INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Real-time TAsk-Splitting scheduling algorithms framework

JOÃO FRANCISCO DE CASTRO CASTANHO CORREIA
Outubro de 2015

POLITÉCNICO
DO PORTO

Real-time TAsk-Splitting scheduling algorithms framework

João Francisco de Castro Castanho Correia

Dissertação para a obtenção do Grau de Mestre em
Engenharia Informática

Área de especialização em **Arquiteturas, Sistemas e Redes**

Orientador: Doutor Paulo Manuel Baltarejo de Sousa

Júri:
Presidente: Xpto1,
Professor Adjunto no Departamento de Engenharia Informática
do Instituto Superior de Engenharia do Porto
Vogais: Xpto2,
Professor Coordenador no Departamento de Engenharia Informática
do Instituto Superior de Engenharia do Porto

Porto, Outubro de 2015

Acknowledgements

Despite the numerous pitfalls that this area of research presents, it can become a less painful experience when one is accompanied by the right people. For that reason, I would like to thank professor Paulo Baltarejo for his great guidance through this voyage in my life. His, in my humble opinion, great wisdom regarding multiprocessor scheduling and Linux kernel development, definitively proved to be of great importance, because if I did not had his help, the same level of quality and usefulness would not be expected out of this work.

Then, I would also like to thank professor Eduardo Tovar and Konstantinos Bletsas, for allowing me to proceed with my work at CISTER. This demands a great level of trust. A special thanks also goes to the technical staff at the research center for their great availability and competence.

To all my family members, specially my parents, because they allowed me to follow my dreams, their resilience towards my conduct at home proved to be of great value, and their emotional support was indispensable. Finally, I am very thankful to all my friends from, Viana do Castelo, and more importantly those from Lanheses, the small town from which I am a proud resident, for keeping me busy at weekends. I would not be able to have outstanding moments in my life without them.

Resumo Alargado

Nos dias de hoje, os sistemas de tempo real crescem em importância e complexidade. Mediante a passagem do ambiente uniprocessador para multiprocessador, o trabalho realizado no primeiro não é completamente aplicável no segundo, dado que o nível de complexidade difere, principalmente devido à existência de múltiplos processadores no sistema. Cedo percebeu-se, que a complexidade do problema não cresce linearmente com a adição destes. Na verdade, esta complexidade apresenta-se como uma barreira ao avanço científico nesta área que, para já, se mantém desconhecida, e isto testemunha-se, essencialmente no caso de escalonamento de tarefas.

A passagem para este novo ambiente, quer se trate de sistemas de tempo real ou não, promete gerar a oportunidade de realizar trabalho que no primeiro caso nunca seria possível, criando assim, novas garantias de desempenho, menos gastos monetários e menores consumos de energia. Este último fator, apresentou-se desde cedo, como, talvez, a maior barreira de desenvolvimento de novos processadores na área uniprocessador, dado que, à medida que novos eram lançados para o mercado, ao mesmo tempo que ofereciam maior performance, foram levando ao conhecimento de um limite de geração de calor que obrigou ao surgimento da área multiprocessador. No futuro, espera-se que o número de processadores num determinado chip venha a aumentar, e como é óbvio, novas técnicas de exploração das suas inerentes vantagens têm de ser desenvolvidas, e a área relacionada com os algoritmos de escalonamento não é exceção.

Ao longo dos anos, diferentes categorias de algoritmos multiprocessador para dar resposta a este problema têm vindo a ser desenvolvidos, destacando-se principalmente estes: globais, partiticionados e semi-partiticionados. A perspetiva global, supõe a existência de uma fila global que é acessível por todos os processadores disponíveis. Este fato torna disponível a migração de tarefas, isto é, é possível parar a execução de uma tarefa e resumir a sua execução num processador distinto. Num dado instante, num grupo de tarefas, m, as tarefas de maior prioridade são selecionadas para execução. Este tipo promete limites de utilização altos, a custo elevado de preempções/migrações de tarefas. Em contraste, os algoritmos particionados, colocam as tarefas em partições, e estas, são atribuídas a um dos processadores disponíveis, isto é, para cada processador, é atribuída uma partição. Por essa razão, a migração de tarefas não é possível, acabando por fazer com que o limite de utilização não seja tão alto quando comparado com o caso anterior, mas o número de preempções de tarefas decresce significativamente.

O esquema semi-particionado, é uma resposta de caráter hibrido entre os casos anteriores, pois existem tarefas que são particionadas, para serem executadas exclusivamente por um grupo

de processadores, e outras que são atribuídas a apenas um processador. Com isto, resulta uma solução que é capaz de distribuir o trabalho a ser realizado de uma forma mais eficiente e balanceada. Infelizmente, para todos estes casos, existe uma discrepância entre a teoria e a prática, pois acaba-se por se assumir conceitos que não são aplicáveis na vida real. Para dar resposta a este problema, é necessário implementar estes algoritmos de escalonamento em sistemas operativos reais e averiguar a sua aplicabilidade, para caso isso não aconteça, as alterações necessárias sejam feitas, quer a nível teórico quer a nível prático. Adicionalmente, os métodos de obtenção de resultados também têm de ser melhorados, para que o trabalho de investigação necessário seja feito com maior graciosidade.

Nesta dissertação, apresenta-se uma framework concebida num sistema operativo real, neste caso Linux, que implementa uma série de algoritmos de escalonamento semi-particionados em ambiente multiprocessador, de uma forma genérica e modular, para que no futuro, novas adições possam ser incluídas. Documentação detalhada também é fornecida, para que um terceiro também possa tirar partido do que foi desenvolvido. Presente também está, um mecanismo de obtenção de resultados, quer a nível do que é feito durante o escalonamento no que diz respeito a eventos, quer a nível de dados estatísticos.

Apesar de no início deste trabalho, uma versão inicial desta framework já ter estado disponível, neste momento tornou-se ainda mais modular, aproveitando algumas das novas características herdadas por uma versão mais recente do núcleo do Linux, e de uma nova organização do código fonte que tinha sido produzido até então, e que, também resultou em novas funcionalidades. Entre estas, destaca-se a capacidade de se poder escolher o algoritmo uniprocessor a ser utilizado durante o escalonamento.

Posto isto, possibilitou-se a utilização de algoritmos de prioridades fixas ao nível da tarefa durante o escalonamento a ser realizado. Isto obrigou à conceção de uma nova análise de escalonabilidade que lida diretamente com este caso e tal também é apresentado neste documento. Até agora, apesar deste trabalho ainda necessitar de alguns melhoramentos, resultados positivos foram obtidos.

Para facilitar enormemente o processo de recolha de resultados, e de melhoramento de produção e/ou testes dos algoritmos implementados, uma ferramenta foi concebida para que, ao consumir os dados produzidos durante o escalonamento, uma representação visual se construa, e isto é feito sobe a forma de um diagrama de Gantt, em que tudo o que é representado é facilmente manipulado/filtrado.

Após se terem feitos vários testes, concluiu-se que o trabalho efetuado por todos estes componentes aqui desenvolvidos, mostram-se bastante promissores no que diz respeito à criação de novo trabalho de investigação nesta área.

Palavras-chave: Sistemas multiprocessador, Escalonamento multiprocessador de tempo-real, Escalonamento semi-particionado, Análise de escalonabilidade, Linux

Abstract

Nowadays, real time systems grow in importance and complexity. While moving from the uniprocessor to the multiprocessor environment, the tools available can not be directly used from one to the other because the level of complexity is significantly greater. This complexity grows with the number of processors available, and surely, this fact is what is keeping this area of research from advancing quicker, and this is even more true for real time task scheduling.

This advancement, from one processor to multiple, promises a number of advantages, such as: doing work that would not be possible in the uniprocessor environment, resulting, in new performance guarantees, lower monetary costs, and less power consumption. Heat generation, imposed a limit in uniprocessor advancement making it an unreliable path to take, and therefore, the multiprocessor is, perhaps, the way to go in the future. It is expected that the number of processors in a given chip is going to grow, and this imposes the necessity to develop new technologies to fully exploit the resources available, and real time multiprocessor scheduling is no exception.

Different multiprocessor scheduling categories were developed throughout the years. The global scheme uses a global queue that is accessible to all processors. This enables task migrations, high utilization bounds but at the cost of many task preemptions and migrations. To reduce these, the partitioned scheme was created. It allocates tasks to partitions, and these are mapped to processors. Unfortunately, this removes the possibility to migrate tasks, and grants lower utilization bounds. To inherit the advantages of both schemes, the semi-partitioned, or task-splitting scheme was devised, allowing some tasks to execute in several processors, and others to be executed by one. This is done to achieve better scheduling guarantees by balancing the work more efficiently.

This dissertation, describes the construction of a framework that implements some of these algorithms. This work aims to deal with the real difficulties found when doing such in a real operating system. Schedulability analysis, which is a must, is also provided for a specific environment that is generated by the framework's usage, namely, for task-fixed priority algorithms as the uniprocessor scheduling algorithm used in the on-line scheduling procedure. Finally, to improve the research process, a piece of software was been conceived that generates a graphical representation of the scheduling phase. It is shown, that by using these tools, realistic and positive results can be obtained.

Keywords: Multiprocessor Systems, Multiprocessor Real-Time Scheduling, Semi-Partitioned Scheduling, Schedulability Analysis, Linux

Contents

List of Figures

List of Tables

Acronyms

CFS	Completely Fair Scheduling.
CPU	Central Processing Unit.
DM	Deadline-Monotonic.
EDF	Earliest Deadline First.
FIFO	First-In-First-Out.
GPOS	General Purpose Operating System.
HRT	High-Resolution Timer.
IPI	Inter-Processor Interrupt.
LLF	Least-Laxity First.
NPS-F	Notional Processor Scheduling - Fractional capacity.
NUMA	Non-Uniform Memory Access.
OS	Operating System.
ReTAS	Real-time TAsk-Splitting Scheduling Algorithms Framework.
RM	Rate-Monotonic.
RR	Round-Robin.
RT	Real-time.
RTA	Response-time Analysis.
RTOS	Real-Time Operating System.
RTS	Real-Time System.

S-EKG Sporadic-EKG.

UMA Uniform Memory Access.

WCET Worst-Case Execution Time.

Chapter 1

Introduction

With the current hardware's advancement, the software available must be prepared for it. Regarding uniprocessor real time computing, that was originated since the launching of the first spaceships to the moon, there is software available and mature scheduling theory. In the multiprocessor case, that is not true, the scheduling theory available is still new, and there are many problems to be addressed. The work presented in this dissertation aims to address some of them.

First, by presenting new unified schedulability analysis for fixed-task uniprocessor algorithms under semi-partitioned scheduling. Second, in depth documentation regarding a framework developed in combination with the Linux kernel, and finally, a piece of software that offers a number of advantages, such as, testing semi-partitioned algorithms.

1.1 Motivation

As it is known, through time, software complexity grows, and to be able to respond with this demand, computer manufacturers simply provided processors with more clock speed. Heat generation and power consumption created a big problem for uniprocessor[1] systems [Lowney, 2006], and therefore, advancements in this area started to be very difficult to make. This problem began to be solved with the introduction of multiprocessor systems, that, allowed multiple cores (or processing units) to work simultaneously, where each core runs at a lower frequency, which in turn, reduces the power consumption and the heat generation. This change for multiprocessor systems, showed itself very promising and widely accepted [Sodan et al., 2010]. The problem with this newer perspective, is that it triggered newer challenges to be met. The techniques used in uniprocessor systems could not be directly applied, and software developers needed to create newer strategies to fully exploit the resources available, that is, the processing units available.

For real time systems too, multiprocessing can be very useful to obtain better performance and less power consumption. In, [Lee et al., 2007], it is shown that these systems are becoming, more and more important due to their pervasiveness, since most infrastructures depend on them. Additionally, multiple processors working together increase the computation capacity.

[1] In this dissertation, uniprocessor or single-processor systems are those were there is only one processing unit/core available. In contrast, in multiprocessor systems there are multiple processing units/cores available.

In [Lee et al., 2007], a definition of a real time system can be found. Their correctness and quality is defined not just by the computation results that they produce, but also by the time in which they are able to do so, that is, for each task that is executed there is a time constraint often called *deadline*. A task may not coexist alone, but yet various can, to which this group is normally designated as *task set*.

Like in any other computational systems, in real time computing there must be software that is prepared to deal with real time functionalities. A general purpose operating system is not prepared for them, time control and exactness is of utmost importance, and the operating system components must be aware of the task's temporal constraints, with special emphasis on the task scheduler. This is why it is needed a *scheduling algorithm*[2] in place to be able to execute and satisfy the task's demands. Scheduling algorithms can also be compared and evaluated using their *utilization bound*, which is a limit that defines the maximum amount of work that can be done by a task scheduled according to one scheduling algorithm, which, if surpassed, will result in tasks not meeting their deadlines. Furthermore, these systems must be designed and implemented so that they do not fail, given that in some instances catastrophic results may occur. This can be assured with a *schedulability test*, which allows, during the system's design phase, to conclude if a task set is *feasible*, that is, that all of its tasks will meet their deadlines.

The uniprocessor scheduling algorithms devised, and their implementation techniques, again, could not be directly used, and this justified the act of new multiprocessor scheduling algorithms schemes being devised, and three main categories were, at least, created: *global*, *partitioned*, and *semi-partitioned*.

In the global scheme there is a global queue in which every processor can access to fetch tasks. This scheme assures that at any time instant the highest priority tasks are executing on m processors (assuming that the system is composed by m processing units). Task migration is possible, that is, a task can change from one processor to another during its execution span. This scheme often allows a very high utilization bound (some can achieve 100%), but the global shared queue's existence imposes the need of a locking mechanism that is used to control its access.

Partitioned scheduling, as its name implies, distributes the existing tasks into partitions, and these are assigned to processors. A processor can only be working with one partition at a given instant which results in tasks not being allowed to migrate among processors. The utilization bound is smaller than the global scheme(only 50% can be achieved). This scheme has the advantage of simplifying the scheduling problem, because each partition is seen as a uniprocessor system. Furthermore, these algorithms are, in fact, composed by two algorithms, one that is used off-line, during the design phase, and another at run-time. The first, allocates tasks to processors, and it presents itself as a bin-packing problem that is also known to be NP-hard [Coffman et al., 1997]. The second, schedules them at run-time, and it is a uniprocessor scheduling algorithm. One example is Earliest Deadline First, where the highest priority task is the one with the most urgent deadline.

In an attempt to inherit the advantages and reduce the disadvantages given by the previous

[2]This term is often used by the real time computing community, not just to describe the algorithm but the overall scheduling mechanism in existence. Another term used is scheduling policy, or simply policy.

schemes, semi-partitioned, or Task-Splitting scheduling, was devised. Under this scheme, most tasks execute in one single processor (non-split tasks), and others are split between them (split tasks). It is important to know that its not the task's code that is split, but yet their execution demand. Like in the partitioned case, these are also composed by two algorithms, one, off-line, to assign tasks to processors, and another, on-line, to schedule tasks at run-time.

Unfortunately, as affirmed in [Baker, 2010], the theory devised, usually does not have correspondence in practice, and it is, therefore, very important to actually try to implement these multiprocessor scheduling algorithms in real operating systems. Task context switches for example, that is, the act of switching from one task to another at run-time, the computational resources spent at the task's release, or even inter-processor communication, impose at least one issue to be addressed. This issue refers to the fact that they can affect the scheduling process in the long run, since they consume computational resources which could lead to deadlines not being met. Ultimately, their behaviour must be studied.

1.2 Contributions

Upon this, the work described in this dissertation, provides several research tools that were created to aid the research process in the area of multi-processor semi-partitioned scheduling:

- Because there is only unified schedulability analysis for semi-partitioned algorithms that uses Earliest Deadline First as its uniprocessor scheduling algorithm, unified schedulability analysis is proposed using uniprocessor fixed-task priority algorithms, such as, Rate Monotonic and Deadline Monotonic. This is motivated by the fact that fixed-task priority algorithms are widely used in the real world and it would be interesting to observe how they behave in combination to semi-partitioned algorithms.

- At the beginning of this work an implementation of the framework was already in place, but during this period, some improvements were done. A new Linux kernel is used.The current framework [3] version is now even more unified and optimized and with a few newer functionalities. The documentation needed for a third-party to understand it also present. This is very important to promote the research methodologies used, so that new research work may be produced.

- The final contribution is a piece of software that uses the scheduling information that is generated during the scheduling process. With this, it is capable to draw a Gantt diagram that provides a visual representation of what happened. The results shown can be navigated and filtered. Finally, the statistical information, is used to draw box-and-whisker plots for testing purposes.

[3] The current version of the framework is available, for installation and testing purposes. Please, visit this website at the url: www.cister.isep.ipp.pt/retas for further information.

1.3 Institutional Support

The work present in this document was devised in the context of the EMC^2 project, belonging to the ARTEMIS program, with the support provided by CISTER (Research Centre in Real-Time and Embedded Computing Systems) which is a top-ranked Research Unit based at the School of Engineering (ISEP) of the Polytechnic Institute of Porto (IPP), Portugal.

The IPP-HURRAY research group, created in mid 1997, is the core and genesis of the CISTER Research Unit. HURRAY stands for HUgging Real-time and Reliable Architectures for computing sYstems. Therefore, the research unit focuses its activity in the analysis, design and implementation of real-time and embedded computing systems.

1.4 Outline

The overall document is organized in the following fashion:

- Chapter 2 provides the fundamental background regarding multiprocessor real-time systems. In addition, it is also offered an explanation regarding the hardware that is used. Linux kernel development basics are also provided.
- Chapter 3 focuses on real-time multiprocessor scheduling. First by providing the basic concepts and then by exploring the most relevant work in the area. At the end, several algorithms are explained, because they are the ones available at the implementation level.
- Chapter 4 provides the overall theory behind unified schedulability analysis for fixed-task scheduling algorithms under semi-partitioned scheduling. This is done by exploring some concepts around response-time analysis and then by providing an in depth explanation to all algorithms and mathematical expressions devised.
- Chapter 5 offers an in depth study over the framework that was devised, focusing on its most important underlying mechanisms.
- Chapter 6 refers to the prototype's implementation used to obtain the visual representation about the scheduling that is done using the framework.
- Chapter 7 explores the experimental procedure that must be employed to develop an application that communicates with the framework to obtain results. Furthermore, several tests are shown, combining all the work devised.
- Chapter 8 is aimed at giving the final conclusions regarding the work that was done, emphasising on what was been achieved, and what can still be done in the future.

Chapter 2

Multiprocessor Real-Time Systems

2.1 Introduction

Nowadays, real-time systems have an important role in society, but unfortunately most people lack the important awareness that this type of computing really deserves. Because more and more organizations depend on these, and with the introduction of multiprocessor systems in the market, it becomes important to transport their existence to this environment and, this is justified, because, it is possible to do more as it is in the latter case.

This chapter is aimed to provide an understanding of what real-time systems are. Furthermore, perhaps the most important knowledge around the software and hardware used in this work is described. This is achieved by exploring some Operating System(OS)'s details, with emphasis on the Linux's kernel scheduling mechanism.

2.2 Definition of real-time computing

A Real-Time System(RTS) is one where its correctness, not only depends on its functionalities but also on its timing constraints [Stankovic, 1988], or in other words, for each computational requirement there is a temporal requirement too, which is usually designated as deadline. These temporal restrictions may differ in nature, an inexact result that is provided faster, might have more quality than another that is exact but slower. In other systems, it might be tolerable to fail some of these constraints, where in others, this cannot happen. This is the case of a hard-RTS, where catastrophic results may occur if such an event ever happens, but in a soft-RTS, such restriction does not exist, allowing the occurrence of deadline misses, which results in decreasing the quality of the result produced.

Physically speaking, usually it is considered the existence of two sub-systems, one which controls, and therefore is the computational system, and another that is controlled, thus constituting the environment that is interacted. A water pump mechanism[1] can be taken as an example.

Information about this environment is then provided, by the usage of, for example, sensors, to the controlling system. This corresponds to its current state and it must be consistent with

[1] A system where its main responsibility is to control the water's flow to pump it in the future.

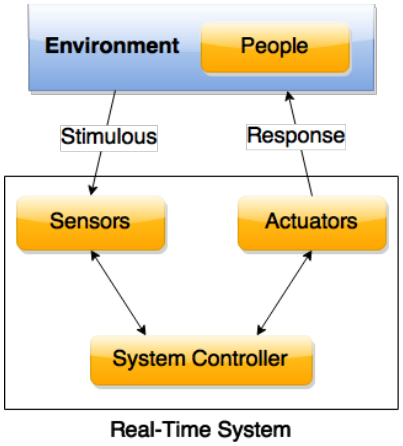


Figure 2.1: Example of a RTS scheme

the data that the controlling system has. If this requirement is not met, then, depending on the system, problematic output may be attained. Figure 2.1 shows a common scheme regarding the layout of a RTS.

Another important characteristic to discuss is reliability. Every computation that must be performed is analysed according to a specific design before being used in the final system, and this results in execution guarantees. Furthermore, the system's characteristics are known in advance before it is put into usage, making its activities to be studied and determined off-line. The system becomes inflexible, but in the other hand, enables higher performance during its operation.

They exist on a wide spectrum of environments such as: robotics, avionics, auto-mobiles and the military, just to mention some. Their complexity shows growth, where they range from simple microprocessors, to highly sophisticated, complex and distributed systems [Stankovic et al., 1992].

Another fact which makes these systems evolve, is the hardware advancement. But this also imposes the creation of more sophisticated design and analysis techniques. With the appearance of multiprocessor systems, it is expected that they should also appear in this realm, inheriting the benefits promised. For example, better performance, the reduction of physical space and less power consumption, enabling the execution of complex tasks, whose feasibility would not be achievable in a uniprocessor environment.

2.3 Multiprocessor computer architectures

This section is focused in giving a succinct explanation in computer architecture, describing some of its evolution, from the uniprocessor, to the more modern, multiprocessor solution. Throughout the last decades, the industry connected to processor development rapidly developed their technologies and made them available to the public. The market did not, in any way, shown any diminishing signs, since more and more computational systems were made available. In the long term, they required the improvement of their power consumption and performance[Gepner and Kowalik, 2006].

Performance, is nothing more than the quantity of time that needs to be spent to execute something, and it is calculated between the product of the processor's clock frequency, and the number of instructions executed during a clock cycle. Increasing the clock frequency is a known path to be taken to increase performance, but this is in fact a very turbulent path, as shown in [Gepner and Kowalik, 2006].

Considering the concepts mentioned, processor designers were able to maintain a balance between what is done during a processor cycle, and in the other hand, good levels of voltage and processor frequency. Unfortunately, by using uniprocessor systems it is impossible to proceed with performance growth, since heat generation is a serious problem. To control and get better power consumption levels, new architecture design methodologies were devised, making, perhaps, the multiprocessor scheme the way to go in the future [Lowney, 2006].

Furthermore, uniprocessor and multiprocessor systems still share some concepts. Most computers are based in the von Neumann's architecture as in: [Eigenmann and Lilja, 1998]. It supposes the existence of a Central Processing Unit(CPU), memory, and input/output devices. Each one is connected to a single bus shared between them.

In terms of responsibilities, the CPU[2] is the component that executes computer programs. It then uses data that is directly stored in the main memory. This makes the application execution speed not only dependable on the processor's capabilities, but also from the memory access speed. Caches are used to reduce computational costs behind memory access, and they are particularly useful because accessing the main memory can be much more slower than the processor speed, originating latency issues. Caches are smaller, and faster, than the their counterpart memory. Inside, it is possible to find temporal copies of data recently requested by the CPU belonging to a particular piece of software. They are also organised in hierarchies, which are then accessed before the CPU fetches the data from the main memory if it fails to find it in the different cache hierarchy levels.

The architecture itself, is nothing more than the common components, for example, the registers and instruction set, conveyed and maintained by the manufacturers during a period of time, and made, usually, publicly available.

With respect to multiprocessor architectures, there are different classification levels. Each of these have repercussions on the interactions between the architecture's main components, but the key idea, is that a multiprocessor system is in fact a single computer system that contains several processors in order to allow simultaneous processing and therefore all processors share the varied components available in the system. Unlike the *multitasking* concept, which allows the execution of tasks simultaneously on a single processor by creating several virtual entities that act as independent processors. In the former case, we have genuine concurrent processing and in the latter, there is a distinction between two or more components that simulate the existence of multiple processors, but only one instruction can be executed at a particular time, given that there is only have one processor.

Another characteristic, is *symmetric multiprocessing*, which assumes the existence of centralized shared memory where multiple *homogeneous* processors are available, that is, processors

[2] In this dissertation, the term CPU is used as a synonym for processor. Both terms are used arbitrarily.

that have exactly the same characteristics: architecture, performance, etc. In contrast, in a system where processors are not identical, the name *heterogeneous* is given.

Each processor has a distinct responsibility, allowing it to execute and deal with with information from a variety of sources. Additionally they can share information through a shared bus. Each processor also has its high-speed memory (cache), for the purpose of reducing traffic on the shared bus. Consider the Figure 2.2. This is an instance of the *shared memory model*, where memory accesses are uniform for each processor, thus, the designation: Uniform Memory Access(UMA) is given.

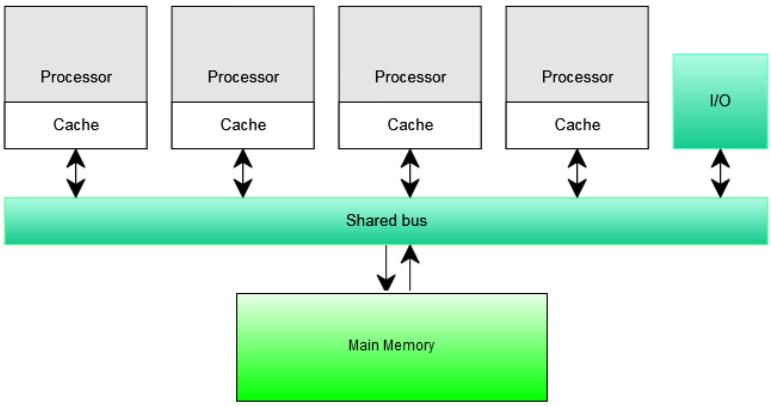


Figure 2.2: Example of a shared memory model with Uniform Memory Access

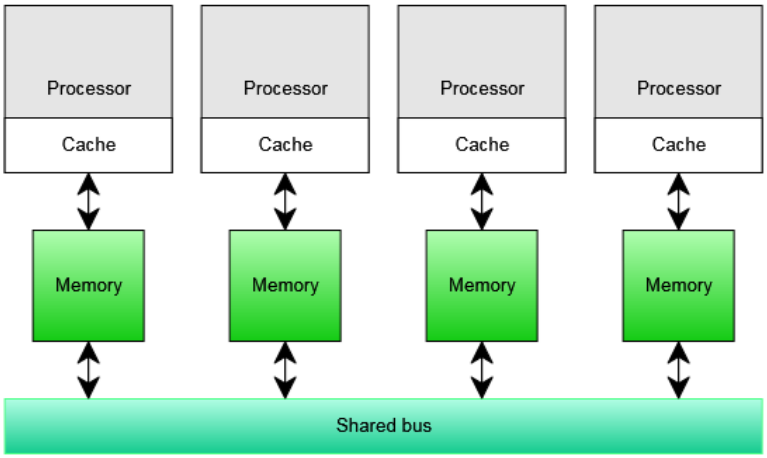


Figure 2.3: Example of a shared memory model with Non-Uniform Memory Access

In other types of systems, there is a high-speed network that interconnects the available processors, and it is used when a processor needs information from another. This is the *distributed memory model*. There is an inherent independence between processors, so the process of adding and removing them is promoted. The big problem is that it is more difficult to communicate among them, which imposes the need of a protocol to define the rules for such communication.

Another example of the shared memory model application, is Non-Uniform Memory Access(NUMA), where each processor has its own private memory that can be accessed rapidly.

By looking at Figure 2.3, the access to the memory via a shared bus is allowed by other processors at a more increased cost. Due to this difference in access speeds, regarding the information resident among various memories, hence, the non-uniform designation, this type of architecture has the advantage were it is very fast to access local memory in contrast to the previous cases.

2.4 The Linux Operating System

In this section, some of the software that can be used to construct a RTS is described. The first piece of software to study is the OS. This software, is responsible to manage the different pieces of hardware available to the computer and controlling the existing functionalities. This introduces a layer of security that is challenged by the development of any particular user-space application, given that in some cases, hardware must be accessed, and it is within the kernel-space that further validations must be performed.

There is software that is mainly responsible about the system's management, and others that provide functionality at a higher level. This first group of software runs on what is called *kernel-space*, and the higher level software, runs at *user-space*.

Currently at least two main groups of OSs exist, General Purpose Operating Systems(GPOSs) and Real-Time Operating Systems(RTOSs). The first type is aimed at the general public. The OS does its work but no temporal constraints exist. The second case, is aimed to RTSs. These must have other extra capabilities, such as, much better time management and a task scheduler that is aware of the time constraints associated with the computational work that must be performed.

One of the most widely used and successful GPOSs is Linux. This OS, is divided in its kernel, a set of user-space software and some additional libraries. It is mainly present in servers, mobile phones, televisions and home computers. The work developed in this dissertation, is implemented in this OS, and, Real-time(RT) capabilities can, therefore, be added. An example of this effort is the PREEMPT-RT patch, which is described in section 2.4.2. In the next subsection, the Linux's task scheduler is introduced, because the work developed uses it extensively.

2.4.1 The Linux kernel scheduler

The most important objective in this work is to use the Linux kernel in combination with the PREEMPT-RT's patch features in order to create better real-time capabilities in this OS, namely provide Linux kernel with real-time scheduling policies. To start this process, there are two main data structures that contain the most basic information needed among the scheduler implementation, the `task_struct` and the `struct_rq` structures. When a process spawns within the kernel, it is a mere instantiation of the `task_struct` structure, and it contains various fields that are vital through its lifespan. From the various parameters available, those in Table 2.1 are used in the context of task scheduling and are, perhaps, important to discuss.

When the system starts, a per-processor run-queue is used, and in resemblance to the previous case, its an instantiation of the `rq` structure. It contains information of interest regarding each task that coexists in a specific processor and this is done through the existence of per-class queues(RT,Completely Fair Scheduling(CFS)) and other parameters. Important examples are:

`curr` which is a pointer to the currently executing task and `lock` a variable of type `spinlock_t` to control the run-queue's access.

The lock variable can't be instantly obtained, it spins until it becomes available and it is used manage and update some of the information in the run-queue. When this happens, only one processor has access to it possessing the lock. This is very important to the scheduler's implementation, specially in the multiprocessor scheme, since various run-queue's locks may be needed to manage task migrations. This must be done always in the same order, so that deadlocks[3] may be mitigated.

Table 2.1: Some of the most important fields in the `task_struct` structure.

Structure field	**Description**
state	Refers to the current process state, some examples are: TASK_RUNNING, when the process is running, and TASK_STOPPED when it stops its execution.
prio	Priority number of a process. It is defined with a value known as Nice. This value is the processor's scheduling priority, where numbers lesser than 20 have higher priority and those greater than 19 less.
sched_class	Process's scheduling class. In the kernel there is a class for real-time tasks(RT) for example, but others are available according to the task's nature. More details are given later.
policy	Scheduling policy chosen according to the scheduling class. The RT scheduling class, as an example, has two policies available: SCHED_FIFO and SCHED_RR.

How time is managed in the kernel is also important to grasp, there are many components that require time management capabilities to ensure that their functionalities may further be exploited for the greater good of the kernel, like for example, data that is stored from a process that is being executed and the mechanism that is responsible to manage the balance for each processor's run-queues. It is important to work with the best time accuracy available. The Linux kernel is capable to use many hardware devices for time management purposes to obtain precise time values by gathering information from all of them, creating a quality metric to each, and normalizing the values obtained.

Upon this, the kernel has a periodic timer to manage the overall system, designated `tick`. In the implementation, there is a macro called `HZ` to specify its resolution. Typically it presents a resolution of one millisecond which clearly does not fulfil the requirements that real-time systems impose. To solve this problem, there are in existence High-Resolution Timers(HRTs) which offer nanosecond timer's resolution. To create a timer of this nature, one needs to go through different phases represented in the following table:

[3] A deadlock occurs when there is a no-go situation between two or more processes. Their execution is halted, that is, they become blocked.

Table 2.2: Phases to implement a HRT.

Phase	Function	Description
1	`my_hrtimer_init`	Initialize the timer data.
2	`my_hrtimer_start`	Activate the timer and its expiring instant.
3	`my_hrtimer_callback`	Callback called whenever the timer expires.

The Linux scheduler implementation is based on a function, `schedule`, and various hierarchically organized scheduling classes, which results in a scheduling mechanism that is now more extensible, and well organized. This function is used for scheduling decisions, when tasks complete their execution or even when tasks become illegible for preemption, but there are other cases. Actually it relegates the scheduling decisions to the scheduling classes. So, it inquires the scheduling classes for ready tasks, by starting the highest scheduling class, and if any of the inquired scheduling classes did not reply with a ready task it goes to the lowest scheduling class(idle) that has always a ready task called idle task. The scheduling class chooses the ready task for execution according to their scheduling policies. Figure 2.4 shows this in more detail. As it can be understood, the kernel offers three levels of scheduling(classes): RT, CFS and Idle that are accessed by the scheduling mechanism for the purpose of obtaining ready tasks, this is done in order until there are no more ready tasks, if this is the case, then the scheduler chooses the Idle one. The illustration shown corresponds to the kernel version that was used in this work, which in this case is 3.10.15.

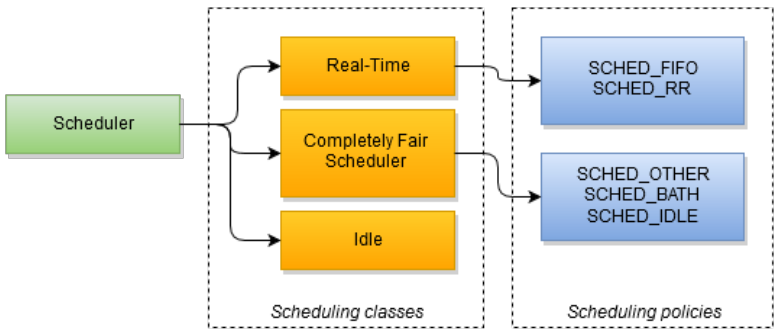


Figure 2.4: Linux scheduling architecture.

The scheduling classes are instantiations of the `sched_class` structure. Its first parameter is `next`, which is a pointer to the scheduling class that points to the next lower priority scheduling class. Scheduling classes are organized by priorities in a linked list, which starts with the class with higher priority. In this case the order is, RT, CFS and finally Idle. This is done, so that when a task is obtained to be executed, so is done visiting all the classes available were those visited first are the ones with higher priority.

Whenever these classes are used, it is mandatory to specify a set of functions that act as callbacks to specific events. Some of the most important, are listed in Table 2.3. It is then possible, by using the appropriate functions to make scheduling decisions:

Table 2.3: Some `sched_class` functions.

Function	Description
`enqueue_task`	Called whenever a task becomes ready. Inserts it in the appropriate processor's run-queue.
`check_preempt_curr`	Checks if the current executing task must be preempted. This varies according to the policy in place, resulting in a scheduler invocation.
`dequeue_task`	Called whenever a task is no longer ready. Removes the task from the respective processor's run-queue.
`pre_schedule`	Remove from another processor's run-queue the highest priority task that isn't executing. Used for work-balance purposes between processors.
`task_tick`	This function is called at a regular time interval so that something can be made. One example is doing task preemptions.
`pick_next_task`	Selects the next task to execute, searching all the scheduling classes.

Because this work uses primarily the RT scheduling class special attention is given to it. Through the source code its name is `rt_sched_class`, and like any other structure it contains various fields. In this case, its the one belonging to CFS's. Each scheduling class also contains its own run-queue, in this case, it is `rt_rq`. There are also three scheduling policies in place, `SCHED_FIFO`, and, `SCHED_RR`, the first schedules tasks according to the First-In-First-Out(FIFO) policy, and the second according to the Round-Robin(RR) policy. The first, which is more favoured by the scheduler, schedules tasks until they are preempted by a task with higher priority or blocked by a I/O event. The second one, executes tasks, until they spend their time slice, and tasks may be preempted or blocked like in the other case.

Finally, there are many other functions and macros used by the scheduler and unfortunately it would be impossible to explain all of them in detail in this document. The objective is just to give a debrief in how some of the components responsible interact. In the next subsection a succinct overview of the PREEMPT-RT patch is offered.

2.4.2 The PREEMPT-RT patch

Because the vanilla Linux kernel does not suffice to ensure the some needs that a RTS requires, a group of programmers, [Fu and Schwebel, 2014], is working and maintaining a patch to add some real-time features to the Linux's kernel.

To eliminate latencies, most of the locking primitives used through the source code are replaced by *mutexes*. A mutex is based in the concept of mutual exclusion which creates a one-to-one relationship between tasks that access critical areas. These are origins for unpredictability issues, because they need to be accessed by different tasks, and doing so with spin locks, as it is in the original kernel, does not suffice to create a single path of execution and in a RTS this is

important. For more information about the Linux locking primitives please refer to [Jones, 2007].

The patch implements the concept of *priority inheritance* together with the original locking primitives replacement by mutexes. This mechanism exists to avoid the origin of *priority inversion* cases in the system. Priority inversion refers to the pathological event when there are tasks with different priorities, and a higher priority task is preempted by another of lower priority, thus making their relative priorities invert. In more detail, because a task of lower priority accesses a shared recourse, and the other with higher priority tries to access it, this one becomes blocked until the lower priority task relinquishes it. At this time, a medium priority one can become ready, preempting the lower one before it releases the resource, the higher priority task then becomes blocked indeterminately. For more information about priority inversion please refer to [Barr, 2002]. Priority inheritance is achieved by changing the task's priority when it is accessing the recourse for a value equal to the maximum priority of the group of tasks that are also trying to access it. Once this timespan finishes, the task's priority returns to normal.

Another interesting feature, is the creation of a series of kernel tasks to handle I/O events. Because these have higher priority than any other task in the system, a task with lower priority that is having I/O events is interfering the execution of higher priority tasks, in the patch, this is solved by decoupling this responsibility to these newly created kernel tasks which have medium priority, thus eliminating this problem, since they have lower priority when compared with the original high priority tasks.

Finally, looking at the RT scheduling class, with the presence of this patch, there are no newly created scheduling policies, and the ones in existence, FIFO and Round-Robin, are not fit of multiprocessor computational systems, because task sets with high workload fail to meet temporal constraints and they require balancing techniques between processors which produce even more unpredictability problems. One of the objectives of the Real-time TAsk-Splitting Scheduling Algorithms Framework(ReTAS) framework, is not only to provide better tools of research but also to fill this particular gap, by offering different multiprocessor task-splitting scheduling algorithms.

2.5 Summary

In this chapter, it was described the most basic important principles regarding real-time systems. Because the multiprocessor environment being used in this work, hardware details about some of the principles regarding hardware architectures were also described. This was reinforced later on, by explaining some OS details, mainly Linux and its task scheduling solution. The PREEMPT-RT patch was explored, completing the knowledge requirements that should be needed to comprehend the concepts that lay ahead. The next chapter, which is focused in task scheduling theory, aims to continue this effort.

Chapter 3

Real-time Multiprocessor Scheduling

3.1 Introduction

One of the most important components of any operating system is the task scheduler, in the real-time context that premise continues to be true. The task scheduler is the component that is responsible to provide processor time to the tasks that need to execute. Typically, it schedules tasks according to some scheduling heuristic or, commonly designated, scheduling algorithm.

In this chapter, the objective is to provide some background regarding the basics of real-time scheduling. In the beginning it is given a basic explanation on how the basic theory can be interpreted, but in a later stage, more advanced concepts are discussed, like multiprocessor scheduling algorithms. By doing this, it is also going to be exposed some of the work that was done during the latest years in this area. Finally, multiple algorithms are explained, in order to ease the interpretation of any results taken at a later stage.

3.2 System model

Before entering in more advanced concepts, it is important to know the basic terminology regarding this particular theory field.

A real-time application is, typically, composed of a static set, τ, of n tasks ($\tau = \{\tau_1, \cdots, \tau_n\}$). Each task, τ_i, generates a sequence of z *jobs* ($\tau_{i,1}, \cdots, \tau_{i,z}$, where z is a non-negative number and potentially $z \to \infty$), and is characterized by a three-tuple (C_i, T_i, D_i).

The *Worst-Case Execution Time(WCET)*, C_i, is the maximum time required by the processor to execute a job of task τ_i without any interruption. T_i defines the frequency at which jobs of task τ_i are released in the system and, according to the nature of T_i, the systems are classified in three broad categories: (i) *periodic*, jobs are released regularly at some known rate (called *period*); (ii) *sporadic*, jobs are released irregularly at some known rate (called *minimal inter-arrival time*); and finally, (iii) *aperiodic* jobs appear with irregular arrival times, typically, at unknown rate. The temporal constraint of each task, τ_i, is defined by its *relative deadline*, D_i, the size of the time window for executing a job of such task τ_i.

A task set, τ, is said to have *implicit deadlines* if the relative deadline of every task is equal

to its period ($D_i = T_i$), *constrained deadlines* if the relative deadline of every task is less than or equal to its period ($D_i \leq T_i$), and *arbitrary deadlines* if there is no such constraint; that is, D_i can be less than, equal to, or greater than T_i.

For implicit deadline task sets, the task's, τ_i, *utilization*, u_i, is given by the ratio between its WCET and its period/minimal inter-arrival time:

$$u_i = \frac{C_i}{T_i} \tag{3.1}$$

Each job $\tau_{i,j}$ (this notation means the j^{th} job of task τ_i) becomes *ready* to be executed at its arrival time, $a_{i,j}$, and continues until its *finishing* (or completion) time, $f_{i,j}$. The duration of this time interval is said to be the *response time* ($r_{i,j} = f_{i,j} - a_{i,j}$) of job $\tau_{i,j}$. The response time (R_i) of task τ_i is defined as being the maximum response time of all its jobs ($R_i = \max_{j=1}^{\infty}(r_{i,j})$). The *absolute deadline*, $d_{i,j}$, of job $\tau_{i,j}$ is computed as: $d_{i,j} = a_{i,j} + D_i$. Therefore, a deadline miss occurs when $f_{i,j} > d_{i,j}$, which means that $R_i > D_i$. Aperiodic tasks are out of the scope of the work presented in this document; hence, the time difference between the arrivals of two consecutive jobs by the same task τ_i must be equal to T_i (for periodic tasks) or at least equal to T_i (for sporadic tasks).

Figure 3.1 illustrates the relation among the timing parameters of the job $\tau_{i,j}$. The execution of the job $\tau_{i,j}$ is represented by a gray rectangle and the sum of all execution chunks ($c^x_{i,j}$), which correspond to the execution time of that job, cannot exceed C_i.

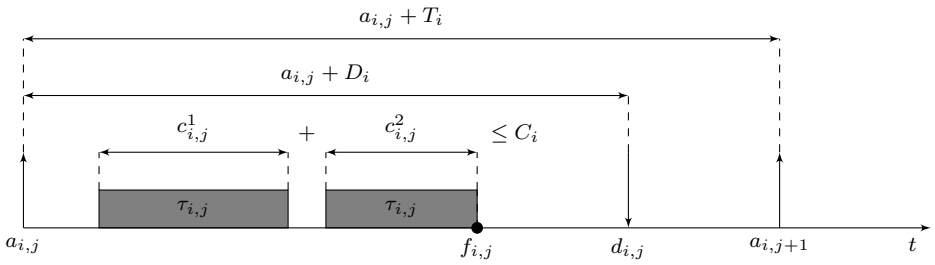


Figure 3.1: Illustration of the job timing parameters.

RT tasks are typically classified as: *hard* or *soft*. A task is said to be a hard real-time task if it is not allowed to miss any job deadline, otherwise undesirable or fatal results could be produced in the system. On the other hand, soft real-time tasks can miss some deadlines and the system is still able to work.

Real-time tasks can be categorized as *dependent*, if they interact with other tasks and its execution can be blocked by those tasks, or *independent*, if they do not.

One of the main RTSs characteristics is the time constraint. Hence, the time instant when the system starts operating is, typically, designated as *time zero* and a task set can be classified as *synchronous* or *asynchronous*. In the first case, all tasks arrive at the system simultaneously, in the second case, they do not, and then, it is needed to specify for each task, τ_i, an offset, O_i, that is the difference between the instant when a task arrives at the system and the time zero.

In the context of scheduling, preemption is one of the fundamental concepts. In a *preemp-*

tive system an executing task can be interrupted in order to execute a "more important" task. That is, the interrupted task relinquishes the processor and the processor is given to the "more important" task to be executed. In a *non-preemptive* system, a task that has been given to the processor for execution cannot be interrupted before its end.

In a multiprocessor environment the number of physical processors must be considered, m, ranging from P_1 to P_m, and upon this, it is possible to devise the expression that calculates the utilization of the system, U_s, which corresponds to the task's utilization sum normalized to the number of processors, m.

$$U_s = \frac{1}{m} \sum_{i=1}^{n} u_i \tag{3.2}$$

Migration is another concept to retain, which refers to the event of moving a task that was partially executed in one processor to be executed on another.

A *scheduling algorithm* is the method for defining the sequence for a set of ready jobs to be scheduled using the resources available, that is, to provide processor time to them. Real-time scheduling algorithms should schedule ready jobs according to their demands such that their deadlines are met.

Taking into account all concepts here mentioned, another characteristic to consider is to determine if system is or not *schedulable*. A system is considered schedulable if the task set time constraints are all met. In other words, if there no deadline miss occur. However, given the particularities of RTSs this must be guaranteed before run time, namely for hard-real time systems (systems composed by one or more hard real-time tasks). For that, it is used a *schedulability test*, which is dictated from a consecutive group of mathematical expressions that ultimately should aid the system's designer(s) in this matter. Finally, in the next section, some scheduling approaches specific for multiprocessor systems are provided.

3.3 Multiprocessor scheduling approaches

In order to study in more detail the performance of a particular scheduling algorithm, one needs to use the concept of *utilization bound*, which is nothing more than a threshold associated with the utilization of the taskset, so that all the tasks can achieve their requirements provided that this limit is not surpassed. It can be used to compare the outcomes of several different scheduling algorithms, and serves as an estimate of the expected performance.

To make this easier to explain, it is better to revisit some of the work that was done in the uniprocessor preemptive systems. Throughout the last decades, several perspectives were enunciated, some with strong adhesion, due to the ease in which anyone can implement. One of the algorithms proposed in [Liu and Layland, 1973] and named Rate-Monotonic(RM), assigns priorities to tasks according to their period (T_i), small period implies higher priority, thus becoming a *fixed-task* priority algorithm because the task's priority does not change. Similar to RM is Deadline-Monotonic(DM) [Liu and Layland, 1973] that assigns priorities to tasks according to their relative deadline (D_i), small relative deadline implies higher priority.

Another example, is Earliest Deadline First(EDF), devised in [Liu and Layland, 1973], which schedules by giving the highest priority to the job with earliest absolute deadline, thus becoming a *fixed-job* priority algorithm because the job's priority does not change during execution.

Finally, a *dynamic-job* priority algorithm example, where the priority of a job can change during its execution, is the Least-Laxity First(LLF) algorithm. LLF was proposed in [Dertouzos and Mok, 1989] and schedules jobs according to their executing urgency, denoted as *laxity*. The smaller the laxity value of a job is, the more urgent it is.

In the area of multi-processor preemptive systems, there a new, and difficult to deal with, layer of complexity. This layer, can be addressed considering the concept of *migration degree*, which defines the methodology employed to address the scheduling problem. For now lets consider these: *global* and *partitioned*. The way in how these two different perspectives work can be seen in the Figure 3.2.

A global scheduling algorithm keeps the information related to the tasks available for execution in a global queue that is shared by multiple processors. Within a certain time instant, multiple jobs are selected to be executed on the available processors. Jobs can migrate from one processor to another during its execution; that is, the execution of a job can be preempted on one processor and resume on another. Some of these scheduling algorithms are said to be *work-conserving*, because it is not possible to have processors stopping while there is a ready job waiting for execution. As a consequence, these algorithms provide good workload balance among processors and also higher utilization bounds (some of them, present a utilization bound of 100%), which may help prevent deadline misses because there is no constraint restricting on which processor a job can be executed. However, this is achieved at the cost of many migrations and preemptions, which could incur a great schedulability penalty when it is running in a real system. The act of Migrating and preempting tasks cost computational resources which can affect the final result. Meanwhile, the use of a global shared queue imposes the use of some locking mechanism to serialize the accesses to it.

Conceptually, partitioned scheduling algorithms transform a multi-processor system into a m uniprocessor system, but their utilization bound is 50%. The task set is divided into partitions and all tasks in a partition are assigned to the same processor. Contrarily to global scheduling al-

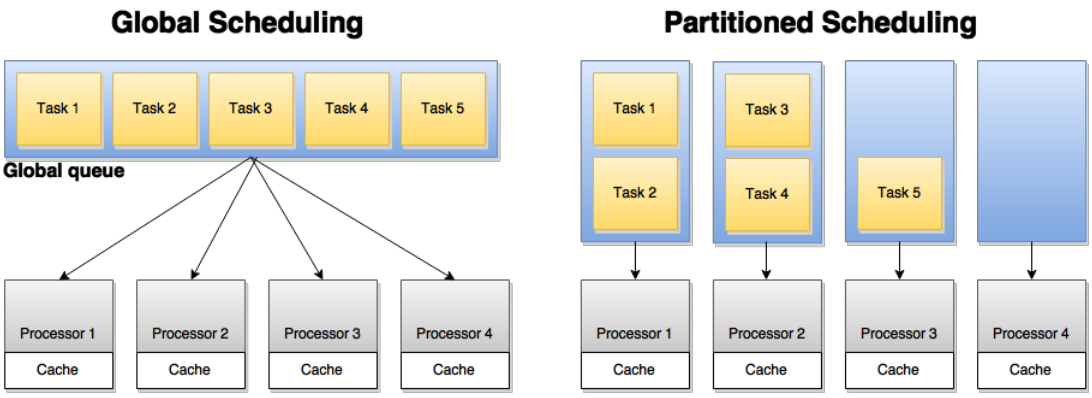


Figure 3.2: Global and partitioned multiprocessor scheduling

gorithms, that are work-conserving, partitioned scheduling algorithms are non work-conserving. Processors can be effectively stopped even if there is some ready job, since the tasks assigned to a partition are always assigned to same processor and tasks can not migrate from one processor to another.

The partitioning process involves assigning tasks to processors, and this is a bin-packing problem. In its theoretical basis, it reflects the need to allocate a group of items with different sizes, in the smallest number of fixed-size containers, without, of course, exceeding their capacities. In this case, the analogy is as follows: Each item is a task from the task set and its size is given by the calculation of the task's utilization. Finally, the containers correspond to the available processors. There are various methodologies to address this problem. The Next-Fit and the First-Fit heuristics are some examples. Unfortunately these represent an NP-hard problem as referenced in [Coffman et al., 1997], and this result in not being possible to obtain an optimal solution.

Partitioned scheduling algorithms, are in fact, composed by two algorithms, one that assigns tasks to processors and another that schedules tasks, the first one is ran *off-line* and the latter at *run-time*. Again, the first is a bin-packing algorithm, and the other is a uniprocessor one, and an example would be EDF.

However, in recent years a new multiprocessor scheduling scheme was developed, and received the name of *semi-partitioned* or *task-splitting*. Obviously, its main objective is to reduce the disadvantages inherent to the usage of global and partitioned algorithms. In this environment most of the tasks are *non-split* ones and are only executed in one processor, in resemblance to the partitioned scheme. The rest, though more rarely, can be executed in various processors, like in the global scheme, and are designated as *split* tasks. It is important to understand that it is the execution requirement that is split and not the code of these tasks. Please refer to Figure 3.3 for further details. The utilization bound of these algorithms is higher than that achieved by partitioned scheduling algorithms. Another advantage is that the number of migrations needed diminishes along with the probability of originating a bottleneck at the locking mechanisms required to manage the access to the run-queues.

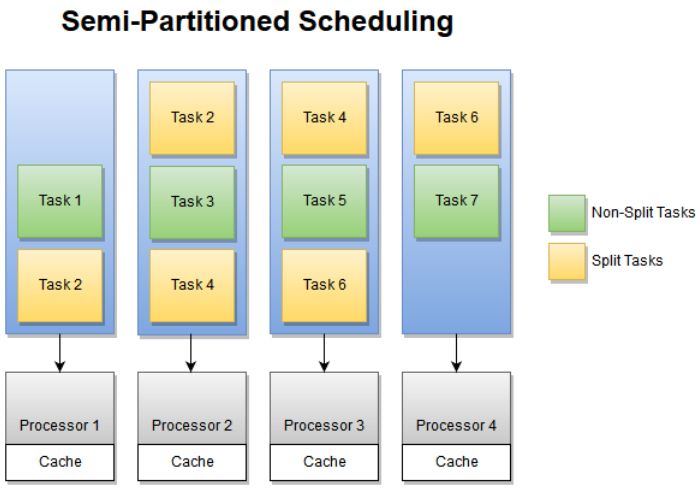


Figure 3.3: Semi-Partitioned multiprocessor scheduling

Generally, semi-partitioned approaches can be classified as *job-based* or *slot-based*. Job-based approaches [Kato and Yamasaki, 2007, Kato and Yamasaki, 2008, Kato and Yamasaki, 2009, Lakshmanan et al., 2009, Guan et al., 2010, Burns et al., 2012] splits a job into two or more sub-jobs and forms a sequence of sub-jobs. The arrival time of a sub-job is set equal to the absolute deadline of its preceding sub-job. Slot-based approaches [Andersson and Tovar, 2006, Andersson et al., 2008, Bletsas and Andersson, 2009a, Baltarejo Sousa et al., 2013] sub-divide time into *time slots* of equal duration. Typically, the beginning of the time slots are synchronized across all processors; Inside of each processor time slot, there are *time reserves* for executing jobs. The ready jobs of a non-split task utilize only one reserve on the processor that task is assigned to. For split tasks, the end of a time slot of processor P_p contains a time reserve and the beginning of a time slot of processor P_{p+1} which contains another time reserve, and these two reserves supply processing capacity for executing jobs of a split task. Slot-based approaches cause more preemptions than the job-based approach but, in return, it offers higher utilization bounds.

So far, some important concepts in the area of scheduling regarding multiprocessor systems were introduced. The following section will be focused on several slot-based scheduling algorithms.

3.4 Slot-based scheduling algorithms

Some examples of slot-based scheduling algorithms are: Sporadic-EKG(S-EKG), Notional Processor Scheduling - Fractional capacity(NPS-F), and Carousel-EDF, although the last one does not fit entirely in this category, due to its different way to approach the scheduling problem. Please refer to [Andersson and Bletsas, 2008, Bletsas and Andersson, 2009b, Sousa et al., 2013], respectively, for further details. These algorithms are an instantiation of a generic algorithm that will be described next.

For every real-time system, when employing any scheduling algorithm, task's computational requirements must be fulfilled by providing computational resources to them, this is done, by providing processor time to each one of them. To do this, there must be a scheduling policy in place, that theoretically determines the method in which to provide those resources.

For slot-based scheduling algorithms, there is a logic component called, *server*, that is used to serve computational resources to the system's tasks that need to be executed.Thus, tasks are firstly assigned to servers (the server's maximum processing capacity is equal to the processor's maximum processing capacity, that is, 100%), and then, these are mapped to processors, by allocating periodic *time reserves*. There are limits to allocations in these steps, for example, for a specific server, it is only possible to map it to, at the most, two processors, and for each processor three different servers can be allocated.

Tasks of a given server are executed in the allocated server's time reserve(s). For obvious reasons, these cannot be overlapped. Then, since all tasks have a period associated to and *time reserves* are theoretically interrelated with them, this also imposes the existence of a *time reserve* period: the *time slot*. In the generic algorithm, *time slot* is always equal for all processors.

This generic algorithm divides responsibilities in a orthodox fashion where during the sys-

tem's design (*off-line*), tasks are assigned to servers by firstly determining the server's characteristics. Next, the server's *time reserves* lengths are calculated, and, finally, they are mapped to processors. At run-time, it is used a uniprocessor algorithm, which usually is EDF, to schedule tasks in each server in resemblance to a uniprocessor system. Server execution is done in each time reserve to which the server was allocated.

Additionally, the off-line procedure is sub-divided in four steps. This is defined also by the generic algorithm, but the methods used to accomplish these, belong to the responsibility of a particular semi-partitioned scheduling algorithm, that is, they can be different between them.

As an example, some results obtained from NPS-F are used next, because this algorithm is graciously related to the generic algorithm. More details about this algorithm are provided in Section 3.4.2. Figure 3.4 is divided in three insets that correspond to several phases of the off-line procedure, the idea that is necessary to retain is that in the first phase, each bar's height corresponds to a specific task utilization, U_i. Each server, is represented by $\tilde{P}_q$. The task set τ example is composed by seven tasks, from τ_1 to τ_7.

The generic algorithm's first stage, is shown in the inset (b) of Figure 3.4. It corresponds to the task's assignment to servers. Again, each algorithm has its own method of doing this procedure, in this case, the First-Fit bin packing heuristic was employed. Next, in the second stage, shown in inset (c), the server's computational requirement is computed. Assuming that the utilization of server $\tilde{P}_q$ is given by equation 3.3.

$$U_{[\tilde{P}_q]} = \sum_{i \in \tau[\tilde{P}_q]} u_i \tag{3.3}$$

Where $\tau[\tilde{P}_q]$ is the set of tasks assigned to server $\tilde{P}_q$. This procedure is done by *inflating* the overall server's utilization, $U^{infl}[\tilde{P}_q]$. This is a fundamental step, server's inflation is required, because there is the need to compensate for the time that one server may have tasks that are ready to be executed, but none of them can, because the server's time reserves are not active, and a processor can only execute tasks of its corresponding time reserve. This happens, because only one server's time reserve can be activated for each processor. Each algorithm defines its own way to inflate server utilization.

Figure 3.5, shows the third stage of the off-line procedure, which is, mapping servers to processors. This mapping is performed taking into account the $U^{infl}[\tilde{P}_q]$. Some, namely, $\tilde{P}_1$ and $\tilde{P}_4$ are mapped to only one processor, and are then, designated as, *non-spit servers*, where in the

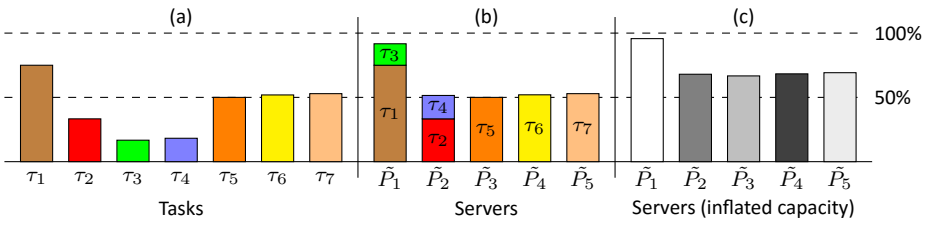


Figure 3.4: Task-to-server mapping.

case of the remaining ones, $\tilde{P}_2$, $\tilde{P}_3$ and $\tilde{P}_5$, they are designated *split servers* because they are split between two processors.

Finally, in the last stage, the time reserves length are computed for each processor P_p. First, time slot length, S, is computed and then each mapped server's time reserve length. Note that, the mapping is performed according to the $U^{infl}[\tilde{P}_q]$. However, it could be done based on server's time reserve length, $Res^{len}[\tilde{P}_q]$. For that, before mapping, each server reserve length could be computed according to Equation 3.4 and the mapping could be performed taking as processor capacity the time slot length, S.

$$Res^{len}[\tilde{P}_q] = U^{infl}[\tilde{P}_q] \cdot S \tag{3.4}$$

In each one, the time slot can be divided so that, at the most, three time reserves can coexist, $x[P_p]$, $y[P_p]$ and $N[P_p]$, as shown in the Figure 3.6. If split servers exist, the time reserves $x[P_p]$ and $y[P_p]$ are assigned to the beginning and the end of a time slot, respectively. In the first case, it has a reserve for the split server shared by two neighbour processors, P_p and P_{p-1}, while in the second case its P_p and P_{p+1}. For non-split servers, a time reserve is provided that is the rest that is available in a processor P_p, namely $N[P_p]$. Once again, the result shown was obtained with the application of NPS-F, other methods can be used, and in this case, the time slots are synchronized. At run-time, the uniprocessor scheduling algorithm is used to execute task's from the server allocated to the currently time reserve, to complete the whole process, allowing the task's computational requirements to be satisfied.

One of the interesting characteristics of these scheduling algorithms is the hierarchical approach that they provide. Actually, from the run-time perspective they produce a two-level hierarchical scheduling. In the first level, they statically schedule servers by the means of time reserve, and in a second level, each server schedules tasks according its scheduling uniprocessor policy.

In this section, the main features of the slot-based scheduling algorithms were abstractly explored. Next, a briefly description of S-EKG, NPS-F, and Carousel-EDF scheduling algorithms is offered highlighting their similarities and differences.

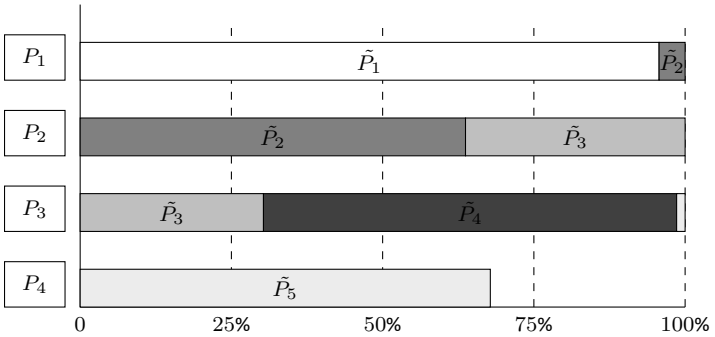


Figure 3.5: Server-to-processor mapping.

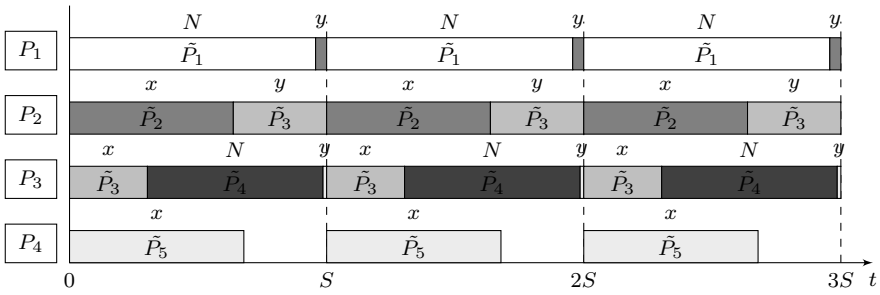


Figure 3.6: Example's run-time dispatching time line produced by NPS-F.

3.4.1 S-EKG

The S-EKG scheduling algorithm [Andersson and Bletsas, 2008], is similar to the generic one, though the original theory does not use the concept of server. The main difference between S-EKG and the generic algorithm is that S-EKG assures that each split server has only one task. Actually, this algorithm generates at most $m-1$ split tasks in a system composed by m processors. When a split task is not available for scheduling, a non-split task's is selected for scheduling. This is achieved employing a different approach to assign tasks to servers and mapping servers to processors in comparison to the one described in the generic algorithm.

In this algorithm the time is divided in time slots of length, S, computed according to Equation 3.5. The δ parameter is chosen at design time to manipulate the utilization bound at the cost of preemptions and migrations.

$$S = \frac{1}{\delta} \cdot min_{\tau_i \epsilon \in \tau} (T_i) \tag{3.5}$$

The utilization bound, U^{bound}_{S-EKG}, can be obtained through the Equation 3.6 and varies from 65% to closely 100% at the cost of more preemptions and migrations.

$$U^{bound}_{S-EKG} = 4 \cdot \left(\sqrt{\delta \cdot (\delta + 1)} - \delta \right) - 1 \tag{3.6}$$

The inflation factor, α, is also dependent of the designer-set parameter δ and is computed according to Equation 3.7

$$\alpha = \frac{1}{2} - \sqrt{\delta \cdot (\delta + 1)} + \delta \tag{3.7}$$

The server inflation, U^{infl}_{S-EKG}, is given by the Equation 3.8. Note that, this is required to assure the schedulability of the task set.

$$U^{infl}_{S-EKG}[\tilde{P}_q] = U[\tilde{P}_q] + 2 \cdot \alpha \tag{3.8}$$

The reserve length of a server, $Res^{len}[\tilde{P}_q]$, is computed according to Equation 3.9.

$$Res^{len}[\tilde{P}_q] = S \cdot U^{infl}_{S-EKG}[\tilde{P}_q] \tag{3.9}$$

3.4.2 NPS-F

The NPS-F scheduling algorithm [Bletsas and Andersson, 2009b] is also very similar to the generic algorithm. Comparatively to S-EKG, this one, has a greater utilization bound, but there is one disadvantage, that is: the number of split tasks are neither known in advance nor it is possible to keep its number to $m - 1$ in m processors. Actually, for most of the cases its bigger.

The time slot length, S, is computed according to Equation 3.5, which is the same equation used for S-EKG and δ parameter has the same meaning. It is used to trade preemptions and migrations with utilization bound. The utilization bound can be configured so that values between 75% to arbitrarily close to 100% can be obtained according to Equation 3.10.

$$U_{NPS-F}^{bound} = \frac{2 \cdot \delta + 1}{2 \cdot \delta + 2} \tag{3.10}$$

To inflate the server capacity it is used the Equation 3.11 and the reserve length of a server is computed according to Equation 3.9 but using $U_{NPS-F}^{infl}\left[\tilde{P}_q\right]$ instead of using $U_{S-EKG}^{infl}\left[\tilde{P}_q\right]$.

$$U_{NPS-F}^{infl}\left[\tilde{P}_q\right] = \frac{(\delta + 1) \cdot U\left[\tilde{P}_q\right]}{U\left[\tilde{P}_q\right] + \delta} \tag{3.11}$$

3.4.3 Carousel-EDF

In order to attempt to improve the already good results obtained by the NPS-F scheduling algorithm, in 2013, [Baltarejo Sousa et al., 2013], devised Carousel-EDF. Carousel-EDF is an offshoot of the generic algorithm. However, contrarily to it, which migrates tasks at time slot and reserve boundaries (see Figure 3.6), Carousel-EDF only migrates tasks at reserve boundaries (see Figure 3.7). Consequently, the time is no longer constrained to the time slot boundaries, but it is only divided into time reserves instead.

As it can be noted in Figure 3.7, server's time reserves are not fixedly mapped to processors. That is, every in time slot they move from processor P_p to P_{p-1}. This feature, the cycling of servers across processors, inspired the name *Carousel*. This approach preserves the utilization bounds of NPS-F, which is the highest among semi-partitioned scheduling algorithms. Furthermore, with respect to NPS-F, Carousel-EDF reduces up to 50% the worst-case number of context switches and the worst-case number of preemptions caused by the time division mechanism.

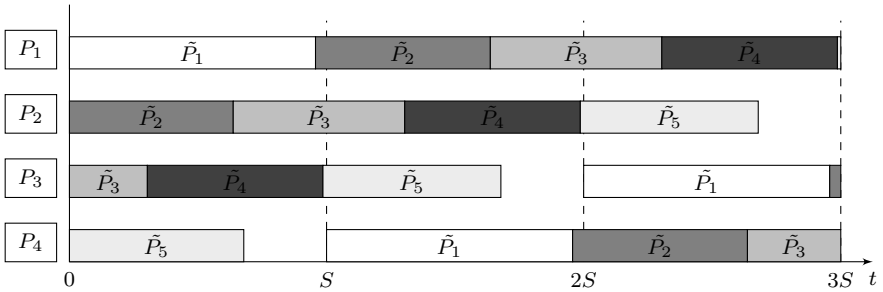


Figure 3.7: Example's run-time dispatching time line produced by Carousel-EDF.

3.5 Summary

In this chapter the most fundamental principles regarding real-time systems and multiprocessor scheduling were explored. First, the most basic concepts, then, the focus was on multiprocessor scheduling algorithms in general, and finally, in slot-based scheduling algorithms, which are those under study in this work. With this, it is hoped that the required knowledge to grasp the following chapters is correctly conveyed.

Chapter 4

Using Fixed-task Priority Policies with Slot-based Scheduling Algorithms

4.1 Introduction

Multiprocessor real-time systems are, perhaps, the next step to be taken in the real-time computing area. Uniprocessor real-time theory is well matured, but, unfortunately, the methodologies employed to achieve the desired results can not be directly applied to multiprocessor systems. Furthermore, uniprocessor fixed-task priority scheduling algorithms. This is why a whole level of research must be promoted to attain the promised advantages inherent to multiprocessor systems.

From what is discussed in [Baltarejo Sousa et al., 2013] and [Baltarejo Sousa et al., 2014], this area of multiprocessor algorithms is fuzzy and it demands the improvement of its research techniques. This of paramount importance, because there are no perfect multiprocessor algorithms for real-time systems. There is always more space for different approaches.This work shows how the basis of the unified scheduling theory presented in [Baltarejo Sousa et al., 2014] can be employed to this case, when uniprocessor fixed-task priority scheduling algorithms are used, by combining it with the classical response-time analysis schedulability test.

Perhaps the best way to justify this, is to look at some of the real-time computing history throughout the years. Nowadays, according to [Buttazzo, 2005], there are more systems in place that use the RM scheduling algorithm as their main scheduler, since most of the commercial RTOSs in existence implement it. Because algorithms like EDF need to keep track of several task's characteristics, being priorities and deadlines the ones with more importance, it makes it less appealing for application in a real environment, even though EDF shows more promising results by making better use of the resources available.

Despite the misconceptions presented in many works, such as promising lesser overheads because of this implementation easiness, RM has received more attention than EDF, and it is extensively studied existing several highly technologically advanced projects that utilize it, being organizations like the European Space Agency just one example. The reader is referred to [Sha et al., 1994] for further information.

Here, the important idea to retain is that fixed-task priority algorithms are really important, either by looking at some of the research work that has done in these years [Andersson and Jonsson, 2000, Guan et al., 2009, Kato and Yamasaki, 2009, Guan and Yi, 2012, Davis et al., 2015], or either by simply understanding that it is extensively used in the real world. As a contribution, we present a new unified schedulability analysis for slot-based algorithms that use fixed-task priority scheduling algorithms, RM and RM, as their run-time scheduling algorithm, in contrast to the original, that uses the EDF scheduling algorithm. It is for this reason, that new variations of the main semi-partitioned scheduling algorithms now exist and must have new nomenclature. For the rest of this document, the suffix *-RM* or *-DM* is added to the algorithm's designation according to the on-line uniprocessor scheduling algorithm, with the exception of Carousel-EDF, which already has one by default, for this case the EDF suffix is replaced by the others previously described. The variations, are therefore these: NPS-F-RM, NPS-F-DM, S-EKG-RM, S-EKG-DM,Carousel-RM and Carousel-DM. The theory, here presented, is applicable to all these variations.

This chapter is structured in the following fashion: Section 4.2, provides an introduction to the classical real-time analysis and the system model used in this work. Section 4.3 contains the schedulability analysis devised. Section 4.4, shows an example that applies the solutions created in the previous section. Finally, Section 4.5, shows some final thoughts, emphasizing the future work that should be done to continue this research effort.

4.2 Response Time Analysis

In the uniprocessor area, fixed-task priority scheduling algorithms have several tests [Audsley et al., 1993, Bini and Buttazzo, 2004, Lehoczky et al., 1989, Liu and Layland, 1973] available to check if a particular task set is feasible, but it was chosen the one that was proposed in [Audsley et al., 1993] designated Response-time Analysis(RTA), because of its simplicity, and its inherent plausibility to find realistic results. It is important to know, that generically these tests do not consider the existence of operating system overheads, sources of jitter, blocking mechanisms, etc... which, in a real world approach and implementation, can not be neglected. Nevertheless, it was made a comparative test in [Min-Allah et al., 2012], that proved that in some cases, RTA produces better results than the rest of the tests proposed until that time. It is shown in [Audsley et al., 1995], that by using it, good results can be obtained, in contrast to the other tests, that, to the best of our knowledge, there is not research work available, much less, in the multiprocessor environment that advocates their usage.

Considering the RTA test, a task set, τ, is said to be schedulable if the Equation 4.1 holds:

$$\forall i \in \tau : R_i \leq D_i \tag{4.1}$$

Thus, it implies the need to compute each task worst-case response time, R_i, which is obtained according to Equation 4.2. The response time (R_i) of task τ_i is defined as being the maxi-

mum response time of all of its jobs ($R_i = \max_{j=1}^{\infty}(r_{i,j})$).

$$R_i^k = C_i + \sum_{j \in hp(i,\tau)} \left\lceil \frac{R_i^{(k-1)}}{T_j} \right\rceil \cdot C_j \tag{4.2}$$

Where k is used for counting the iterations ($k \geq 0$) and $hp(i,\tau)$ is the τ's sub set of tasks with a priority higher than the priority of the task, τ_i, under analysis. Equation 4.2 is an iterative recurrent equation, because R_i^k term is computed as a function of the preceding term, $R_i^{(k-1)}$. Therefore, there is the need to define the initial value for R_i^0 as well as the conditions to break the iteration. The initial value is calculated according to Equation 4.3.

$$R_i^0 = C_i \tag{4.3}$$

The breaking iteration conditions are (in this sequence):

$$(1) \quad R_i^k > D_i \tag{4.4}$$

$$(2) \quad R_i^k = R_i^{k-1} \tag{4.5}$$

In the former case ($R_i^k > D_i$), the task is considered unschedulable, and schedulable in the latter case ($R_i^k = R_i^{k-1}$). Note that, a task set, τ, is considered schedulable if all tasks are schedulable.

4.3 RTA-based Schedulability Test

All slot-based scheduling algorithms (S-EKG, NPS-F and Carousel-EDF) considered use as the run-time task dispatching algorithm, EDF, which is a dynamic-task priority scheme. As mentioned in Section 4.2 there are several good reasons to consider fixed-task priority schemes for task's dispatching algorithm. With this purpose, in the next subsections we present the required tools to employ fixed-task priority schemes to the slot-based scheduling algorithms under consideration, namely a new task to server assignment and the server to processor mapping procedures both based on RTA.

4.3.1 RTA-based Task-to-Server Assignment Procedure

The first step of the off-line procedure is to assign tasks to servers, and then, once the server's characteristics are found, in the second step, servers are mapped to processors. Recall that, system schedulability is mandatory in real-time systems. With this in mind, we devise a function, `rta_server_check` (see Algorithm 1) that checks the schedulability of a server $\tilde{P}_q$. That is, to check if the set of tasks, $\tau[\tilde{P}_q]$, assigned to a server, $\tilde{P}_q$, is schedulable.

For the sake of simplicity, we define `rta_task_check` function (see Algorithm 2) that checks the schedulability of a given task, τ_i, in a set of tasks $\tau[\tilde{P}_q]$ assigned to server $\tilde{P}_q$. Note that, $hp(i,\tau[\tilde{P}_q])$ is the set of tasks assigned to server $\tilde{P}_q$ that has higher priority than τ_i.

Recall that the first step is to assign tasks to servers. This assignment is performed according

Algorithm 1 Pseudo-code of the server schedulability check function, `rta_server_check`

Input: $\tilde{P}_q$ {server to analyse}
Returns: **true** if $\tilde{P}_q$ is schedulable, **false** otherwise

for each $\tau_i \in \tau[\tilde{P}_q]$ **do**
  if not $rta\_task\_check(\tau_i, \tau[\tilde{P}_q])$ **then**
    return $false$
  end if
end for
return $true$

Algorithm 2 Pseudo-code of the task schedulability check function, `rta_task_check`

Inputs: τ_i {task to analyse}
  $\tau[\tilde{P}_q]$ {set of tasks}
Returns: **true** if τ_i is schedulable, **false** otherwise

$R_i^0 \leftarrow C_i$
while $R_i^0 \leq D_i$ **do**
  $R_i^1 \leftarrow= C_i + \sum_{j \in hp(i,\tau[\tilde{P}_q])} \left\lceil \frac{R_i^{(k-1)}}{T_j} \right\rceil \cdot C_j$
  if $R_i^1 = R_i^0$ **then**
    return $true$
  end if
  $R_i^0 \leftarrow R_i^1$
end while
return $false$

to the First-Fit bin packing heuristic. Algorithm 2 shows that, when a task has a worst response time bigger than its deadline ($R_i > D_i$), it halts the mechanism in place, resulting in a unschedulable task set. The remaining procedure is done according to the original RTA theory.

Algorithm 3 Pseudo-code modification of the task-to-server mapping algorithm.

Input: set of n tasks τ_i, with $1 \leq i \leq n$
Output: set of k servers, with $k \geq 0$ ($k = 0$ means failure)

$k \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  $scheduled \leftarrow 0$
  for $q \leftarrow 1$ to k **do**
    add_task_to_server($\tau_i, \tilde{P}_q$)
    if rta_server_check($\tilde{P}_q$) **then**
      $scheduled \leftarrow 1$
      break
    else
      remove_task_from_server($\tau_i, \tilde{P}_q$)
    end if
  end for
  if $scheduled = 0$ **then**
    $k \leftarrow k + 1$ {add a new server}
    add_task_to_server($\tau_i, \tilde{P}_k$)
    if not rta_server_check($\tilde{P}_k$) **then**
      $k \leftarrow 0$
      break {failure}
    end if
  end if
end for

Upon this, we devised a task-to-server assignment mechanism by using the First-Fit bin-packing heuristic. Algorithm 3 shows its pseudo-code. The function receives the task set information and tasks are gradually added to servers, and for each server, upon addition, it checks if it is schedulable by using the `rta_server_check` function. During this process, a server is constructed, and when it becomes unschedulable, or in other words, it is not possible to add any task to it, we save it, and create a newer one, so that the process can be repeated, until no more tasks remain to be assigned. Additionally, if we have a task that cannot be added to a server with only one task, we also consider that the task set is unschedulable (actually, this must not be possible). Therefore, after servers being created the next step is to map them to processors.

4.3.2 RTA-based Server-to-Processor Mapping Procedure

In this section, it is described the second step of the off-line procedure, where all servers are mapped to processors by the means of time reserves. For that purpose, the computational requirement of each server is obtained, by calculating each server's time reserve length, ($Res^{len}[\tilde{P}_q]$). Originally, for instance, the authors of the S-EKG and NPS-F scheduling algorithms found a way to analytically calculate it. In contrast, we followed the approach used in [Baltarejo Sousa et al., 2014], which considers the existence of a *fake task*. This one, is constructed according to the remaining execution time available until this particular server, with a group of tasks already assigned, remains schedulable. In the end the server's time reserve length is computed as:

$$Res^{len}[\tilde{P}_q] = S - C_{fake} \tag{4.6}$$

As mentioned before, $S = \min(T_1, \cdots, T_n)$ specifies the period of each server time reserve as well as its maximum length. Taking this into account, the parameters of the fake task are:

$$\begin{aligned} C_{fake} &= S - Res^{len}[\tilde{P}_q] \\ T_{fake} &= S - \epsilon \\ D_{fake} &= C_{fake} \end{aligned} \tag{4.7}$$

Where $\epsilon \to 0^+$. Note that, in the context of fixed-task priority scheduling schemes, this assures that the fake task has the highest priority among tasks assigned to a server. However, there is a small problem to deal with. We want to compute the server $\tilde{P}_q$ time reserve length ($Res^{len}[\tilde{P}_q]$) considering C_{fake} and according to Equation 4.6, it is used to calculate $Res^{len}[\tilde{P}_q]$.

The solution found to solve this problem was the *bisection method*. As shown in Algorithm 4, the bisection method is used to determine C_{fake}. It is considered that at the most it has the length of the time slot (S), thus the interval will be [0,S]. Due to this method's nature, its convergence will be of great value during the algorithm's execution. Depending on the scheduling result provided by the scheduling test already developed in the **rta_server_check** function, the midpoint is computed, which serves has the lower bound or higher bound. The last step of this algorithm, is to determine the reserve length, which is given by the difference between the size of the time slot, S, and the execution requirement of this fake task, C_{fake}.

Algorithm 4 Pseudo-code of the server reserve length computation algorithm.

Inputs: $\tilde{P}_q$ {server to analyse}
Δ {desired precision}
S {size of the timeslot}
Output: $Res^{len}[\tilde{P}_q]$ {server's reserve length}

$C_{fake} \leftarrow 0$
$T_{fake} \leftarrow S - \epsilon$
$C_{fake}^{min} \leftarrow 0$
$C_{fake}^{max} \leftarrow S$
while $C_{fake}^{max} - C_{fake}^{min} > \Delta$ **do**
  $C_{fake} \leftarrow (C_{fake}^{max} + C_{fake}^{min})/2$
  $D_{fake} \leftarrow C_{fake}$
  add_task_to_server($\tau_{fake}, \tilde{P}_q$)
  if rta_server_check($\tilde{P}_q$) **then**
    $C_{fake}^{min} = C_{fake}$
  else
    $C_{fake}^{max} = C_{fake}$
  end if
end while
$Res^{len}[\tilde{P}_q] \leftarrow S - C_{fake}^{min}$

Algorithm 5 Pseudo-code of the server-to-processor mapping algorithm.

Input: set of k servers, with $k \geq 1$ {set of servers}
S {size of the timeslot}
Output: set of p processors with servers assigned

$p \leftarrow 1$
for $q \leftarrow 1$ to k **do**
  if $Res^{len}[\tilde{P}_q] = S$ **then**
    $Type[\tilde{P}_q] \leftarrow$ SINGLE
    add_server_processor($P_p, \tilde{P}_q$)
    $p \leftarrow p + 1$
  end if
end for
for $q \leftarrow 1$ to k **do**
  if $Type[\tilde{P}_q] \neq$ SINGLE **then**
    $S_{rem}^{len} \leftarrow S_{rem}^{len}[P_p]$
    if $S_{rem}^{len} \leq Res^{len}[\tilde{P}_q]$ **then**
      $Type[\tilde{P}_q] \leftarrow$ NON_SPLIT
      add_server_processor_N($P_p, \tilde{P}_q, Res^{len}[\tilde{P}_q]$)
      if $S_{rem}^{len}[P_p] = 0$ **then**
        $p \leftarrow p + 1$
      end if
    else
      $Type[\tilde{P}_q] \leftarrow$ SPLIT
      add_server_processor_Y($P_p, \tilde{P}_q, S_{rem}^{len}$)
      $p \leftarrow p + 1$
      add_server_processor_X($P_p, \tilde{P}_q, Res^{len}[\tilde{P}_q] - S_{rem}^{len}$)
    end if
  end if
end for

The final step is to map servers to processors using the First-Fit bin-packing heuristic. For this part, we consider that the processing units available act as the bins to be filled, and the objective is to use the minimum possible, by packing the servers into them. This is an off-line bin packing procedure, since all items are known in advance, because the servers characteristics were already calculated. Algorithm 5 shows the pseudo-code of this procedure. The first goal of

the first server set loop iteration, is to classify the SINGLE servers. Single servers are mapped to only one processor and that processor has only one server, which amounts to pure partitioning for the respective server tasks. In the second server set loop iteration the remaining servers are mapped to the remaining processors, by allocating time reserves to servers according to the respective $Res^{len}[\tilde{P}_q]$. Servers are classified as NON − SPLIT or SPLIT according to the number of time reserves allocated, one or two respectively.

4.4 Applying the new RTA-based Schedulabity Test

In this section, we apply the RTA-based schedulability test presented in Section 4.3. Note that this RTA-based schedulability test is greatly simplified, because no sources of overheads are considered (In a real environment these cannot be neglected, so, it is our plan to deal with in the future).

Table 4.1: Task set example

Task	C_i	$T_i = D_i$	U_i
τ_1	4	8	50%
τ_2	3	10	30%
τ_3	10	15	66%
τ_4	9	17	53%
τ_5	2	19	11%
τ_6	38	49	78%
τ_7	30	42	72%

We create a simple group of tasks to simplify this demonstration, shown in Table 4.1. No particular criterion was used for define the task's parameters. Each task is defined by its worst execution time C_i, its period T_i, and its utilization U_i. Inset (a) of Figure 4.1 shows graphically the utilization of each task.

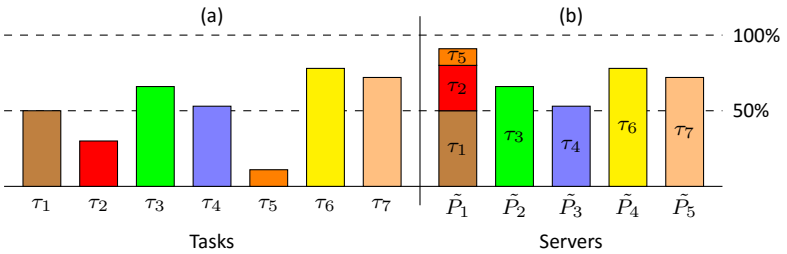


Figure 4.1: Example of the task-to-server procedure.

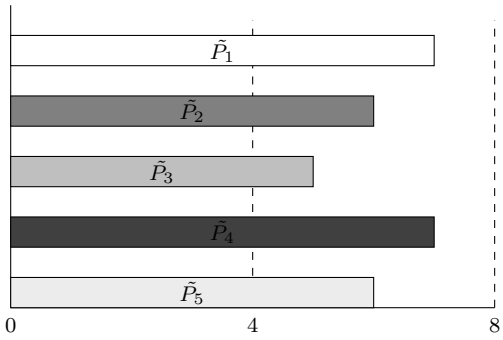


Figure 4.2: Servers' time reserve length ($Res^{len}[\tilde{P}_q]$) and S equal to 8.

As mentioned before, the first step is to assign tasks to servers. Inset (b) Figure 4.1 shows the outcome of the task-to-server assignment procedure for this task set. The second step is to map servers to processors. For that purpose, first, we have to calculate the time slot length, S. For this particular task set S is equal to 8 ($S = \min(T_1, \cdots, T_7) = 8$). Second, for each server we have to calculate the $Res^{len}[\tilde{P}_q]$ according to the Algorithm 4. The outcome is illustrated in the Figure 4.2. This means that at every S time units the system provides to each server $Res^{len}[\tilde{P}_q]$ time units that are enough to guarantee the schedulability of the entire task set.

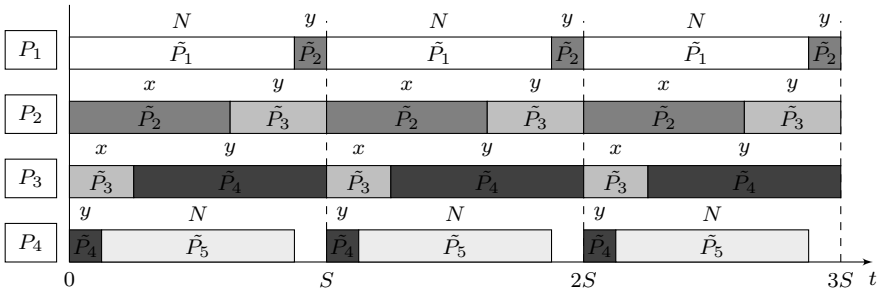


Figure 4.3: Example run-time dispatching time line.

Third, we map servers to processors. As shown in Figure 4.3, we were able to fit seven tasks inside five servers, and these are mapped to four processors. In this example, we have no single servers, two non-split servers ($\tilde{P}_1$ and $\tilde{P}_5$) and three split servers ($\tilde{P}_2$, $\tilde{P}_3$ and $\tilde{P}_4$).

4.5 Conclusions and Future Work

Without a doubt, slot-based algorithms show that they are, probably the way to go in the multiprocessor scheduling area for real-time systems. They use a more realistic approach to solve the problem imposed by the usage of multiple processing units. They do this, they combine some of the benefits from the older scheduling schemes, that at the end, result in a more efficient method to use the processing power available.

So far, the only unified scheduling analysis in existence is dedicated to the usage of EDF as the run-time scheduler. We decided to improve this, because, fixed-task priority scheduling algorithms, like RM and DM, are widely used, and thus they are an obvious attempt to produce even

more research work in this area. This work, uses partially the work previously devised, and is the first, to the best of our knowledge, to produce unified schedulabiblity analysis for slot-based scheduling algorithms that use fixed-task priority scheduling algorithms as their uniprocessor run-time scheduler.

The example provided, shows that all the steps devised can initially be employed and implemented in a real operating system, and successfully obtain positive results. Nevertheless, because the sources of overheads, that are already known to us, are not, at this time, being considered, this cannot be taken as a final answer. As future work, we will consider the existence of those overheads and create schedulability analysis based on RTA that takes them into account. Our goal is to bridge the gap between theory and practice.

4.6 Summary

In this chapter the new schedulability analysis devised was presented together with all the algorithms devised. This theory was created combining the classical RTA, with the already present unified scheduling analysis. At the end, an example was shown to prove its usefulness. In the next chapter the algorithm's core implementation in the Linux kernel is described.

Chapter 5

Framework implementation

5.1 Introduction

To bridge the gap between theory and practice, as it was been discussed in Chapter 1, the attempt to implement semi-partitioned algorithms in a real OS is a must. In Chapter 2 and 3 it was laid down the foundations to begin this attempt, first by providing some hardware details and Linux kernel development knowledge, and then, by describing, perhaps, the most important theory regarding semi-partitioned scheduling used in this work.

The ReTAS framework, is composed mainly by a kernel patch, that once applied correctly, provides the implementation of several multiprocessor scheduling algorithms (S-EKG,NPS-F and Carousel-EDF). This framework distinguishes itself, by being unified and modular, because the implementation is the same for all the scheduling algorithms available, and it aims to promote the addition of new functionalities to it. Furthermore, it is accompanied by a tracing and statistics gathering mechanism to improve the research process involved.

In this chapter, it is described the methodologies used to implement all the steps described by the generic semi-partitioned algorithm studied in Chapter 3. It is shown also, how all the algorithms coexist in the same implementation, explaining the decisions made whenever possible. Not every aspect of the implementation will be explained, only the main data structures and the most important functions developed. Finally, it is provided the way in which data is collected for testing purposes.

5.2 Approach

This section focuses primarily in discussing the challenges found while implementing the scheduling algorithms that are based in the slot-based generic algorithm. It is important to know, that the information determined off-line, that is, the task's characteristics, the servers, the reserve lengths, and the server-to-processor bin-packing procedure, is already in existence, either manually or automatically.

The implementation, obviously adds new components that hold the required information, from the off-line procedure, and several other mechanisms that are indispensable, such as, the

task's release mechanism or the scheduling algorithms in existence. To begin with, all of the framework's data is added to the main, per-processor run-queue, which is the scheduler's data structure that contains most of the system's scheduling information that exists in a given instant. This is advantageous, because it conserves the work already in existence, and maintains coherence with the kernel's scheduling module. Similarly, the framework's tasks information is also stored in the same data structure like any other system task.

The final product that results from the off-line procedure, is the computed reserves and their allocation to processors. The reserve information is stored and accessed in a fashion that allows each semi-partitioned algorithm to use the same implementation. It depends only in the way in which this is conveyed from user-space to kernel-space.

Each processor has its own reserve run-queue, because it must be able to select its next reserve. Each reserve is then stored in a global data structure, and each one has its server and its length. The reserve run-queue, allows the framework to gradually select the next reserves to schedule, so that the server that was allocated to it allows its tasks to have processor time. The reserve run-queue, maintains information regarding the current reserve executing in that processor, its number of reserves, and the current reserve index in the global data structure.

In the case of S-EKG and NPS-F, a global reserve structure is not required, each processor has its own reserves, and these do not need to be accessible by all processors. To permit Carousel-EDF to exist, a global queue was added, and this resulted in a reserve accessing strategy that is different for this algorithm. This is shown in some detail in Figure 5.2.

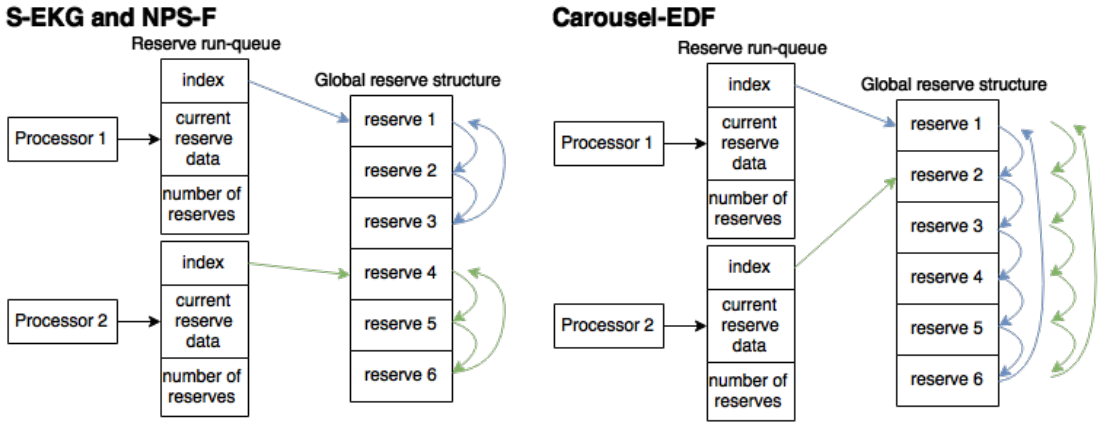


Figure 5.1: Reserve accessing method for every algorithm.

Again, this global data structure, in every case, holds each reserve for each processor in order, but in the case of S-EKG and NPS-F, each processor's reserve run-queue only accesses its exclusive reserves, using the total that were allocated, and the information about the first one that was stored for that processor. In contrast, Carousel-EDF, uses as the total of reserves, the ones that were stored for every processor, and each processor's run-queue accesses every reserve sequentially, and each processor starts executing an adjacent reserve.

The reserve mechanism is not complete until it is possible to act upon switching from one reserve to another. To deal with this specific problem, a timer for each processor is used. A

timer of this type always depends on its start and on its expiring instants. The first is computed according to the timezero, that is, the time in which the scheduling starts, plus an offset.

The second is updated when the timer expires, by adding this instant to the next reserve length. Therefore, when the timer expires, it selects the next reserve and computes the next expiring instant according to its length. Then, it updates the current server with the server associated to such reserve. Additionally, the current executing task is marked to be preempted in order to force a new scheduling decision.

To allow tasks to be scheduled and eventually released, two types of queues are used, and they are ready and release queues. Ready queues are associated only to servers, that is, there is a ready queue for each server in existence, and release queues, exist for each server, or each processor, depending on the way in which tasks are released. This is a choice that is possible to make in the framework, conceived to eliminate cases where more computational resources are spent. It is expected that when releases are made at the server level, less resources are used because releases are only done when servers are executing. Still, it is possible to experiment with both, more details about this will be given later. Please refer to 5.2 for details about the associations and the components involved.

Both ready and release queues are implemented using a red-black tree for each one in existence. The red-back tree data structure is a good choice, because it is a self-balancing binary search tree were its nodes are sorted by a key. The lowest key is the one belonging to the leftmost node. Search operations are done in $O(log_n)$ time, where n is the number of existing elements. This makes insertion and deletion of nodes faster when compared to other data structures. Additionally, it is supported and used natively.

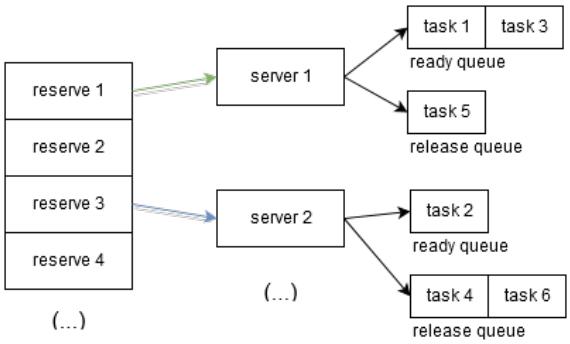


Figure 5.2: Tasks coexisting in their ready and release queues.

When a task enters in the system, it is enqueued in its corresponding server ready queue. This is possible because each server has their own tasks associated, and each server has its ready queue. Upon enqueue, the task's absolute deadline is computed according to the sum between its arrival and the task's relative deadline. While tasks are added to ready queues, they are sorted according to a uniprocessor scheduling policy, either EDF, according to the task's absolute deadline, RM, relative period, or DM, relative deadline, so that the top node in the tree corresponds to the highest priority task in that server.

At task execution compilation, it is dequeued from the ready queue and enqueued to the

release queue. In the release queue tasks are sorted by absolute arrival time. At that time each task is released, that is, dequeued from the release queue and enqueued in the ready queue. Again, timers are used when moving tasks between both types of queues, or, in other words, to allow tasks waiting for execution to be released.

Again, upon the choice made from user-space, the release method is done either only when servers are executing or not. In the second case, the task is automatically added to its release queue, and if it is the most urgent one, the timer's expiring instant is updated according to the next release. Otherwise, when using release queues at the server level, the release enqueue is done checking the next release instant from the current server. Furthermore, this instant must be within the reserve span, so that tasks cannot be released out of their reserve bounds.

Tasks are woken, that is, a new job is ran, once release timers expire, or during the time when switching from one reserve to the next. This results in tasks being removed from their release queue and being again added to it using the updated arrival instant, then ready queues are sorted accordingly upon reactivating tasks. During this, the next expiring instant is computed and updated, and the task is marked for rescheduling.

There is a main run-queue for each processor available it manages all the previously described timers, the reserve run-queue for a given processor, and optionally, release queues, which, again, may be used depending on the way in which tasks are released. Additionally, it has another timer, used to mark the currently executing task for rescheduling. These timers, which expire in a fixed interval, are used so that when the task with more priority may be obtained correctly, because in some cases it might not yet have been relinquished by the last processor that executed it. This happens, because, when the task is obtained, its last server may still be locked, and the timer is used to to wait until the server is unlocked.

When a new task is woken, it may be the one with higher priority, in this case, the currently executing task is preempted. This is done, by first checking if the currently executing server is the same belonging to the newly woken task, and if it is not, then the task itself is checked, if one of these steps is not true, then a preemption is needed.

Finally, when tasks of a given server finish their execution, a post schedule procedure is done, which ends up going through every reserve, and if a different server is executing in a processor that is different than the one that was executing, it unlocks it.

5.3 Implementation

In this section, a brief description of every component will be given, focusing on the data structures that were created, each major mechanism, and the system calls created. The system calls, compose an interface in which it is possible to communicate with the framework and produce an experiment, provided that everything is done in its correct order.

To achieve the desired functionality, and considering that it was been used the vanilla Linux kernel 3.10.15 in combination with the PREEMPT-RT patch, it is known that it has three main scheduling classes: RT(Real-Time), CFS(Completely Fair scheduler), and Idle. There is an hierarchy between them. The highest being RT, so the scheduler tries to obtain runnable tasks in a specific

order, thus RT tasks are more urgent than CFS tasks and so on. For this implementation, it was been added a newer scheduling class within the RT scheduling class, denominated SCHED_RETAS which contains a unified implementation of all the algorithms available, please refer to Listing 5.1.

```
#define SCHED_NORMAL    0
#define SCHED_FIFO    1
#define SCHED_RR    2
#define SCHED_BATCH    3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE    5
/* Can be ORed in to make sure the process is reverted back to SCHED_NORMAL on
     fork */
#define SCHED_RESET_ON_FORK     0x40000000

#ifdef CONFIG_RETAS
#define SCHED_RETAS    7
#endif
```

Listing 5.1: Newly added scheduling policy.

To add more information to a task that is to be scheduled according to the newly added scheduling policies, the system must be able to distinguish between system's tasks and the framework ones. This is done by modifying in the data structure `task_struct`. This one, is used to manage any task's vital information. It is included a new field named `retas` of type `retas_task`, which contains all the important information regarding a ReTAS tasks.

Furthermore, because the framework needs high-resolution timers, since they provide finer resolution and accuracy depending on the system's hardware, they are activated at system start-up, in order to switch from standard timers to these. During this procedure, every framework timers are also initialized.

Then, there is a data structure called, `rq` that contains the information about every, per processor, run-queue. Because there are new functionalities being added, a new field in this structure is also added. This new parameter is of type `retas_rq`, the framework's new main run-queue type.

5.3.1 Tasks

As it was already mentioned, the ReTAS framework strives for unification. Its source code is segregated in a organized fashion. Please refer to Listing 5.2 to see the ReTAS task data structure implementation.

The `task_id` and `server_id` fields are used to store the current task's identification number, and then, because tasks are allocated to servers, there is also the `server_id` parameter, which, has its identification number.

The next two parameters are `node_ready` and `node_release`. These are of type, `rb_node`, and thus they are nodes from a red-black tree. For now, it is important to understand, that each task can belong to ready and release queues, and these are represented using a red-black tree.

```
struct retas_task {
  int task_id;
  int server_id;
  struct rb_node node_ready;
  struct rb_node node_release;
  struct job_param {
    unsigned long long deadline;
    unsigned long long arrival;
    unsigned int nr;
#ifdef CONFIG_RETAS_STATISTICAL
    unsigned char state;
#endif
  }job_param;
  struct task_param {
    unsigned long long deadline;
    unsigned long long period;
  }task_param;
#ifdef CONFIG_RETAS_STATISTICAL
  struct task_stat *stat;
#endif
};
```

Listing 5.2: retas_task data structure.

Then there is the `job_param` data structure, that contains a series of fields: `deadline`, `arrival`, `nr` and `state`. The first three ones, belong to several job characteristics, the task's absolute deadline, the task's absolute arrival, both in nanoseconds, and the job's number. The last parameter is used for statistics and it used to keep track of the task state.

Next, the `task_param` data structure, that, in contrast to the previous one, stores tasks characteristics. Those are: `deadline` and `period`. They are the task's relative deadline and its relative period accordingly.

To conclude, the last field, `stat`, is a data structure of type, `task_stat`. It belongs to the statistics mechanism and it will be explained in more detail in the statistics section of this chapter. Succinctly, it contains a specific task statistical information.

To update the task's information according to the data obtained from the off-line procedure, there are several system calls available. The `sys_retas_set_task_param` is a system call used to set the scheduling policy and its priority according to the framework's new policy using the function `sched_setscheduler`. Then, there is, `sys_retas_set_task` to get the framework task's information from user-space and store it its data structure.

5.3.2 Servers

Listing 5.3 shows the declaration of the main server's data structure. Tasks and servers are identified with a number. This structure, maintains the one belonging to a server using its first parameter, `id`. Then, because a server may execute in one or two processors, an array containing their identification numbers is used. The structure, `cpus_allowed` is used for this purpose. This array is accessed later, to check if a server does not execute when it is not allowed. Note that,

for Carousel-EDF [1] this field is not considered since all servers execute in all processors.

The `curr_cpu` parameter contains the processor's identification number that is currently holding the server. Because a processor may only be executing one server at a given instant, there is a variable `flag`, that serves to know when the server is being used by a processor, it holds zero if it is unlocked, one otherwise. Both of these variables are essential, to provide a way in which to guarantee that servers do not execute out of their bounds, or when a processor tries to access it and it was not been relinquished yet. The next processor executing that server must wait until the server is unlocked.

```
struct server {
  int id;
  int cpus_allowed[2];
  atomic_t curr_cpu;
  atomic_t flag;
  unsigned char type;
  struct ready_queue ready;
  struct release_queue release;
};
```

Listing 5.3: ReTAS servers data structure.

Furthermore, comes the server's type, using the `type` variable because a server may be single, non-split or split. Finally, the `ready` and `release` data structures, holding all the information about the ready and release queues for the server in question. The purpose of these queues was provided in the last section, though it is important to remember that ready queues only exist at the server level, but for release queues, they may be optionally used at the server level, or at the processor level in the main framework's run-queue. These queues are instanced in both of these places, and may or not be used depending on the way in which tasks are released.

When a task is added to a server, using the function `server_enqueue_ready_job`, it is enqueued in its corresponding ready queue, which later sorts the queue's red-black tree nodes according to the uniprocessor policy chosen for that server. Additionally, in resemblance to the ready queue example, there is also a function, `server_enqueue_release_job`, to add a task to its release queue.

When a server is fetched for execution, a checking procedure is done to ensure that the server was already been relinquished. The function `server_is_executing_on_other_cpu`, is used for that purpose. Servers may be locked and unlocked to manage their access and schedule their tasks inside their reserves using the functions, `server_lock` and `server_unlock`.

If the task's releases are done at the server level and when the reserve timer expires to change from one reserve to another, the tasks belonging to the next server to execute allocated in that reserve must be woken, so that they start to execute. The `server_try_wakeup_jobs` function is called to achieve this by gradually emptying the server's release queue, putting the next task to be release into a running state and activating it. This results in a new job being executed, which also makes the contents of the server's ready queue to be updated upon activation.

[1] The original designation is maintained, but in the implementation there are two more variants that could be called, Carousel-RM and Carousel-DM, as it is explained in Chapter 4

To conclude, the system call named `sys_retas_set_server_param`, is used to obtain the information from a server from user-space and store it in the server's main data structure.

5.3.3 Reserves

Focusing now on reserves, they are time intervals in which a group of tasks allocated to a server are available for scheduling. Listing 5.4 shows the reserves main data structure declaration and the reserve run-queue data structure that is used for each processor in order to do the scheduling. Reserves added to their run-queue are fetched and accessed in a specific order which follows several rules for the different algorithms available.

```
struct reserve {
  struct server * server;
  struct server * alt_server;
  unsigned long long length;
};
struct reserve_rq {
  int idx_curr_reserve;
  struct reserve *reserves;
  int nr_reserves;
};
```

Listing 5.4: main reserve and reserve run-queue data structures.

For each reserve there can be, the main server assigned to the reserve, represented by the variable of name, `server`, and the alternate which might be used if needed, called, `alt_server`. Reserves have lengths computed during the scheduling off-line stage, and the `length` variable contains its value.

There are two system calls available to prepare reserves before the scheduling starts. They are `sys_retas_init_reserves` and `sys_retas_set_reserve_param`. The first is used to simply to clean them, and the second, to pass the information about the servers that were allocated, main and alternate, plus the reserve length from user-space.

The reserve run-queue data structure, contains, index of the currently executing reserve, `idx_curr_reserve`, for that processor. Again, there is an array that contains every reserve that is going to be scheduled, and every processor has access to it, therefore, a processor must start executing a reserve that is accessible by using this index in this array. The currently executing reserve, must be easily accessible, and to solve this, a pointer is used, which is known as `reserves`. Finally, there is the `nr_reserves`, which is the number of reserves allocated to that processor.

The current and next executing reserves are easily accessible, by using the functions `reserve_rq_get_curr_reserve` and `reserve_rq_get_next_reserve` which basically, work with the number of reserves and the index that is being used for the currently executing reserve. The index is incremented by one unit and it is reset to zero if it surpasses the total number of reserves. Using this method, it is possible to only allow the processor to access its reserves and not every one of them, if the main algorithm used is S-EKG or NPS-F. At the same time, a global array may be accessible by all processors, which is necessary for Carousel-EDF.

The index of the first reserve for a processor in the global reserve array can be set from user-space by using the `sys_retas_set_cpu_start_reserve` system call. The system call `sys_retas_set_cpu_reserves` uses the first reserve to execute, using the same index sent from user-space to initialize the pointer to the first reserve that will be scheduled for that processor. At the same time, the total number of reserves allocated is also sent and stored.

Reserve timers are used when decisions in how the next reserve should be picked for execution and its subsequent server. When these timers are set up by using the function `reserve_timer_setup_timer` their callback is set too, in this case `reserve_timer_reserve_fn`, and they remain inactive. Two values must be passed from user-space, the timezero and the offset, using the `sys_retas_set_timezero` and `sys_retas_set_cpu_offset` system calls. Both of these values are used to correctly start these timers and compute the next expiring instant, using the function `reserve_timer_start`, and adding both of them to the size of the timeslot, which is computed by the sum of the size of each reserve allocated to the current processor. Additionally, these timers remain inactive for single servers.

The callback is called upon timer expiration, which basically selects the next reserve to be executed, and computes the next expiring instant by adding the current expiring instant to the next reserve length. Then, if the task's releases are made at the server level, it tries to wake them so that tasks may start to execute as it was explained previously. The last scheduled and executed server, is now free to be unlocked, which in a later stage will be, because the current task is marked for rescheduling and this results in a new scheduling decision. When this happens, the main scheduling function is called, which at the end, after selecting the next task to be executed, executes another function named, `post_schedule_retas`, that, unlocks the previously scheduled server, by using the task that is currently executing and comparing its server with the one that belongs to the currently executing reserve and check if they differ. If they do, an unlock is required, because the other server is no longer being used.

5.3.4 Ready and release run-queues

Ready and release queues data structures are very similar. They are, because both types will have, during the scheduling, tasks associated to them and sorted in a specific fashion, were each task is a node in a red-black tree. Both of these data structures are declared according to Listing 5.5.

Both queues have the `lock` variable which is a spinlock to control its access by other framework mechanisms. Similarly, they both have the queue itself represented by a red-black tree, a variable of type `rb_root` and with the name `queue`. For each case, there is a pointer, `highest_prio`, for ready queues, and `erf` for release queues. They are used to easily access the tree's top node, that is, the highest priority task according to a uniprocessor policy for ready queues, and the next task to be released for release queues. In ready queues there is an extra parameter, `policy`, that specifies the uniprocessor scheduling policy that is going to be used to schedule tasks that belong to the current ready queue.

To enqueue a task by according to an uniprocessor scheduling policy, the function `ready_`

```
struct ready_queue {
  raw_spinlock_t lock;
  struct rb_root queue;
  struct task_struct *highest_prio;
  unsigned char policy;
};
struct release_queue {
  raw_spinlock_t lock;
  struct rb_root queue;
  struct task_struct *erf;

};
```

Listing 5.5: ready_queue and release queue data structures.

`enqueue_job` was conceived. This function obviously uses the lock to manage the access to the tree, and goes through each node. According to the policy chosen, it reorganizes the nodes by checking the parameters from the task that is being enqueued, and those that are already inside the tree. It then changes the ready queue `highest_prio` field properly so that it can be accessed to get the new highest priority task, later to be used to execute a new job. Furthermore, the task enqueue process is similar in the release queue's case, but instead of sorting according to a uniprocessor policy it sorts according to the task's absolute arrival.

To ensure that tasks are released, a timer for each processor is used. When this timer expires, the task with the most urgent arrival is released, and is given processor time, which results in one job being executed for that processor. When this timer expires, the task is added to a release queue, which results in the queue being sorted again by absolute arrival. It is important to know, that while this is happening, and when a task begins its execution, job by job, the timer is started and its next expire instant corresponds to the absolute arrival from the next task to be released. The job's arrival is passed from user-space, using the system call `sys_retas_delay_until`, that starts the timer according to the release method chosen and this can be done using the system call `sys_retas_set_release_policy`. If this arrival is already late according to the current time, then a deadline miss occurs.

Again, there are two methods available to release tasks, the difference is in how the timer starts and were the next task to be released is obtained. In the simple case, where the release queues are used in each processor, the task is enqueued in the release queue and the timer is started from that moment. The second case, which is a little more complex, selectively starts the release timer according to the currently executing reserve. In this case, release queues are used at the server level and timers only start and expire only when they are executing. Please refer to Listing 5.6 for details.

When one of these timers expires, it uses its callback act differently for both release methods, but still, they are very similar. Upon expiration, the next task to be released must be woken, that is, it dequeues the task from the release queue, and if everything went well, the task starts to run executing a new job.

Then, the expiring instant of the release timer must be updated using, `release_timer_get_next_release`. This function, gets the release value from the task that will be executed

```
if(server_enqueue_release_job(server, p)){
      next_release     = server_get_next_release(server);
      reserve_timer_state    = reserve_timer_get_state(reserve_timer);
      if(reserve_timer_state == RESERVE_TIMER_ACTIVE_NO_STARTED){
        __hrtimer_start_range_ns(&release_timer->timer, ns_to_ktime(
            next_release),0, HRTIMER_MODE_ABS_PINNED,0);

      }else{//RESERVE_TIMER_ACTIVE_STARTED
        reserve_timer_expires = reserve_timer_get_expires(reserve_timer);
        if(next_release < reserve_timer_expires - RELEASE_SLACK){
          __hrtimer_start_range_ns(&release_timer->timer, ns_to_ktime(
              next_release),0, HRTIMER_MODE_ABS_PINNED,0);
        }
      }

    }
```

Listing 5.6: release_timer_do_delay_until2 function implementation.

next, using the proper release queue according to the release method. Finally, the task is set to be rescheduled, so that it can enter again in the system, and its execution demands fulfilled.

5.3.5 Main framework run-queue

The framework extends the original system per processor run-queues, and to do so, adds a new field at the `rq` data structure present in the original scheduling module. This new field, is of type `retas_rq`, and it adds information to it using a new data structure that communicates with several of the framework's mechanisms. This new data structure, is visible in Listing 5.7.

```
struct retas_rq {
  unsigned char post_schedule;
  int prev_server_id;
  struct reserve_rq reserve_rq;
  struct reserve_timer reserve_timer;
  struct release_queue release;
  struct release_timer release_timer;
  struct resched_curr_cpu_timer resched_timer;
};
```

Listing 5.7: main ReTAS run-queue data structure.

The first parameter, `post_schedule`, is used to know if a server unlock should be performed when it ends its execution when a reserve timer expires. The second, `prev_server_id`, is used to keep track of the last server that was executed which is useful during the time where a server unlock is required, that is, an unlock is required if the currently executing server is different from the one that executed before and this variable is used to make this comparison. The rest of the parameters are straightforward, they are used to access the reserve run-queue, the release queues, if used, and the timers already studied. Nevertheless, the parameter, `resched_timer`, holds access to another kind of timers which are described in the next section.

5.3.6 Scheduling module

In this section, the core framework's scheduling functionalities are analysed. The first mechanism to be studied corresponds to the act of getting a new task to be scheduled. The `_pick_next_task_retas` function is used for this purpose. It is here, that the next task to be scheduled is returned, that is, the one with higher priority. Listing 5.8 shows the implementation of this mechanism. Upon accessing the currently executing server, if it is not possible to obtain the next task, the `next` variable is returned with `NULL`. It is in here that the task rescheduling timers are used.

```
server = reserve_get_server(rsv);
  if(server){
    next = server_get_highest_prio_job(server);
    if(next){
      if(server_lock(server, rq) == LOCKED_BY_OTHER_CPU){
        resched_curr_cpu_start_timer(resched_tm);
        next = NULL;
      }
    }
    if(!next){
      server = reserve_get_alt_server(rsv);
      if(server){
        next = server_get_highest_prio_job(server);
        if(server_lock(server, rq) == LOCKED_BY_OTHER_CPU){
          next = NULL;
        }
      }
    }
  }
  if(next){
    smp_wmb();
    next->on_rq = 1;
    task_thread_info(next)->cpu = rq->cpu;
  }
  return next;
```

Listing 5.8: _pick_next_task_retas implementation extract.

These timers have their implementation which follows the same philosophy in resemblance to the other timers already described. Still, they work slightly different, they don't expire at a computed time instant, instead they do so in a previously defined interval. This interval can be modified using the system call, `sys_retas_set_resched_curr_cpu_timer_interval`. Once the callback is called, which in this case, is, `resched_curr_cpu_timer_fn`, the currently executing task is targeted for rescheduling using the function, `retas_resched_task`. As it can be seen from the source code extract, if the server is locked, these timers are started. The reasoning behind this solution, is that the server might need some time until it becomes unlocked, which results in the `next` variable being finally populated.

Meanwhile, when a new task is woken, a checking procedure is done to know if this task has higher priority than the one that is currently executing. The function, `_check_preempt_curr_retas`, is called and it returns, one or zero, either if the current task needs to preempted or not, respectively. The method in now this function operates is simple, it obtains the highest priority job from the currently executing server, and checks if it is the same job as the one currently executing. If this is true, no preemption is needed. If a preemption is needed, the current task is marked to be rescheduled.

When the system adds a task to a processor's run-queue, because for example, it enters in the system for the first time or it is marked for rescheduling the function `_enqueue_task_retas` is called. Once it starts executing, `__enqueue_task_retas` is called, and it enqueues the current task to its respective server ready queue, following a specific uniprocessor scheduling policy. It does so, by also computing the current task's absolute deadline accordingly, and storing it, using `task_set_job_deadline`. Similarly, when a job finishes its execution the task must be dequeued from the ready queue associated using another function, `dequeue_task_retas`, which just accesses its server and removes it.

5.4 Tracing mechanism

When scheduling events are recorded with the aid of the tracing mechanism that was devised, it allow later, to use this information, in order to make even more studies about an experiment.

To store this information, a data structure, named `trace_record`, containing it was created, Listing 5.9 shows it in detail. This structure is instanced for each processor available, so that when the data is collected, the results will have a processor associated.

As it can be seen, there is also another structure with the name `trace_log`, which represents a single entry in the list, or in other words, a single event traced. The parameters `front` and `rear` correspond to the index of the last and first items recorded, `nitems`, is used to know the number of items already in existence, and, finally `tracing` is used to check if the tracing mechanism is enabled or not.

Revisiting the `trace_log` structure, it is composed by a header, which is another data structure that contains two parameters, `event` and `time`. The first parameter is defined by an enumeration which dictates available events to store. Examples are: `T_START_RES`, when a reserve timer expires and another reserve begins its execution, and `T_UNLOCK_SERVER`, when a server is unlocked, but there are other types available. The second field, is the time instant in which the event was recorded in nanoseconds.

Whenever an event needs to be traced, a function, designated, `trace_event` is called. These are, in order: `cpu`, the current processor's identification number, `time`, the event's time instant, `arg`, an additional argument which contains a pointer to something where additional data must be taken from, and finally, `event`, the event's type.

The way in which this function operates is simple. Once a `trace_log` variable is composed with the information to be traced, the function `trace_insert` is called. While, this function obtains the tracing information from the current processor, it updates the `front` index by incre-

```
struct trace_log {
  struct trace_header header;
  union {
    struct trace_job job;
    struct trace_reserve reserve;
    struct trace_server_locking lock;
    struct trace_resched_cpu resched;
  } data;
};
struct trace_record {
  struct trace_log log[RETAS_TRACE_SIZE];
  atomic_t front;
  atomic_t rear;
  atomic_t nitems;
  atomic_t tracing;
};
```

Listing 5.9: trace_log and trace_record declaration.

menting it. After this, the old `rear` value is obtained, and this index is used to store the new tracing data and finally, `rear` is incremented. There is a limit to store events, which makes the `rear` variable to be reset if surpassed.

The method in which the tracing information is obtained from user-space and successfully written to a series of files, is through the creation of a device driver, and the trace reading process from user-space is done in `retas_trace_read`. The function obtains the current tracing information and if there are existing events recorded, it reads and copies it to user-space using a buffer, updating the `front` variable so that new data can be constantly read and written to the buffer, later to be accessed from user-space.

5.5 Statistics mechanism

Besides events being traced, several statistics are taken too, for the purpose of analysing the behaviour of several components during an experiment. In this section, it is explained the solution devised to achieve this. Perhaps, the first data structure to be studied is: `stat_feature`, which defines the parameters that are to be stored for each feature in which values are taken. A feature is nothing more than a element of study, and examples of these would be: `relJ`, `RelO`, both the release jitter and the release overhead, `ipiL`, the latency generated by Inter-Processor Interrupts(IPIs)[2], but many others are recorded and every feature is explained with some detail later on. Listing 5.10 shows its declaration.

Whenever a new value from this structure needs to be updated, `stat_insert_feature`, is called, with the information regarding the feature to be updated and a new value to be saved, upon which, all of its parameters are updated. The `min` and `max` parameters are the minimum and maximum values stored until that moment, `sum` stores the sum of every value added, `sum2` stores the sum of all the values multiplied by themselves, and finally, `nr_regs`, just maintains

[2]It is a type of interrupt that is used when a processor interrupts another, if, the interrupting one requires action from it.

```
struct stat_feature {
  unsigned long min;
  unsigned long max;
  unsigned long long sum;
  unsigned long long sum2;
  unsigned long nr_regs;
};
```

Listing 5.10: stat_feature data structure.

the number of times a new value was been added for that feature. These values are then used to calculate the mean and standard deviation.

The statistics belonging to each task reside in a data structure of type `task_stat_list`. This structure is composed of several parameters, being them: `enabled`, `nr_wasted_regs`, `lock` and `list`. They are used to check if the statistics are enabled, the number of records to be wasted[3], a spinlock to manage the list's access, and finally, a list containing all the tasks statistical data respectively. Each item of the latter contains a data structure of type, `task_stat`, and this one contains the all the statistics information being recorded for a single task. Additionally, it also has a parameter designated, `aux`, of type `task_stat_aux`, which might be used to pass additional information for some events. Next, it is shown the method that is used to compute each task's statistical information.

Statistical information for each processor is also obtained, and is saved in a data structure of type `cpu_stat`. Within this structure, the are two more parameters of importance, `reserve_stat` and `tick_stat`. The first refers to the statistical information for each reserve that was scheduled, and the second belongs to the statistics gathered when the no idle tick timer expires. This timer is used while a processor is in an idle state.

In chapter 6, it is analysed a graphical application that shows these statistics in the form of a box-and-whisker plot for each task and processor in study. Furthermore, in 7, some results taken are shown. To understand these results, one needs to grasp the nature, that is, the way in how they are measured, and each one is represented by an alias, and it is this identification that is used in these plots. They are, for each task, the following: `PREE`, `C`, `T`, `RT`, `RelJ`, `RelO`, `CtswO`, `TickO` and `IpiL`, and for each processor: `TickC`, `TickT`, `ResJ`, `ResO`, `ResS` and `ResL`. Finally, it is described, what they are, and how they are obtained.

- `PREE`: refers to the number of preemptions that a task as suffered. Whenever a task is preempted, the function, `stat_task_stat_preempted`, which updates this value, when a task finishes its execution the final value is stored.

- `C`: it is the task's execution time. When a task begins its execution, `stat_task_stat_start` is called, and it saves the time instant when the task started to execute. Later on, when it finishes, `stat_task_stat_completed` is called, and the the final value is equal to the difference between the current time instant and its start saved earlier minus the

[3] this parameter is used, to ensure that less records not belonging to an experiment are recorded. To achieve this, a number of wasted records is numerically set, and only those that come after are to be accounted as statistics.

current `TickO`.

- T: measures the task's period if the last job deadline was not missed. When a task becomes ready, the `stat_task_stat_ready` function is used, which keeps track of the previous enqueue instant. If this value exists, the period is equal to the difference between the current instant and that value.

- RT: is the time interval when a task as become ready and its completion. Again, when a task becomes ready, `stat_task_stat_ready` is used and stores the instant when it became ready.

- RelJ: measures the task's release jitter, that is, the interval when the release timer was fired, that is saved with `stat_task_stat_release`, and its current job arrival instant. The final value is stored when the task becomes ready using `stat_task_stat_ready`.

- RelO: corresponds to the task's release overhead given by the difference of the instant when the task became ready and the release timer's expiring instant. Once again, this is stored in `stat_task_stat_ready`.

- CtswO: measures the context switch overhead, which is computed according to the difference between the task's re-execution, and the scheduler's invocation. This is measured when a task begins its execution and `stat_task_stat_start` is called.

- TickO: it is the overhead incurred by the no idle timer. This one is used while the processor is not executing anything. When it expires, `stat_task_stat_tick` is called. The latter, keeps track and constantly sums the timer's overhead, that is given by the difference between the current time and when the its expiring instant. The summed value is used to store the task's execution time correctly, by not using the time while the processor was idle, and it is reset whenever a deadline miss or a preemption occurs. Finally, when a single timer expiration occurs, `TickO`, is updated.

- IpiL: a task's release in a split server can cause an IPI which is used to notify a task's release between processors that share it. This may generate a latency created by the IPI, which is designated here as `IpiL`. This happens only in special cases, where the arrival of a task's, shared between P_p and P_{p-1}, job occurs in P_p and is the one with higher priority in the current server, while at the same time, falling the arrival inside the reserve from processor P_{p-1}. This results in an IPI being sent from P_p to P_{p-1} to notify about this arrival. This process end up calling `stat_task_stat_ipi_fired` which sets a flag about this occurrence. This IPI latency may, or may not happen, and this is why the `CtswO` is measured separately. At the end, the final latency value is stored in `stat_task_stat_start`, when a task begins execution. The flag is accessed, and the latency is computed by the difference between the scheduler's invocation and the previous enqueue instant, that is, the last time when the task became ready.

- TickC: it measures the execution of the no idle tick. This is computed in the function `retas_stat_cpu_tick` which is called after the no idle tick callback finishes its execution. In the callback both its initial instant and its finishing instant are stored, and the final value is the difference.

- TickT: it is the period between the no idle tick expire instants. Also calculated in `retas_stat_cpu_tick` which keeps track of the last no idle tick expire instants, the period is the difference between two adjacent instants.

- ResJ: measures the reserve release jitter that is the difference between the time when the reserve timer is fired, or in other words, when the next reserve should have started, and its next reserve timer expiring instant. This value is saved in `retas_stat_cpu_reserve` and this function is used when the reserve timer callback finishes.

- ResO: corresponds to the reserve overhead, calculated by the difference between the instant where the reserve timer callback finished its execution and its next expiring instant. Again, the final value for this overhead is also saved in `retas_stat_cpu_reserve`.

- ResS: is the reserve context switch true interval, it is calculated using the instant when the scheduler is invoked where a context switch happens, which results in `retas_stat_cpu_start_reserve` being called when a reserve starts to execute. The final value is given by the difference between this instant and the time when the previous reserve truly started its execution.

- ResL: is the reserve latency, it is also calculated using the instant when the scheduler is invoked where a context switch happens, and the latency is given by the difference between this instant and the time when the previous reserve should have started its execution. Again it is also computed in `retas_stat_cpu_start_reserve` when a reserve starts to execute.

Succinctly, in order to copy all of this information to user-space, system calls are available which simply go through all the list entries. Furthermore, they finds the entries which have the information from the task or processor needed. When this happens, it is used a function to convert the statistics feature to a string. And while they are converted, they are written to a buffer, which will, at a later stage, be copied to user-space.

5.6 Summary

In this chapter the implementation of the ReTAS framework was been studied, first, by describing it main features and solutions devised more abstractly, while at the same time emphasizing the responsibility behind their presence. For each framework's components, their data structures where dissected, so that when exploring the operations done in the various functions available, it would be possible to understand how they work more efficiently. At the end, it is explored the method in which to obtain results later to be examined.

The next chapter is provides an analysis of an application that was been constructed which uses these results. It serves to aid the process of testing in this area of research, by providing a visual representation of what is done while the framework is scheduling tasks. This is done primarily in the form of a Gantt diagram.

Chapter 6

Visualization Tool Analysis

6.1 Introduction

The last chapter provided a brief explanation in how data is collected. The mechanisms responsible, are the genesis to a new level of research experiments now possible to produce. Furthermore, when one looks at this raw information, it is not directly appealing nor humanly easy to extract real information. Nevertheless, it is possible to create a tool that automates this process and creates a visual representation of what is happening while tasks are scheduled. This is expressed in the form of a Gantt diagram. The tool described in this chapter promises this, perhaps, never done before feature in this area of research, and a new group of functionalities that allows easy navigation and results filtering. A box and whisker plot is generated for each task and each processor available, showing their statistics attractively, which results in even more conclusions to be done. The main objective is to obtain empirical evidence in this area of study. Additionally, it serves as a debugging tool, that is, to know if the implementation is correctly devised. This chapter provides an overview of the piece of software that was developed, explaining the approach taken for each functionality created, the advantages, and the real conclusions that can be learned once an experiment is made.

6.2 Application overview

This section provides an overview of the features that are included in the final application. Each one was obviously conceived to promote the idea that the information created during a scheduling experiment is in fact easily accessible and well represented.

The main feature is to draw a Gantt diagram were for each row, and each one, belonging to a processor, has its scheduled elements drawn, such as, tasks and reserves. Fortunately, for each processor, a file is created containing its traced contents, and therefore a parser for them was conceived so that each event is properly read and stored, later to be used during the drawing process. Once this is is done, the drawing process can start.

Unfortunately, it is impossible to represent the hole diagram in one single screen and be able to visualize its contents correctly. Each event is traced in nanoseconds, and there are, in normal

circumstances, hundreds of events to be drawn. his small granularity, demands that there must be a zooming mechanism that can be used to allow one to visualize a bigger or smaller timespan of the scheduling events. Plus, it is also possible to navigate, according to the current zoom level, by incrementing or decrementing a fixed number of nanoseconds, so that it also it is possible to navigate forth and backwards in the graph. Additionally, provided a value in nanoseconds one can jump to that instant.

Upon this, content filtering features were created. If a dedicated look to a specific item's type is needed, one can choose what to be drawn, so for example, if the user does not want to draw job arrivals or reserves he can do so. Furthermore, a specific task or reserve can be enabled or disabled. Tasks and reserves are represented through randomly chosen colors and these can also be defined arbitrarily.

Content can be clicked, to allow the user to easily get detailed information about it. A table with the tracing information for each processor is also shown in the application. This permits the visualization of its contents easily, and also aids in the inclusion of filtering capabilities. The user can filter the contents by choosing what types of events to show. Then, if he double clicks in a table row he will jump to that specific time instant in the graphic.

If statistical information is provided, the application draws a box-and-whisker plot for each task or processor, and at the same time visualise the tables that contained the data that was been used to do so. In the next subsections, an in depth description in how these features were conceived is provided and their main advantages too focusing on the real information that can be obtained.

6.3 Working with the tracing and statistical data

When an experiment is done, several files are generated at the end, it is known already that for each processor there is a file containing its scheduling information, section 5.4, shows this procedure in detail. The user then specifies a folder containing all of these files, and the next step to make, is to properly store the information inside these in data structures so that they can be easily fetched in the future. Once the folder containing these is provided, every single piece of data is stored properly by following a specific methodology, so that the initial drawing process starts and some user interface modifications are also made in this application.

Once the directory path is known, it is possible to know how much processors were used during the scheduling phase, this is because it corresponds to the number of existing files known with a similar alias. This number is important to know in order to allow the Gantt diagram to be drawn in a dynamically.

Using this solution the application is ready to draw a Gantt diagram using the information of an arbitrary number of processors. This is a huge advantage of this application, because it makes it possible to draw the scheduling that was been traced from different systems were the ReTAS framework resides.

When the parsing process starts, each line from each file is stored in a data structure, so that, again, for each item in it, it can be obtained in its raw state. This then allows to determine

the time zero value, which is the time instant of the first event that was recorded and not to be confused with the time zero from the ReTAS kernel module. This is done by fetching the first line of each processor's tracing file and sorting them from lowest to highest so that the the lowest time instant is the time zero instant. This value is stored, and it serves as the beginning time instant of the experiment. Its main responsibility is to aid in navigation process.

Each line is dissected, and according to the event type, identified by a number, the remaining parsing process is made according to it. One data structure is used for each type of event, and further information is stored in an organized way. For every event, its processor identification number, the instant, calculated through the time zero value, and additional information is stored. This additional data is used in the future to show in an unified way information, upon mouse clicking an item in the diagram.

At least two great advantages can be extracted, because if a new event is added in the future the employed mechanism can be used with just a few modifications. An event is, again, identified by a number, and to allow its parsing a new conditional statement is what remains to be added in order to achieve this. Furthermore, if new information must be added to a single item, that can also be done easily.

Nevertheless, despite this being true for all the elements being drawn, tasks and reserves are not entirely dealt in the exact same fashion, each reserve end instant is the beginning of the next one, and this is solved in this part of the prototype, that is, each start and end is within one single entry in the tracing information and not in two. Task's jobs also have beginning instants and ending ones represented by different events. This requires even more dedication, but so is also done. At the end of the parsing process, as an additional step, a random color is given for each task found during the parsing process.

Then the processor's and task's statistics files are accessed, and their information, unaltered, is stored in different data structures, later to be used upon drawing the statistics box-and-whisker plots.

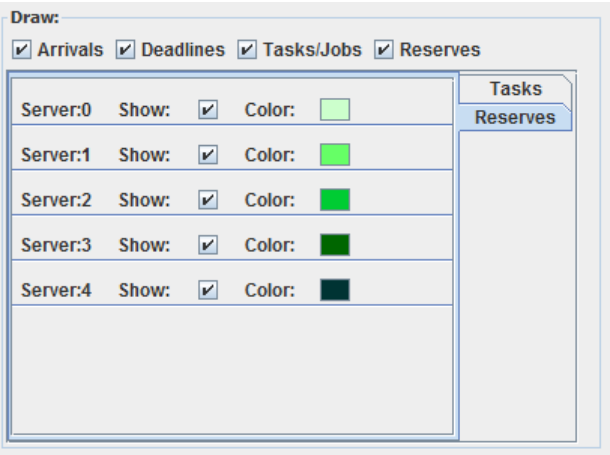


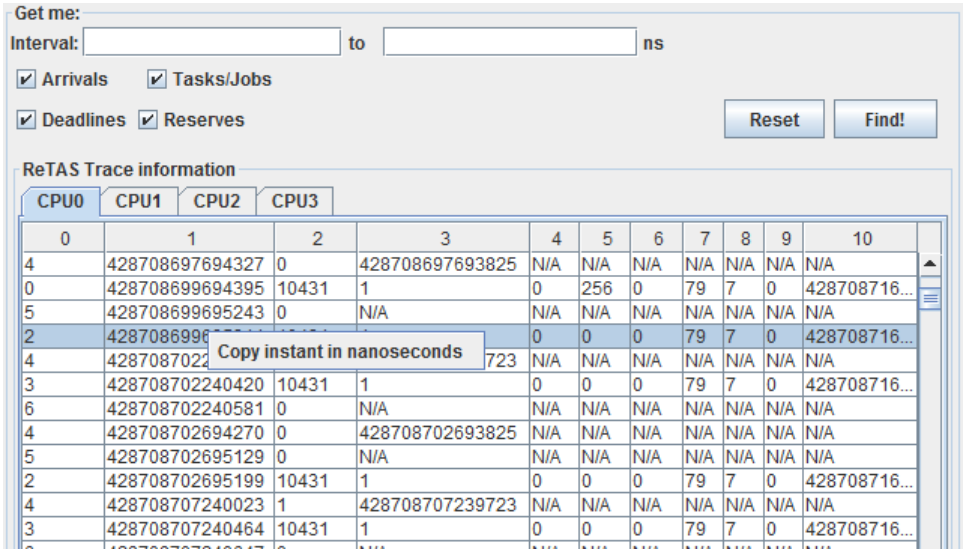
Figure 6.1: Visualization tool draw panel.

Once all files are parsed, it is known for trivial reasons, the preparations that must be done in the user-interface. If they fail to exist, the features enabled by their existence are disabled.

Again, if the main scheduling trace files are present and well constructed, a panel is populated with each task and reserve that was scheduled during the experiment. Please refer to Figure 6.1 to see this graphically. As it can be noted, for each task/reserve detected, a check box is added, and a color button. Once the check box is clicked, the item in question can be drawn or not, and the button serves to change the item's color. The checkboxes above are utilized to enable or disable one type of item in the diagram.

Again, some advantages can be seen in this mechanism, if the researcher desires only to analyse one single, task, reserve, or any arbitrary item type he can do so very easily. This is very useful to check for pathological behaviour and proof gathering, this is were mistakes during the algorithm's implementation in the kernel can be easily detected, because one can simply look at the diagram and check for obvious mistakes, such as for example, items being obviously out of place, or reserves overlapping. Plus, there are no limits on the number of tasks or reserves.

Next, it will be described, perhaps, one the most interesting features available. Once the initial data is read, it is also offered the possibility to directly visualize or filter the data that was been obtained from each processor's tracing file.



0	1	2	3	4	5	6	7	8	9	10
4	428708697694327	0	428708697693825	N/A	N/A	N/A	N/A	N/A	N/A	N/A
0	428708699694395	10431	1	0	256	0	79	7	0	428708716...
5	428708699695243	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2	428708699			0	0	0	79	7	0	428708716...
4	428708702		723	N/A	N/A	N/A	N/A	N/A	N/A	N/A
3	428708702240420	10431	1	0	0	0	79	7	0	428708716...
6	428708702240581	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
4	428708702694270	0	428708702693825	N/A	N/A	N/A	N/A	N/A	N/A	N/A
5	428708702695129	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2	428708702695199	10431	1	0	0	0	79	7	0	428708716...
4	428708707240023	1	428708707239723	N/A	N/A	N/A	N/A	N/A	N/A	N/A
3	428708707240464	10431	1	0	0	0	79	7	0	428708716...

Figure 6.2: An example of the visualization tool populated tables.

The way in how this is achieved, is to receive all the processor's tracing data in its unaltered state. This process begins to, by knowing the number of processors that were used in the experiment, to go through the data of each one of them, populating a table with a fixed number of rows equal to the number of entries read and a number of columns equal to eleven. This last number is used, because at most, eleven pieces of information are stored for each record. As shown in Figure 6.2, some records do not contain all of them, and thus, when this happens, a "not-available" string is put in place to warn the user. This illustration also shows the table's filtering capabilities.

These tables are non-editable for the reason that this would imply an automatic refresh of the drawing made, unfortunately this feature was not planed in advance, and it could be interesting

to manipulate this data to see how the scheduling phase could have gone if other results were taken. Despite that, other features are available. The table, upon double-clicking in one of its rows, allows the user to automatically jump to the record's time instant in nanoseconds.

Additionally, the user can also right-click one row to be able to copy the record's time instant directly to the operating system's clipboard. Refer again to Figure 6.2 to see this feature being used.

These filtering features are used upon clicking the ”find” button, which make the application to start to read the instant values from the two text boxes shown in Figure 6.2, and obtaining each record from each processor between those, plus the event types selected from the group of check-boxes available in the same panel. Which, in turn, will show the records filtered, in contrast to all the original information in the table. The user can also reset the values by clicking the ”reset” button.

The tables being shown, and the filtering capabilities provided, despite their initial state, allow the user to see the results taken, and stored in the files in their original state. This, with the combination of the filters available, allows the user to see a given group of events by type within a given timespan. This is very advantageous, because now, it is possible to see if one type of event is being recorded correctly without the visual interference of the other records present in the same file.

6.4 Gantt diagram visualization

Once the applications starts, and the tracing data read, a panel with the diagram is drawn were its size is determined by the number of existing processors that were used in the experiment, and other variables that can be manipulated to change the drawing at will. For example, the size of the drawing could be changed to fit more information, or perhaps to change the size of the space taken by processors or jobs. This can be done only at the source code level, but the way in how it was been devised was such to allow in the future to represent even more data easily. A mouse listener is added to it, to allow the user to click an item in the drawing and show additional information.

The drawing process starts, and it is composed by two phases, the first one, draws the axis of the Gantt diagram, and the second, draws and prepares the hit-boxes to be used with the mouse listener.

By looking at Figure 6.3, one can see the axis being drawn and some scheduling items. The scheduling shown in the picture, may not represent a positive experiment, it is just an example of something that could be obtained. The drawing is done in a simple fashion. Still, there are some considerations to be made. As it can be seen, the user may wish to navigate through it and thus it is in this phase that the limits to be drawn must be controlled, and the time instant in which to begin the drawing is defined.

The values in which to start and end the drawing are tracked. Only the items that belong inside this timespan are to be drawn. To do this, a global variable containing the number of nanoseconds represented inside each vertical line of the Gantt diagram with width of one single

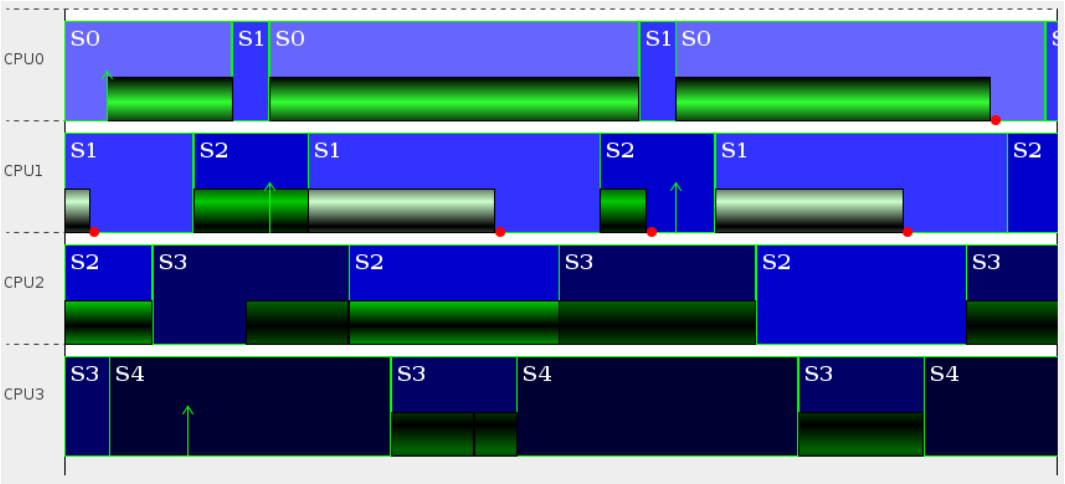


Figure 6.3: Scheduling drawing example.

pixel is used, representing therefore, the scale.

Inside the main drawing process, each data structure containing each type of elements to be drawn is accessed, and it is here that the application decides either or not to draw a specific type if the user enabled of disabled it. The solution employed is very similar to every case, and what is important to know is that, there is information about the current element hit-box and additional information. This information is to be displayed if a mouse click is made in it, if indeed the element is clickable. Then, in the mouse adapter implementation, the hit-boxes information is accessed, and if a mouse click is inside in one of them, a message is shown to the user, displaying a string containing the additional information.

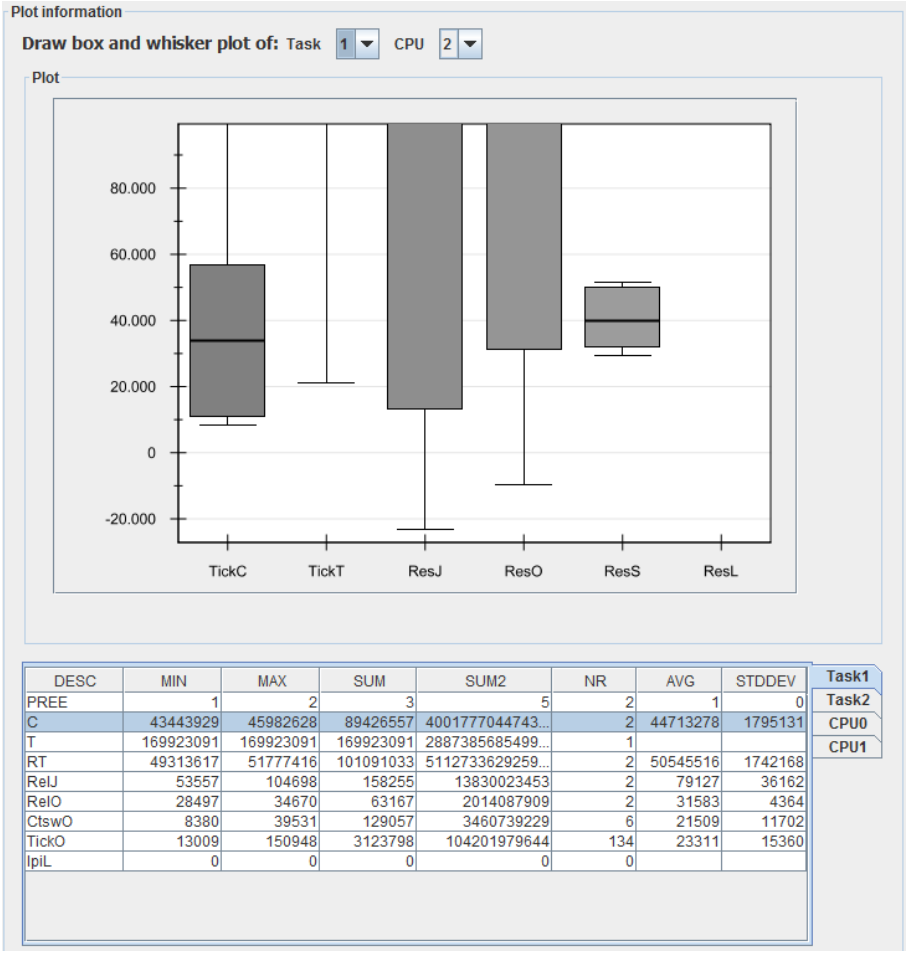
It is trivial to figure that the functionalities provided by the drawing are indeed useful. The Gantt diagram, can be navigated, zoomed in and zoomed out to obtain even more detail and show a desired quantity of information. These functionalities are possible, because if what is needed is to zoom in the drawing, the scale used, is divided by a determined zoom factor. In contrast, it is multiplied, if the user wants to zoom out. In both of these procedures, the end bound of the diagram is recalculated, and the drawing repainted. Again, if the user wants he can to navigate to freely. All of these capabilities result in a pleasing method to obtain empirical evidence of what is happening during the scheduling phase. In the next subsection, the reasoning behind the construction of box-and-whisker plots will be provided.

6.5 Box-and-Whisker plot creation

In order to generate a box-and-whisker plot from the statistics that were been taken initially, the application uses a third-party library designated *gral* [1]. Figure 6.4 shows an example of a plot that was been obtained. The interface allows the user to choose what plot to draw, either being it one from a given task or a processor. Once he chooses one of them, the plot is drawn. Then he can

[1]The gral library can be used to draw plots with scientific data in Java applications. For more details the reader can visit: http://trac.erichseifert.de/gral/

choose, using a tab panel in the interface, to show a table containing the statistics information that he wants, either to see the one that corresponds to the current plot, or to compare it. It is important to know, that each row corresponds to each statistic taken from those that were been discussed in the previous chapter, and each column has these values in this order: the minimum value, the maximum value, the sum, the sum of all the values multiplied by themselves, the number of values that were used in the statistic, the average, and finally, the standard deviation.



DESC	MIN	MAX	SUM	SUM2	NR	AVG	STDDEV
PREE	1	2	3	5	2	1	0
C	43443929	45982628	89426557	4001777044743...	2	44713278	1795131
T	169923091	169923091	169923091	2887385685499...	1		
RT	49313617	51777416	101091033	5112733629259...	2	50545516	1742168
RelJ	53557	104698	158255	13830023453	2	79127	36162
RelO	28497	34670	63167	2014087909	2	31583	4364
CtswO	8380	39531	129057	3460739229	6	21509	11702
TickO	13009	150948	3123798	104201979644	134	23311	15360
IpiL	0	0	0	0	0		

Figure 6.4: Box-and-whisker plot example.

Obviously, the main reason why the plot is being drawn is that it allows one to graphically see the way in how each statistic taken behaves during the scheduling. For example, some values are expected to be higher than others, and to see the reverse happening, would allow the researcher to know if something is wrong automatically. Additionally, the standard deviation is used to understand, if the data that was collected is indeed close to the average, and this can be seen by the upper and lower limits of the boxes in the plot. The rest of the information is very easy to grasp also, because the average is represented by the line that is being drawn in the middle of those boxes, and the minimum and maximum values are the lower and higher whiskers for each statistic respectively.

6.6 Summary

This chapter was conceived to give an ample view in how the visualization tool receives the data generated during an experiment, and successfully produces visual results. The benefits from this procedure were been discussed, emphasizing the idea that by analysing the raw files directly would prove to be a daunting task. Indeed, a researcher would jeopardize a great amount of time.

Each feature, and the methodologies employed in the resulting piece of software, were studied one by one, and additionally, a critical analysis was also done, by showing the way, in some cases, how the application could be improved. Furthermore, the advantages that each one of them brings to the whole research process was also been described. At the end, it was shown how the statistics are visually reproduced in a box-and-whisker plot, and the information that someone can take from them.

Chapter 7

Experimental Procedure

7.1 Introduction

This chapter is aimed to offer the necessary knowledge in how, abstractly, one could develop a piece of software that uses the framework functionalities, so that an experiment can be made. For this to happen, a methodology must be followed.

An application of this kind, communicates from user-space to kernel-space through the usage of a set the system calls that were devised. Recall that, scheduling algorithms under consideration require a set of information that has to be provided before runtime. As mentioned before, that information is given by the off-line procedure described in several parts of this document. The purpose of this information is to configure the different parts of the framework in order to, at runtime, produce the correct scheduling.

Next, it is explained the way in which one can read the trace and statistics information that can be gathered during the runtime scheduling phase, so that, after an experiment is performed, one can analyse the results gathered.

Finally, in order to provide some evidence that the framework can be used, results are shown through experiments that were conducted.

7.2 Preparation

A real time system is composed by a static set of tasks that has to be scheduled according to one scheduling algorithm in a system composed by a set of processors. This kind of systems require that, at off-line, their schedulability must be guaranteed. For that purpose, an off-line set of analytical tools are used to ensure that at runtime all the tasks real-time constraints will be met.

Therefore, in the ReTAS framework before any experiment starts, the information obtained from the off-line procedure, that is, the final task to server and server to processor accompanied with the computed reserves using the multiprocessor scheduling algorithms available must exist. Algorithm 6 shows, abstractly, the way in which the system preparation could be done, using the system calls available.

The first step is to prepare the server information. To do this, for each server, the system call

Algorithm 6 System preparation procedure

Input: Set of n tasks τ_i with $1 \leq i \leq n$, set of k servers P_q with $1 \leq q \leq k$, set of r reserves with lengths $Res^{len}[\tilde{P}_q]$, set of m processors P_j with $1 \leq j \leq m$. {off-line scheduling information}

for $q \leftarrow 1$ to k **do**
  $sys\_retas\_set\_server\_param(Id[\tilde{P}_q], Policy[\tilde{P}_q], Type[\tilde{P}_q], CpuAllowed1[\tilde{P}_q], CpuAllowed2[\tilde{P}_q])$
end for
for $q \leftarrow 1$ to r **do**
  $sys\_retas\_set\_reserve\_param(Id[\tilde{P}_q], AltId[\tilde{P}_q], Res^{len}[\tilde{P}_q])$
end for
for $j \leftarrow 1$ to m **do**
  $sys\_retas\_set\_cpu\_reserves(Id[P_j], IndexFirstReserve[P_j], NrReserves[P_j])$
  $sys\_retas\_set\_cpu\_start_reserve(Id[P_j], IndexStartReserve[P_j])$
end for
$sys\_retas\_set\_timezero(get\_current\_time\_nanoseconds())$
$sys\_retas\_start\_cpu\_reserves(P, m)$
$policy \leftarrow 1$
$sys\_retas\_set\_release\_policy(policy)$
for $i \leftarrow 1$ to n **do**
  $sys\_retas\_set\_task\_param(Id[\tau_i], ServerId[\tau_i], Deadline[\tau_i], Period[\tau_i])$
  $sys\_retas\_set\_task(ProcessId[P_j], Prio[P_j])$
end for
$sys\_retas\_stop\_cpu\_reserves()$

`sys_retas_set_server_param` should be used. This system call receives the current server identification number, the server's uniprocessor scheduling policy, which can be zero for EDF, one for RM and two for DM, then the server's type, which is also a digit, one for a NON-SPLIT server, two for a SPLIT and three for a SINGLE, and, finally, the processor's identifiers where the server was allocated (note that, processor's identifiers are not considered when the scheduling algorithm is any of the Carousel versions, because servers execute on all processors available).

After the first step, the preparation phase now needs to have the reserves prepared and this requires several steps. The first one, is to convey their most basic information, that is, the identification number of the server that was allocated to it, and the reserve length. To do this, the system call, `sys_retas_set_reserve_param` is available. Then, in order to have all of the algorithms available in one implementation, several information should be set for each processor available, the system call `sys_retas_set_cpu_reserves` should be called to specify the identification number of the processor's first reserve to schedule, used to initialize the pointer to it in kernel-space, and the number of reserves that the processor was mapped to. Finally, the index of the first reserve in the global reserve array is passed using, `sys_retas_set_cpu_start_reserve`.

Now, the timezero instant must be set with, `sys_retas_set_timezero` which receives the current time in nanoseconds and marks the time instant in which the scheduling procedure started. The reasoning behind this, is to synchronize the reserves mechanism among processors and other related information.

Before preparing the tasks that are to be scheduled and begin their execution, the reserve timers must be started with `sys_retas_start_cpu_reserves`. This is done at this moment, first, because they require the information that was conveyed previously to operate, and second, because the existing tasks require that these timers must be already active. Indeed, the reserve scheduling mechanism depends on them to switch from one to another.

It is important to remember that the implementation offers different strategies in how tasks are released, and before tasks start to be scheduled, the release policy must be set with `set_release_policy`, which receives one digit equal to one or two according to the release method chosen, again the first implements the release queues at the per-processor run-queue level and the other at the server level.

After all this, each task must have its own Linux process created in the system. For each one created, the task's main parameters must be set using, `sys_retas_set_task_param`, which receives the task's identification number, the server's identification number where the task was allocated, its relative deadline, and finally, its relative period, both in nanoseconds. Furthermore, the scheduling class of each task must be set to the RT scheduling class. Then, it is required to set a priority level according to it. This is performed by invoking `sys_retas_set_task`, which receives the task's process id, and the priority number desired.

Typically, real-time tasks are recurrent. Their algorithm is based on a loop where they perform some work and then wait until the next release. In the real-time context, each activation is called a job. The ReTAS framework has a system call that puts the calling task in a waiting state until the next release `sys_retas_delay_until`. The release is performed depending on the release policy chosen. Therefore, after one job's work is finished, it should compute the next release instant, which is equal to the sum between the previous arrival and the task's period. This happens, until a task fulfils its computational and temporal demands.

After this, when every task ends its execution, its trace and statistical information can be obtained using a filesystem node, which can be read like a regular file, and additionally, the reserve timers should be stopped.

Later in this chapter, some tests are presented that were made using an application that does all the steps described until this moment. The objective is to demonstrate how final results can be taken and analysed to prove that the framework can be used in a real OS.

7.3 Environment

The experiments were performed in a host platform using a 4-core Intel Core i7-920 Processor with 6GB of main memory where each core is able to run at 2.66 GHz. The Linux distribution used, was Ubuntu version 10.04 LTS and the kernel's version 3.10.15 in combination with the PREEMPT-RT patch.

The first three experiments were performed using the same task set included in Section 4.4 and in the next one it will be revisited. The reasoning behind this, was to test the framework in conjunction with the schedulability analysis included in chapter 4. This task set was constructed by hand in order to have a few high utilization tasks coexisting with several light utilization ones, for the sake of simplicity. It is already known, that these tasks reproduce a schedulable task set and therefore no unusual behaviour should be expected provided that the algorithms are working correctly.

To run an experiment, a piece of software written in C was used which implements the steps described in the last section and spawns a new process for each task, where each one starts at

the same time. Within each process, a loop is made where each iteration corresponds to a single job executed, and it is here, in a single iteration, that a task's work is simulated. A job's deadline miss is registered using the current system time. The next job's release is calculated, and the task sleeps until that instant, so that a new job can be executed.

7.4 Task Set example

Revisiting the task set information again, as it is possible to see in Figure 7.1, it is divided by three insets. The first one, that is, Inset (a), is a visual representation of each task's utilization. Inset (b), shows the results obtained from the task-to-server assignment. Each task is assigned to servers according to the First-Fit bin packing heuristic, while their schedulability is checked according to RTA(described in Chapter 4).

Finally, Inset (c), shows the reserve lengths computed for each server, to do so, the existence of a fake task was used, as it is explained in Subsection 4.3.2.

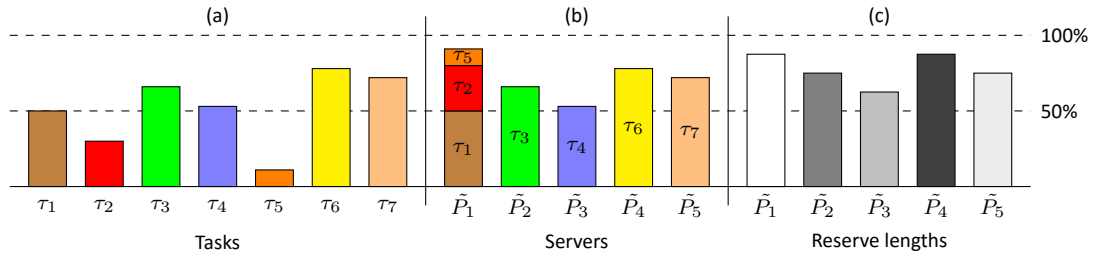


Figure 7.1: Task set example.

Next, the experiments done are analysed, showing some of the results taken, providing also an interpretation of what happened during their execution. For each one, an extract of the Gantt diagram generated by the visualization tool is provided, giving some evidence that the components involved result in tasks being scheduled. Then, it is also shown the behaviour of a processor and a task as examples, knowing that each one behaves similarly to the remaining ones. Another objective, and that should be accomplished in the last experiment, is to have a grasp of the magnitude for each overhead already discussed in 5.5.

7.5 Experiment with NPS-F-RM

Again, this experiment was done using the task set shown in Section 4.4. It was constructed and its feasibility checked using the other application that was conceived to produce the example shown in chapter 4. The latter, is able to, not just to perform the schedulability test required, but also to produce the off-line information. The experiment ran during a 100 second interval, and each value's resolution in the task set is in milliseconds. The Figure 7.2 shows an extract of the output obtained from the user-space application described in Section 7.3 after the experiment finished.

The output shows that the application, designated as `LAUNCH`, successfully read the off-line

```
LAUNCH with PID:1853: gets experiment config [./ts_3_d/config]
LAUNCH:1853: PRIO 20
LAUNCH:1853: starting on 4 cpus
LAUNCH:1853: set servers param
LAUNCH:1853: init reserves
LAUNCH:1853: set reserves
LAUNCH:1853: set cpu reserves
LAUNCH:1853: set cpu start reserve
LAUNCH:1853: set cpu offset
LAUNCH:1853: set time zero
LAUNCH:1853: start cpu reserves
LAUNCH:1853: set release policy
LAUNCH:1853: forks 7 retas_tasks
LAUNCH:1853: changes the scheduling policy
LAUNCH:1853: experiment started at: Wed Oct  7 11:13:42 2015
LAUNCH:1853: wakes up retas_tasks
LAUNCH:1853: pid range of retas_tasks (1854--1860)
LAUNCH:1853: is waiting ...
TASK:1854:1:0:12502
TASK:1855:2:0:10002
TASK:1856:3:0:6668
TASK:1858:5:0:5265
TASK:1857:4:0:5884
TASK:1860:7:0:2382
TASK:1859:6:0:2042
LAUNCH:1853: stop cpu reserves
LAUNCH:1853: finishes
LAUNCH:1853: experiment finished at: Wed Oct  7 11:15:31 2015
```

Figure 7.2: User-space application's output while running the experiment.

information and did the proper system preparation, upon which it launched seven tasks and each process is called, `TASK`. It also shows, for each one, upon finishing its execution, additional information. For each entry, one can read in this order: the task process id, the task id, deadline misses occurred, and finally, the number of jobs executed. As it can be observed, no deadline misses were registered.

To proceed with this experiment, the application described in Chapter 6 is now used for testing purposes. Figure 7.3 shows an extract of the Gantt diagram created by this tool, were six complete timeslots are possible to observe. The time interval shown is arbitrary.

Furthermore, in these experiments, tasks are numbered according to the task set, that is, they begin from τ_1 to τ_n where n is the number of existing tasks, though this is not quite possible to see in the following figure but they are represented by different colors and each task's job is represented by a rectangle with gradient. Each task's arrival is represented by a light green coloured arrow pointing upwards and each task's deadline is drawn using a small red circle.

In contrast, each server, $\tilde{P}_k$, numbered in the task set from $\tilde{P}_1$ to k where k is the number of servers in existence, is now numbered, and represented in the figure, from 0 to $k-1$, that is, $\tilde{P}_1$, is actually indexed in the experiment with the alias $S0$ and so on. This representation is combined with each reserve rectangle that is also color coded.

Each processor is also numbered in this fashion. Aside from this, the algorithm's implementation requires that an extra server is added, to fill the remaining computational space not used by the last server in the last processor and this is also visible in the same diagrams. In some instances, it is not possible to visualize the server's alias completely, and therefore, it is important to know that each reserve is represented by a distinct color, as it was said earlier. Reserves from different processors, that have the same color, are those to which the same server was allocated.

As it can be observed, it is possible to know that each task is executing inside their respective reserves, and these are scheduled correctly according to the example that was shown in chap-

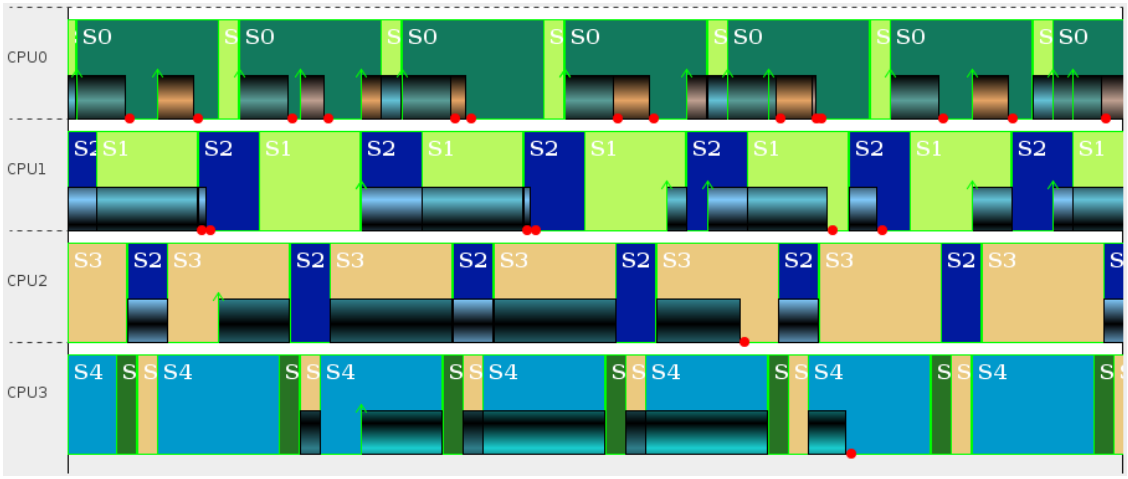


Figure 7.3: First experiment Gantt diagram's extract.

ter 4. This is possible to prove because each task is shown in a color that was chosen previously. Each server is also allocated in its respective reserve correctly. The figure also shows at least one task's arrival and one task's finish, which as it is possible to see, follows what is expected. The tasks that were allocated to the first server, are light ones and therefore they appear more often in the overall scheduling phase, and, in contrast, the task that was allocated to the fifth server, is a heavy one, and it takes more computational resources until it finishes its execution.

Clearly, the objective of this tool was fulfilled, it certainly helps some of the testing and debugging process that the framework demands, since if it was observed something strange in the drawing, the component(s) that were wrongfully drawn would provide an hint to what to check for software problems, provided, of course, that this tool is correctly generating the Gantt diagram.

The next step, is to have an idea about the behaviour of each statistical information gathered. Again, the same tool allows one to generate a box-and-whisker plot for each task and processor. As an example, please refer to Figure 7.4, and Figure 7.10 to observe what was been obtained for the fifth scheduled task in the original task set.

To continue with this example, please observe the Figure 7.6 and 7.7 which focuses on the statistics of a processor instead of a task.

The main objective of these results is to have a perception in how some overheads measured behave and other components too, the way in which these are measured was already discussed

DESC	MIN	MAX	SUM	SUM2	NR	AVG	STDDEV
PREE	1	7	6133	8057	5262	1	0
C	1190964	1232698	6271195...	7473958...	5262	1191789	1716
T	18996573	19003095	9995899...	1899220...	5261	18999999	341
RT	1192934	6971268	1812169...	7927903...	5262	3443879	1790708
RelJ	132	3237	2016084	975191804	5262	383	196
RelO	243	1112	1822100	656510428	5262	346	69
CtswO	308	1929	4011167	2803709...	6135	653	171
TickO	402	2096	3380040	2260764...	5513	613	184
IpiL	0	0	0	0	0		

Figure 7.4: Visualization tool task 5 table with the statistics information.

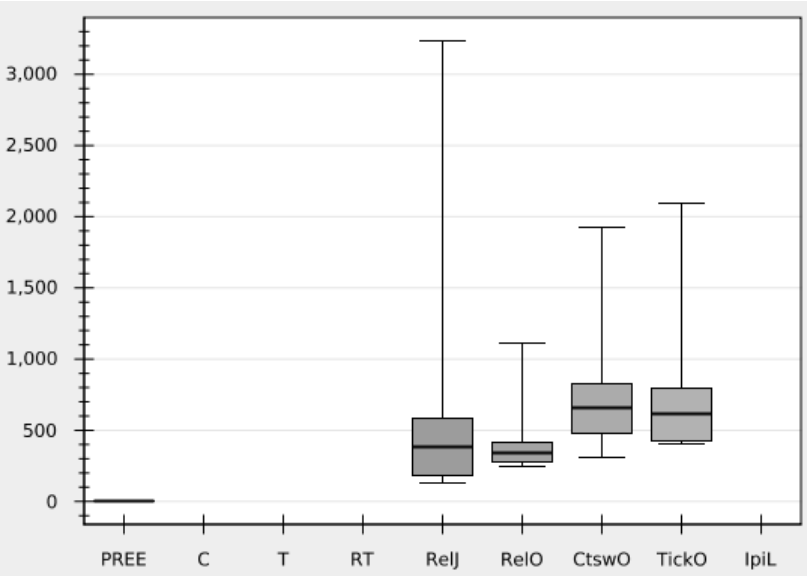


Figure 7.5: Visualization tool task 5 Box-and-whisker plot.

DESC	MIN	MAX	SUM	SUM2	NR	AVG	STDDEV
TickC	143	7095	68629966	5830458...	134142	511	415
TickT	991143	1012657	1341409...	1341410...	134141	999999	301
ResJ	128	5238	15537137	9910984...	33509	463	284
ResO	231	5592	22132620	1806500...	33509	660	320
ResS	137	1363	3415646	1005348...	12472	273	74
ResL	435	4487	10451900	9843925...	12472	838	294

Figure 7.6: Visualization tool processor 1 table with the statistics information.

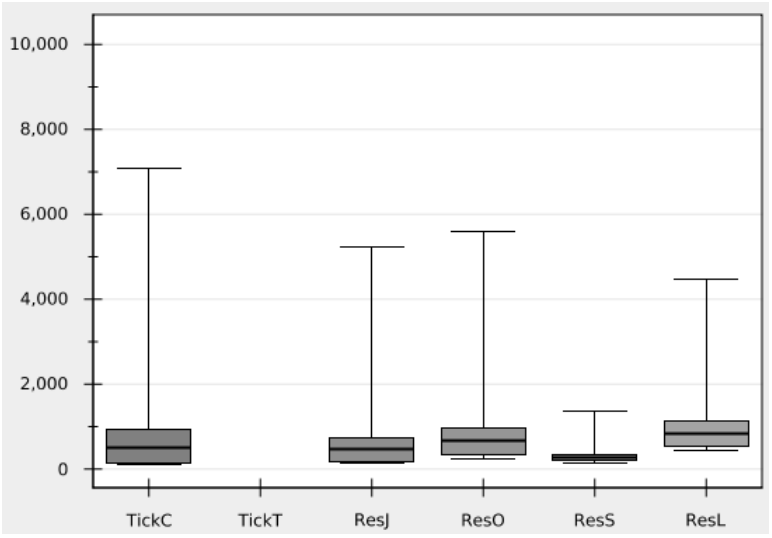


Figure 7.7: Visualization tool processor 1 Box-and-whisker plot.

in Section 5.5. By looking at the plots, for each one of them, one is able to see the range of values that were obtained and have an idea how much they, perhaps, can really be, considering also the computed average and standard deviation to understand the how they diverge. Some results are not shown in the plots. Their range of values differs much when compared to the others, as it can be observed from the tables provided. Otherwise, when using the zooming capabilities of this application the rest would not be possible to observe either. The objective is to show the most information possible. In the other hand, `IpiL`, is not being collected, because no IPIs occurred during the experiment. Again, this only happens in special cases, and for an experiment that only ran for a brief period of time, it is completely normal.

7.6 Experiment with S-EKG-RM

For this experiment, the only difference is that the S-EKG algorithm is used. Again, S-EKG and NPS-F share the same low-level implementation, and they differ in the off-line information that is conveyed from user-space to kernel-space. The S-EKG algorithm has a requirement, that is, only one task can coexist in a SPLIT server, and when this task is not available for scheduling a task inside the NON-SPLIT server in that processor is chosen instead. For this to be possible, the off-line information as to be changed in order to reflect this requirement. For each processor, whenever a split server also coexists with a non-split server, the alternate server for the split ones must be set, which actually is, the non-split one. The Gantt diagram's extract shown through Figure 7.8 illustrates this with some detail.

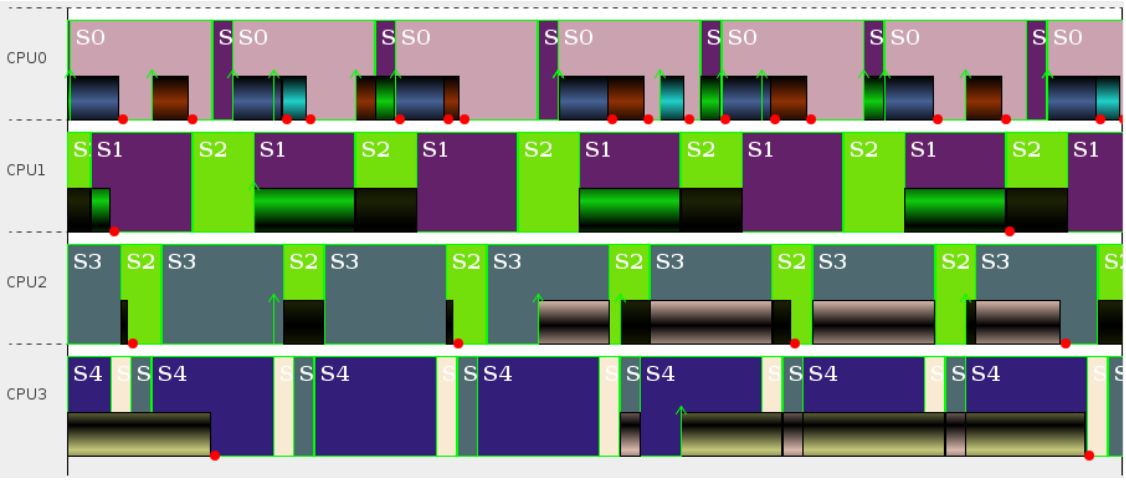


Figure 7.8: Second experiment Gantt diagram's extract.

When one looks at the beginning of the Gantt diagram, in the last processor(CPU3), it is possible to see that the non-split task, in this case, τ_7, is executing on all the servers allocated to this processor: the additional one used to fill the remaining space, and the other in which the SPLIT task (the task assigned to SPLIT server) is not available. The NON-SPLIT task (the task assigned to NON-SPLIT server), which was mapped to the $\tilde{P}_5$, here represented by the alias S4, is executing on server $\tilde{P}_5$ here represented as S3. Later on, in the same processor, the task allocated

to that split server arrived in the previous processor and is shared normally between them. Every task and processor is behaving similarly to the previous case, without any significant difference, so no further statistical information is shown.

7.7 Experiment with Carousel-RM

It was also performed an experiment using Carousel-RM as scheduling algorithm. Again, the same task set was used and the experiment ran for 100 seconds. The final purpose was to prove that the Carousel philosophy can also be combined with the newly added uniprocessor algorithms with the scheduling analysis produced. Figure 7.9 shows a random timespan of the scheduling obtained using the application already mentioned.

As it is possible to visualize, the big change is that all processors share every server, and each is sequentially accessed, no split servers exist, and therefore, there are less reserves in consideration, in fact, only one for each server. Each processor then starts executing the first reserve that is allowed to.

Taking a look at Figure 7.10, one can visualize that IPIs occurred between processors, in contrast to the last experiments. Indeed, this fact is more probable to happen in this case, since every task is shared by every processor available. Please refer to section 5.5 for more details. Observing the information taken, it is now possible to understand that the computational resources spent in an IPIs are, in fact, quite similar to the other overheads.

Concluding, these three experiments ran without any problems, and despite being unrealistic because they were done in such a small time interval, no deadline misses occurred. Unfortunately, no big differences are possible to see, because the measured statistics are similar among processors and tasks. Nevertheless, if these experiments were done in longer time intervals differences may start to reveal, and it is with this strategy and tools, that they can be detected and analysed. Again, what these served to prove, is that now one is able to obtain visual evidence that it is, indeed, possible to schedule a task set successfully using the ReTAS framework.

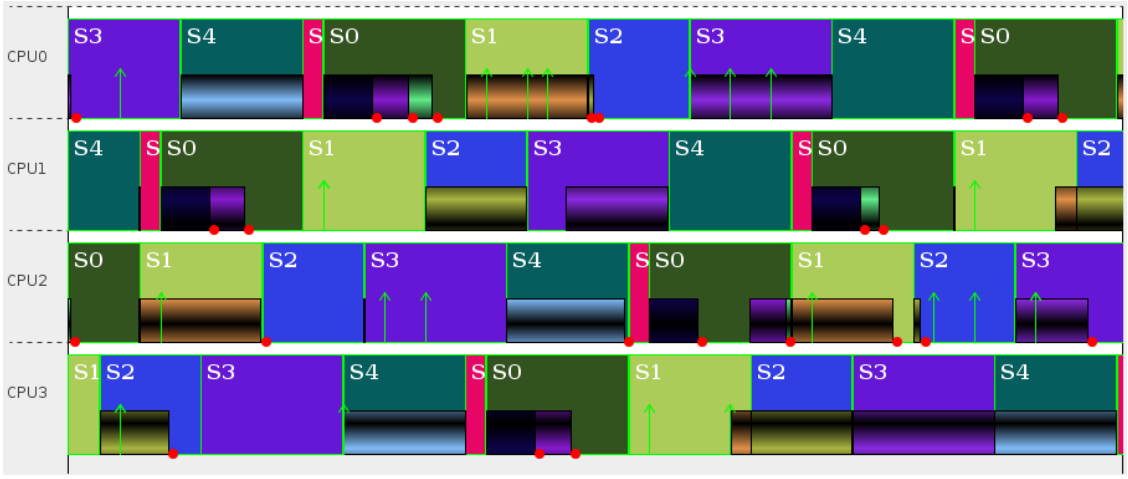


Figure 7.9: Third experiment Gantt diagram's extract.

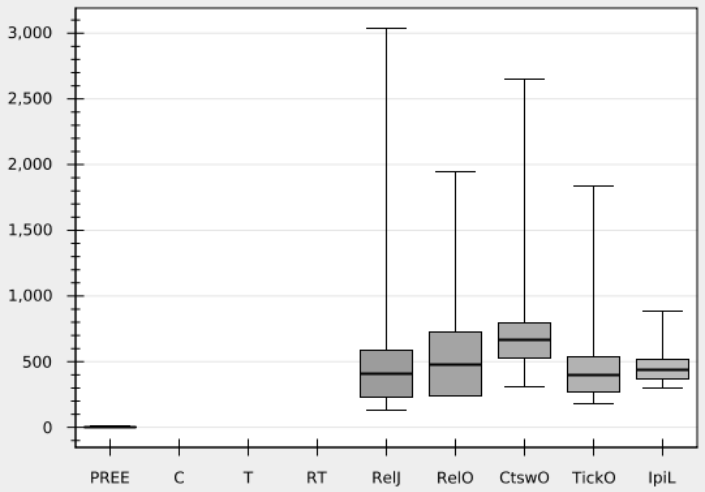


Figure 7.10: Visualization tool task 2 Box-and-whisker plot.

In the next section, a different kind of experiment is performed. Most newly added variations, firstly described in 4.1 are now tested, and all of their results combined, so that there is now a primordial methodology to quantify the overheads described earlier.

7.8 Experimenting with several scheduling variations

For this case, instead of only using one task set, ten were randomly generated for each scheduling variation used. Each task set is also composed by a group of tasks that is also random, where in some cases, more than ten tasks were scheduled. Furthermore, each task is mixed in nature, where their utilization, u_i, can range from 0.05 to 0.95. Because the task set generator, is not capable to generate task sets that result in having only one task in the resulting split servers, that is the requirement for S-EKG, only NPS-F and Carousel were considered, using, of course, the uniprocessor policies added in this work, that is, RM and DM.

Each task set ran for a total of 100 seconds, and for each one, the task's and processor's overhead measures were taken and combined, considering only the maximum values of each one of them. The reasoning behind this choice is that, due to the criticality of RTSs, only the worst values should be considered. Figure 7.11 shows a box-and-whisker plot that represents graphically each overhead. Unfortunately, due to the large range of quantities between them, a non-linear base

Table 7.1: Information taken for each overhead.

Description	**ResJ**	**ResO**	**ResS**	**ResL**	**RelJ**	**RelO**	**CtswO**	**TickO**	**IpiL**
Elements	160	160	160	160	288	288	288	288	54
Average	12421.4	12696.1	26710.1	31789.6	8721.7	1293.6	2389.1	1661.3	965117.9
Std. Deviation	3966.9	3956.8	81378.9	80235.2	4029.6	917.8	1471.8	935.9	3799139.6
Maximum	25669	25868	486747	487084	21963	5596	7016	6717	27999193
Upper Quartile	14963.3	15167	10696	16597.3	9387.8	1306.8	2193.3	1716.5	1317.5
Mean	13220	13428.5	2288.5	9899.5	7145.5	1076	1808.5	1417	705
Lower Quartile	9234	9446.3	1173.3	6961.3	5298	796	1649	1161.3	538.8
Minimum	5637	5916	561	1552	1028	590	893	760	354

10 logarithmic scale for the y-axis was used to show all the information conveniently. Nevertheless, the reader is asked to look at the table 7.1 to visualize the real information that is illustrated in the plot.

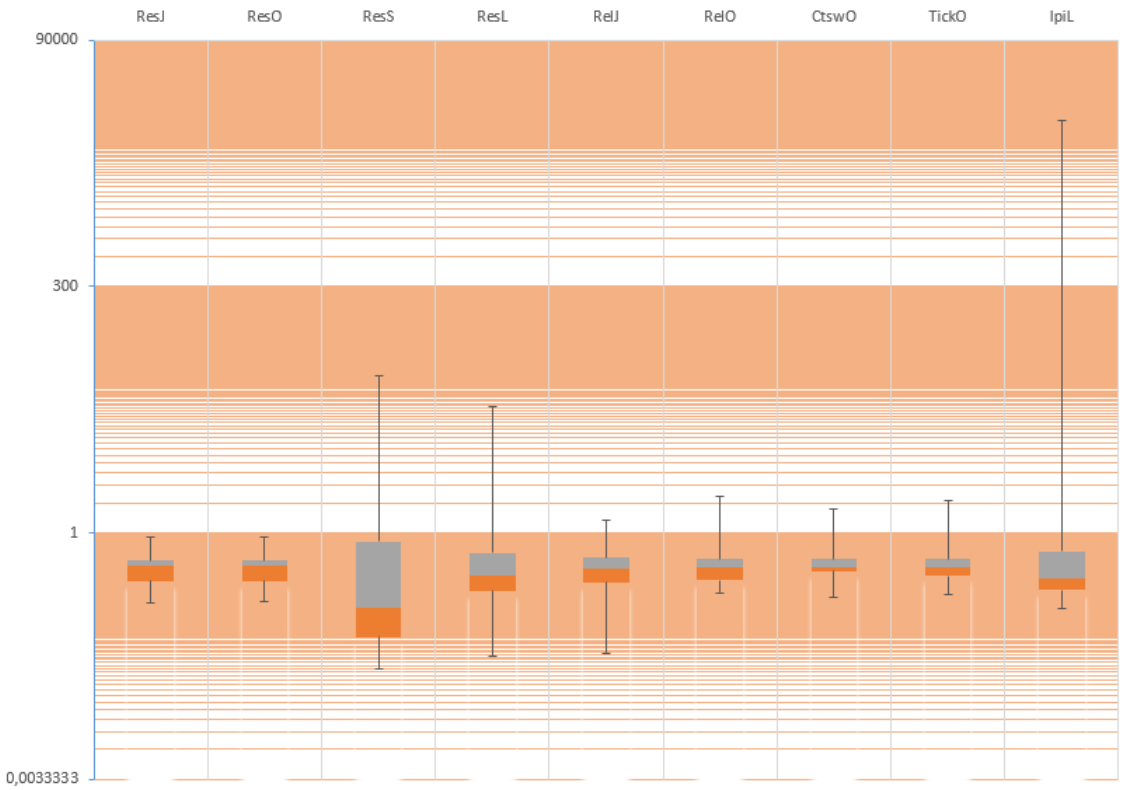


Figure 7.11: Box-and-whisker plot showing graphically each overhead.

Despite the interest being finding the worst case possible for each overhead, the box-and-whisker plot information is useful to understand several information about the overheads. First, the upper and lower whiskers represent the maximum and minimum values taken. The median, the line in the middle of the two different coloured rectangles, is the expected value, shows that at least 50% of the obtained information for that overhead is actually greater than it, and the upper and lower quartiles, shown as the upper and lower coloured rectangle limits, represent that at least 25% of the values used are greater or lower respectively than them. The standard deviation, is useful to know the quantity of variation from the set of values used, being those closer to 0 the ones with less dispersion. Using this knowledge, it is possible to visualize that all overheads are quite similar, with the exception of `ResS`,`ResL` and `IpiL` which diverge much more than the rest. Additionally, these have maximum vales, that are much greater, specially in the last case, which sugest that they should perhaps be treated differently in future developments.

The point to prove with this experiment, is that, this methodology, could perhaps be used to realistically quantify each one of them, considering the worst-cases possible as a basis to do so. Enabling the possibility to continue with the work presented in Chapter 4 which requires that overheads should be considered because otherwise problematic results occur.

7.9 Summary

In this chapter, it was explored the method which was used to run each experiment. This was important because there are specific steps to be taken in an order, and the off-line information should be conveyed from user-space to kernel-space correctly. Then, a few experiments were made, each one, showing a different case for the purpose of comparison and observing the behaviour of the framework. But this was also performed to test the work here devised, that is, to testify the synergy created by each tool here presented.

In the next chapter, the final conclusions are drawn, focusing on what, and in how the work devised can generate an improvement in the research process involved. To complement this effort, future improvements are also discussed.

Chapter 8

Conclusion

Semi-partitioned scheduling shows itself as a promising method to obtain satisfying results, and this is done by dealing with the scheduling process more efficiently by combining the advantages from the other scheduling schemes, namely the global and partitioned scheduling methods. This dissertation aimed to provide a real implementation of those algorithms and by demonstrating, that it is possible, by using the tools described, to do experiments and to obtain results from them. The big objective was to bridge the gap between theory and practice.

This chapter is, therefore, divided in two parts, first, conclusions are presented, and finally, future work that must be done to continue with the development in this area of research.

8.1 Conclusions

With the development of modern operating systems, either in the general or the real time area, more and more computational demands are generated because of the software's complexity. This factor demands that the processors being used must be more capable. Heat generation eventually stopped the advancement in the uniprocessor area and this is why this scheme is not suitable any more. To obtain better performance, less financial costs and less power consumption, the multiprocessor scheme was introduced.

With this hardware evolution (or revolution) imposes the evolution (or revolution) of the underlying software, namely those that are responsible for the management of the hardware resources.

This dissertation deals with an important component of such software: the *scheduler*. The scheduler is responsible for the management of CPU time. It defines which task execute on it and when it relinquishes it, and which is the next one to be executed. This is called process scheduling, that is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular scheduling algorithm.

Scheduling algorithms for uniprocessor systems are considered mature due to their usage in the last decades. But, with the advent of multiprocessor systems, new software started to appear for multiprocessor systems, but the uniprocessor techniques could not be directly applied,

because of the new layer of complexity created by the existence of multiple processors.

One of the areas that needed more development was real time scheduling, which enjoys mature work for uniprocessor systems. A task set could be successfully scheduled without problems as long an optimal scheduling algorithm was employed together with robust schedulability analysis. This is highly important for hard real time systems were failing a task's deadline could result in a catastrophic event.

Meanwhile, new multiprocessor scheduling perspectives started to be developed, with the big example of global and partitioned scheduling. Both offered a number of advantages and limitations. However, semi-partitioned scheduling was developed to inherit their advantages. Unfortunately, to prove their usefulness in a real operating system requires much work to be done.

The work presented in this dissertation, demonstrates how some of these semi-partitioned scheduling algorithms can be implemented in a real operating system, which in this case, was Linux. This resulted in a framework that provides an unified implementation of some multiprocessor scheduling algorithms.

With this, this work also shows how information is gathered during a scheduling experiment. This information, which is difficult to assess with the human eye in its raw state, is used in an application conceived that draws a Gantt diagram representing scheduling events, and this is what ends up aiding in the research process, because it is now possible to actually see the scheduling produced, and to conclude if something wrong occurred, provided, of course, that this application is well implemented.

Despite this area of research being complex, and trying to implement multiprocessor scheduling algorithms can be a daunting task, the Linux's kernel version used certainly helped, due to its modularity and unification, specially in its scheduling core. In the beginning of this work, a framework's version already existed, but it was not as organized as it is now, and it did not offered the same functionalities, with emphasis to the new uniprocessor algorithms added, which resulted in new scheduling variations never used before, during the runtime procedure. Because this last feature generated an environment that was different, new scheduling analysis was conceived to contemplate the newly created scheduling variations, since until then, only unified schedulability analysis that considered EDF as the on-line uniprocessor algorithm was in existence. This was done combining RTA, which was known to be a good schedulability test for the new uniprocessor algorithms added to the framework. At the end, it is believed, that the big objective of this work was completely achieved. Indeed, positive results were obtained, experimenting and using all the tools here created.

8.2 Future Work

The schedulability analysis presented, although showing that it can be applied, still needs to be completed. For now, it does not include the existence of operating system overheads. This is a must, overheads affect the scheduling, and they cannot be neglected. This is why, in the chapter 7.8, the worst possible cases for each overhead were obtained, and used to conclude that they

could be used to serve as the basis to complement them in the schedulability analysis required. The initial steps for future development, are therefore, available.

The OS version, and processor's architecture used(x86_64), were conceived for general use, and therefore, are not suitable for RT computing. To further prove that this work is truly useful, it is necessary to actually use the work devised in an environment acceptable for this area. Fortunately, at least, in the second case, this should not be that difficult, since the source code produced is not too dependent from the architecture with the exception from the system calls.

The visualization tool, unfortunately, has some performance issues when it is working with a lot of information, even when showing information from four processors, which nowadays is a small number, still it is be prepared to be able to draw the scheduling of any arbitrary number of processors. It is predicted, that for more than eight, unusual behaviour can occur.

The next big addition to this work, would be to use a real application together with the framework to schedule its tasks according to the scheduling policies offered. It was shown how to conceive an application that communicates with the framework, and so was done to perform the experiments needed. The task sets used were previously prepared, or, randomly generated. To use the framework with a real application requires that a method must be conceived to construct the task set, which is something that, for now, was not explored.

In conclusion, the software produced, allowed the construction of several experimental tests that ended up showing promising results, it was possible to successfully schedule task sets and to use the scheduling information in the visualization tool to see what really happened during that process. This fact, allows one to conclude that semi-partitioned scheduling can indeed be implemented in a real operating system.

Bibliography

[Andersson and Bletsas, 2008] Andersson, B. and Bletsas, K. (2008). Sporadic multiprocessor scheduling with few preemptions. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ECRTS '08, pages 243–252, Washington, DC, USA. IEEE Computer Society.

[Andersson et al., 2008] Andersson, B., Bletsas, K., and Baruah, S. (2008). Scheduling arbitrary-deadline sporadic tasks on multiprocessors. In *proc. of the 29th IEEE Real-Time Systems Symposium (RTSS'08)*, pages 385–394, Barcelona, Spain.

[Andersson and Jonsson, 2000] Andersson, B. and Jonsson, J. (2000). Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *7th International Workshop on Real-Time Computing and Applications Symposium (RTCSA 2000), 12-14 December 2000, Cheju Island, South Korea*, pages 337–346.

[Andersson and Tovar, 2006] Andersson, B. and Tovar, E. (2006). Multiprocessor scheduling with few preemptions. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '06, pages 322–334, Washington, DC, USA. IEEE Computer Society.

[Audsley et al., 1993] Audsley, N., Burns, A., Richardson, M., Tindell, K., and Wellings, A. (1993). Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292.

[Audsley et al., 1995] Audsley, N. C., Burns, A., Davis, R. I., Tindell, K. W., and Wellings, A. J. (1995). Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Syst.*, 8(2-3):173–198.

[Baker, 2010] Baker, T. (2010). What to make of multicore processors for reliable real-time systems? In Real, J. and Vardanega, T., editors, *Reliable Software Technologiey â€“ Ada-Europe 2010*, volume 6106 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg.

[Baltarejo Sousa et al., 2014] Baltarejo Sousa, P., Bletsas, K., Tovar, E., Souto, P., and Akesson, B. (2014). Unified overhead-aware schedulability analysis for slot-based task-splitting. *Real-Time Syst.*, 50(5-6):680–735.

[Baltarejo Sousa et al., 2013] Baltarejo Sousa, P., Souto, P., Tovar, E., and Bletsas, K. (2013). The carousel-edf scheduling algorithm for multiprocessor systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on*, pages 12–21.

[Barr, 2002] Barr, M. (2002). Introduction to priority inversion. `http://www.embedded.com/electronics-blogs/beginner-s-corner/4023947/Introduction-to-Priority-Inversion`.

[Bini and Buttazzo, 2004] Bini, E. and Buttazzo, G. (2004). Schedulability analysis of periodic fixed priority systems. *Computers, IEEE Transactions on*, 53(11):1462–1473.

[Bletsas and Andersson, 2009a] Bletsas, K. and Andersson, B. (2009a). Notional processors: an approach for multiprocessor scheduling. In *proc. of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'09)*, pages 3–12, San Francisco, CA, USA.

[Bletsas and Andersson, 2009b] Bletsas, K. and Andersson, B. (2009b). Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In *Real-Time Systems Symposium (RTSS), 2009 IEEE 30th*, pages 447–456. IEEE.

[Burns et al., 2012] Burns, A., Davis, R., Wang, P., and Zhang, F. (2012). Partitioned EDF scheduling for multiprocessors using a $C = D$ task splitting scheme. *Real-Time Syst.*, 48(1):3–33.

[Buttazzo, 2005] Buttazzo, G. C. (2005). Rate monotonic vs. edf: Judgment day. *Real-Time Syst.*, 29(1):5–26.

[Coffman et al., 1997] Coffman, Jr., E. G., Garey, M. R., and Johnson, D. S. (1997). Approximation algorithms for np-hard problems. chapter Approximation Algorithms for Bin Packing: A Survey, pages 46–93. PWS Publishing Co., Boston, MA, USA.

[Davis et al., 2015] Davis, R. I., Burns, A., Marinho, J., Nelis, V., Petters, S. M., and Bertogna, M. (2015). Global and partitioned multiprocessor fixed priority scheduling with deferred preemption. *ACM Trans. Embed. Comput. Syst.*, 14(3):47:1–47:28.

[Dertouzos and Mok, 1989] Dertouzos, M. L. and Mok, A. K. (1989). Multiprocessor online scheduling of hard-real-time tasks. *IEEE Trans. Softw. Eng.*, 15(12):1497–1506.

[Eigenmann and Lilja, 1998] Eigenmann, R. and Lilja, D. J. (1998). Von neumann computers. *Wiley Encyclopedia of Electrical and Electronics Engineering*.

[Fu and Schwebel, 2014] Fu, L. and Schwebel, R. (2014). Real-time linux wiki. `https://rt.wiki.kernel.org/index.php/Main_Page/`.

[Gepner and Kowalik, 2006] Gepner, P. and Kowalik, M. F. (2006). Multi-core processors: New way to achieve high system performance. In *PARELEC*, pages 9–13. IEEE Computer Society.

[Guan et al., 2009] Guan, N., Stigge, M., Yi, W., and Yu, G. (2009). New response time bounds for fixed priority multiprocessor scheduling. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, pages 387–397.

[Guan et al., 2010] Guan, N., Stigge, M., Yi, W., and Yu, G. (2010). Fixed-priority multiprocessor scheduling with liu & layland's utilization bound. In *proc. of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*, pages 165–174, Stockholm, Sweden.

[Guan and Yi, 2012] Guan, N. and Yi, W. (2012). Fixed-priority multiprocessor scheduling: Critical instant, response time and utilization bound. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2470–2473.

[Jones, 2007] Jones, M. T. (2007). Anatomy of linux synchronization methods: Kernel atomics, spinlocks, and mutexes. `http://web.archive.org/web/20090209170415/http://ibm.com/developerworks/linux/library/l-linux-synchronization.html`.

[Kato and Yamasaki, 2007] Kato, S. and Yamasaki, N. (2007). Real-time scheduling with task splitting on multiprocessors. In *proc. of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, pages 441–450, Daegu, Korea.

[Kato and Yamasaki, 2008] Kato, S. and Yamasaki, N. (2008). Portioned EDF-based scheduling on multiprocessors. In *proc. of the 8th ACM/IEEE International Conference on Embedded Software (EMSOFT'08)*, pages 139–148, Atlanta, GA, USA.

[Kato and Yamasaki, 2009] Kato, S. and Yamasaki, N. (2009). Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *proc. of the 21st IEEE Euromicro Conference on Real-Time Systems (ECRTS'09)*, pages 239–248, Dublin, Ireland.

[Lakshmanan et al., 2009] Lakshmanan, K., Rajkumar, R., and Lehoczky, J. (2009). Partitioned fixed-priority preemptive scheduling for multi-core processors. In *proc. of the 21st Euromicro Conference on Real-Time Systems (ECRTS'09)*, pages 239–248, Dublin, Ireland.

[Lee et al., 2007] Lee, I., Leung, J. Y.-T., and Son, S. H. (2007). *Handbook of Real-Time and Embedded Systems*. Chapman & Hall/CRC, 1st edition.

[Lehoczky et al., 1989] Lehoczky, J., Sha, L., and Ding, Y. (1989). The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171.

[Liu and Layland, 1973] Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61.

[Lowney, 2006] Lowney, G. (2006). Why intel is designing multi-core processors. In Gibbons, P. B. and Vishkin, U., editors, *SPAA 2006: Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Cambridge, Massachusetts, USA, July 30 - August 2, 2006*, page 113. ACM.

[Min-Allah et al., 2012] Min-Allah, N., Khan, S. U., Ghani, N., Li, J., Wang, L., and Bouvry, P. (2012). A comparative study of rate monotonic schedulability tests. *J. Supercomput.*, 59(3):1419–1430.

[Sha et al., 1994] Sha, L., Rajkumar, R., and Sathaye, S. (1994). Generalized rate-monotonic scheduling theory: a framework for developing real-time systems. *Proceedings of the IEEE*, 82(1):68–82.

[Sodan et al., 2010] Sodan, A., Machina, J., Deshmeh, A., Macnaughton, K., and Esbaugh, B. (2010). Parallelism via multithreaded and multicore cpus. *Computer*, 43(3):24–32.

[Sousa et al., 2013] Sousa, P. B., Souto, P., Tovar, E., and Bletsas, K. (2013). The Carousel-EDF scheduling algorithm for multiprocessor systems. In *proc. of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'13)*, pages –, Taipei, Taiwan.

[Stankovic, 1988] Stankovic, J. A. (1988). Misconceptions about real-time computing - A serious problem for next-generation systems. *IEEE Computer*, 21(10):10–19.

[Stankovic et al., 1992] Stankovic, J. A. et al. (1992). Real-time computing. *Byte, pág*, pages 155–162.

Appendix A

The ReTAS framework installation and testing procedure

In this appendix are given instructions to install the framework and successfully make experiments with it. The guidelines provided here, are the same available in the framework's main web site. For each file needed, the reader is asked to go this page.

It is important to know, that before doing anything, the reader must have the proper Linux distribution available, the correct kernel version, and its corresponding PREEMPT-RT patch. In our experiments we have used the following:

- Ubuntu 14.04.2 LTS(ubuntu-14.04.2-desktop-amd64.iso)
- PREMMPT_RT patch (rt11)

A.1 Compilation and Installation

First, one needs to obtain the software required, these are either libraries or extra software dedicated to kernel development. To do so, please type in the terminal:

```
$ sudo apt-get install fakeroot
$ sudo apt-get install kernel-wedge
$ sudo apt-get install build-essential
$ sudo apt-get install makedumpfile
$ sudo apt-get install kernel-package
$ sudo apt-get install libncurses5-dev
```

Before compiling, the /etc/default/grub file must be changed to show the GRUB menu. For that purpose please type:

```
$ sudo gedit /etc/default/grub
```

and comment the GRUB_HIDDEN_TIMEOUT option with a # like in the example:

```
# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.

GRUB_DEFAULT=0
#GRUB_HIDDEN_TIMEOUT=0
GRUB_HIDDEN_TIMEOUT_QUIET=true
GRUB_TIMEOUT=10
GRUB_DISTRIBUTOR='lsb_release -i -s 2> /dev/null || echo Debian'
```

Next, the correct kernel source code must be downloaded from http://kernel.org/. Please type in the terminal:

```
$ wget http://www.kernel.org/pub/linux/kernel/v3.0/linux-3.10.15.tar.bz2
```

Extract tar (.tar.bz2) file:

```
$ tar -xjvf linux-3.10.15.tar.bz2
```

and rename directory from linux-3.10.15 to linux-3.10.15-rt11-retas-reserve

```
$ mv linux-3.10.15 linux-3.10.15-rt11-retas-reserve
```

download the linux-3.10.15-rt11-retas-reserve patch and change to the linux-3.10.15-rt11-retas-reserve directory:

```
$ cd linux-3.10.15-rt11-retas-reserve
```

apply the patch:

```
$ patch -p1 < ../linux-3.10.15-rt11-retas-reserve.patch
```

The first step to compile the kernel is to choose the compilation options. To make this process easier, a configuration file is provided with some modifications to make the compilation process quicker. Please, copy it to the linux-3.10.15-rt11-retas-reserve directory changing the file name to .config:

```
$ cp ../Downloads/linux-3.10.15-rt11-retas-reserve/config linux-3.10.15-rt11-
    retas-reserve/.config
```

When this is done, navigate to the modified kernel and type:

```
$ cd linux-3.10.15-rt11-retas-reserve
$ make menuconfig
```

Exit the menuconfig app and save the .config file. A shell script that automates the compilation process is provided but this is optional. Copy the script file to your Linux kernel parent folder and please type the following in the terminal:

```
$ cd ..
$ sudo ./kcompile.sh
```

Everything should be done at this moment, there should be a new option in the GRUB menu(check the advanced options section) that will allow one to boot to the modified kernel.

A.2 Testing

The first thing to do, is to download the user-space application. It allows one to feed it with a configuration file containing the scheduling information that will be used during the scheduling phase. It is of utmost importance that in order to use the ReTAS framework properly, one understands the contents of a valid file of this nature, for that purpose we provide an explanation in how to construct one in the website. Uncompress the archive:

```
$ unzip linux-3.10.15-rt11-retas-reserve.zip
```

Change to the application directory, and compile the application as follows:

```
$ cd linux-3.10.15-rt11-retas-reserve/tasks
$ ./compile.sh
$ cd ..
```

To perform an experiment all is needed to do is to execute a simple script file(run_exp) that is present in the extracted folder. Some configuration files are provided in the archive. They are named config# (ex: config1, config2, etc...). File permissions must be changed. There is also another script file(log.sh), that performs some logging activities, still, the run_exp script automatically uses it, so there is no need to execute it directly.

```
$ chmod 755 log.sh
$ chmod 755 run_exp.sh
$ ./sudo run_exp.sh
```

This finishes the overall procedure, results should start to show in the terminal. Again, further details in how to interpret them, are provided in the web page.