure 5.2):

- Un processus Linux non périodique lit les entrées du fichier d'entrée et les écrit dans le tube d'entrée.

- Un autre processus Linux non périodique lit les sorties du tube de sortie et les compare aux sorties attendues du fichier de sortie.

AOCS a été choisi comme cas d'utilisation car il possède des propriétés intéressantes. Tout d'abord, il est représentatif d'une application utilisée dans l'environnement spatial puisque tous les engins spatiaux disposent du logiciel AOCS. Deuxièmement, son exécution est très complète, il possède une partie calcul mais aussi d'autres parties qui ont besoin d'accéder à la mémoire aussi bien en lecture qu'en écriture. Enfin, il s'agit d'une application périodique avec des étapes très différentes, ce qui signifie que chaque période exécutera quelque chose de différent de la précédente.

Plateforme matérielle

Les expériences présentées ici sont réalisées sur un Zynq Ultrascale + fabriqué par Xilinx qui est l'une des plates-formes matérielles COTS envisagées par l'industrie spatiale pour la prochaine génération de satellites. Le schéma de principe de l'architecture Zynq Ultrascale + est présenté dans la figure 5.3.

Le SoC sélectionné dispose de deux unités de traitement principales (application et temps réel) ainsi que d'une PMU (unité de gestion de plateforme), d'une CSU (unité de configuration et de sécurité) et d'un GPU (unité de traitement graphique). Le Zynq ultrascale+ embarque également un FPGA accessible par toutes les E/S et unités de traitement.

En outre, l'unité de traitement des applications fonctionnera sous Linux. Cette unité de traitement est composée d'un processeur ARM Cortex-A53 avec une fréquence allant jusqu'à 1,5 GHz, qui implémente l'architecture ARMv8-A. Il s'agit d'un quad-core, avec un cache L1 d'instructions et de données ainsi qu'un cache L2 partagé et une MMU (unité de gestion de la mémoire). Le cache L2 est un cache associatif à 16 voies de 1 Mo (1024 Kio) avec ECC partagé entre les processeurs. Le cache L2 est unifié, c'est-à-dire qu'il contient à la fois des données et des instructions du système de mémoire L1. Le cache d'instructions L1 est un cache associatif à 2 voies de 32 Ko avec ECC indépendant pour chaque processeur. Il contient 500 lignes de 64 octets. Le cache de données L1 est un cache associatif à 4 voies de 32 Ko avec ECC indépendant pour chaque CPU. Il peut contenir 500 lignes de 64 octets.

A.5.2 Methodologie

Protocole

Pour chaque métrique, nous voulons récupérer deux mesures : une mesure avant d'appeler la fonction de calcul et une mesure après. Ensuite, la différence entre ces deux mesures est faite pour obtenir la valeur de la métrique lors de l'exécution de la fonction de calcul. Notez que les mesures ne sont prises que sur le processus AOCS et non sur les processus I/Os. Cela signifie que l'écriture et la lecture des pipes ne sont pas directement prises en compte dans les mesures. De plus, toutes les mesures sont prises sur un seul processus, qui est considéré comme la victime de l'expérience. Tous les autres processus (utilisateur et noyau) et toutes les activités du noyau sont considérés comme les attaquants du système. La seule mesure prise avant de lire le tuyau d'entrée provient de l'horloge car une métrique importante à analyser est la date de réveil du processus. Le code des mesures est montré dans la figure 5.4.

De nombreuses données sont collectées à partir des compteurs de performance, principalement sur les événements mémoire matériels (accès au bus, accès au cache, recharges de cache, ...). Toutes ces expérimentations tournent sur une distribution Linux "light" faite avec Yocto. Le noyau Linux utilisé est une version 5.4 du noyau Linux Xilinx trouvé sur le Github officiel Xilinx [80]. Selon les recommandations Xilinx ([81]), la version Yocto Zeus du dépôt officiel Yocto ([82]) a été utilisée.

Impacts des mesures

Chaque mesure décrite dans le cadre de ces expériences est prise à l'intérieur d'une période d'exécution. Cela signifie que chaque code en dehors de la boucle périodique n'est pas pris en compte dans les résultats de mesure. En particulier, les phases de chargement et d'initialisation de l'application n'entrent pas dans le champ des mesures.

Les mesures présentées ici proviennent de deux sources différentes. Les horloges, gérées par le système d'exploitation et servant à mesurer le temps d'exécution, le temps de réveil relatif et le temps de traitement c'est-à-dire le temps passé sur le CPU entre le début et la fin d'un cycle. Récupérer la valeur de ces horloges prend approximativement le même temps à chaque fois. L'impact de la récupération d'une valeur temporelle est alors constant. La deuxième source pour les mesures sont les compteurs de performance. L'activation de ces compteurs ne modifie pas l'exécution du code. Récupérer la valeur de ces compteurs implique de lire des registres CPU qui ne sont pas dans la mémoire principale. Comme pour les horloges, les impacts de la récupération des valeurs des compteurs de performance sont constants dans le temps. Pour les deux mesures, les impacts des sources sur la récupération des données sont constants et n'affectent pas la mémoire principale, c'est-à-dire qu'aucun accès à

la mémoire n'est effectué. Pour être analysées, ces données récupérées doivent être stockées et mises à disposition après l'expérience. La procédure de stockage peut induire d'énormes impacts sur les mesures, car les résultats doivent être stockés dans la mémoire principale. Pour réduire les effets de ces accès mémoire sur l'expérience, les données mesurées sont placées dans des variables locales placées sur la pile du processus puis copiées dans la mémoire en dehors de la section mesurée. Cette méthode générera moins d'impacts sur le code exécuté mais modifiera un peu l'état matériel de la mémoire.

Enfin, les impacts de la récupération des mesures sont constants et le stockage des résultats se fait en dehors de la boucle périodique. Par conséquent, pour décider si les impacts des mesures sont négligeables dans ce contexte, nous devons mesurer la partie constante des impacts. La figure 5.5 montre les résultats lorsqu'aucun code n'est exécuté entre deux points de mesure. Ce graphique représente le temps d'exécution induit par la mesure elle-même qui est compris entre $7,5\mu s$ et $9,0\mu s$. Comme le WCET du processus AOCS est d'environ 1 ms, nous pouvons conclure que l'impact des mesures est négligeable.

A.5.3 Expériences

Mono fichier versus multi fichiers

Dans cette expérience, deux groupes d'applications ont été créés sur la base du code SCAO. Le premier groupe est composé d'applications SCAO utilisant le même fichier exécutable et les mêmes fichiers d'E/S. Pour le second groupe, chaque exécutable et fichiers d'E/S sont dupliqués. Pour chaque groupe, la même expérience a été réalisée et est décrite comme suit. L'expérience commence par l'exécution d'une application SCAO et par la mesure du temps d'exécution, du temps de traitement, du temps de réveil et des compteurs de performances liés à la mémoire, tels que le remplissage du cache L2 ou les accès au bus. Les résultats de cette première étape sont utilisés à des fins de comparaison et doivent être équivalents dans les deux groupes. La prochaine étape de l'expérience consiste à augmenter le nombre d'exécutions parallèles. Le nombre maximum d'applications lancées est de 30, ce qui représente 90 processus sur le système d'exploitation (30 processus AOCS, 30 processus d'entrée et 30 processus de sortie). Dans le cadre de cette expérience, les mesures sont prises sur un seul processus AOCS considéré comme victime alors que tous les autres processus AOCS et I/Os sont considérés comme attaquants.

La figure 5.6 montre le temps d'exécution moyen du processus AOCS lorsque d'autres processus AOCS sont exécutés en parallèle. Il y a une courbe pour le premier groupe et l'autre courbe pour l'autre groupe. Les résultats de cette expérience ont montré que lorsque les fichiers sont les mêmes, les performances sont meilleures en

moyenne. La pire performance est meilleure que pour l'autre groupe de processus. Dans ces courbes, on peut remarquer des points aberrants, par exemple le point 3 de la courbe mono file. Dans cette expérience, l'accent est mis sur les tendances de la courbe et non sur les valeurs particulières. Ainsi, la non-reproductibilité de Linux n'est pas vraiment un problème ici.

La modification de l'état d'un périphérique matériel à mémoire partagée est un autre type d'interférence. La figure 5.7 montre le nombre de remplissages du cache de données L2, c'est-à-dire le nombre de fois qu'une ligne de cache doit être chargée depuis la mémoire principale, à partir des deux mêmes groupes de processus. Quant à la courbe précédente, elle a été obtenue en exécutant les processus SCAO en parallèle pendant 16 000 pas à une fréquence de 8Hz. La courbe est similaire à la précédente, mais sa forme autour du point de pivot est plus lisse.

Maintenant que la forme des courbes est expliquée, concentrons-nous sur les différences entre les deux groupes de processus. Dans les deux graphiques, il y a une différence non négligeable entre la courbe mono-fichier et celle multi-fichiers. La courbe de fichier mono montre moins de remplissage de cache L2 avec un maximum de 1000 tandis que le maximum de remplissage de cache de données pour la courbe multi-fichiers dépasse 3000. Cela signifie qu'en utilisant les mêmes fichiers, il y a une meilleure réutilisabilité des données qui peuvent être trouvées dans les caches. Mais l'espace d'adressage des différents processus sous Linux est complètement séparé. En d'autres termes, les données du processus A ne peuvent pas être partagées avec le processus B, et donc le processus B ne peut pas utiliser les données du cache déjà chargées par le processus A. Cela soulève la question de l'explication des résultats précédents.

Pour conclure, l'utilisation des mêmes fichiers à la fois comme exécutables et entrées/sorties de données pour des processus parallèles modifie radicalement les performances observées. Cette section a montré que l'impact de l'utilisation ou non des mêmes fichiers dans l'exécution parallèle de processus est impacté par deux phénomènes distincts.

Pics periodiques d'interférences

Lors de la première expérience, une observation a été faite que lorsqu'un processus SCAO était exécuté seul sur le système, son temps d'exécution était impacté par des interférences. En fait, des pics périodiques sont présents dans la courbe de temps d'exécution comme le montre la figure 5.8. Dans cette section, une enquête est faite pour savoir si ces pics proviennent du code de l'application lui-même ou s'il s'agit d'interférences générées par le système d'exploitation. Dans cette expérience, il n'y a pas de différence entre mono ou multi fichiers car nous ne considérons qu'un seul processus AOCS

Observation du phénomène Pour déterminer l'origine de ces pics, deux observations ont été faites. Premièrement, les pics n'apparaissent pas aux mêmes étapes pour chaque exécution du processus AOCS. On sait que le code AOCS n'a pas le même comportement pour chaque étape mais une étape exécutera toujours le même code si ses entrées sont les mêmes. Dans le cadre de ces expérimentations, les fichiers d'E/S ne sont pas modifiés. Pour cette raison, l'hypothèse que les pics ne proviennent pas du code AOCS lui-même semble plausible. Une autre hypothèse simple pourrait être que ces pics proviennent d'un changement de contexte

Pour savoir quelle hypothèse est vraie, les mesures ont été faites une deuxième fois avec une période modifiée. Dans ce cas, la période des pics n'a pas changé. La figure 5.9 montre les résultats pour une période de 50ms. En fait, il y a 20 pics en 625 secondes dans les deux courbes.

Recherche de la source Maintenant, la question est de trouver d'où viennent les pics. On peut noter sur la figure 5.8 qu'il existe deux types de pics. Il y a des pics plus petits avec des périodes plus courtes et des pics plus grands avec des périodes plus longues. Il semble que deux phénomènes périodiques impactent le processus AOCS. Il y a 2 hypothèses pour le deuxième phénomène avec les plus gros pics. Soit il s'agit d'une exécution particulière du premier phénomène se produisant tous les 10 pics. Ou cela peut provenir d'une source complètement différente et les 2 phénomènes sont synchronisés pour une raison quelconque

Pour répondre à ces questions un autre algorithme a été créé pour aider à trouver la source des pics observés. La nouvelle application lit à partir d'un tableau d'entiers stocké en mémoire. Les mesures de ce nouvel algorithme donnent les mêmes pics dans les données de temps d'exécution. Les deux types de pics apparaissent avec la même intensité et la même période. Cela signifie que la modification du code exécuté n'a pas modifié l'occurrence et l'intensité de l'impact observé. La seule chose qui reste la même entre ces deux expériences est le système Linux sur lequel les processus sont exécutés. Cela a confirmé une fois de plus que le système d'exploitation modifie le comportement des processus. Comme le montre la figure 5.11, il existe également des pics dans le remplissage du cache de données L2. Ces pics se produisent aux mêmes étapes que celles observées dans la courbe de temps d'exécution.

La figure 5.12 montre le résultat d'une trace LTTng effectuée lors d'une courte exécution du processus SCAO (environ 100 étapes). La période présentée dans la capture d'écran est juste avant l'exécution du processus SCAO. Il semble que plus d'un processus soit exécuté pendant cette période de temps. En fait, cela est dû à un service systemd Linux particulier appelé timesyncd. Ce service est responsable de la synchronisation de l'horloge système. Pour ce faire, il utilise le réseau pour obtenir des informations du protocole NTP. Le premier autre service est resolved,

il s'agit d'un autre service systemd Linux. Le dernier service montré dans la figure 5.12 est networkd, qui est encore une fois un autre service Linux systemd. Enfin, les services découverts en regardant la trace sont tous liés au réseau. En effet, le SoC est connecté à un réseau local via la prise ethernet. Cependant, cela signifie que les paquets Ethernet doivent être traités et potentiellement stockés dans la mémoire à un moment donné. Ainsi, des paquets Ethernet peuvent avoir été chargés sur le cache l2 partagé pendant cette période. Ceci confirmerait que le pic présent dans la courbe d'exécution AOCS est bien induit par un pic dans la courbe de remplissage du cache l2.

A.6 Expériences sur les cgroups

A.6.1 Enoncé du problème

Dans un satellite, l'algorithme SCAO est critique, ce qui signifie que sans lui, le comportement de l'engin spatial est incontrôlé. Ainsi, il doit fonctionner indépendamment de ce qui se passe dans les autres cœurs et le système d'exploitation. Un processus peut être impacté par son environnement de différentes manières :

- En modifiant son comportement direct (accès à ses données privées, modification des entrées/sorties)
- En bloquant son accès aux ressources ce qui modifie son comportement dynamique et peut modifier son comportement fonctionnel.

Ce chapitre se concentre sur le second aspect et plus particulièrement sur la modification du comportement dynamique du processus SCAO. Le but des expériences présentées ci-dessous est de savoir si les cgroups peuvent être utilisés pour réduire l'impact d'autres processus utilisant le temps CPU et la mémoire lors de l'exécution du processus SCAO. Le comportement attendu est que le mécanisme de limitation du temps CPU apporté par les Cgroups stabilise le comportement de l'ensemble du système tout en empêchant les processus de priver le processus SCAO d'accéder au CPU.

A.6.2 Cgroups

Comme expliqué dans la section 2.4, les cgroups permettent de contrôler les ressources auxquelles accède un ensemble de processus. En particulier, les cgroups peuvent être utilisés pour limiter l'utilisation d'une certaine ressource. Ce mécanisme peut être utilisé pour améliorer l'isolation du processus SCAO dans le système. Elle n'est pas totalement isolée comme celle apportée par la virtualisation qui rend le système plus

dynamique. Tous les processus du système utilisent les mêmes ressources matérielles et les mêmes cœurs de processeur, ce qui signifie qu'ils peuvent toujours avoir un impact les uns sur les autres par interférence. L'utilisation de cgroups pour limiter l'utilisation de ressources spécifiques devrait théoriquement réduire ces impacts en arrêtant les processus qui tentent d'utiliser plus de ressources qu'ils ne peuvent y accéder. Ce chapitre se concentre sur un aspect spécifique des cgroups, la limite dans l'utilisation du temps CPU. Cet aspect a été sélectionné car empêcher une application d'utiliser un processeur est une bonne solution pour réduire les interférences mémoire. En effet, moins une application tourne, moins elle impacte les autres applications.

A.6.3 Cas d'utilisation et benchmarks

Dans ce chapitre, deux types d'applications sont utilisées pour les expériences. La première est une version modifiée de l'algorithme SCAO du chapitre précédent, jouant le rôle de victime (c.a.d qui subit les interférences). Le deuxième type d'applications joue le rôle d'attaquants (c.a.d qui génère les interférences) et sont des benchmarks qui se focalisent sur un aspect très précis des ressources partagées.

SCAO

La version de l'algorithme de SCAO utilisée est simplement plus courte (seulement 800 pas de calculs) pour pouvoir répeter les expériences. De plus, les pas de calcul exécutés ont tous à peu prés le même temps d'exécution. Pour ce faire, les pas choisis vont du 3000ème au 3800ème.

benchmarks

Il y a deux types de benchmarks utilisés dans ces expériences. Le premier est nommé benchmark pi, sa particularité est qu'il ne fait aucun accès mémoire. De ce fait, il n'impacte que le temps d'occupation du processeur. Ce temps d'occupation est configurable.

Le deuxième type de benchmark est le benchmark load, qui a pour but de remplir le cache L2 partagé avec des accès mémoire bien configurés. Son impact sur l'occupation du processeur est donc minimal.

A.6.4 Approche expérimentale

Méthodologie

La méthode utilisée pour ces expériences consiste à faire varier deux types de paramètres indépendamment l'un de l'autre. La première configuration qui variera est le nombre

de benchmarks exécutés en parallèle avec le processus SCAO. Cette configuration a été choisie car l'augmentation du nombre de processus exécutés en parallèle augmentera probablement le nombre d'interférences. La deuxième configuration est la quantité de ressources partagées utilisées par tous les benchmarks. Cette configuration a été choisie pour cibler les ressources partagées où des interférences sont susceptibles de se produire. Ici, la ressource partagée considérée dépend du type de benchmark.

Le tableau présenté dans la figure 6.1 montre un exemple de la variation des paramètres pour les benchmarks Pi. Chaque cellule avec un "X" dans ce tableau représente une itération des 800 pas du processus AOCS. Pour chaque colonne et ligne de ce tableau, un seul paramètre varie.

Mesures

La sous-section précédente décrivait l'approche expérimentale, mais pour bien comprendre comment ces expériences fonctionnent, il est important de comprendre ce est mesuré et comment les mesures sont effectuées. Les compteurs de performance de l'ARM Cortex A53 et les horloges internes Linux donnent des informations substantielles sur le comportement des processus. Comme pour les expériences précédentes, une mesure est effectuée avant l'étape AOCS et une après afin d'évaluer la différence entre ces deux valeurs. Cependant, par rapport aux expérimentations précédentes, des mesures liées à l'ordonnanceur sont ajoutées. Dans l'ordonnancement multi-cœur périodique, l'allocation des cœurs et les échéances manquées donnent beaucoup d'informations sur la santé de l'ensemble du système. Quelques modifications ont été apportées au processus AOCS pour rendre ces mesures possibles.

Chaque combinaison de paramètres de la table 6.1 est utilisée dans une expérience avec 800 mesures de temps d'exécution et de compteurs de performance. Ces mesures sont ensuite répétées 50 fois afin de réduire la volatilité des mesures induite par Linux. Le nombre d'exécutions et le nombre de migrations représentent une mesure pour chaque processus et 800 étapes. Cela signifie que pour chaque cellule du tableau6.1, il existe une mesure du nombre d'exécutions et de migrations par processus.

A.6.5 Expériences avec les benchmarks Pi

Expérience SCHED_DEADLINE

Description de l'expérience Les valeurs des paramètres utilisées dans cette expérience sont les suivantes:

- Nombre de benchmarks: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 20, 30, 40, 50, 60, 70, 80.
- Temps d'exécution global de tous les benchmarks (en ms): 1, 2, 3, 5, 7, 8, 10, 15, 20, 25, 30.

Dépassements d'échéance Le premier objectif de cette expérience est de savoir si la politique SCHED_DEADLINE peut planifier les processus sans aucun dépassement de délai. Le graphique présenté dans la figure 6.5 donne la vue du nombre de pas exécutés pendant une exécution complète, il a été obtenu en faisant la moyenne des pas exécutés pour tous les processus de chaque expérience. Les valeurs inférieures à 800 indiquent des échéances manqués potentiels induits par une surcharge de processus à exécuter pendant la période de 10 ms.

La figure 6.5 montre le nombre de pas exécutés pour chaque configuration de cette expérience. Une observation intéressante est que pour des valeurs faibles du temps d'exécution global, le nombre de pas exécutés diminue légèrement. Pour essayer de comprendre ce phénomène, concentrons-nous sur l'impact du nombre de benchmarks dans ce graphique. Pour voir plus clairement ce résultat, il est possible d'intervertir le paramètre dans le graphique, ce qui signifie que le nombre de benchmarks est maintenant représenté sur l'axe des abscisses et que le temps d'exécution global de tous les benchmarks est représenté par la couleur de la courbe. La figure 6.6 montre le résultat de cette nouvelle configuration.

Ainsi, il existe une relation entre le nombre de pas exécutés et les deux valeurs des paramètres de configuration. Le nombre de pas exécutés est toujours égal à 800 sauf pour les valeurs faibles du temps d'exécution global et les valeurs élevées du nombre de benchmarks. Plus de benchmarks avec moins de temps d'exécution global signifie que chaque benchmark a un temps d'exécution inférieur. Ainsi, lorsque le temps d'exécution d'un benchmark est faible l'ordonnanceur n'ordonnance pas tous les benchmarks à chaque période. En effet, l'ordonnanceur ne parvient pas à ordonnancer tous les benchmarks dans la situation où le CPU est moins sollicité.

Impact sur le temps d'exécution Un autre aspect important de cette expérience est l'impact sur le temps d'exécution du processus SCAO. La figure 6.8 montre le temps d'exécution du processus SCAO avec le temps d'exécution global de tous les benchmarks sur l'axe des abscisses et le nombre de benchmarks représenté par la couleur des courbes.

Le premier sixième du graphique montre le comportement décrit dans le paragraphe précédent. Comme le nombre d'étapes exécutées diminue lorsque le temps d'exécution d'un benchmark est faible, l'impact sur le SCAO induit par les benchmarks est alors réduit. C'est pourquoi le temps d'exécution de le SCAO est plus faible dans cette première partie du graphique.

Concentrons-nous sur l'autre partie du graphique. La première chose à dire est que toutes les courbes semblent assez plates. Il y a encore quelques variations au milieu des courbes "bench 80" et "bench 70" mais ce n'est pas spécifique à ces courbes puisqu'en refaisant cette expérience ces variations n'apparaissent pas au même endroit. Ce

qui est intéressant derrière le fait que ces courbes soient plates, c'est que le temps d'exécution du processus AOCS n'est impacté que par les variations du nombre de benchmarks et non par la quantité de temps CPU qu'ils utilisent. Ce phénomène peut être observé en changeant l'axe des abscisses et la couleur des courbes. Les résultats sont présentés dans la figure 6.9. Ici, nous pouvons voir que toutes les courbes (sauf celles représentant les faibles valeurs du temps d'exécution global) ont la même forme et les mêmes valeurs.

Source de l'impact Dans le chapitre 5 il était clair que le cache partagé L2 avait un gros impact sur le temps d'exécution d'un processus sur la plate-forme matérielle utilisée. C'est pourquoi les "refills" du cache L2 ont également été mesurés dans cette expérience. Les résultats sont présentés dans la Figure 6.10.

Le même phénomène pour les valeurs inférieures du temps d'exécution global est présent dans ce graphique. Considérons seulement la partie des courbes après le point 5ms. Dans la Figure 6.10 il est difficile de distinguer les différents groupes de courbes, mais il semble que ces groupes aient disparu. En regardant la Figure 6.11 où le temps d'exécution global est représenté par la couleur des courbes et le nombre de repères en abscisse on peut voir que la forme globale des courbes est complètement différente de la ceux de la figure 6.9. Les lignes en escalier ont disparu et les courbes ont une forme quadratique. Cela signifie que l'impact constaté sur le temps d'exécution du processus AOCS ne provient pas uniquement de l'interférence sur le cache partagé L2.

Pour trouver l'autre canal d'interférence qui impacte le processus AOCS, il est possible de regarder les autres compteurs de performance. La figure 6.12 montre le nombre d'accès au cache de données L1 qui n'est pas partagé par les quatre cœurs. Ce graphique montre les mêmes groupes de courbes que celui de la figure 6.8. Les interférences induites par le cache L1 ne se produisent que lorsque plusieurs processus sont exécutés sur le même cœur. Ainsi, l'impact augmente avec le nombre de processus par cœur, ce qui crée ces groupes de 4 courbes.

L'impact sur le temps d'exécution du processus AOCS induit par l'exécution des benchmarks Pi est principalement induit par le système de cache mémoire à la fois L1 et L2.

Migrations de coeur Enfin, la dernière métrique importante à mesurer est le nombre de migrations car elle permet de comprendre si le système est stable. Ces mesures ont été récupérées en prenant le nombre de migrations qui se sont produites pour chaque processus au cours des 800 étapes exécutées. Ensuite, une moyenne sur tous les processus d'une même itération d'expérience a été calculée. Pour chaque couple de paramètres de configuration de l'expérience, il y avait 50 valeurs moyennes calculées

qui ont été agrégées en prenant la moyenne de ces 50 points. Les résultats présentés dans les graphiques présentés dans cette sous-section sont issus de cette méthode. La Figure 6.13 montre le nombre de migrations avec le nombre de benchmarks représenté par la couleur des courbes et le temps d'exécution global de tous les benchmarks en abscisse. Les courbes présentées dans ce graphique sont divisées en deux parties, la première partie pour les valeurs du temps d'exécution global inférieures à 15 ms est assez chaotique tandis que la seconde partie est plus stable.

La figure 6.15 montre le nombre de migrations pour le SCAO et tous les processus de référence pour chaque lot de l'expérience avec 10 tests de référence et un temps d'exécution global de 10 ms. Dans ce graphique, la plupart des points sont inférieurs à 100 migrations, ce qui signifie que la plupart du temps un processus est migré moins de 100 fois lors de l'exécution de 800 étapes. En fait, il y a 83 points supérieurs à 100, ce qui représente 16% de tous les points de cette figure. De plus, il n'y a que 23 points au dessus de 400 migrations soit 4% de tous les points. Cependant, cet exemple montre qu'il y a beaucoup de variabilité dans le nombre de migrations au cours d'une expérience. C'est, encore une fois, un résultat peu intuitif d'autant que dans cette configuration, 10 benchmarks pour 10ms de temps d'exécution global, le système n'est pas surchargé et le temps d'exécution d'un benchmark est suffisamment élevé pour ne pas tomber dans l'effet vu précédemment.

Conclusion Pour conclure cette première expérimentation, la politique SCHED_DEADLINE est capable de programmer des candidatures en temps réel tout en respectant les délais et périodes. Cependant, deux remarques sont importantes à expliquer.

La première est que lorsqu'un benchmark a un temps d'exécution supérieur à son temps d'exécution configuré, le planificateur ne le programme pas correctement. Cela entraîne des dépassements de délais pour les processus de référence.

Le second est le nombre de migrations de cœur qui se sont produites pendant les expériences. Au cours de cette expérimentation, les processus de référence ont beaucoup migré. Ce comportement rend le système instable et moins prévisible. Cependant, le processus SCAO n'a pas beaucoup migré et c'est un bon point pour la politique SCHED_DEADLINE même s'il n'y a aucune explication sur la raison pour laquelle la politique SCHED_DEADLINE a choisi le processus SCAO pour ne pas migrer beaucoup.

La plupart des impacts induits par l'exécution des benchmarks proviennent du cache partagé L2 et du cache de données L1. Cette politique permet une configuration simple pour exécuter des processus basés sur des délais, cependant, certains comportements peuvent rendre le système instable. Les sections suivantes se concentreront sur d'autres politiques de planification et mécanismes en temps réel et compareront les résultats avec celui-ci.

Expérience SCHED_RR

Description de l'expérience Comme pour l'expérience présentée dans la section 6.7.1, le nombre de benchmarks et le temps d'exécution global de tous les benchmarks varieront. Les variations de ces paramètres seront les mêmes que pour la dernière expérience (section 6.7.1) avec des benchmarks pi. Cependant, même si le temps d'exécution des benchmarks suit toujours la même équation (voir section 6.7.1), il n'y a pas de runtime à configurer pour la politique SCHED_RR. En effet, à la fin d'une étape de benchmark, le code de benchmark calcule la date du prochain réveil et la durée de son sommeil. Ce petit changement modifie le comportement des cas-tests puisque leur code est changé. La modification est considérée comme négligeable car le code ajouté ne dépend pas du nombre d'itérations de Pi. Cela signifie que le surcoût est constant pendant l'expérience. Un surcoût différent doit être pris en compte lorsqu'une comparaison sera faite entre les différentes expériences.

Dépassements d'échéance La figure 6.19 montre le nombre moyen de pas exécutés pour tous les processus. Chaque courbe du graphique est constante avec une valeur de 800, ce qui signifie que chaque processus a exécuté toutes ses pas.

Impact sur le temps d'exécution La Figure 6.20 montre le temps d'exécution du SCAO pour les différentes valeurs du nombre de benchmarks et le temps d'exécution global de tous les benchmarks. Le constat fait sur la Figure 6.19 se poursuit ici car la première partie du graphique est similaire à la dernière partie contrairement à la première partie de la Figure 6.20. Cependant, il semble que le temps d'exécution soit un peu plus élevé pour des valeurs faibles du temps d'exécution global et des valeurs élevées du nombre de benchmarks.

Le groupe de 4 courbes observé dans l'expérience précédente (dans la section 6.7.1), a disparu, et on ne peut observer qu'un seul groupe qui regroupe toutes les configurations jusqu'à 10 benchmarks (il n'est pas explicite que le point des 10 repères est la valeur seuil car elle est en fait très progressive). Toutes les autres courbes sont assez distinctes les unes des autres. Cette observation est confirmée par le graphique présenté dans la Figure 6.21. Les lignes en escalier ont complètement disparu et la forme globale de la courbe est complètement quadratique. De plus, les valeurs des courbes sont inférieures à celles de l'expérience SCHED_DEADLINE.

Enfin, à ce stade de l'exploration des résultats de SCHED_RR, il est possible de faire une comparaison entre les deux premières expériences présentées dans la section 6.7.1 et 6.7.2 . Ces expériences ont exactement le même schéma et le même protocole. La seule différence est la politique d'ordonnancement utilisée sous Linux. Ce qui a été appris jusqu'à présent est que le comportement du système est moins dépendant de la variation des paramètres de configuration et que l'absence de paramètres de

configuration de politique d'ordonnancement arbitraire génère un ordonnancement plus efficace. De plus, le temps d'exécution du processus SCAO est moins impacté par les activités des benchmarks exécutés entre-temps.

Source des impacts Même si l'impact de l'exécution de tous les processus de benchmark sur le temps d'exécution du SCAO est réduit par rapport à l'expérience SCHED_DEADLINE, il y a toujours un impact. Comme cela a été beaucoup observé dans cette thèse (chapitre 5), le cache L2 partagé est une source majeure d'interférences sur le Xilinx Zynq Ultrascale+. La Figure 6.23 montre le nombre de refills de cache L2 en moyenne lorsque le nombre de benchmarks et le temps d'exécution de tous les benchmarks varient.

Deux observations peuvent être faites à partir de ces courbes. Premièrement, les courbes sont constantes et ne dépendent pas du temps d'exécution global de tous les benchmarks. Deuxièmement, le nombre de "refills" de cache est légèrement supérieur à celui de la dernière expérience. Comme la seule chose qui a changé entre la première expérience et celle-ci est l'ordonnancement, cela signifie que la politique d'ordonnancement a un impact non nul.

Migrations de coeur La figure 6.25 montre le nombre de migrations pour tous les processus en fonction du nombre de benchmarks et du temps d'exécution global de tous les benchmarks. Tout d'abord, le nombre de migrations est beaucoup plus faible que dans la première expérience (dans la section 6.7.1). Le système a une meilleure stabilité avec cette nouvelle politique de d'ordonnancement. Cependant, une seule courbe, la courbe à 5 benchmarks, est toujours nettement supérieure aux autres comme le confirme la figure 6.26.

L'ordonnanceur semble avoir plus de problèmes pour ordonnancer le système lorsqu'il y a 5 benchmarks. La courbe des « 9 benchmarks » est également globalement plus élevée que les autres. Plus généralement, la Figure 6.26 montre que l'ordonnanceur devient plus stable après 10 benchmarks avant qu'il y ait des valeurs plus élevées du nombre de migrations. Jusqu'à 4 processus (soit 3 benchmarks + SCAO) c'est assez simple pour l'ordonnanceur car chaque processus peut avoir un seul cœur pour lui-même. Pour une raison quelconque, l'ordonnanceur a du mal à trouver plus de stabilité entre 4 et 10 benchmarks, quel que soit le temps d'exécution global de tous les benchmarks.

Un nombre de migrations plus faible signifie que les processus ont tendance à s'exécuter sur le même cœur, ce qui entraîne moins de remplissage dans les caches L1. C'est pourquoi dans cette expérience l'impact sur le temps d'exécution du processus AOCS n'a pas les mêmes groupes de comportement de courbe. En effet, les impacts des différentes caches ont la même forme et les lignes en escalier ont disparu.

Enfin, la Figure 6.27 montre le nombre de migrations du processus SCAO. Le processus SCAO migre plus que les autres processus en moyenne, surtout lorsqu'il n'y a pas trop de benchmarks exécutés en parallèle.

Conclusion Pour conclure cette expérience, la politique SCHED_RR est capable d'ordonnancer des applications basées sur des délais à l'aide de l'appel système *clock_nanosleep*. L'impact sur le temps d'exécution est quasiment le même que pour l'expérience SCHED_DEADLINE mais le phénomène qui s'est produit avec la faible valeur du temps d'exécution global a disparu. De plus, le système est plus stable en ce qui concerne les migrations de cœur, ce qui se traduit par une réduction de l'impact du cache de données L1. Cependant, le nombre de migrations du processus SCAO est supérieur aux autres processus qui sont moins bons que l'expérience SCHED_DEADLINE.

L'utilisation de cette politique d'ordonnancement rend le système plus prévisible car les processus sont moins susceptibles de migrer et l'impact sur le temps d'exécution est réduit.

Expérience SCHED_RR + cgroups

Description de l'expérience La configuration de l'expérience est la même que celle décrite dans la section 6.7.2. La création et l'initialisation des cgroups se font depuis le terminal bash avant de lancer le processus parent.

Dans cette expérience, il y a un cgroup avec tous les benchmarks et un autre avec le processus SCAO. Les cgroups sont configurés pour limiter uniquement le temps CPU. La période/échéance des deux cgroups, représentée par le paramètre 'sched_rt_period_us" est fixée à 10 ms. Le temps d'exécution représenté par le paramètre 'sched_rt_runtime_us" est la quantité de temps CPU qui peut être utilisée par les processus à l'intérieur du cgroup. Ce paramètre est fixé à 1 ms pour le cgroup SCAO et 8,5 ms pour le cgroup benchmarks.

Dépassements d'écheance Comme pour les autres expériences, le nombre d'échéances manquées est approximé en comptant le nombre de pas exécutés et en soustrayant du nombre de pas de base (800). La Figure 6.28 montre le nombre de pas exécutés de tous les processus exécutés au cours de cette expérience. Il semble que l'ajout des cgroups n'ait rien changé par rapport à la dernière expérience. L'ordonnanceur utilise toujours la même stratégie round-robin. Comme les courbes sont toutes constantes avec une valeur de 800, il est sûr de dire que toutes les étapes ont été exécutées et qu'il est peu probable que des dépassements d'échéance se produisent. Cela signifie que l'ajout de la configuration des cgroups ne modifie pas le comportement de l'ordonnanceur Linux avec la politique SCHED_RR.

Impacts sur le temps d'exécution L'impact sur le temps d'exécution du processus SCAO suit approximativement le même schéma que dans l'expérience précédente. La Figure 6.29 montre les courbes du temps d'exécution du SCAO avec le temps d'exécution global de tous les benchmarks représentés par l'axe des abscisses et le nombre de benchmarks par la couleur des courbes.

Tout d'abord, le phénomène de décroissance pour les faibles valeurs du temps d'exécution global découvert dans l'expérience décrite dans la section 6.7.2 est toujours présent. Ainsi le temps d'exécution global de tous les benchmarks a un impact mais seulement lorsqu'il est suffisamment faible. Ce phénomène peut apparaître à cause de la politique SCHED_RR.

Deuxièmement, il n'y a toujours pas de groupes de courbes mais la courbe « 1 benchmark » est supérieure à toutes les courbes inférieures à 30 repères. Ce phénomène est plus clair dans la Figure 6.30. On peut observer une ligne droite décroissante au début du graphique. Les cgroups n'ont pas le même comportement lorsqu'il n'y a qu'un seul benchmark exécuté en parallèle.

Enfin, les cgroups ont ajouté une surcharge constante dans le temps d'exécution du processus SCAO. Toutes les courbes sont environ 2% plus élevées que dans l'expérience précédente.

Source des impacts Comme pour l'expérience précédente présentée dans la section 6.7.2, les valeurs des mesures où le nombre de "refills" de cache L2 est supérieur à 10 000 ont été retirées du pool de données. Il y a 2404 sur 7 680 000 mesures qui sont au-dessus des seuils de 10 000, ce qui représente 0,03% de toutes les mesures. C'est presque le même pourcentage que l'expérience précédente.

La figure 6.31 montre le nombre de "refills" du cache L2 après la suppression des valeurs incohérentes. Les courbes ressemblent exactement à celles de l'expérience précédente. Cela signifie que les différences d'impacts observées dans les courbes de temps d'exécution AOCS ne proviennent pas de l'interférence du cache L2. Ainsi, cela laisse les interférences du cache L1.

La figure 6.32 montre le nombre d'accès L1 effectués par le processus SCAO. Dans ce graphique, deux observations majeures peuvent être faites. La première est que la courbe "1 benchmark" est plus haute que le groupe de courbes. La deuxième observation est que le début de toutes les autres courbes est plus plat.

La première observation explique pourquoi l'impact sur le temps d'exécution du SCAO est différent pour 1 benchmark. Une hypothèse pourrait être que l'ordonnanceur place les deux processus sur le même noyau alors que les cgroups n'ont qu'un seul processus. La deuxième observation pourrait expliquer le phénomène se produisant au début des courbes de temps d'exécution. Le nombre d'accès L1 est d'autant plus élevé que le temps d'exécution global est plus faible ce qui peut induire la partie

décroissante des courbes de temps d'exécution.

Migrations de coeur La Figure 6.33 montre le nombre de migrations du processus SCAO. Par rapport à l'expérience précédente présentée dans la section 6.7.2, le nombre moyen de migration a un peu diminué. Cependant, la courbe des "5 benchmarks" est toujours supérieure aux autres courbes comme le montre la Figure 6.34.

La figure 6.35 montre le nombre de migrations du processus SCAO. Le maximum de ces courbes est plus faible que dans l'expérience précédente mais la valeur moyenne de toutes les courbes est la même. Globalement, il n'y a pas beaucoup de différences entre cette expérience et la précédente en termes de migrations.

Conclusion L'ajout des cgroups au système n'a pas modifié le comportement du système de manière significative. Globalement, les impacts sont les mêmes que sans les cgroups. Il y a cependant un comportement spécial qui se produit lorsqu'un seul benchmark s'exécute en parallèle avec le processus SCAO. Le nombre de migration est également un peu plus faible que sans les cgroups.

L'ajout des cgroups n'a pas un impact énorme par rapport à l'expérience précédente et la stabilité du système est la même que dans l'expérience avec la politique SCHED_RR.

Expérience avec le patch académique

Le patch académique Abeni *et al.* ont développé un correctif de l'ordonnanceur Linux, présenté dans [76]. Ce patch vise à créer un ordonnanceur à deux niveaux. Le premier niveau utilise SCHED_DEADLINE et planifie les cgroups. La deuxième couche utilise SCHED_RR et planifie les processus. Ce patch permet d'utiliser la politique SCHED_DEADLINE avec le mécanisme des cgroups ce qui est normalement impossible.

Description de l'expérience Cette expérience utilise le patch décrit dans la sous-section précédente. Cela permet à l'ordonnanceur de fonctionner à deux niveaux différents. La première consiste à ordonnancer les cgroups et utilise la politique SCHED_DEADLINE tandis que l'autre ordonnance les processus à l'intérieur de chaque cgroup avec la politique SCHED_RR. Le but de cette expérience est de trouver si l'utilisation de ce patch diminue les interférences entre les processus de la victime et ceux de l'attaquant.

Le protocole de l'expérience et la configuration des cgroups sont exactement les mêmes que l'expérience précédente de la section 6.7.3. Il y a toujours un cgroup pour l'AOCS et un autre pour tous les benchmarks. La seule chose qui change est le noyau lui-même qui est modifié par le patch de Luca Abeni *et al.*

Dépassements d'échéance La figure 6.36 montre le nombre moyen de pas exécutés par tous les processus. Les processus exécutent toutes leurs 800 pas, ce qui signifie qu'il est peu probable que les échéances soient dépassées. La nouvelle politique d'ordonnancement ajoutée par le patch de l'ordonnanceur ne modifie pas l'impact sur les échéances manquées.

Impacts sur le temps d'exécution L'impact sur le temps d'exécution est illustré dans la Figure 6.37. Ce graphique est similaire à la deuxième expérience avec uniquement SCHED_RR. Cependant, les courbes sont un peu plus élevées. Le phénomène qui s'est produit avec la courbe "1 benchmark", vu dans l'expérience précédente, a disparu.

La Figure 6.38 montre le temps d'exécution du processus SCAO avec le nombre de benchmarks sur l'axe des abscisses et le temps d'exécution global de tous les benchmarks représentés par la couleur des courbes. Globalement, l'allure des courbes est toujours la même que dans les deux expériences précédentes. Ces trois expériences montrent le même impact sur le temps d'exécution du processus SCAO.

Source des impacts Comme pour les trois autres expériences (section 6.7.1, 6.7.2 et 6.7.3), la première source d'impact est le cache partagé L2. La figure 6.39 montre les "refills" l2 du processus AOCS. On constate que cette fois il n'y a pas de points incohérents, c'est-à-dire qu'il n'y a pas de migrations du processus SCAO lors de l'exécution d'un pas. En ce qui concerne le graphique de temps d'exécution, les courbes ressemblent aux autres graphiques de "refills" l2 présentés dans les deux expériences précédentes mais sont un peu plus élevées. Cette différence entre cette expérience et les deux précédentes peut expliquer la différence des courbes de temps d'exécution. Comme les courbes de "refills" L2 sont un peu plus élevées, le temps d'exécution est également un peu plus élevé.

Migrations de coeur Comme indiqué dans le paragraphe précédent, le processus SCAO ne migre pas lors de l'exécution d'une étape de cette expérience. Cela signifie que même si toutes les courbes précédentes, présentées dans cette section ressemblent aux courbes des expériences précédentes, cela pourrait ne pas être le cas pour les courbes de migrations de cœur.

La Figure 6.40 montre le nombre de migrations en moyenne pour tous les processus. Deux remarques peuvent être faites sur ces courbes. Premièrement, les courbes "5 benchmarks" et "6 benchmarks" sont plus hautes que les autres courbes mais aussi plus hautes que les courbes correspondantes dans le graphique des deux expériences précédentes présentées dans la section 6.7.2 et 6.7.3. Deuxièmement, le reste des courbes est inférieur aux courbes correspondantes des deux expériences précédentes.

Ce second constat est plus visible sur la Figure 6.41. L'ajout du patch de l'ordonnanceur rend le système beaucoup plus stable en termes de migrations de base. Mais cela accentue aussi le phénomène avec les courbes à 5 et 6 repères.

Enfin, la Figure 6.42 montre le nombre de migrations pour le processus AOCS uniquement. Les courbes sont comprises entre 2,5 et 5 migrations ce qui est très faible par rapport aux autres expériences. Le phénomène pour les courbes à 5 et 6 repères a disparu et toutes les courbes ont les mêmes valeurs. Le patch de l'ordonnanceur protège le processus SCAO des migrations principales et réduit ainsi l'impact l'ordonnanceur sur le processus SCAO.

Conclusion Pour conclure cette expérience, le patch de l'ordonnanceur réalisé par Luca Abeni *et al.* apporte de belles améliorations. Alors que l'impact sur le temps d'exécution est sensiblement le même, le nombre de migrations est beaucoup plus faible. Surtout, le nombre de migrations du processus SCAO est inférieur à 5 pour 800 exécutions ce qui est vraiment faible. Il semble que le correctif combine les avantages de la politique SCHED_DEADLINE et de la politique SCHED_RR tout en utilisant le mécanisme des cgroups pour limiter l'utilisation du temps CPU.

A.6.6 Expériences avec les benchmarks load

Expérience SCHED_DEADLINE

Description de l'expérience Dans cette expérience, les paramètres variants dans la configuration du système sont le nombre de processus de benchmarks et le nombre total d'accès load effectués par tous les processus benchmarks combinés. Les attaquants utilisés dans cette expérimentation sont les benchmarks load présentés dans la section 6.4. Ces benchmarks sont configurés pour effectuer une quantité spécifique d'accès mémoire afin de remplir le cache partagé L2.

Les valeurs de configuration qui seront utilisées pour cette expérience sont décrites comme suit :

- Nombre de benchmarks: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 20, 30, 40, 50, 60, 70, 80
- Nombre de loads: 256, 512, 1024, 2048, 3072, 4096, 6144, 8192

Le nombre de chargements est paramétré pour remplir progressivement le cache L2.

Dépassements d'échéance La Figure 6.44 montre le nombre de pas exécutés en fonction du nombre de benchmarks et du nombre de loads effectués par tous les benchmarks. Globalement toutes les courbes sont constantes avec une valeur de 800 sauf deux d'entre elles : les courbes « 1 benchmark » et « 2 benchmarks ». La première courbe commence à décroître vers 4000 loads et tombe à 610 pas exécutés

ce qui représente une perte de 25%. La deuxième courbe diminue après environ 6000 loads et tombe à 780 pas exécutés, soit une perte de 2,5%.

Ce phénomène peut être confirmé en regardant la Figure 6.45. Cela ne se produit que pour un très petit nombre de benchmarks (inférieur à 3) et un très grand nombre de loads (supérieur à 4096 loads). Cela signifie qu'il s'est produit lorsque le nombre de loads pour chaque benchmarks est très élevé (supérieur à 3000 loads par benchmark). Lorsque le nombre de loads est élevé pour un processus, son impact sur le temps d'exécution est plus important. Si le temps d'exécution dépasse le paramètre d'exécution donné à la politique d'ordonnancement, l'ordonnanceur peut ne pas ordonnancer le processus. Ce phénomène a déjà été constaté dans l'expérience SCHED_DEADLINE avec les benchmarks pi. Ainsi, cela pourrait expliquer le phénomène observé dans ces graphiques.

Impacts sur le temps d'exécution La Figure 6.46 montre les courbes de temps d'exécution. Dans ce graphique, il y a deux groupes de courbes : le premier est composé des courbes en dessous de 20 benchmarks et le second des courbes au dessus de 20 benchmarks. Cependant, il reste une courbe qui est seule et n'appartient à aucun de ces groupes, c'est la courbe « 1 benchmark ». En effet, comme vu dans la sous-section précédente, lorsqu'il n'y a qu'un seul benchmark, il n'exécute pas tous ses pas ce qui réduit l'impact de ce benchmark sur le temps d'exécution du processus SCAO. Toutes les courbes sont croissantes mais le premier groupe de courbes a une pente plus positive que l'autre groupe.

Cette différence entre les deux groupes de courbes est visible sur la figure 6.47. On constate une cassure des courbes aux point 20 benchmarks. La première partie des courbes est légèrement décroissante, légèrement croissante ou constante (sauf pour le premier point) et la deuxième partie des courbes est linéairement croissante d'environ 10 000 ns tous les 10 benchmarks. L'explication probable de ce phénomène est que les benchmarks ont plus de chances de remplir le cache lorsqu'il y a moins de benchmarks en parallèle. En effet, l'espace d'adressage virtuel des différents processus est complètement différent et la traduction de ces adresses virtuelles peut se chevaucher et donc aller vers les mêmes lignes de cache.

Comme pour les expérimentations de benchmarks pi, l'impact du nombre de benchmarks est supérieur à l'impact de la quantité utilisée de la ressource partagée. Cependant, avec les benchmarks load, il y a un impact non négligeable du nombre de chargements ce qui n'était pas le cas avec le temps d'exécution global de tous les benchmarks dans les benchmarks pi expérimentaux. Ceci peut s'expliquer par le rôle prépondérant du cache L2 dans l'impact sur le temps d'exécution AOCS

Source des impacts Comme pour toutes les autres expériences, le cache partagé L2 a un grand rôle dans les impacts sur le temps d'exécution. La Figure 6.48 montre les courbes du nombre de "refills" de cache L2. Les deux groupes de courbes qui étaient décrits dans les courbes de temps d'exécution sont toujours présents ici. De plus, les courbes de "refills" du cache L2 ressemblent beaucoup aux courbes de temps d'exécution, ce qui confirmerait que l'impact sur le temps d'exécution provient principalement du phénomène de contention du cache L2.

Les autres courbes de "refills" du cache L2 présentées dans la figure 6.49 ressemblent également à la courbe du temps d'exécution. La cassure qui se produit au point des 20 benchmarks est moins raide mais la forme globale des courbes est la même.

Globalement, on peut dire que les impacts sur le temps d'exécution, vus dans la sous-section précédente, proviennent majoritairement du cache partagé L2. Ces impacts sont plus importants que dans les expériences de benchmarks pi car dans cette expérience les benchmarks accèdent à la mémoire ce qui augmente l'utilisation du cache L2.

Migrations de coeur Les principales migrations de tous les processus sont présentées dans la figure 6.50. Toutes les courbes sont élevées par rapport au nombre de pas exécutés (800), avec un nombre moyen d'environ 200 migrations. Cependant, même si les courbes n'ont pas une forme claire, elles sont moins chaotiques que les courbes de migrations du cœur de l'expérience SCHED_DEADLINE avec les benchmarks pi. Sur ce graphique, il est difficile d'observer un phénomène clair.

Afin d'observer un phénomène plus clair, regardons la Figure 6.51 qui montre également le nombre de migrations mais le nombre de benchmarks et le nombre de loads ont changé de représentation. On voit que pour des valeurs faibles du nombre de benchmarks (inférieures à 5) il n'y a pas beaucoup de migrations (moins de 100 migrations). Aussi lorsque le nombre de benchmarks augmente, les courbes se séparent. Après environ 20 benchmarks, il semble que les courbes soient plus stabilisées autour de la valeur 250.

Comme pour les autres courbes de migrations de cœur, il est difficile de comprendre le comportement de l'ordonnanceur en regardant ce graphique. De plus, une métrique plus importante est le nombre de migrations pour le processus SCAO en particulier. La figure 6.52 montre le nombre de migrations du processus SCAO uniquement. Bien que les courbes soient encore chaotiques, le nombre moyen de migrations a beaucoup diminué par rapport au nombre moyen de migrations montré dans les figures 6.50 et 6.14. En effet, il y a moins de 3 migrations en moyenne durant les 800 pas du processus SCAO.

Conclusion L'impact induit par les benchmarks load dans l'expérience SCHED_DEADLINE est un peu différent qu'avec les benchmarks pi. Cela dépend à la fois du nombre de benchmarks et du nombre de loads et est principalement induit par les "refills" du cache L2. Cependant, le phénomène avec le nombre de pas exécutés est toujours présent mais il se produit dans moins de cas. Le comportement de SCHED_DEADLINE concernant la migration de cœur est un peu différent par rapport aux expériences de benchmarks pi. En moyenne, il y a plus de migrations mais les courbes sont plus stables.

Expérience SCHED_RR

Description de l'expérience Cette expérience est la même que la précédente mais au lieu de la politique SCHED_DEADLINE, elle utilise la politique SCHED_RR.

L'objectif principal de cette expérience est de savoir si les benchmarks load modifient le comportement de la politique SCHED_RR. Comme la nature des benchmarks a changé, on s'attend à ce que la politique SCHED_RR change de comportement.

Dépassements d'échéance La figure 6.53 montre le nombre de pas exécutés et toutes les courbes sont constantes avec la valeur maximale de 800 pas exécutés. Comme pour la même expérience pour les benchmarks pi (section 6.7.2), cette politique d'ordonnancement parvient à ordonnancer tous les processus sans problème.

Impacts sur le temps d'exécution La figure 6.54 montre le temps d'exécution du processus SCAO. Les courbes présentées dans ce graphique ont la même forme que celles de l'expérience SCHED_DEADLINE. Les deux groupes de courbes sont toujours présents mais l'exception pour la première courbe a disparu et la courbe « 1 benchmark » est supérieure à toutes les courbes du premier groupe. Ceci est lié au fait que dans l'expérience SCHED_DEADLINE lorsqu'il n'y a qu'un seul benchmark il n'est pas exécuté pendant 800 pas lorsque le nombre de loads augmente.

La figure 6.55 montre le temps d'exécution du processus SCAO mais le nombre de benchmarks est représenté sur l'axe des abscisses. La forme globale des courbes est toujours la même que celle de l'expérience SCHED_DEADLINE. Cependant, la rupture qui s'est produite autour des 20 benchmarks semble avoir disparu. La pente de la courbe est toujours la même avec 10 000 ns pour 10 benchmarks.

Le phénomène au début du graphique se produit toujours, ce qui ne contredit pas l'explication prospective décrite dans la dernière expérience de chargement des repères présentée dans la section 6.7.1. Comme pour les benchmarks pi, les courbes de temps d'exécution de l'expérience SCHED_RR sont inférieures à celles de l'expérience SCHED_DEADLINE. L'impact du changement de type de benchmarks est visible principalement au début du graphique, mais également dans la différence entre chaque

courbe. En effet, dans les expériences de benchmarks pi, le temps d'exécution ne dépendait que du nombre de benchmarks alors que dans ces expériences il dépend aussi du nombre de loads.

Source des impacts Pour comprendre la source des impacts sur le temps d'exécution, analysons les courbes de "refills" du cache L2. Comme pour les expériences précédentes avec SCHED_RR, des migrations de cœur peuvent se produire lors de l'exécution d'une étape SCAO. Ainsi, cela peut rendre les mesures des compteurs de performance inutilisables. C'est pourquoi les courbes présentées dans cette sous-section ne prendront en compte que les mesures où la valeur de "refills" du cache L2 est inférieure à 10 000.

La figure 6.56 montre le nombre de "refills" L2 du processus SCAO. Il est également possible de changer l'axe des abscisses et la couleur des courbes. Les résultats sont présentés dans la Figure 6.57. Dans les deux cas, les courbes ressemblent exactement à celles des courbes de temps d'exécution présentées dans les figures 6.54 et 6.55. Ainsi les impacts sur le temps d'exécution du processus SCAO proviennent du cache L2.

Migrations de coeur La figure 6.58 montre le nombre de migrations de tous les processus. Les courbes sont pour la plupart constantes avec une valeur comprise entre 0 et 20 migrations. Par rapport aux expériences SCHED_DEADLINE avec benchmarks load, il y a beaucoup moins de migrations et les courbes semblent plus stables. En fait, il n'y a pas beaucoup de points qui dépassent 30 migrations, ce qui n'est pas beaucoup pour 800 pas. Par rapport aux mêmes courbes mais avec des benchmarks pi, le phénomène où deux courbes étaient au-dessus de toutes les autres a disparu. L'hypothèse est que la politique SCHED_DEADLINE a du mal à trouver un ordonnancement optimal pour un certain nombre de processus exécutés.

Cependant, comme le montre la figure 6.59, il y a encore beaucoup de migrations lorsque le nombre de benchmarks est inférieur à 10 mais que le nombre de migrations est inférieur (entre 10 et 70). De plus, à la fin du graphique, on peut voir que certaines courbes augmentent de moins de 10 migrations à entre 10 et 20 migrations mais ce phénomène n'est pas assez important pour être analysé ici.

Enfin, regardons le nombre de migrations du processus SCAO uniquement. La figure 6.60 montre ce nombre de migrations. Le nombre de migrations du processus SCAO ne ressemblait qu'à la moyenne. Cependant, le pic autour des 10 benchmarks est beaucoup plus élevé pour le SCAO uniquement. Le SCAO migre plus que les autres processus avec la politique SCHED_RR, c'est la même tendance qui a été démontrée dans les benchmarks pi avec la politique SCHED_RR présentés dans la section 6.7.2. Pour expliquer ce comportement, l'hypothèse est que

ce n'est pas la politique SCHED_RR qui migre le plus le processus AOCS mais celle SCHED_DEADLINE qui le migre beaucoup moins du fait de son temps d'exécution plus élevé. Le nombre de migrations du processus SCAO avec la politique SCHED_RR ne dépend que de sa place dans la file d'attente roud-robin puisque tous les processus ont la même priorité.

Conclusion Pour conclure, le nombre de loads impacte différemment les "refills" du cache L2 que les benchmarks pi. Cela a un impact sur le temps d'exécution du SCAO. En termes de migrations et de nombre d'exécutions, la politique SCHED_RR se comporte exactement comme lors de l'expérience des benchmarks pi. Les différences entre la politique SCHED_RR et SCHED_DEADLINE présentées dans cette sous-section sont comparables à celles observées entre les deux mêmes expériences mais avec les benchmarks pi présentés dans les sections 6.7.1 et 6.7.2.

Expérience SCHED_RR + cgroups

Description de l'expérience Les cgroups sont configurés comme dans l'expérience pi des benchmarks correspondants (voir section 6.7.3). Le premier cgroup est composé du processus SCAO uniquement et le second est composé de tous les benchmarks. Ils ont tous les deux une échéance de 10 ms et un temps d'exécution de 1 ms pour le cgroup SCAO et de 8,5 ms pour le cgroup benchmarks, comme expliqué dans la section 6.7.3.

L'objectif principal de cette expérience est de savoir si les impacts des charges de référence peuvent être réduits par les cgroups. Les benchmarks load n'utilisent pas le temps CPU comme les benchmarks pi et on pourrait donc avancer qu'essayer de réduire leurs impacts en configurant un mécanisme de limitation du temps CPU ne serait pas la meilleure solution. Cependant, il a été vu auparavant que les cgroups ont un impact sur le comportement des migrations principales de l'ordonnanceur et que les migrations principales ont un impact sur l'utilisation des caches. Enfin, cette expérience vise à déterminer si les cgroups sont utiles dans cette configuration particulière

Dépassements de l'échéance La figure 6.61 montre le nombre de pas exécutés qui est l'approximation des échéances manquée. Toutes les courbes sont constantes avec la valeur de 800 qui est le maximum possible.

Ainsi, il est possible de conclure que l'ajout de la configuration des cgroups dans le système n'a pas ajouté d'échéances manquées. Cela signifie que tous les processus ont respecté leurs échéances.

Impacts sur le temps d'exécution La figure 6.62 montre le temps d'exécution du processus SCAO au cours de cette expérience. Les courbes ont exactement la même forme et les mêmes valeurs que celles de l'expérience précédente (section 6.8.2).

La figure 6.63 confirme également la similitude des deux expériences. Cela signifie qu'en termes de temps d'exécution, la configuration des cgroups n'a eu aucun impact notable sur ces mesures.

Source des impacts Les courbes suivantes sont présentées dans les figures 6.64 et 6.65 montrent le nombre de "refills" de cache L2 du processus SCAO. Les points aberrants ont été supprimés de ces graphiques.

Sans surprise, ces courbes sont très similaires aux courbes de temps d'exécution mais aussi aux courbes de "refills" du cache L2 de l'expérience précédente. Le cache L1 n'a pas un impact énorme sur le temps d'exécution du SCAO. Le cache L2 est toujours un point de contention spécifiquement avec les benchmarks load. Le fait que ces benchmarks effectuent plus d'accès à la mémoire rend les cgroups moins capables de réduire les interférences. En fait, la mémoire sera accessible par les benchmarks, quelle que soit la décision de l'ordonnanceur.

Migrations des coeur Les courbes de migrations de cœur sont présentées dans la Figure 6.66. Ce graphique ressemble presque exactement à celui correspondant dans l'expérience précédente. Pour confirmer ce sentiment, il est possible de regarder les autres courbes de migrations de la Figure 6.67. Là encore, ces courbes sont très proches de celles des expériences précédentes.

La figure 6.68 montre le nombre de migrations du processus SCAO uniquement. Les courbes présentées sur cette figure ont la même allure que celles correspondantes de l'expérience précédente. Le pic autour du point des « 10 repères » est toujours là et le nombre de migrations est globalement supérieur à la moyenne. Ainsi, l'ajout des cgroups n'a eu aucun impact sur la migration principale du processus AOCS uniquement

Conclusion Pour conclure cette expérience, l'ajout des cgroups n'a pas modifié significativement les impacts des benchmarks sur le processus SCAO. Le comportement de l'ensemble du système est globalement similaire à celui de l'expérience sans les cgroups

Expérience avec le patch académique

Impacts sur le temps d'exécution La Figure 6.70 montre les courbes du temps d'exécution SCAO. Encore une fois avec les benchmarks load, les courbes de temps d'exécution sont très similaires à celles de l'expérience SCHED_DEADLINE avec les

benchmarks load présentées dans la section 6.8.1. Il existe encore deux groupes de courbes : l'une avant 20 benchmarks et l'autre après 20 benchmarks. Les formes des courbes sont les mêmes que celles des expériences précédentes. Cependant, les valeurs des courbes sont inférieures à celles de l'expérience SCHED_DEADLINE.

Cette tendance est également confirmée par les autres courbes de temps d'exécution présentées dans la Figure 6.71. Nous pouvons voir l'augmentation des courbes lorsque le nombre de loads augmente. De plus, la phase décroissante au début du graphique est toujours présente et le pic de cette phase est plus élevé que dans l'expérience SCHED_DEADLINE.

En termes d'impacts sur le temps d'exécution, l'ajout du patch académique a réduit le temps d'exécution en moyenne par rapport à l'expérience SCHED_DEADLINE mais l'a augmenté pour les faibles valeurs du nombre de benchmarks.

Source des impacts Les figures 6.72 et 6.73 montrent le nombre de "refills" de cache du processus SCAO. Ces courbes ressemblent aux courbes de temps d'exécution montrant que la plupart des impacts sur le temps d'exécution proviennent des "refills" du cache L2.

Cependant, dans le premier groupe de courbes, il existe de petites différences entre le graphique de "refills" du cache L2 et le graphique du temps d'exécution. Certaines courbes appartenant au premier groupe de courbes semblent plus basses dans le graphique de remplissage du cache L2 que dans le graphique des temps d'exécution. Cela peut signifier qu'il existe une autre source d'impact sur le temps d'exécution qui peut être les "refills" du cache de données L1.

Migrations de coeur La Figure 6.74 montre le nombre de migrations de tous les processus au cours de cette expérience. A l'exception d'un point anormal, le nombre moyen de migrations est très faible pour toutes les courbes. Le maximum est d'environ 30 migrations ce qui ne représente pas un montant énorme pour des exécutions en 800 pas. Par rapport aux autres expériences de benchmarks load avec SCHED_RR, le nombre moyen de migrations est un peu plus élevé mais les points maximaux sont plus bas. Ceci est confirmé par la figure 6.75 où l'on peut voir qu'il y a moins de gros pics avant le point des 10 benchmarks.

En termes de nombre de migrations du processus AOCS uniquement, comme le montre la figure 6.76, il y a beaucoup moins de migrations qu'avec les autres expériences SCHED_RR. Le système est plus stable avec le processus SCAO puisqu'il migre beaucoup moins ce qui réduit les interférences. La politique SCHED_DEADLINE a des informations sur le temps d'exécution du processus SCAO et décide donc de ne pas trop le migrer car il a le temps d'exécution le plus élevé de tous les processus.

Conclusion Pour conclure, comme pour l'expérience de benchmarks pi correspondante, cette expérience a les mêmes impacts que les expériences SCHED_RR mais le nombre de migrations du SCAO est le même que celui de l'expérience SCHED_DEADLINE. Le patch académique combine les avantages des politiques SCHED_RR et SCHED_DEADLINE.

A.7 Synthèse d'analyse et recommandations methodologiques

A.7.1 Recommandations sur les expériences

L'importance du cache L2

Presque toutes les expériences présentées dans cette thèse ont montré que le cache L2 est un goulot d'étranglement dans le système. Le temps d'exécution observé du processus SCAO dépend du cache L2 dans la mesure où les courbes de temps d'exécution et les courbes de "refills" du cache L2 ont, la plupart du temps, des formes très similaires. Par définition, lorsqu'un "refill" de cache L2 se produit, le cache va transférer l'accès mémoire du bus vers la mémoire principale. Cela pourrait expliquer la relation entre le temps d'exécution et les remplissages du cache L2. Deux options sont possibles pour réduire cet impact : soit réduire le nombre de remplissages du cache L2, soit réduire le temps d'accès pour passer du cache L2 à la mémoire principale. La réduction du temps d'accès du cache L2 à la mémoire principale est un problème matériel et n'entre pas dans le cadre de cette thèse.

Cependant, il existe des solutions pour réduire le nombre de recharges de cache L2. Une solution matérielle serait d'augmenter la taille du cache en ajoutant soit plus de lignes, soit plus de voies. La deuxième solution pour réduire le nombre de recharges de cache L2 est d'utiliser des techniques de partitionnement de cache.

Impacts des paquets réseau

Les différentes expériences présentées dans cette thèse ont montré un impact significatif du code lié au réseau sur le processus SCAO. La section 5.3.2 montrait les impacts des services *timesyncd* et *ntpd*. Il existe deux points communs entre ces deux services. La première est qu'ils sont tous les deux des services systemd (décrits dans la section 2.4). La seconde est qu'ils doivent envoyer et recevoir des paquets réseau.

Globalement, l'utilisation d'une pile réseau complexe associée à la gestion des démons systemd induit un impact énorme sur le processus SCAO. Ces services réseau ont été conçus pour gérer les opérations réseau d'un ordinateur de bureau ou d'un serveur connecté à un réseau dynamique. Cependant, un système embarqué a une

taille de réseau réduite et le réseau n'est pas modifié dans le temps. Il n'y a pas de nouveaux routeurs d'extrémités dans un satellite. La mise en œuvre d'une pile réseau personnalisée optimisée devrait réduire l'impact des applications liées au réseau.

Impacts de la charge relative du CPU

Le chapitre 6 montre des résultats expérimentaux de l'impact de la charge relative du CPU. Les benchmarks pi ont été conçus pour augmenter la charge relative du CPU. Ces expériences concluent que la charge relative du CPU n'a pas d'impact sur le temps d'exécution du processus AOCS. Cela pourrait sembler être un résultat contre-intuitif, mais ce n'est pas le cas.

Lorsque l'on pense à la charge relative du processeur, la logique est que l'augmenter augmentera la quantité de travail que le système doit effectuer. Bien que cela soit vrai pour le processeur, ce n'est pas vrai pour le système d'exploitation. En d'autres termes, un processus avec un temps d'exécution de 0,1ms avec une période de 10ms et le même processus avec un temps d'exécution de 8ms ont le même impact sur le processus AOCS tant qu'ils ne sont pas interrompus. La seule limitation est que le système d'exploitation a besoin d'un certain temps pour s'exécuter, ce qui signifie que le processus ne peut pas être exécuté pendant 9,9 ms sur une période de 10 ms. Ceci est confirmé par les résultats présentés dans le chapitre 6. C'est un résultat contre-intuitif car la plupart du temps l'augmentation du temps d'exécution est liée à une augmentation de l'utilisation des ressources (mémoire, I/Os, caches, bus, ...) qui induit plus d'interférences et a donc un impact plus important sur le processus de la victime.

Les applications logicielles qui n'utilisent pas de ressources partagées autres que la puissance du processeur ne sont pas très courantes. La grande majorité des applications logicielles utilisent de la mémoire. Cependant, si les processus sont isolés et que les accès aux ressources partagées n'ont pas d'impact sur les autres processus, alors l'augmentation du temps d'exécution d'un processus n'affecte pas les autres processus. Cela signifie qu'il est possible de réduire la marge de sécurité de la puissance CPU prise pour garantir que chaque processus aura le temps de s'exécuter, quels que soient les impacts auxquels il est confronté. Ainsi, il y a plus de puissance CPU disponible sur la plate-forme.

A.7.2 Configurations Linux

Configuration de SCHED_DEADLINE

La politique SCHED_DEADLINE permet d'ajouter une période et une échéance aux processus. Comme expliqué dans la section 2.4 la politique SCHED_DEADLINE

doit être configurée pour être fonctionnelle. La politique ne devine pas la période et l'échéance requis par le processus. Les trois paramètres nécessaires pour configurer cette politique d'ordonnancement modifient complètement le comportement de l'ordonnanceur. Les résultats de l'expérience présentés dans le chapitre 6 montrent que la configuration de cette politique d'échéance peut induire de nombreux dépassements d'échéance. En particulier, le paramètre "runtime" a beaucoup d'impacts sur le comportement de l'ordonnanceur avec la politique SCHED_DEADLINE. Si ce paramètre est inférieur au WCET de l'application cela peut induire des dépassements de délais et au contraire, s'il est supérieur au WCET de l'application cela signifie que le temps CPU est "réservé" pour rien.

Trouver le WCET d'une application donnée est un problème bien connu dans la littérature et il est très difficile d'avoir un WCET précis. De plus, dans le cadre de cette thèse, le WCET du processus AOCS dépend non seulement du code AOCS mais aussi du système d'exploitation Linux. En raison de la non-reproductibilité du comportement de Linux entre deux exécutions différentes, il semble impossible de trouver le WCET précis du processus SCAO. Par conséquent, le meilleur paramètre d'exécution pour configurer correctement SCHED_DEADLINE sera nécessairement supérieur au WCET en prenant une marge sur le plus grand temps d'exécution observé.

Le patch PREEMPT-RT dans un contexte spatial

L'objectif principal du patch PREEMPT-RT est d'améliorer le comportement temps réel de Linux. Le comportement temps réel est caractérisé par le nombre de dépassements d'échéance et le respect de la période.

Dans le chapitre 5 il a été montré que le patch PREEMPT-RT n'a pas d'impact sur les résultats de mesure. Pour comprendre pourquoi le patch n'a pas d'impact sur le temps d'exécution il faut regarder le chapitre expérimentations 6. Dans ces expériences, le nombre de pas exécutés du processus SCAO a été mesuré. Comme expliqué dans le chapitre 6, le nombre de pas exécutés est une bonne approximation du nombre d'échéances manquées. Ces mesures ont montré qu'il n'y a la plupart du temps aucune échéance manqué. Le seul cas où les échéances sont dépassées est dû à un problème de configuration de la politique SCHED_DEADLINE comme cela a été expliqué dans le chapitre 6. Par conséquent, l'ajout du patch PREEMPT-RT n'aura pas un impact énorme sur les performances mesurées du processus SCAO. Notez que les expériences présentées dans le chapitre 5 ne sont pas conçues pour tester le patch PREEMPT_RT.

A.8 Conclusion

Le chapitre 4 présente une modélisation des interférences qui permet d'identifier et de compter les interférences mémoire d'un ensemble d'applications donné. Réalisée avec des automates temporisé, cette modélisation fournit un comptage précis des interférences mémoire et peut être reliée à une méthode académique décrite dans [17].

Le chapitre 5 fournit des expériences sur une application SCAO fonctionnant sous Linux sur un SoC Zynq Ultrascale+. Les résultats des expérimentations ont montré les impacts de Linux sur une application AOCS. Plus précisément, il a été prouvé que l'utilisation des mêmes fichiers pour différentes applications concurrentes a de meilleures performances que l'utilisation de plusieurs fichiers. C'est un exemple d'impact positif du système d'exploitation. A l'inverse, les services systemd ont un impact négatif sur l'application SCAO. Il a été montré que cet impact se caractérise par l'augmentation des "refills" de cache L2. Mais l'application SCAO n'est pas la seule partie à être impactée, en effet l'ordonnanceur Linux ne se comporte pas de la même manière lorsque certains services sont désactivés. Il en résulte une dérive de la date de réveil des processus.

Le chapitre 6 a pour but de proposer des expériences sur les mécanismes d'isolation sous Linux. L'objectif était d'évaluer un mécanisme d'isolement avec différentes politiques d'ordonnancement en utilisant deux types de processus d'attaquants. Les résultats des expériences ont montré que le mécanisme d'isolement n'a pas d'impact négatif sur le processus de la victime. De plus, lorsqu'il est utilisé avec une politique d'ordonnancement personnalisée (présentée dans [76]), il offre de meilleures performances avec une combinaison des avantages de SCHED_DEADLINE et SCHED_RR.

Enfin, le chapitre 7 propose une synthèse des résultats d'expérimentations et des recommandations méthodologiques sur l'utilisation de Linux dans un environnement spatial.

Bibliography

[1] Why space radiation matters, 2022.

[2] Thomas Beck, Frédéric Boniol, Jérôme Ermont, and Luc Maillet. Impact of environment on the execution of a real-time linux process on a multicore platform. In *Proceeding of the 11th European Congress on Embedded Real Time Software and Systems*, 2022.

[3] Thomas Beck, Frédéric Boniol, Jérôme Ermont, Luc Maillet, and Franck Wartel. An automata-based method for interference analysis in multi-core processors. In *15th Junior Researcher Workshop on Real-Time Computing 2022*, page 7, 2022.

[4] Thomas Beck, Frédéric Boniol, Jérôme Ermont, and Luc Maillet. Mutual perturbations of fprime applications in a space context. *ETR 20*, page 29, 2021.

[5] Elisa project, 2022.

[6] Sil2linux, 2014.

[7] Automotive grade linux, 2022.

[8] The official gihub repository of linux, 2022.

[9] Xavier Jean, Laurence Mutuel, Didier Regis, Hélène Misson, Guy Berthon, and Marc Fumey. White Paper on Issues Associated with Interference Applied to Multicore Processors, 2016. Retrieved from `http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/media/SDS_DO005_White_Paper.pdf`.

[10] Certification Authorities Software Team. Multi-core Processors - Position Paper. Technical Report CAST 32-A, Certification Authorities Software Team, November 2016.

[11] Irune Agirre, Jaume Abella, Mikel Azkarate-askasua, and Francisco J. Cazorla. On the tailoring of CAST-32A certification guidance to real COTS multicore architectures. In *12th IEEE International Symposium on Industrial Embedded Systems, SIES 2017, Toulouse, France, June 14-16, 2017*, pages 1–8. IEEE, 2017.

[12] Xavier Jean, Laurence Mutuel, and Vincent Brindejonc. Assurance methods for cots multi-cores in avionics. In *35th Digital Avionics Systems Conference (DASC'16)*, 2016.

[13] Frédéric Boniol, Claire Pagetti, and Nathanaël Sensfelder. Identification of multi-core interference. In *Proceedings of the 19th IEEE High Assurance Systems Engineering Symposium (HASE'19)*, 2019.

[14] Frédéric Boniol, Youcef Bouchebaba, Julien Brunel, Kevin Delmas, Thomas Loquen, Alfonso Mascarenas Gonzalez, Claire Pagetti, Thomas Polacsek, and Nathanaël Sensfelder. PHYLOG certification methodology: a sane way to embed multi-core processors. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, Toulouse, France, January 2020.

[15] Claire Maiza, Hamza Rihani, Juan Maria Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Comput. Surv.*, 52(3):56:1–56:38, 2019.

[16] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In *10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

[17] Thomas Carle and Hugues Cassé. Reducing timing interferences in real-time applications running on multicore architectures. In *18th International Workshop on Worst-Case Execution Time Analysis*, 2018.

[18] Georgia Giannopoulou, Kai Lampka, Nikolay Stoimenov, and Lothar Thiele. Timed model checking with abstractions: towards worst-case response time analysis in resource-sharing manycore systems. In Ahmed Jerraya, Luca P. Carloni, Florence Maraninchi, and John Regehr, editors, *Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012*, pages 63–72. ACM, 2012.

[19] Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In *Proceedings of the 2012 Ninth European Dependable Computing Conference*, EDCC '12, pages 132–143, Washington, DC, USA, 2012. IEEE Computer Society.

[20] Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4), January 2012.

[21] Bertrand Putigny. Benchmark-driven approaches to performance modeling of multi-core architectures. 03 2014.

[22] Jingyi Bin, Sylvain Girbal, Daniel Gracia Perez, Arnaud Grasset, and Alain Merigot. Studying co-running avionic real-time applications on multi-core cots architectures. In *Embedded Real Time Software and System Conference (ERTS'14)*, 2014.

[23] Sylvain Girbal, Jingyi Bin, Daniel Gracia Perez, and Alain Merigot. Using monitors to predict co-running safety-critical hard real-time benchmark behavior. In *Conference on Information and Communication Technology for Embedded Systems (ICITES'14)*, 01 2014.

[24] Cédric Courtaud, Julien Sopena, Gilles Muller, and Daniel Gracia. Improving Prediction Accuracy of Memory Interferences for Multicore Platforms. In *RTSS 2019 - 40th IEEE Real-Time Systems Symposium*, Hong-Kong, China, December 2019. IEEE.

[25] Sylvain Girbal, Jimmy le Rhun, and Hadi Saoud. METrICS: a measurement environment for multi-core time critical systems. In *9th European Congress on Embedded Real Time Software and Systems (ERTS'18)*, 2018.

[26] Alfonso Mascareñas González, Youcef Bouchebaba, and Luca Santinelli. Multicore shared memory interference analysis through hardware performance counters. In *10thEuropean Congress on Embedded Real Time Software andSystems(ERTS 2020)*, Toulouse, France, January 2020.

[27] Roy Jamil, Emmanuel Grolleau, Bernard Dautrevaux, and Antoine Bertout. Measurement-based timing analysis on heterogeneous mpsocs: A practical approach. In Henry Muccini, Paris Avgeriou, Barbora Buhnova, Javier Camara, Mauro Caporuscio, Mirco Franzago, Anne Koziolek, Patrizia Scandurra, Catia Trubiani, Danny Weyns, and Uwe Zdun, editors, *Software Architecture*, pages 279–293, Cham, 2020. Springer International Publishing.

[28] Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. Modeling Cache Coherence to Expose Interference. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International*

Proceedings in Informatics (LIPIcs), pages 18:1–18:22, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[29] Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. On how to identify cache coherence: Case of the NXP qoriq T4240. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems, ECRTS 2020, July 7-10, 2020, Virtual Conference*, volume 165 of *LIPIcs*, pages 13:1–13:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[30] Viet anh Nguyen, Eric Jenn, Wendelin Serwe, Frederic Lang, and Radu Mateescu. Using Model Checking to Identify Timing Interferences on Multicore Processors. In *ERTS 2020 - 10th European Congress on Embedded Real Time Software and Systems*, pages 1–10, Toulouse, France, January 2020.

[31] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Åkesson, Vincent Nélis, and Thomas Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *28th Euromicro Conference on Real-Time Systems*, July 2016.

[32] Quentin Perret, Pascal Maurère, Éric Noulard, Claire Pagetti, Pascal Sainrat, and Benoît Triquet. Mapping hard real-time applications on many-core processors. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016*, pages 235–244, 2016.

[33] Etienne Borde, Smail Rahmoun, Fabien Cadoret, Laurent Pautet, Frank Singhoff, and Pierre Dissaux. Architecture models refinement for fine grain timing analysis of embedded systems. In *2014 25nd IEEE International Symposium on Rapid System Prototyping*, pages 44–50. IEEE, 2014.

[34] Jérôme Hugues, Thomas Vergnaud, Laurent Pautet, Yann Thierry-Mieg, Soheib Baarir, and Fabrice Kordon. On the formal verification of middleware behavioral properties. *Electronic Notes in Theoretical Computer Science*, 133:139–157, 2005.

[35] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, jan 1973.

[36] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (bvt) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, page 261–276, New York, NY, USA, 1999. Association for Computing Machinery.

[37] Stéphane Rubini, Christian Fotsing, Frank Singhoff, Hai Nam Tran, and Pierre Dissaux. Scheduling analysis from architectural models of embedded multiprocessor systems. *ACM SIGBED Review*, 11(1):68–73, 2014.

[38] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar: A flexible real time scheduling framework. In *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time Distributed Systems Using Ada and Related Technologies*, SIGAda '04, page 1–8, New York, NY, USA, 2004. Association for Computing Machinery.

[39] Joël Goossens, Emmanuel Grolleau, and Liliana Cucu-Grosjean. Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms. *Real-time systems*, 52(6):808–832, 2016.

[40] Boris Teabe, Alain Tchana, and Daniel Hagimont. Application-specific quantum for multi-core platform scheduler. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–14, 2016.

[41] John Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB, 2000.

[42] Jan Bredereke. A survey of time and space partitioning for space avionics. Technical report, Hochschule Bremen, 2017.

[43] ARINC Specification. 653: Avionics application software standard interface, part 1-required services, aeronautical radio. *Inc., November*, 2010.

[44] Julien Delange, Laurent Pautet, Alain Plantec, Mickael Kerboeuf, Frank Singhoff, and Fabrice Kordon. Validate, simulate, and implement arinc653 systems using the aadl. *Ada Lett.*, 29(3):31–44, nov 2009.

[45] Xavier Jean, David Faura, Marc Gatti, Laurent Pautet, and Thomas Robert. Ensuring robust partitioning in multicore platforms for ima systems. In *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pages 7A4–1. IEEE, 2012.

[46] James Windsor. Integrated modular avionics for spacecraft — user requirements, architecture and role definition. In *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*, pages 1–17, 2011.

[47] James Windsor and Kjeld Hjortnaes. Time and space partitioning in spacecraft avionics. *Space Mission Challenges for Information Technology, IEEE International Conference on*, 0:13–20, 07 2009.

[48] Sean Campbell and Michael Jeronimo. An introduction to virtualization. *Published in "Applied Virtualization", Intel*, pages 1–15, 2006.

[49] Katherine K. Sheridan-Barbian. A survey of real-time operating systems and virtualization solutions for space systems, 2015-03.

[50] 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*, 55(1):435–439, 2008.

[51] Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 209–222, New York, NY, USA, 2010. Association for Computing Machinery.

[52] Gernot Heiser and Ben Leslie. The okl4 microvisor: Convergence point of microkernels and hypervisors abstract. pages 19–24, 06 2010.

[53] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 164–177, New York, NY, USA, 2003. Association for Computing Machinery.

[54] Alfons Crespo, Patricia Balbastre, José Simó, Javier Coronel, Daniel Gracia Pérez, and Philippe Bonnot. Hypervisor-based multicore feedback control of mixed-criticality systems. *IEEE Access*, 6:50627–50640, 2018.

[55] Geetishree Mishra, V Ashwini, and Soumya Sunkara. Virtualization on arm embedded platform codezero hypervisor-a case study. In *2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS)*, pages 1–4, 2018.

[56] Gaurav Banga, Peter Druschel, and Jeffrey C Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, volume 99, pages 45–58, 1999.

[57] Konstantinos Konstantopoulos. Cache-partitioning for cots multicorearchitecture, 2017.

[58] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong 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, PACT '12, page 367–376, New York, NY, USA, 2012. Association for Computing Machinery.

[59] Hannu Leppinen. Current use of linux in spacecraft flight software. *IEEE Aerospace and Electronic Systems Magazine*, 32(10):4–13, 2017.

[60] Håvard Grip, Johnny Lam, David Bayard, Dylan Conway, Gurkirpal Singh, Roland Brockers, Jeff Delaune, Larry Matthies, Carlos Malpica, Travis Brown, Abhinandan Jain, Alejandro Martin, and Gene Merewether. Flight control system for nasa's mars helicopter. In *AIAA Scitech 2019 Forum*, 01 2019.

[61] Björn Brandenburg and James Anderson. Joint opportunities for real-time linux and real-time systems research. *11th Real Time Linux Workshop*, 01 2009.

[62] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The real-time linux kernel: A survey on preempt-rt. *ACM Comput. Surv.*, 52(1), feb 2019.

[63] Daniel Bristot de Oliveira and Romulo Oliveira. Timing analysis of the preempt rt linux kernel. *Software: Practice and Experience*, 46:n/a–n/a, 05 2015.

[64] Daniel Bristot de Oliveira, Daniel Casini, Rômulo Oliveira, and Tommaso Cucinotta. Demystifying the real-time linux scheduling latency. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, 07 2020.

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

[66] 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*, pages 133–142, 2002.

[67] Paul Mckenney and Dipankar Sarma. Towards hard realtime response from the linux kernel on smp hardware. In *In linux.conf.au 2005*, 01 2005.

[68] Jeremy H Brown and Brad 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)*, pages 1–17, 2010.

[69] Dario Faggioli, Fabio Checconi, Scuola Superiore, Sant Anna, Michael Trimarchi, and Claudio Scordino. An edf scheduling class for the linux kernel. In *proc. of the 11th Real-Time Linux Workshop (RTLWS'11)*, 01 2009.

[70] Tommaso Cucinotta. An efficient and scalable implementation of global edf in linux. In *proc. of the 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'11)*, pages 6–15, 01 2011.

[71] Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline scheduling in the linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016.

[72] Roman Obermaisser and Bernhard Leiner. Temporal and spatial partitioning of a time-triggered operating system based on real-time linux. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 429–435, 2008.

[73] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. *SIGPLAN Not.*, 53(2):405–418, mar 2018.

[74] Jungho Kim, Philkyue Shin, Myungsun Kim, and Seongsoo Hong. Memory-aware fair-share scheduling for improved performance isolation in the linux kernel. *IEEE Access*, 8:98874–98886, 2020.

[75] Jungho Kim, Philkyue Shin, Soonhyun Noh, Daesik Ham, and Seongsoo 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)*, pages 215–220, 2018.

[76] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. Container-based real-time scheduling in the linux kernel. *ACM SIGBED Review*, 16(3):33–38, 2019.

[77] Imanol Allende, Nicholas Mc Guire, Jon Perez-Cerrolaza, Lisandro G. Monsalve, Jens Petersohn, and Roman Obermaisser. Statistical test coverage for linux-based next-generation autonomous safety-related systems. *IEEE Access*, 9:106065–106078, 2021.

[78] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994.

[79] Gerd Behrmann, Alexandre David, and Kim Larsen. A tutorial on uppaal. In *Formal Methods for the Design of Real-Time Systems*, volume 3185, pages 200–236, 01 2004.

[80] The github repository of the xilinx linux kernel, 2021.

[81] Xilinx wiki on yocto, 2021.

[82] Yocto zeus version repository, 2021.

[83] Lttng website, 2022.

[84] Trace compass website, 2022.

[85] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[86] Frédéric Boniol, Youcef Bouchebaba, Julien Brunel, Jean-Loup Bussenot, Kevin Delmas, Anthony Fernandes-Pires, Claire Pagetti, and Thomas Polacsek. Plan de management et programme prévisionnel de travail de la convention PHYLOG 2 – Etude de la certifiabilité des architectures logiciel-matériel reposant sur des calculateurs hybrides (DOW). Technical Report RT 1/31759 DTIS, ONERA, August 2021.

[87] The Linux man-pages project. clock_nanosleep(2) linux man page, 2021.

[88] Sparsh Mittal. A survey of techniques for cache partitioning in multicore processors. *ACM Comput. Surv.*, 50(2), may 2017.

Acronyms

ALU Arithmetic Logic Unit.

AOCS Attitude and Orbit Control System.

ASIC Application-Specific Integrated Circuit.

COTS Commercial-off-the-shelf.

CPU Central Processing Unit.

CSU Configuration and Security Unit.

ECC Error Correction Code.

ECSS European Cooperation for Space Standardization.

ESA European Space Agency.

FPGA Field-Programmable Gate Array.

GPU Graphical Processing Unit.

IMA Integrated Modular Avionics.

MCP Multi Core Platform.

MMU Memory Management Unit.

NASA National Aeronautics and Space Administration.

PMU Platform Management Unit.

RAM Random-access Memory.

SoC System on Chip.

TLB Translation Lookaside Buffer.

WCET Worst-Case Execution Time.