Université Pierre & Marie Curie
TELECOM-ParisTech

Master Informatique

Systèmes et Applications Répartis

Design & Development of Safety-Critical Systems for Multi-core Architectures

Supervisor:
Dr. Thomas Robert

Author:
Roberto Medina

Referent professor:
Dr. Julien Sopena

September 10, 2015

Contents

List of Abbreviations

AADL	Architecture Analysis and Design Language
API	Application Programming Interface
CPU	Central Process Unit
EDF	Earliest Deadline First
F&I/D	Fetch-And-Increment/Decrement
HLP	Highest Locker Protocol
IMA	Integrated Modular Avionics
IPC	Inter-Process Communication
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
P-EDF	Partitioned Earliest Deadline First
PCP	Priority Ceiling Protocol
Pfair	Performance Fair
PIP	Priority Inheritance Protocol
RMS	Rate-Monotonic Scheduling
RMW	Read-Modify and Write
RR	Round-Robin
RTOS	Real-Time Operating System
SLR	Systematic Literature Review
TAS	Test-And-Set
WCET	Worst-Case Execution Time

List of Figures

PART I

Introduction

1 Internship context

TELECOM-ParisTech (ex École Nationale Supérieure de Télécommunications) is an engineering school founded in 1878 under the name École Supérieure de Télégraphie. Nowadays the school is member of the Paris Institute of Technology (ParisTech) and Institut Mines-Télécom. Research laboratories of TELECOM-ParisTech are attached to the Centre National de la Recherche Scientifique (CNRS).

This internship takes place in the Systems, Software, Services team (S3) of the Network and Computer Science department (INFRES). The S3 team works on the architecture, conception, modelization, development, validation and evaluation of systems, software and services, taking into account different constraints like time, limited resources, mobility, scalability, interoperability, robustness and reliability. The main domains treated by its team members are:

- Real-time systems.
- Embedded systems.
- Distributed computing.

This internship is supervised by Dr. Thomas Robert, Assistant Professor at TELECOM-ParisTech.

2 Motivation and objective

The main focus of this internship is to evaluate how multi-core processors question the integrity of safety-critical systems, and to determine requirements to have an efficient safety-critical system in a multi-core architecture.

Safety-critical systems are in charge of controlling, monitoring and reacting to real physical events, *e.g.* thermal output in nuclear plants, aircraft positioning for autopilot, *etc.* A high level of confidence is given to safety-critical software behavior. This can be achieved thanks to high integrity software implementation, predictable execution and fault tolerance mechanisms. Systems and processes in the safety-critical domain have to comply with strict time requirements. To assure predictable execution, real-time computing is used in safety-critical systems. Real-time computing is subject to time constraints, *i.e.* the services of an application have to be delivered before a deadline. However, safety related applications also require fault tolerance mechanisms, the presence of faults cannot be overseen and must not impact the reliability of these systems. This means that deterministic execution coupled with robust fault tolerant platforms should be used in this domain.

Safety-critical software tend to be deployed on operating systems and middlewares providing these key features. Yet, most mature solutions featuring time determinism and justified high confidence implementations are available in operating systems developed for uniprocessors. The adoption of multi-core platform is key for the domain, this architecture will most likely replace single-core processor chips in the upcoming

years. Nevertheless, safety-critical operating systems lack from satisfactory support for multi-core architectures nowadays.

Recently, a push to cope with *off-the-shelf* multi-core processors is a tendency in safety-critical systems. In fact most architectures supported by these systems tend to have a multi-core version of their products, and most of the time, the multi-core version is a juxtaposition of single-core processors, what leads developers to believe that obtaining a multi-core version of their safety-critical system might not be very difficult because the modular code existent can be reused. However, this type of hardware affects the high confidence given to software: high integrity is not assured anymore and safety is compromised (*e.g.* fault propagation from one core to another).

In the S3 team, its members work actively around the different problematics of safety-critical systems. For example the POK operating system is the result of the team's research around high integrity operating systems. New scheduling policies integrating reliability, efficiency and performance are other examples of the teams collaboration to the domain. This internship is a clear example of the team interest in the new challenges of real-time systems. The internship aims to leverage obstacles for capitalizing legacy uniprocessor portions of code and identify key services that would help the adoption of multi-core processors, increasing their attractiveness.

This document is decomposed as follows: the first part gives an extensive literature review around safety-critical systems and multi-core architectures, the major difficulties with the new hardware are explained, how they differ and become more complex when it comes to executing on multi-core platforms. We also describe the current works proposed by the real-time community, what can be expected of the multi-core architecture, how it is beneficial for real-time applications, and what are the current methods to adopt this architecture in safety-critical systems. The second chapter explains the main problem we try to tackle throughout this document: what is needed to obtain parallelism at a kernel level, in particular how can system calls run in different execution contexts without compromising the system's integrity. To formalize the requirements of the new multi-core implementation, a specification language is proposed in the next chapter, putting in evidence dependencies between the newly introduced execution contexts, the services offered by the operating system and the internal data structures, who play a major role in representing the state of the system. We also explain how critical sections are handled, having potential parallelism can clearly lead to data races and the kernel should be modified to handle them in the best way possible (*i.e.* only locking when necessary, minimize blocking time, not introduce deadlocks, ...). Modification patterns are the main contribution of this work, and a chapter describes how they were obtained and how they can be implemented with adapted tools. Assisting engineers in the porting phase is the goal of those patterns. Most adaptations are done manually and might need a new verification process to determine if the multi-core version complies with real-time standards, this is one of the main motivations behind the proposed patterns: having some sort of verified automation can ease the verification process. We conclude by showing our results for two different real-time systems studied, the POK and seL4 micro kernels.

3 Focus of the literature review

The literature review regarding the main subject of this internship was an important part of the work. The review has for first objective to identify the state of contributions in terms of safety-critical operating systems features for real-time execution platforms. The second objective consists in identifying the specific features and obstacles to extend opportunities for capitalization of safety-critical uniprocessor mechanisms.

The review is also necessary to assist the development process that will come in future work. Its

quality is very important as well, many review process are done "by hand" and some relevant work can be easily overseen. To address this problem, the community tends to use more automated methods, helping reviewers with the quality of their findings. It was decided that one of this automated approaches, as opposed to just getting references from my supervisor, would be used during the research period. The automated method, called *Systematic Literature Review* was used to present this bibliographic study.

3.1 Description of the Systematic Literature Review

In order to establish the literature related to safety-critical operating systems in multi-core architectures , I adapted a research method called *Systematic Literature Review* (SLR), mostly used in health-care.

This review method is explained in several articles, in particular Khan *et al.* [25] describe the basic steps to make this kind of research.

- First of all, questions for the review have to be framed. A free-form question is the start point, from there, the problematic needs to be structured: what are the main problems in the question, what was done to solve one of these problems, what are the outcomes of the proposed solutions. Well formulated questions have to be precise, clear and unambiguous.
- The second step consists in searching for relevant work. Investigation needs to be extensive: multiple resources should be consulted with no restriction on the language. A selection criteria for the sources is specified *a priori*. A record with the criterion of inclusion/exclusion of a reference should be kept as well.
- Third, once studies are chosen, refined quality assessments will be used to filter the first findings. This can be done using assessment tables and checking just the abstracts of the references to see if the topics really correspond to the questions formulated at the beginning of this procedure.
- Fourth step consists in summarizing the findings. The data synthesis aims to give a perspective around the problems that were stated in step one.
- Fifth, a discussion around the findings needs to be done. Interpreting results and drawing conclusions will ideally give the answer to the question in step one. Health-care problematics are mostly yes/no questions, but for this work the final implementation presented in the last part should be seen as the answers to the questions of my SLR implementation.

We considered respecting the recommendations of this method and using its tools to carry out the literature review. It was the opportunity to evaluate this new tendency and check if it could be reliable for establishing a review process.

3.2 Review implementation

The main problem and entry point to begin with the SLR is:

Why is it difficult to use multi-core processors in safety-critical systems?

To understand this problem we need to focus on the key points of safety-critical systems: the design of operating systems used in the safety-critical domain and the theory around real-time programming used in these systems. Following what is described in [25], the first question that I framed had to do with real-time tasks and scheduling policies for multi-core platforms:

How can real-time tasks be scheduled in multi-core platforms?

In fact, the scheduler could be seen as the principal component of an operating system. Many scheduling algorithms are present in the literature, it would be interesting to see if a common tendency is present or if a consensus is made about the best way to schedule real-time tasks in various cores. When we consider scheduling in multi-core processors a big problem that needs to be considered is *synchronization* between tasks. Race conditions can occur when it comes to multi-threading and the hypothesis of tasks not sharing resources or not communicating is highly restricting. Therefore, this problem is relevant as well:

How can real-time tasks synchronize in multi-core architectures?

These two problematics were the main axes to conduct the SLR. The structured questions are as follow:

- *Safety-critical systems*: safety-critical operating systems executed on multi-core platforms.
- *Real-time tasks*: processes with real-time properties running concurrently.
- *Scheduling algorithms*: modified or new algorithms to schedule real-time tasks.
- *Synchronization methods*: the best way to synchronize and how should it be done in safety-critical systems.

To identify relevant work I used an adapted software called SLR Tool [9]. This search engine aims to make SLR easier without compromising the strengths of the procedure. The tool can create a project providing the different functionalities of SLR: users can join a review procedure, research questions can be specified, thorough search of the literature is done using several sources, criteria for inclusion and exclusion can be included, quality assessment can be used and other visual reports are available as well. To use this software we need a set of keywords that will generate a list of sources, the query can then be optimized by including more sets. In this work, the most used keywords were: **multi-core, scheduling, real-time, safety-critical** and **synchronization**. Sources are presented to the user after the query goes throw several data bases, by default, the result contains ten references but the list can grow until the tool does not generate more results or until the user estimates that the output is not really relevant to the question anymore. The reviewer then selects titles that seem to be relevant for the project and the tool will create a list of references with all the sources. Once a first selection is done, the project will be created and saved into the SLR Tool database. This list of findings can then be exported to a BibTeX file. After generating this file, it was imported to JabRef [4], a software that allowed me to follow the third step of the systematic review: quality assessment. If a source was interesting enough it was marked, if it was not it would stay in the reference file because SLR requires to keep a record of the consulted sources. Once the final selection was done, the `Comment` annotation was used to take the fourth step in SLR: summarize the findings. The final bibliography was exported to a `.bib` file again and the final result is what you can see as the **References** for this work. The fifth step of the SLR, the discussion of the results, is presented in the second chapter of this work. Hundreds of articles were found by the tool, twenty of them had hard real-time as their main topic, but at the end only eleven were selected mainly because the others were previous works of the final batch, were more related to hardware or their tasks models were not applicable to safety-critical systems.

Some limits were visible while performing this review method. Adapted tools to achieve SLR are hard to find: many papers talk about the subject but when it comes to finding the software associated with it, binaries do not work, links are dead, web pages have not been updated in years or simply there is no trace of the program. Even SLR Tool had several issues: only one search through the data bases could be done at a time, the search engine was broken and gave no output at some point, PDFs where not found by the tool and manual search had to be done for all sources. The same keyword sets were used using Google Scholar to compare the results. The same sources were found but SLR Tool performed better and found more relevant results. To overcome these limitations other references were added, my background knowledge and my supervisor's advice allowed me to find articles that are relevant to the subject as well. We are aware

that the subject is vast and this study may not treat all the aspects related to safety-critical software in multi-core architectures. However, the contributions for this internship are related to the findings that are presented in this work.

3.3 Organization of the bibliographic study

In order to understand why adapting safety-critical systems to multi-core processors is hard, the literature review is decomposed in two parts:

The first part explains how safety-critical systems assure that their execution will be safe and how robustness for the system is assured. Predicting and controlling execution is possible using real-time computing and safety-critical operating systems. This type of programming has to respect different constraints and it becomes more complex when programs have dependencies (*e.g.* sharing resources, communicating, using devices, accessing to memory). A section explaining how real-time operating systems are implemented demonstrates that their development is also an essential step to assure a predictable safe execution. The chief design goal is not to have a high throughput, but rather guarantee a hard real-time performance category, *i.e.* processes and operating system components should meet a deadline deterministically. Errors and faults are part of the real world and will most certainly happen, even if the system was carefully designed. Handling non predicted behavior is vital for safety-critical systems. Different mechanisms are deployed to increase their reliability. Real-time software development is confronted to specific standards to assure a certain safety level. In depth testing leads to a certification for the system's software allowing it to be deployed into the hardware. All these aspects show that adopting a new architecture is not straightforward and many aspects need to be considered during the porting phase.

The second part discusses the current state of multi-core platforms and safety-critical systems. The main objective being to efficiently use multi-core architectures. The different difficulties with this type of processors are presented. These challenges directly affect the main axes of this work: scheduling and synchronization. Achieving true parallelism with real-time tasks has a direct impact in real-time computing, new task models need to be designed in order to predict safe execution and allow concurrency between tasks. Adapting known scheduling policies and capitalizing legacy code from single-core approaches was the first step taken to support multi-core platforms. Algorithms were modified to check for schedulability in multiple CPUs, locking mechanisms were analyzed to check if waiting time for a resource could be bounded. Nonetheless, adapting known methods might not be enough to support this type of architecture, therefore innovation around multi-core architectures is also very important. New algorithms are introduced to take advantage and overcome difficulties with these platforms. New hardware is being developed, new scheduling policies are proposed, innovative synchronization methods are analyzed. Another important aspect are standards, norms and certifications, that should be validated when adopting this new architecture. This second part could be seen as the discussion of the SLR implementation.

PART II

Literature review

Software for safety related applications is complicated, hardware components like execution units, memory, disks, captors, cameras, *etc*, are shared between the applications. Hence, the software has to be deployed in a operating system controlling the shared resources and providing common services for safety related programs. A Real-Time Operating System (RTOS) offers the usual components of any operating system, however, since RTOS are executing critical programs, it is their responsibility to guarantee that applications will deliver their services *on time* and *without errors*. Consequently, using a RTOS is an essential step to assure safety.

In safety-critical systems, program's correctness not only depends upon its functional correctness, but it must also guarantee response within strict time constraints, *i.e.* its services should be delivered before a deadline. Not reacting in a certain interval of time would cause great loss in some manner, *e.g.* damaging the surroundings, great financial lost or threatening human lives. Real-time programming is used in safety-critical systems to predict safe execution. The theory of real-time computing allows developers to determine that their software design meets time requirements. Real-time scheduling policies will be an obligatory feature of a RTOS.

Abnormal behavior should not interfere with the safe execution of the system. Mechanisms to detect errors and protecting the safety-critical system are integrated as services. Each RTOS implementation has very specific ways to detect and handle non-predicted events, this depends on the environment that the system will be confronted to and the maturity of the operating system. Standards and norms will define very precisely the services that need to be provided by the system in order to assure that its implementation is safe. The development process will be guided by this documentation. With regard to be certified with a safety integrity level, the system will also be put into a set of tests, measures and procedures.

1 Predictable execution of applications

To develop software in safety-critical systems, real-time scheduling policies should be used. A scheduling analysis allows developers to assure that execution of a given set of real-time programs will be correct, *i.e.* deadlines will be respected. Nevertheless, the mathematical formulas have to integrate more variables when the services delivered by programs depend on shared resource, communications, devices, other programs' services, *etc*. Or in the case of multi-core architectures, completely redefine a new scheduling analysis. This process should verify that the system can assure a safe execution. However, schedulability tests need to take into consideration the overhead of the operating systems' operations. This is true for synchronization mechanisms that will be deployed in the RTOS and system calls that are not executed instantly. Real-time properties need to be assured for the components of the operating system as well. The implementation of the safety-critical operating system is therefore critical.

1.1 Scheduling policies and schedulability tests

Using a task model is the basis to assure that a set of processes can be executed in a processor. Liu and Layland inspired the main task models used in hard-real time [32] for this purpose. This work does

not present the latest advances of scheduling policies, considering their implementation might be difficult or even impossible to integrate in a safety-critical operating system.

A) Uniprocessor independent policies

For n tasks, let $\tau = \langle \tau_1, ..., \tau_n \rangle$ be the real-time task set, τ_i the i-th task, T_i its period, C_i its computation time, S_i its activation date, D_i its deadline and $U_i = C_i/T_i$ its processor usage. In particular the global CPU usage is given by $U = \sum_{i=0}^{n} U_i$. Worst-Case Execution Time (WCET) needs to be known for all the processes running on the critical system. Particularly, $WCET_i \leq D_i$ must be true at all times.

With this model, schedulability for a task set can be assured using scheduling analysis. Scheduling real-time tasks can be done with a static approach, *i.e.* using a scheduling table. Nonetheless, this approach is not really flexible, integrating new tasks to the set is difficult and variations to the tasks' properties might lead to an incorrect table. The most common method for scheduling is using well-known algorithms like RMS, that has a sufficient condition to check schedulability of a task set in one processor:

$$\sum_{i=0}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \leq \lim_{n \to \infty} n(2^{\frac{1}{n}} - 1) \approx 69.3\%. \tag{1}$$

Priorities are given to all the tasks depending on the scheduling approach that is taken. These priorities are either fixed (off-line scheduling) or change during execution (on-line scheduling). On-line schedulers tend to improve CPU usage. EDF is one scheduling policy that can achieve 100% of CPU usage and still guarantee tasks deadlines by giving the highest priority to the task with the closest deadline, its sufficient and necessary condition is given by:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1. \tag{2}$$

B) Uniprocessor policies, dependent tasks with shared resources

However, the task model becomes more complex when processes share resources and critical sections have to be protected. B_i, the longest possible waiting time a tasks is blocked by another process, needs to be taken into account when testing schedulability. For example, in RMS the sufficient condition becomes:

$$\forall i, 1 \leq i \leq n, \sum_{j=1}^{j=i} \frac{C_j}{T_j} + \frac{B_i}{T_i} \leq n(2^{\frac{1}{n}} - 1). \tag{3}$$

Different protocols [33], *e.g.* Priority Inheritance Protocol (PIP), Priority Ceiling Protocol (PCP), Highest Locker Protocol (HLP), avoid deadlocks, priority inversions and try to minimize B_i. These protocols will affect a priority to the locking structures protecting the critical section. For the sake of simplicity only the semaphore will be presented, other locking mechanisms will have the same procedure. Let s be a semaphore protecting a certain critical section and $c(s)$ its priority. Tasks will be divided into jobs: $\tau_i = \{j_{i,1}, j_{i,2}, ..., j_{i,l}\}$ where l is the number of jobs for a the task i and $c(j_{i,m})$ the priority of the job m. Jobs can use shared resources, therefore they are in a critical sections. A job can be preempted by another task's job if the priority of this task is higher. To avoid deadlock, priority inversions and to minimize blocking time caused by jobs that cannot access to critical sections, dynamic priorities are in place. The job will inherit either the priority of the highest task that needs the resource: $c(j_{i,m}) := c_t(s)$ (PIP), a higher priority $c(j_{i,m}) := c_h(s)$ (HLP) or a defined ceiling priority $c(j_{i,m}) := c_c(s)$ (PCP). These protocols combined with a scheduling policy are proven to respect hard real-time properties. For example, as it is demonstrated in [16], schedulability

with EDF and PCP lead to a new sufficient and necessary condition integrating waiting time for a task:

$$\sum_{i=1}^{n} \frac{C_i + B_i}{T_i} \leq 1. \tag{4}$$

The various events used by the scheduling policies will have to be implemented in the RTOS, *e.g.* deadlines will be specified beforehand for tasks using RMS, dynamic priorities will be calculated by the scheduler for EDF, *etc.* Thus, selecting an adapted scheduling policy does not only depend on the performance of the algorithm, but also on the possibility of implementing the policy in the operating system's scheduler and on the data structures that are used by the scheduler. The overhead of the scheduler has to be taken into consideration as well, for preemptive policies the cost of a context switch is important for estimating the time of the operating systems' operations. The WCET has to integrate the maximum time the scheduler takes to select the next program that will run on the processor.

C) Multiprocessor policies

Two perspectives are used to implement real-time scheduling in multi-core processors: a **partitioned** point of view, where tasks are assigned to a specific core and cannot migrate to another CPU, and a **global** standpoint where the scheduler allows migration of tasks between CPUs and is aware of the processor's workload. This gave birth to a first generation of modified schedulers [18]: Partitioned-EDF (P-EDF), Global-EDF (G-EDF), Partitioned-Rate Monotonic Scheduling (P-RMS) and so on.

Planned execution is still assured using the partitioned approach, the scheduling analysis used by the single-core version of the algorithms can be applied to all the CPUs, *i.e.* for each CPU the sufficient (and necessary) condition will be calculated. Although when the number of tasks increases developers are confronted with a classical bin-packing problem. Heuristics like First Fit, Next Fit or Best Fit are used to confront this problem. To calculate schedulability using P-EDF for multi-core processors we can adapt the task model as follows:

Let m be the number of CPUs in the processor, $\pi = \{\pi_1, ..., \pi_m\}$ the multi-core processor and n_i the number of tasks placed in processor π_i. The necessary and sufficient condition for P-EDF becomes:

$$\forall \pi_i \mid \pi_i \in \pi, \sum_{j=1}^{n_i} \frac{C_{i,j} + B_{i,j}}{T_{i,j}} \leq 1. \tag{5}$$

Placing tasks is the principal challenge for this standpoint, different constraints about task precedence, communication and shared resources need to be taken into consideration in order to minimize blocking time and to assure that the CPU's task set is schedulable. The processing power of the processor will be affected if the tasks spend most of their time in idle mode waiting for a given resource.

Modified scheduling policies that were designed for single-core architectures might not perform well on multi-core architectures. In fact, new scheduling policies are developed to gain in performance and fairness. One example is Performance Fair (Pfair), an optimal scheduling policy for multiple processors running at the same speed. This algorithm distributes tasks among processors assuring a fair distribution. Tasks are decomposed into sub-tasks (one sub-task for time unit) and priorities $p(\tau_i, j)$ and wake times $w(\tau_i, j)$ are calculated dynamically:

$$p(\tau_i, j) = \left\lceil \frac{j}{C_i/T_i} \right\rceil. \tag{6}$$

$$w(\tau_i, j) = \left\lceil \frac{j-1}{C_i/T_i} \right\rceil. \tag{7}$$

Where j is the sub-task number. For Pfair, deadlines are satisfied and it is optimal for multi-core processors. It also has the advantage of scheduling real-time and non-real-time tasks. The problem with this type of algorithm is the overhead introduced by the scheduler that will implement them. The scheduler will have to be awaken every time unit. If the scheduler determines that a new task has to run, then a context switch will take place, increasing even more the overhead of the algorithm.

Due to their predictability and the well established theory, these scheduling policies are included in operating systems used in safety-critical components. Nonetheless, implementing these scheduling policies is not free: data structures need to be introduced, locking mechanisms to share resources need to be implemented and the performance of the scheduler and system calls will have an impact on the WCET of the application as well. Schedulability tests need to take these costs into account to really predict if a task set can be schedulable for a given processor.

1.2 Real-time synchronization mechanisms

Implementing shared objects in an operating system can be done in two different manners: using locking mechanisms (*e.g.* semaphores, mutexes), or using non-blocking operations (*e.g.* wait-free or lock-free implementation of shared objects). For hard real-time systems, correctness of the scheduling policies presented in the previous section, depend on blocking time for tasks sharing resources and communicating. RTOS usually implement one of the following approaches: data flow communications avoiding blocking mechanisms or shared objects with bounded synchronization time.

Blocking implementations are the most common way to protect a given resource. When a task tries to get a resource, it will have to gain access through a locking object, *e.g.* semaphore, mutex, spinlock. This can potentially change the state of the thread and preempt it with a more important task. The spinlock is particular because threads wait actively until the lock becomes available. This is rather useful if the waiting time is short, the task will not have to be reschedule avoiding the overcost of the scheduling algorithm and the context switch. Using spinlocks is widely used in general public operating systems like Linux. When it comes to RTOS, locks have typically a timeout. If a task has been blocked for too long, then another thread will have access to the resource after a given time. The unlocking policy can be specified as well, if threads are waiting for a given resource, knowing which thread will have access to the resource next is important for hard real-time predictability. Dynamic priority protocols (*e.g.* PCP, HLP, PIP) will be a feature of locking mechanisms too.

Wait-freedom is the strongest non-blocking guarantee of progress. It has the interesting property that every operation has a bound on the number of steps the algorithm will take before the operation completes, guaranteeing system-wide throughput and starvation-freedom. If the number of steps is known then this algorithm can be analyzed to calculate the maximum time a task has to wait for a shared resource, making WCET calculable. Implementing a wait-free algorithm for a shared resource is possible, however it is complex and depends on the shared object's nature mostly.

Lock-freedom guarantees only system-wide throughput. A lock-free algorithm satisfies that when threads have run sufficiently long at least one of the threads makes progress. In general, a lock-free algorithm can run in four phases: completing one's own operation, assisting an obstructing operation, aborting an obstructing operation, and waiting. Correct concurrent assistance is typically the most complex part of a lock-free algorithm, and often very costly to execute.

Non-blocking algorithms can be very interesting in terms of performance and scalability. Nonetheless their implementation is difficult and the underlaying mechanisms that are used to implement them might not be present in all architectures or RTOS.

A study to determine the best way to synchronize in multi-threaded real-time programming is presented in [15]. Busy-waiting with spin-locks is widely used in operating systems to protect critical sections but other options like lock-free execution, wait-free execution and suspension-based locking can perform better than spin-based locking. The results of this study showed that non-blocking algorithms perform better for small critical sections, and wait-free and spin-based implementations are preferable for large and complex objects. An important observation made by Brandergurg *et al.* is that suspended-based locking should be avoided under partitioned scheduling policies. This result will be of great importance for the porting phase of the ARINC 653 operating system to a multi-core processor.

Time predictability for shared resources in addition to scheduling policies makes the application execution more predictable. Nevertheless, correctness of scheduling policies and blocking mechanisms assume that there is no true parallelism at the operating system level. In uniprocessor platforms, when a program is executing a kernel code it will not be preempted, system calls are usually *atomic* and preemption occurs afterwards. This atomic execution will have to remain true in a multi-core processor where system calls can happen simultaneously.

1.3 High integrity implementation

In real-time operating systems, the task models presented in the previous section are used to implement internal data structures (*e.g.* threads, processes, semaphores, mutexes), schedulers and synchronization mechanisms. It is the system responsibility to guarantee good behavior during execution and implementing a task model is not enough. The aspects related to the underlaying hardware have to be taken into consideration as well, *e.g.* the Instruction-Set Architecture (ISA), sharing devices, communicating with exterior components. The hardware also needs to guarantee that the operating system implementation will be real-time capable. RTOS's implementation is therefore critical as well. Developers use different approaches to assure that their operating system is error-free.

For example seL4 [12], an L4 microkernel developed by NICTA, has its implementation formally proven and is correct against its specification [26]. This implies that it is free of implementation bugs like deadlocks, livelocks, buffer overflows, arithmetic exception or use of uninitialized variables. The seL4 is intended to be a general-purpose micro-kernel, although the main target for this operating system are embedded systems with security or reliability requirements. The seL4 follows a very specific design process, it uses a Haskell prototype of the system, manually rewritten in C for the implementation and automatically translated into the theorem proving tool. The internal data structures are presented as objects: threads, capability nodes, IPC endpoints, virtual address spaces and interrupts. High-level kernel services use these kernel objects. This composition of data structures is used for the verification of the operating system. An effort to prove that kernel operations are bounded on their WCET is also a feature of this system, making the seL4 kernel a candidate for hard real-time systems.

The seL4 microkernel has an experimental support for multi-core architectures. The main feature of this system is its kernel, formally proven to be correct. In [36], the two possible approaches that were considered to support multiple cores for the x86 architecture are presented. The first one consisted in accessing a CPU with a Round Robin (RR) policy, this implies that only one core runs at a time. This approach is not acceptable at all, the interest of having tasks running in parallel is completely lost. The seL4 developers decided to take a *multikernel* approach for their operating system: separate instances of uniprocessor seL4 run in each CPU. To support dynamic resource allocation, one master CPU can access all the resources after the system has been initialized and affect them to the desired instance. Sharing resources in this experimental branch of the seL4 operating system is not possible, once the resource is affected, it cannot be transferred to another instance. This approach allows the proofs made for the seL4 to be conserved when using multiple

CPUs. Only the bootstrap process is affected and this part is not covered by the formal verification of the system, the hypothesis that the system boots correctly is made.

Nevertheless, formal verification is expensive in resources and time. For these reasons other methods to ensure real-time behavior and to prevent error introduction are used on the majority of operating systems.

Model design using Architecture Analysis & Design Language (AADL) is used in POK [6]. This partitioned microkernel provides partitions with isolation in time and space and gives them a priority, *i.e.* high level partitions are more important in terms of safety. Each partition can use real-time scheduling algorithms, shared resources and communication mechanisms through a specific runtime called *libpok*. The different services delivered by this operating system makes it ARINC 653-compliant. This standard imposes that different types of services can be used inside a partition (intra-partition) or between partitions (inter-partition). For communication for example, the protocols included at the kernel level are: blackboards and buffers for intra-partition communication, queuing and sampling ports for inter-partition communication. Configuration for the kernel is done automatically depending on the system design, *e.g.* if partitions do not use share resources, no locking objects will be created. This configuration is very flexible and the fine-grained policy can reduce the set of functionalities to a minimum and minimize memory footprint as well. This flexibility also assures that no dead-code can be triggered at execution. Verification of the partitions implementation is done during the compilation process and code is automatically generated to avoid programming errors that might be introduced by developers. The objects that will be used by the partitions are declared statically, everything that needs to be present in memory will be there. This approach assures that no memory leaks are introduced. To make POK a high integrity RTOS, the operating system relies on different tools: Ocarina [5] for model-checking and code generation and Cheddar [1] for real-time scheduling analysis. Recently, RAMSES [7] can perform the model transformation and code generation for POK. This complex tool-chain verifies that specifications are met. The following figure shows an overall view of the development process for POK:

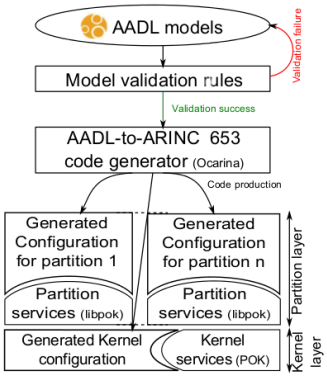

Figure 1: POK development process

Despite the methods that are adopted to ensure correct execution, errors, faults and interruptions will still occur. If a device driver starts to send corrupted data to the operating system, it has to be able to detect the fault and keep execution in a safe state. Programmers do not stop at the implementation of the RTOS for assuring reliability, a fault model is included in their systems to advert users of an unpredicted and/or abnormal behavior. Additional mechanisms to protect the RTOS are therefore essential.

2 Robust execution of operating systems

Handling errors in real-time systems is vital. As a matter of fact, failure detection and error handling are present in all kinds of operating systems. Even in the seL4 microkernel, the Application Programming Interface (API) defines error codes, protecting the operating system from incorrect utilization. In safety-critical applications, detecting and treating an unpredicted event should not impact the schedulability of the system. To handle the different types of abnormal behavior, several services are introduced to guarantee real-time properties in the presence of faults, *e.g.* isolation in time and space, drivers executed in user-space, fault handling routines, hooks, *etc.* In addition to the capability of detecting and treating faults, standards and norms define very specifically how developers should program safety-critical operating systems, what services should be included, what tests should be passed, what hardware should be used. The combination of these measures assure that the safety-critical system is as safe as it could be.

2.1 Detecting timing failures on scheduling policies

Characterizing an abnormal behavior is very important for RTOS. The taxonomy for this type of conduct is well defined in the literature [13]. Fault-tolerant scheduling policies are hard to implement, thus safety-critical systems either include protocols to bring the system back to a safe state or assure that the system will not arrive to a non-functional state.

When it comes to the task model, for a task i, its $WCET_i$ is affected with the time that the operating systems takes to handle an error. Let Δ_i be the time spent to get the system back to a safe state: $WCET_i := WCET_{i,noerror} + \Delta_i$. Therefore, new considerations should be taken into account for dimensioning the safety-critical system, tasks deadlines will be directly affected when a fault model is introduced. The works of [17] showed that efficiency of the error detection mechanism is also an important consideration for implementing the system. Achieving coverage of failures comes with a CPU and memory costs, making WCET even more complex to estimate and bigger in value.

There are two ways for a system to recover to a safe state: rollback and roll-forward protocols. Rollback recovery is a simpler mechanisms to implement and deals with transient faults [35], the operating system can store a state where the execution was correct, *i.e.* deadlines were satisfied, and come back to it when it detects an abnormal behavior. This concept relies on the fact that occurrence of faults is rare, hence the same error is very unlikely to happen again. However, if the state that was recorded is already faulty, then the rollback might not have any effect or will have a greater scope. On the other hand, mechanisms of forward recovery can be implemented in embedded real-time systems [37] and are more interesting because tasks need to complete their execution before their deadlines. This mechanism uses distributed computing by making checkpoints of spare processors. For example, it could be interesting to apply roll-forward recovery in a multi-core processor in order to increase reliability for the RTOS.

For an OSEK/VDX compliant system, a standard used in the automotive industry, interrupt processing is divided into two categories [10] (p. 25, 33). The first category corresponds to interruptions with least overhead that do not require an operating system service. The second category provides an Interrupt Service Routine (ISR) that prepares a run-time environment for a dedicated user routine. Interruptions are threated *atomically* and rescheduling takes place after termination of the ISR. These aperiodic events can also be affected with a priority level, allowing urgent interruptions to be treated immediately or delayed until the system can treat them. Trampoline [11], a RTOS compliant with the OSEK/VDX specification, features the capability to detect an overrun in a program. This is possible thanks to the compilation process of the system: an `.oil` file is used to declare the task set that will run in the RTOS. If execution takes more time than the deadline declared for a task, an exception is risen. These different services delivered by

the OSEK/VDX operating system allow developers to detect errors and to treat them in a way that allows the system to be reliable even in the presence of faults.

For commercial RTOS, fault tolerance mechanisms is what differentiates them from more academic RTOS. For example for VxWorks ® the error handling framework will be verified by an external organization to certify its functionality.

2.2 Dedicated services for fault tolerance

Standards and norms define what level of fault tolerance needs to be present in the application. For airborne systems for example, we have the DO-178C certification. This standard defines a *Software Level*, in the interest of examining the effects of a failure condition in the system. Failure conditions are categorized by their effects on the aircraft, crew and passengers. There is a number of objectives to be satisfied in order to certify a component with a software level. The A level is the highest one, where 71 objectives need to be passed, if a failure condition occurs at this level the consequences can be catastrophic. The E level is the lowest one where no objectives need to be passed, it has no impact on the safety of the aircraft or crew. This certification procedures aims to assure that the software deployed in the airship will have a very low probability of failure. The proportion of A-level code that will be present in the airborne system will have to be minimal. If the code is too big then certification and tests will take longer and be more expensive.

In addition to the DO-178C certification, the ARINC 653 standard imposes a Health Monitor. This operating system component monitors hardware, operating system and application faults and failures. Its main task is to isolate faults and prevent failure propagation. The Health Monitor can restart a partition when it detects an application fault.

Back to POK, in [20] the services that the operating system provides are presented. A fault handling mechanism at the kernel level is present, it basically detects when a fault or an exception is risen and routes the call to the appropriate handler. Since POK is an ARINC 653-compliant operating system, a Health Monitor can be declared by the user. Error handling can be specified for each partition through a recovery subprogram. Following the principle of microkernel design, device drivers are executed in user-space: a specific partition is created just for the driver. This way if the device fails or the driver encounters an error, the Health Monitor can reboot or shutdown the partition, avoiding the impact from a device error and ensuring data isolation (*i.e.* higher level partitions are protected against modification on the driver from another lower level partition). POK also provides security analysis rules, the architecture is tested against a security policy. Depending on the classification level and the data the partition produces or receives, it may be compliant with a specific security policy. This policy defines which operations are allowed to be executed by a partition.

Another example is the AUTOSAR standard [8], providing memory partition for software applications. This mechanism avoids any data corruption between applications. Partitions are used as fault containment regions, they can be terminated or restarted during run-time as a result of a detected error.

Handling errors, faults and interruptions has a direct impact on the capabilities of the safety-critical system. New services need to be included to assure that real-time properties are conserved. Safety standards and certifications also take part in assuring that the system is functionally safe.

Safety-critical operating systems are a key component in predicting execution of applications for real-time programming. RTOS differ from general public operating system due to their rich components: a real-time scheduler, synchronization mechanisms with predictable behavior, error handling mechanisms and recovery protocols. The confidence in these components is very important. Their quality, their predictability and their correctness are guaranteed in different manners, *e.g.* formal proving for seL4, model design verification for POK. An especial care in implementing these systems is also an important step towards safety. Adopting a new architecture has to assure that all of these components will conserve their good properties in order to guarantee safety. This is clearly not the case for multi-core architectures.

The number of requirements that need to be satisfied in this kind of systems makes their implementation a complicated task. These requirements become more complex when the architecture considered contains many execution units and shared hardware components. Scheduling policies have to be modified to take advantage of the processing power of the CPUs. The maturity in this field of real-time systems has achieved a satisfactory stage, many scheduling policies are present in the literature and can exploit the parallelism offered by multi-core architectures. However this is clearly not the case for RTOS services and components.

The most mature and complete safety-critical operating systems are implemented with a single-core perspective. The adoption of multi-core platforms highly relies on mechanisms that are already present in the uniprocessor version of the RTOS. Nonetheless, using the results achieved with uniprocessors might not give a satisfactory result in terms of schedulability, WCET and safety. The integrity of the safety-critical operating system components is compromised as well. What was achieved for RTOS in single-core processors needs to be updated and upgraded according to the properties of multi-core platforms.

3 Obstacles to the multi-core adoption

Operating systems used now-a-days by the general public were adapted to multi-core architectures. This new type of processor has been a real challenge for developers and even today we can not say there is an ideal scheduler for multi-core platforms [34] (p. 278). Researchers have gone as far as saying operating systems design has to be remade to fully exploit the advantages and overcome the difficulties of multi-core platforms. This last statement seems to be too harsh and new ways to get rid of bottle-necks have been proposed for general public systems [14]. Zhao and Gu [39] made a survey of the different problems regarding real-time and embedded systems virtualization, the preferred method to handle several CPUs.

Multi-core platforms can be very promising for RTOS, however using its resources in an efficient way is not simple. In fact, bad programming methods in a multi-core processors can lead to bad performance, losing all the interest of using this platform in the first place. The internal architecture of the processor and the fact that resources are shared between CPUs, make programming for multi-core processors a whole different problem than programming with a single-core perspective.

A lot of effort has been dedicated to assure safety properties in RTOS components. These services have to remain deliverable by the safety-critical operating system in its multi-core version. A first step was to capitalize these components by adding an extra layer taking care of the parallelism offered by multi-core processors. This idea of conserving well-established services was also applied to scheduling policies, synchronization mechanisms and even the entirely RTOS. Nevertheless, the approach that has been taken is not ideal, since many limiting aspects (*e.g.* no global state for the RTOS in all cores), are still present.

Many challenges need to be taken into consideration when porting a RTOS to a multi-core platform, in fact certain guarantees were given freely by uniprocessors, *e.g.* no parallel system calls. New components and mechanisms will have to be integrated in the interest of supporting multi-core processors and assuring predictability for the safety-critical system.

3.1 Efficiently use multi-core processors

Multi-core architectures are promising for safety-critical systems. For example if each process of a task set is independent, they can be scheduled in one core and the system will always be schedulable. Even if today processors scale to hundreds of cores, the one-task-one-CPU approach is very unlikely to happen. Tasks also need to access some kind of shared resource at a software or hardware level, independent tasks are very unlikely to be deployed in a safety-critical system. Distributing tasks among the available CPUs could be beneficial for real-time tasks: if a task set was schedulable for one single-core processor, then it should be schedulable in a multi-core architecture and ideally perform better. However, mastering this platforms is very challenging and introduces new problematics that need to be tackled by developers.

3.1.1 Challenges of multi-core platforms

Due to their internal architecture, multi-core platforms are hard to master. In most cases this architecture consists of several CPUs in one die, most likely four or eight. These physical cores usually run at the same clock frequency and can execute tasks in parallel, in theory all cores can run at the same time. Asymmetric architectures with cores running at different speeds and supporting heterogeneous ISAs are also present in the market. Hyper-Threading technologies allow physical cores to be decomposed into two virtual cores, taking advantage of superscalar pipeline. One main problem is *shared resources*: caches can be CPU-independent, data and instruction specific for a certain level, shared and mixed in the next and completely shared and mixed in the last level. Even in a single-core processor cache predictability can be an

issue for schedulability of the system, it can make WCET inestimable. Having a process running in a CPU that is heavy in memory consumption, the cache will be filled with its data, and the other processes will most likely generate cache misses forcing process to load data from the main memory.

Buses, devices, main memory, sensors and disks are also shared in this type of architecture, tasks are in a permanent race to access the hardware resources. Multi-core architecture can introduce time indeterminism due to these shared components. In particular the interconnect bus has to assure real-time properties, like predictability in time. All the different difficulties are enumerated in [29], it goes from errors with manufacturing procedures to programming complexity. In fact, programming with a single-core perspective is very different to developing a scalable software. Race conditions can occur when a process is multi-threaded. If the code is not optimized for running in a multi-core processors it loses all the performance that could be achieved with this architecture. The benefits that can be achieved with multi-core architectures are also presented in this paper: lowering costs (less hardware has to be deployed), less effort for certification (replace multiple executing units with one multi-core processor), improve reliability (deploy the same software many times to make it fault tolerant).

After taking care of the physical properties with this architecture another key problem has to be addressed: the RTOS' services will be requested by different tasks at the same time.

3.1.2 Concurrency in operating system functions

In a single-core operating system the global state of the system was only accessible by one execution unit, updating data structures, delivering services and system calls in general were executed by one single thread. In multi-core platforms, various tasks can try to make a system call, hence the global state of the RTOS becomes a shared resource and locking mechanisms to assure coherence will have to be introduced.

Parallelize an operating system is no easy task. Even in general public operating systems their support for multi-core platforms is confronted to bottle-necks when it comes to deliver its services [14]. For RTOS this is a major problem. When two different tasks are running concurrently and perform a system call, they could be in a race condition for the requested service. The operating system components and services will become shared resources for the task set. For example, if the scheduler is running and selecting a ready task in a queue, more critical tasks could be introduced in the data structure just before the context switch between the task and the scheduler. The *atomicity* of the operating system operations will have to be guaranteed in multi-core platforms, however adding a locking mechanism for the whole system will impact greatly blocking time for the task set. Synchronization mechanisms should be introduced only in the most critical parts of the kernel and will need to be efficient, *e.g.* choose a wait-free implementation of the ready queue rather than protect the object with a semaphore.

Bounding in time the services delivered by the RTOS in multi-core will have to incorporate the synchronization mechanisms introduced. The schedulability and the dimensioning of the system should be updated, due to the integration of new mechanisms. Nonetheless, this shared global state of the system has not really been tackled by the available RTOS. Other approaches, like incorporating components responsible for the parallelism of the multi-core architecture are chosen by developers.

3.2 Capitalizing on single-core implementations

The first step in adopting multi-core processors was to recycle the existing concepts related to real-time programming. The different algorithms that assure good behavior and predictability were reused or modified to be executable in multiple CPUs. For safety-critical systems, the principal idea was to introduce

support for multi-core processors as an experimental branch. Not many changes were done to the RTOS in order to be executable in this architecture. The adoption of multi-core platforms for RTOS is in fact not very satisfactory.

3.2.1 Capitalizing all components

A clear example of capitalization of components for supporting multiprocessors is RUN [28], an optimal scheduler that reduces the multi-core problem to a series of uniprocessor problems. Once the reductions are made RUN will solve the uniprocessor problems with EDF. The overhead of the scheduler is low enough to be considered as a feasible implementation of the scheduler for multi-core safety-critical systems.

When it comes to RTOS supporting multi-core platforms, there is not really a candidate to execute real-time computing concurrently. The seL4 kernel presented in the previous part uses the multikernel approach: for each CPU a single-core kernel is running on top of it [36]. Resources cannot be affected dynamically after boot and CPUs cannot communicate between them. The same procedure was used in the ERIKA Enterprise [3] operating system, aiming to make programming for multi-core processors transparent, *i.e.* programmers should not be worried that their code is executed in a multi-core platform. The adoption of multi-core processors in these RTOS is not very satisfying, there is no global state of the system accessible by all cores of the architecture. Having exactly the same operating system n times wastes memory and can increase the possibilities of failure. A safety-critical system supporting multi-core processors should have a common state accessible by all cores: the RTOS should be aware that it is executing in a multi-core platform with regard to take advantage of this architecture.

3.2.2 Modified scheduling policies

Algorithms that assure real-time behavior are proven to be correct for single-core implementations. Scheduling analysis can be used to assure that a task set is schedulable for a given processor. An effort to adapt them to multiple processors lead to modified scheduling policies. Using the partitioned approach to schedule many tasks in a multi-core processor has the advantage that single-core schedulability analysis can be applied to each CPU. Implementing this type of algorithm is also easier since the mechanics are the same for each CPU. However, the bin-packing problem that this approach is confronted to, can be an issue when the number of tasks increases. An heuristic has to be chosen, but it might not be ideal for a certain number of tasks. Precedence, communication and synchronization make this approach difficult as well. Determining that a dependable task set can be schedulable in a multi-core processor is more complex. Therefore, modified scheduling policies might not be a good idea for a safety-critical operating system.

In [23], models are used to create static scheduling tables for task sets among different processors in a safety-critical system. Actually, Hilbrich *et al.* method goes a step further: the scheduling tables can be created for different types of processor running at the same time and these platforms can be multi-cored as well. This approach is rather complex and is not very efficient due to the underlying NP-problem of static schedules and multi-core platforms.

Real-time parallelism could be achieved by dividing tasks into parallel sub-tasks executed concurrently, in [30] this approach was taken in order to improve performance and minimize WCET. Tasks are divided into segments and are categorized into two types: heavy and light segments. Where heavy segments have the right to be use the slack of the processor if needed. However, parallelizing a single real-time task and not having inter-process communication is very restricting. This proposed task model is not really adaptable to programs that would run in safety-critical systems. The task model that we would like to integrate consists of tasks running at the same time on different CPUs.

Automation could be a solution to support multi-core processors. Binary rewriting is a possibility that could be exploited to parallelize sequential code. A dynamic binary rewriter is presented in [21], this low overhead binary code rewriting method allows threads to dynamically migrate to a specialized CPU supporting performance enhanced instructions. This procedure could be used to place threads in different cores at runtime. Nonetheless, this approach is very limited. Strong hypothesis about the threads' nature make the implementation of this method unrealistic for a safety-critical operating system.

4 Integrating new mechanisms to overcome dependencies

Research around hard real-time computing and multi-core processors is very active. New hardware is being developed, new scheduling policies are proposed, new synchronization mechanisms are implemented. This need for innovation is a direct consequence of the difficulties that need to be mastered in order to efficiently use multiple CPU processors.

4.1 Innovation around multi-core

Not all development around multi-core platforms consisted in adapting existing algorithms and mechanisms. New approaches with very different paradigms are proposed in order to tackle the problems with this type of architecture.

A) Time-predictable hardware

General propose hardware is not really adapted to safety-critical systems, as a matter of fact, it introduces temporal indeterminism, certifiability becomes more complex and security can be compromised (*e.g.* a CPU executing low safety level tasks could compromise other CPUs running high level security programs). To solve this problematic, a new type of multi-core architecture supporting hard real-time systems was developed by the European ARTEMIS ACROSS project [31]. This new multi-core processor assures safety properties at a hardware level. Its design is based in distributed computing with message passing communications between micro components. The micro components are considered as nodes that can execute mixed-criticality programs without compromising safety: a reliable partitioning mechanism is built in the processor. Nodes can be stopped or restarted if they are considered to be faulty. Each CPU contains a micro ROM that allows the system to restart whenever is needed. To address the problem of temporal indeterminism, communications are bounded in time: a time-triggered Network-On-Chip constitutes the core of the architecture. Communications follow an *a priori* time-triggered message schedule. Certification for this architecture is a main concern as well, it is designed to make this process easier: each micro component can be certified individually.

B) New scheduling paradigms

Like it was presented in [29], multi-core processors can be very promising when it comes to deploy different type of services at the same time in a single processor. *Mixed-criticality* systems are a trend in the literature and scheduling policies capable of scheduling real-time tasks and other type of services are proposed [19]. Mixed-criticality algorithms have to assure that critical processes will always deliver their services respecting hard real-time constraints, they must also assure that no data corruption will occur between critical and non-critical tasks. The idea behind mixed-criticality algorithms is that WCET is rare and there is a waste of utilization for the CPU, the remaining slack will be distributed to low-criticality tasks.

De Niz *et al.* also present the problem of *criticality inversion*: temporal isolation techniques can stop a high-criticality task to allow a low-criticality task to run, making the forming miss its deadline.

Yao *et al.* [38] decided to create a new scheduling technique for real-time tasks focusing on memory resources. In fact, WCET of memory intensive tasks can grow linearly with the number of cores due to bottle-necks. This scheduling policy gives the priority to the task that is memory intensive, improving the utilization of the key memory resource and reducing the stall time on each core. The isolation among the different cores is assured by a coarse-grained Time Division Multiple Access memory schedule. Nevertheless, the task model used in this work implies that tasks are independent and this is highly restricting for a RTOS.

C) Synchronization mechanisms

Synchronization mechanisms rely on the ISA of the platform they will be executed on, *e.g.* a spinlock will not be implemented the same way on different architectures, the hardware instructions will differ. The locking mechanisms rely on *atomic* instructions that might not be available in multi-core processors. Like it was stated before, assuring atomicity can have a great impact on the performance of the applications that use it, however locking mechanisms are necessary and contributions aim to improve their performance by reducing blocking time.

In hard real-time, the upper bound of the WCET should be determined, but synchronization introduces indeterminism, it is difficult to know when a task it is going to get access to the shared resource. In [22], three synchronization mechanisms are implemented using hard real-time capable hardware: mutex locks, binary semaphores and ticket locks. These mechanisms should be time analyzable to assure hard real-time behavior. To achieve this, their implementation is based on well known Read-Modify-Write (RMW) operations, like Test-And-Set (TAS) and Fetch-and-Increment/Decrement (F&I/F&D). Nonetheless, these RMW operations are implemented in the memory controller, that way the ISA of the targeted processor does not have to support additional instructions. A static WCET analysis tool was used to evaluate performance. It was found that primitives implemented using F&I/F&D instructions perform noticeably better than those using TAS when considering WCET. It was also stated that busy waiting strategy for ticket locks improves their WCET considerably.

4.2 Desired properties and conserving standards

As it was stated before, safety-critical systems are confronted to safety standards, need to be certified with regard to obtain a security level and their development process is very guided through norms. The documentation regarding safety-critical systems in multi-core platforms needs to be followed as well. Certification for this type of systems can become more complex when they are deployed in multi-core processors.

When it comes to POK, porting the operating system to a multi-core platform needs to assure that the system will still be ARINC 653-compliant. This basically means that the ARINC 653 API should be supported. Inter-partition communication needs to be conserved: if a partition is being executed in a CPU, it should be able to send a message through a queue to another partition executing in another CPU. Intra-partition communication can also be affected, for example if there are more cores than partitions, it could be interesting to execute a partition in various cores. If a process is running in one CPU it should be able to post a value in the blackboard so other processes running concurrently on other cores can read it. The same problem applies to synchronization mechanisms, drivers and shared resources.

For partitioned systems, the works of [24] present how robustness can be assured in a Integrated Modular Avionic (IMA) system running on a multi-core processor. The two standpoints of parallelism

are presented: *Intra-Partition Parallelism*, where processes of a partition can run concurrently and *Inter-Partition Parallelism* where different partitions are executed at the same time. It was stated that parallelism within a partition may be unrealistic due to time determinism and application design, favoring parallelism at the level of partitions. A methodology for robust partitioning insurance is proposed, a fault model describing the issues related to the multi-core architecture is presented. With this model, the different aspects to assure robustness can be implemented. An important observation made by Jean *et al.* is that for industrial reasons, the design of the underlaying architecture might not be available for developers due to industrial secrets.

Nonetheless, the recent works of [27] tackle the problem of Intra-Partition Parallelism for IMA systems and propose an extension for software partitions by introducing *parallel* software partitions. The execution environment of the partitions is supported by a hardware feature called *guaranteed resource partition*, with this feature interferences in the accesses of shared hardware resources are controlled and time prediction can be guaranteed. Parallel software partitions will prevent intra-partition activities from affecting remote intra-partition activity. The extra cost of inter-partition activities will be known, making the integration process easier. The guaranteed resource partition is the main mechanism that makes this possible: by freezing intra-partition conflicts to let inter-partition request proceed, measuring the impact of inter-partition communication is feasible.

In this part we analyzed the different aspects that need to be confronted when adopting multi-core architectures in safety-critical systems. It is clear that this type of system could benefit in many ways from the properties offered by multi-core processors. However, we cannot say there is a common approach to tackle the problem of hard real-time systems executing in multiple CPUs. Real-time computing is heavily affected by this type of architecture, thus, RTOS have to change to assure safety. Properties assured by the system components need to be conserved, and their different requirements is what leads to the diversification of works proposed by researches. This is basically due to the different challenges that need to be mastered and their diverse nature. As a result of my research, a *general* model that can be used as a guide for developers when they plan to port a real-time operating system to a multi-core architecture is proposed.

PART III

Problem statement

Thanks to the SLR process, the main problems regarding multi-core processors and safety-critical systems were found. This architecture has an important impact in all components of the RTOS, nevertheless it is vital for safety-critical systems to adopt this architecture, manufacturers are clearly going in the direction of multi-core chips. The series of challenges raises many questions that need to be addressed in the domain, *e.g.* how can we assure high integrity in safety-critical software deployed in multi-core processors, what additional mechanisms should be included to provide fault tolerance and so on.

During the internship it was decided that we would like to tackle the problem of safety-critical systems and multi-core architectures in a general way. We would like to be able to assist developers in the porting procedure: how can they transform and adapt their current implementation of the RTOS to obtain a functional safety-critical multi-core system and how can they keep the good properties assured by the single-core implementation of the OS.

To achieve this, we assume developers have access to the whole source code of their RTOS, changes will be made in this code and would then be proposed as a series of patches to be reviewed (this is a very common developing method for many OSes in the market).

There are two main problems to overcome when porting the safety-critical system to a multi-core architecture: the booting process of the system will be different and is very specific to each architecture, most of the time, the code tends to be written in assembly language and is not really reusable nor modular. We will assume this part of the porting procedure is already taken care of for the rest of the work. We will also assume that initialization of the safety-critical system is done with one bootstrap node responsible for initiating the other cores and the internal data structures of the kernel.

On the other hand, we will focus on the main kernel mechanisms and try to extend the execution model of the RTOS and the tasks in order to obtain *true parallelism* for kernel modules: scheduler, locking functions, communication protocols, ... Their source code tends to be modular and architecture independent, making modifications clearer and possibly applicable for all the kernel components. We will be interested in the necessary modifications that extend the execution model of the RTOS: what needs to be implemented, added or extracted in the different kernel modules in order to support multiple execution contexts that can potentially run at the same time. The execution model will be extended and the bibliographic study demonstrated that recycling and adapting the existing code is a viable option to obtain a safety-critical system that conserves the good properties already assured for the single-core implementation. This approach is also the most mature and more realistic to implement.

What the final user is most interest in, is to have a functional RTOS with the same properties (time predictability, robustness, ...) and at least the same functionalities as before (*i.e.* the single-core API should also be available in the multi-core version of the safety-critical system). However, to achieve this, low level kernel mechanisms used by the API functions will need modifications, and be adapted to support multi-core architectures. By the means of *refinement* and *relaxation* of the low level functions, the existing API can be delivered in the multi-core version of the safety-critical system. For example, functions of the API setting new properties in a thread (set a new priority, a new deadline, a new time budget, ...) will

use *refined* low level functions assuring *atomicity* in the modifications of the thread data structure. Likewise, *relaxation* of low level functions will be needed to support checkpoints or mostly any safety mechanism.

The remaining chapters of this work will describe, how system calls can be executed in parallel, their functionalities highly rely on the manipulation of data structures. These data structures can be purely logical, *i.e.* they are not binded or do not represent a state of the underlying hardware resources, or they can be completely dependent of the hardware. For the second case, modifications in the source code will be most likely necessary, while in the first case, due to code modularity, functions might be reusable as they are. The evolution of the services can be described by the relation of the system calls, the data structures manipulated and the underlying hardware, if applicable. For these reasons, in the following chapter, we propose a language to describe dependencies between OS' system calls, kernel data structures and hardware resources. This language will allow us to determine what services will need modifications, and will make appear possible data races/critical sections caused by the new architecture. Developers will be able to explain how their kernel works and how services are linked between each other. For this work, we will focus on the partitioned approach to adapt existing mono-core implementations of safety-critical system. With partitioning, the source code can be reused in the multi-core implementation, however changes will still be needed in order to assure that the new version of the safety-critical system complies with the same standards as the single-core RTOS. This work aims to give a first perspective of the changes that can be automated by the introduction of modification templates. In the final chapters we will present these templates and the obtained results of our implementation applied to two kernels: POK and seL4.

PART IV

Modeling the multi-core evolution: hypothesis and objectives

Since we are basing modifications with the partitioned approach, we would like to identify how system calls differ at execution time when they are performed in a multi-core architecture. Functions were programmed with a mono-core standpoint, and having various executions units with the new architecture might lead to functional problems. With the partitioned approach we would like to know in which cases the source can be capitalized, when it needs slight modifications and when the system call/service will need to be reimplemented from scratch. To determine if changes are needed, we need to represent (with a model), dependencies between the new hardware architecture, the services delivered by the safety-critical system and the kernel data structures.

1 Motivation of the approach

In the bibliographic study, the desired properties for the multi-core implementation of the RTOS were explained: developers want to *efficiently* use the new architecture, *capitalize* as many source code as they can and *conserve* all the safety standards and guarantees the single-core implementation offers. A lot of effort has been dedicated to prove that single-core execution of safety-critical system is *functional*, *i.e.* the code works as expected, *safe*, *i.e.* the code will not cause any issues regarding the security/safety of the safety, and *modular*, all data structures have a very precise role in the kernel.

Capitalizing existing source code can be a viable option for adapting safety-critical systems because their implementations has been highly controlled and confronted to some sort of validation (*e.g.* standards, external validation organisms, formal proofs). More precisely, each data structure exists in the kernel space to deliver some sort of functionality. What needs to be determined is if execution conditions are going to be the same in the new architecture.

Actually, since there is a single execution unit for a single-core processor, only one execution context can be executed at any time, *i.e.* there is only one task being executed at a given time t. Nonetheless, concurrent access to data structures and critical sections are already a problem in the single-core implementations of RTOS due to *preemption*. This issue is already taken care of in uniprocessor OSes: interruptions might be disabled making system calls atomic and non-interruptible, locking mechanisms can also be added to protect data structures. However, since we would like to have multiple system calls executing in different cores these protection methods are not sufficient: more system calls might need atomicity and synchronization mechanisms will have to protect shared data structures being accessed by different cores at the same time.

For these reasons a representation of dependencies between logical resources (data structures), execution units (CPUs) and other physical resources (sensors, memory) needs to be proposed and analyzed. Having such representation should allow engineers to identify what key components of their OS need changes, what data structures are at risk and how the source code will be modified. This model will also allow us to identify different configurations of the new underlying hardware, their relation between the service and the clients, in order to detect possible critical sections that might appear. With this information the required

changes to protect such sections can be determined. In fact, having a way to describe the new situations that the kernel is confronted to, will allow us to determine what it is needed to keep integrity at execution (rewriting a module from scratch or adapting the existing implementation with small modifications). Having such model will also allow us to see how clients will take advantage of the new available resources.

2 Kernel Execution and State Descriptors

Different resources represented by data structures are used inside the kernel space to deliver essential functionalities for each service. Some of them will represent a pure logical state, *e.g.* the state of a semaphore for synchronization, creating a task with time properties ... or a representation of a hardware resource, *e.g.* memory mapping accessing to a specific sensor. In the first case the specification of the service that uses the logical resource will not be affected by the multi-core architecture. Functionalities will need to deliver the same behavior but in a context of multiple execution units. For this case we say that *data structures will not be correlated to the hardware*, they represent a pure logical state. On the other hand, the service taking care of the MMU configuration for example, will have a correlation between the data structure, the memory and the new introduced execution units. The correlation is represented by data structures that mirror the hardware resource. However, multi-core architectures are usually a basic evolution of single-core chips, in which verification has been proved to be correct. Therefore we would like to capitalize source code that has been submitted to verification; and when this complete capitalization is not possible we would like to know what changes are needed to keep the good properties assured by this code. To achieve this we need to model how kernel services use data structures and how the introduction of multiple execution units will affect these resources.

A general model for resource utilization can be defined with the following notations. Let S be the service in the safety-critical system. Let R be the hardware resource. We have two types of data structures: D_m the mapping data structure (or organized data, *e.g.* macros) responsible of initializing, configuring and/or giving access to the resource R, this data structure is a direct *mirror* of the hardware into the memory and is static after initialization of the system. On the other hand let D_a be the data structure controlling the access to the resource R. This data structure can also be the resource itself (it represents a pure logical state in the kernel). We will represent the different clients $EC_1, \ldots, EC_N$ in a box, also called execution units.

2.1 Single execution unit representation

The following images show a graphic representation of the resource utilization described by the general model. The mapping data structure represented by the left image, controls and configures the resource R, clients just have to read/write the values directly from the memory where the data structure is located. We also have only one execution context EC_1 able to access the service S and its data structure.

On the other hand, the image on the right represents the access control data structure, clients C_1 and C_2 have to register in the service S. D_m will handle the clients and let them know when the resource is available. Again only one execution context can request the service S and only one client can have access to the resource at a given point of execution.

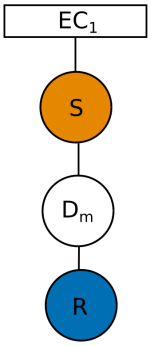

Figure 2: Mapping data structure

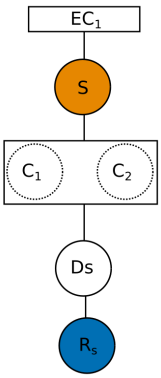

Figure 3: Access controll data structure

The final case is represented by the following image, the data structure D_a is the resource that is being used by the different execution units, there is no underlying hardware implicated in this model.

These images are the clear representation of what we have in the current single-core implementation of the safety-critical system, one execution unit able to request various services, assuring that access to any data structure is exclusive to this unit and therefore integrity is not compromised (assuming protection mechanisms handling multiple execution contexts for a single-core implementation are integrated in the RTOS).

2.2 Resources confronted to multiple execution units

When the number of execution units increase, developers need to adapt the kernel services making them able to handle the duplicated execution units. For example having multiple CPUs will need multiple interrupt management tables. When the underlying hardware is not duplicated and is confronted to multiple execution units, data structures can still be instantiated several times, a clear example is the MMU configuration, each core can have an assigned region of the memory. Depending in the approach taken by engineers, the source code will be modified differently. Duplicate resources and partitioning the kernel is the most common approach used by the real-time community and is understandable because legacy code can be capitalized. Having partitioned memory spaces assures exclusive access to resources for every core, it also means that less modifications might be needed in the source code.

On the other hand having a global pool for clients and only one data structure taking care of them, might need serious modifications in the source code and more effort to assure mutual exclusion for the underlying resource. The service will be constraint to perform modification in only one execution context at a time. For this last case automating changes seems very difficult as opposed to the partitioned approach where legacy code can be reused with or without slight modifications.

During the conception phase, engineers need to choose an adaptation policy to implement, in this section the different options are explained:

- The single hardware was mapped by a data structure. The binding with the hardware is strong, therefore *data structures need to be multiplied* when the hardware is multiplied.
- The hardware will remain unique but it will be confronted to various execution contexts, therefore duplication of data structures is needed, one new data structure for each new context.

- The final option consists in conserving one data structure to control access to all the new hardware resources. Nonetheless, this will lead to specification changes.

2.2.1 Duplicating mapping data structures

When there is a strong binding between the data structure in the kernel and the hardware resources, *i.e.* the data structure maps in the memory a region to control the hardware, developers are forced to duplicate the instance of the data structure. In fact, each new data structure will map a different resource in the memory. For example adding sensors to the system will need new mapped data structures. For each resource, the system will have a dedicated data structure D_m.

A) Underlying hardware is duplicated

The following image shows how the resources R_1, R_2 and R_3 are mapped in the kernel through data structures D_{m1}, D_{m2} and D_{m3}. We have three execution units represented by EC_1, EC_2 and EC_3. The last two units have access to two different services S, potentially all execution contexts can request functionalities from one of the instances of the service and we will have true parallelism.

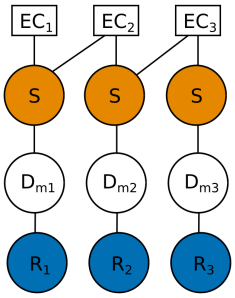

Figure 4: Duplication of the allocation policy and multiplication of the service

B) Underlying hardware is not duplicated

The following image shows how the resource R mapped in the kernel through data structures D_{m1}, D_{m2} and D_{m3}, these data structures are usually configurations of the underlying resource (*e.g.* different MMU configurations for the main memory). We have three execution units represented by EC_1, EC_2 and EC_3. For this case, the hardware may have an integrated protection mechanism, for example for the memory, the interconnect bus can be locked to other cores when one performing read and writes. If this is not the case then S will control the access to the resource and only allow modifications on one data structure at any time.

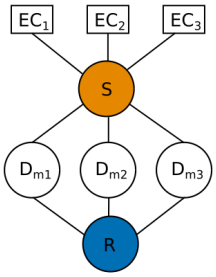

Figure 5: Handling a single resource with multiple execution contexts

2.2.2 Duplicating access control data structures

If developers choose to multiply the access control data structures, this will allow the service to be *multiplied*, *i.e.* each data structure can deliver the functionalities of the service S in parallel. This is not the case for duplicated data structures controlling a single hardware, the MMU configuration is an example of unavailable parallelism with duplicated data structures. Nonetheless, when each duplicated data structure represents a newly introduced hardware, clients will only need to register into the right data structure. This approach can be advantageous because the code that is already implemented in the single-core version of the RTOS can be recycled: the code just needs to be executed in the right data structure.

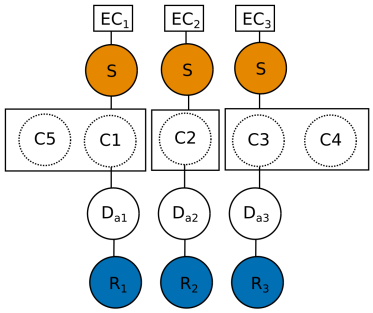

Figure 6: Duplication of the access control data structure

2.3 Conserving one access control data structure

The final case is represented by the following image: it consists in conserving one access control data structure. The improved version of the service will conserve the single data structure, but will take care of the additional parallel execution contexts at the same time. In this case with a global pool for clients and with one service for all the execution contexts, the source code for the service might need serious modifications to support the new execution model. These changes will be most likely be done by hand by engineers. The following figure shows a representation of this approach. A clear example for this is a semaphore, the resource will be unique even in the multi-core version of the kernel, however it will be confronted to various executions units and changes might be needed.

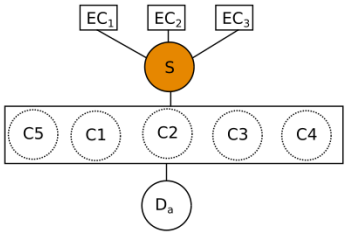

Figure 7: One access control data structure for many resources

Now that we have presented the different approaches to adapt services and their data structures for the newly introduced execution units. We would like to evaluate if critical sections can affect the data structures and in which case developers need to adapt the source code to protect the shared resources.

3 Handling critical sections inside the kernel

After presenting the different resource utilization models for a multi-core architecture, this section will explain how kernel data structures can be confronted to data races and critical sections in this new architecture. If the data structures are shared by different execution contexts, then a protection/synchronization mechanism is necessary. This is true when the execution contexts belong to the same core, this was actually a problem in the single-core implementation, synchronization/protection mechanisms are deployed to avoid incoherence due to interruptions. We can give the following definition of critical section for data structures in safety-critical systems:

Critical section: A part of code in a system call manipulating one or more kernel data structure that may not be concurrently executed or preempted by an interruption.

Since multi-core architectures allow code to run in parallel, being vulnerable to critical sections becomes more likely, now external execution contexts can potentially interfere in data structures, however this is only true when the data structure is shared. For single-core implementations, critical sections were already a problem due to interruptions and preemption, specially in a RTOS where handling an interruption might be the highest priority.

This work will explain what synchronization and/or protection mechanisms have to be included in order to assure safe critical sections. To solve this problem occurring at runtime, different methods can be applied: *avoiding*, *detecting* and *preventing* data races. Avoiding and detecting might need an extensive analysis of the source code to determine if the kernel can arrive to a data race at runtime, which in a multi-core architecture is highly likely to happen. Protecting critical sections is the method proposed in this work. This sections explains how data races can occur and what methods and mechanisms can be included by developers to assure that the system remains reliable.

3.1 System calls concurrence

Having new resource utilization cases forces developers to incorporate new mechanisms able to assure *safety* and *predictable behavior*. If we take the example of the unique data structure taking care of all the clients for the service, we can see that if two execution contexts request functionalities for the resource, they can be in a data race for the D_a data structure. Therefore, when this approach is implemented in the RTOS, **a synchronization mechanism is mandatory.**

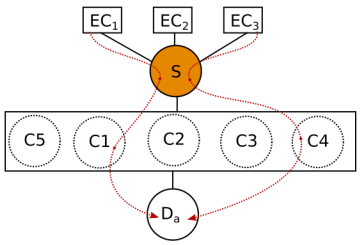

Figure 8: Two system calls trying to access the allocation data structure

If developers choose to multiply data structures to handle the multiplied resources and the execution units, then the service can run in parallel, *i.e.* one service for each data structure duplicated running in different execution units. This approach assures that if two allocable resources R_{a1} and R_{a2} can deliver the

same service S locally, we will have two instances of the service S: S_1 and S_2 running in two different cores.

The following picture represents a case scenario, where three execution units request three different instances of the service S_1, S_2 and S_3, having an influence on three different data structures D_{a1}, D_{a2} and D_{a3}. In this case, services S_1, S_2 and S_3 can deliver their functionalities in parallel.

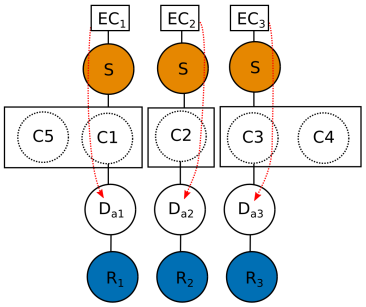

Figure 9: System calls able to run in parallel

Synchronization might still be needed if two execution units need the same service of a given resource R_1 for example. This last figure shows two execution units EC_1 and EC_2, requesting the S service. The service will need to dispatch these requests and assure correctness, making synchronization necessary even when the data structure and services are duplicated.

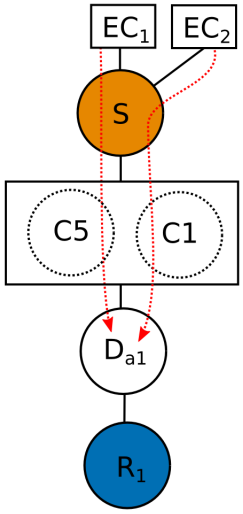

Figure 10: Synchronization for a multiplied service

3.2 Sharing data structures between execution units

In multi-core RTOS, sharing resources might be essential for some applications, *e.g.* having a communication mechanism (blackboard, queue, buffer...), protecting a shared resource with a lock, *etc...* Having the ability to share data between cores leads to the problem of data race, making synchronization mechanisms necessary. Nonetheless, synchronizing for a write access into a data structure can have an influence in the execution of the service requesting the write, *e.g.* the resource is being modified by another service that has blocked its access.

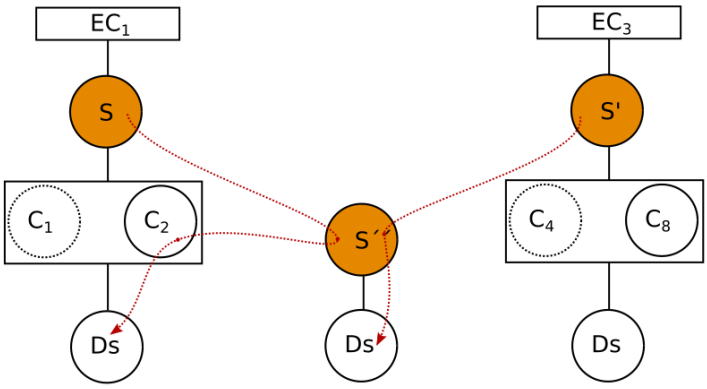

Figure 11: Shared data structure between services

This figure represents how two entities of the same service S share a resource DS that will influence the state of their own data structures D_s and D'_s. Trying to access to the resource DS will generate some sort of feedback from the service S'', this feedback can be positive *e.g.* the service can access the resource in mutual exclusion, or the S instance can receive a delaying feedback, *e.g.* the resource is busy, S will need to execute the corresponding event, like waiting for the lock to be freed or calling the scheduler.

3.3 Disabling local preemption to assure coherence

While a service is running in a given core an interruption might occur, for RTOS, their kernel is fully preemptive most of the time. *I.e.* if the interruption has an important priority execution of a given thread might be stopped in favor of the handler. This raises another problem of data race: if the handler routine leads to a context switch, and if the task was modifying a shared or local data structure its logic state might be compromised. Using only a lock with inheritance protocols solves this problem but the implementation needs to be feature in the RTOS which is not available in any RTOS kernel to my knowledge.

In the case of a RTOS, the most common interruption is the clock tick. For some scheduling policies the clock tick is the basis of the algorithm and a context switch can occur at every tick, (missing a deadline for an important task can have severe consequences). Therefore, when developers need to assure critical sections are protected, masking interruptions might not be a reliable option.

In order to assist developers in the porting procedure of the safety-critical operating system, a design pattern for a partitioned scheduling policy and for the synchronization mechanism are proposed in this work. These patterns aim to be general but some hypothesis about the RTOS implementation are made: there is a priority based scheduling policy that can be adapted to a partitioned approach, the task model allows programs to communicate between them. Adaptation is described using the POK and seL4 kernels in the next chapter.

PART V

Obtaining parallelism in system calls

Like it was explained before in this document, the approach that we would like to implement when porting a RTOS to a multi-core architecture is to obtain a kernel able to execute parallel system calls. We modeled the resource utilization of the newly introduced execution units and showed how they impact the shared data structures. Having this clear view of the potential critical sections allow us to propose a series of patches that will protect these critical sections. The first part for these patches consists in partitioning the services that can be multiplied, for instance the scheduler. And the second part consists in performing the necessary changes for shared data structures, *i.e.* adding synchronization mechanisms and masking interruptions. While developing the patches for the POK kernel, a comparison with the seL4 kernel was made, and we deduced that similar protection patterns were necessary in both implementations. The results of this observation is explained in the second section of this chapter, we explain how we can identify unsafe code patterns and how to protect them. Finally we take advantage of the similarities of the source code in RTOS' kernels to propose a patch generator that could be the basis to adapt single-core implementations of safety-critical systems.

1 Description of the approach

To adapt the current single-core implementation of the RTOS we would like to patch the existing source code. In the first place, we would like to recycle legacy code of the implementation: detect what can be reused in the current code. And in the second place, we would describe what additional mechanisms are integrated to the data structures in order to protect concurrent changes.

1.1 Partitioning services

During the porting procedure we need to identify what type of data structures will be used and if they will be duplicated or not. The following list describes the data structures used by the scheduling system, the first service of the kernel that we would like to adapt to the multi-core version. We explain if any modifications are required for a partitioned scheduler.

A) Data structures types:

We describe with the following list the main data structures used by the scheduling service and if they will be exposed to any functional change that might lead to code modifications.

- **Task descriptors**: This data structure is used to describe a task (stack pointer, priority, period, deadline,...). Task descriptors can be used as they are implemented in the single core version. There are not any functional changes that would cause any alteration in this resource.
- **Task containers**: This data structure might need to be introduced in the multi-core version of the operating system. Tasks need to be grouped, in order to bind each group to a given CPU.
- **Task index**: In order to iterate through the task descriptors, indexes are used. Each task container needs an index to iterate through the groups. This index is usually represented as a macro in the code.

The type will remain the same.

- **Scheduling policy data structure**: The algorithm implemented by the scheduling policy might use a dynamic data structure (queue, tree, ...), representing the execution state of the different tasks (ready, waiting, locked,...) in the operating system. This data structure will remain the same as well, the functionalities do not change *a priori*. **Note**: Some systems might use the task container as the dynamic data structure for the scheduling policy. It is very likely that the porting procedure to the multi-core procedure will keep the scheduling policy that was already implemented in the single-core version of the RTOS, *e.g.* if a RMS scheduler was implemented, the algorithm will be extended to become a partitioned-RMS

In practice: For the POK operating system we can see that almost all these data structures are used: threads, partitions (for task containers and grouping) and tables for indexing the threads. The scheduling policy uses the task containers directly to elect the next task, also the state of the task is included in the task descriptor.

B) Instantiating data structures

After deciding what types of data structures will be used in the multi-core, engineers need to know how many instances need to be declared in the code.

- **Task descriptors**: Each task needs to have an unique representation in the system. Therefore these data structures will not be duplicated.
- **Task containers**: Containers need to be *duplicated*, in order to group the different tasks that will be binded to one CPU. *N.b.:* if the RTOS uses partitions, they do not need to be duplicated, however if the number of CPUs is smaller to the number of partitions, the scheduler will need to be adapted to handle partitions in each core as well.
- **Task index**: For each container an index needs to allow the iteration of tasks. This data structure will be *duplicated*.
- **Scheduling policy data structure**: Since the scheduling policy can be requested at the same time by different CPUs, *duplicating* the dynamic data structure is necessary: the code of the scheduler can be shared but the algorithm will be able to modify different data regions.

C) Initialization data structures

The initialization process in a multi-core RTOS will be quite similar to a general public OS. The kernel will be loaded into memory and one CPU will be activated. This CPU, called the bootstrap node, will be responsible for initializing the whole system (*i.e.* hardware and software resources). Once resources are initialized, the system is synchronized and CPUs start their execution after the startup interruption is sent.

- **Task descriptors**: The initialization does not changed in the multi-core version of the OS.
- **Task containers**: These data structures need to be initialized containing the task descriptors that will be executed in the different cores. This need a previous study, *i.e.* schedulability tests with the given tasks.
- **Task index**: Each index will be pointing to the right task container.

- **Scheduling policy data structure**: For these data structures, the initialization depends entirely in the scheduling policy, *i.e.* the data structure might be initialized containing just one idle thread and one kernel thread, or include all the threads that are able to run in that core.

To give an overview of the data structures, the following two images show the adaptation of the RTOS scheduling data structures to a multi-core version using the partitioned approach.

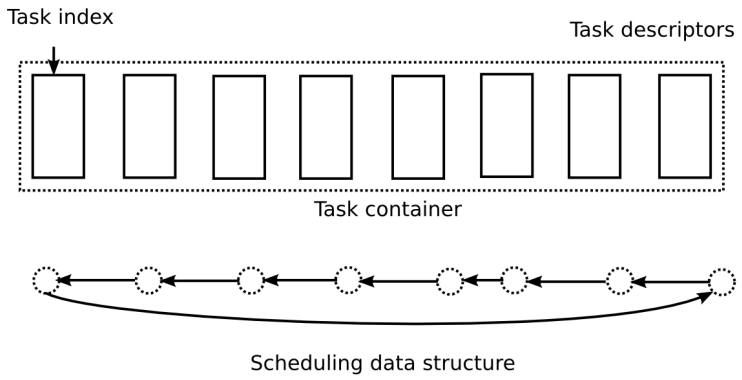

Figure 12: Overview of the scheduling data structures

Like it was described in this section, we have the four major data structures needed by the scheduling service: task descriptor, index, container and scheduling data structure.

After the adaptation, the organization of the data structures will look like the following image. Duplication of data structures is clearly visible. We suppose we have a tri-core processor, therefore when duplication is needed, three instances will be created.

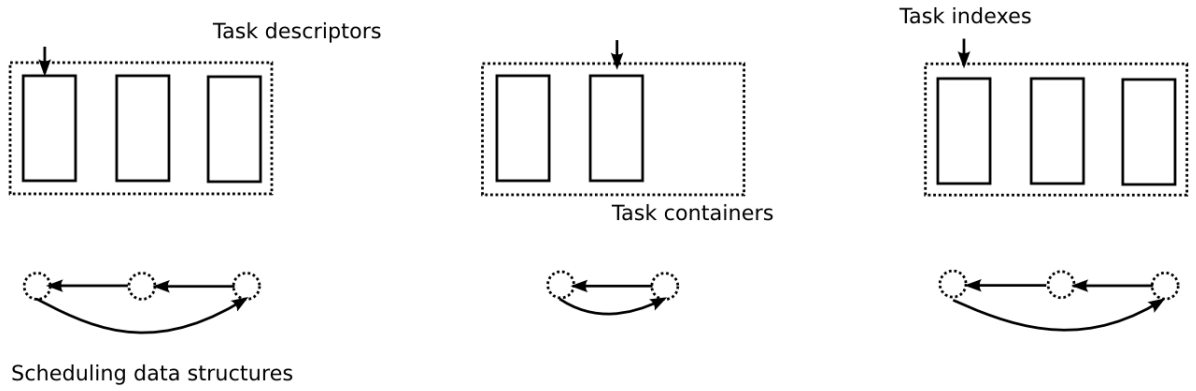

Figure 13: Overview of the data structures adaptation

1.2 Providing system calls with atomicity

Protection of data structures can be assured in two different ways: *synchronizing* changes, for shared resources and *disabling* preemption for local resources of a given CPU.

Since the targeted OS needs to comply with strict time requirements (deadlines should always be met) masking interruptions in the kernel code is not always recommended: if a very important interruption

occurs, the system needs to execute the handler, assuring the safety of the RTOS. However, coming back from the handler and choosing another task might lead to an incoherence in the data structure. Developers need to choose and estimate what is safer for the end users, either assure coherent execution at all times and mask interruptions, or allow preemption at any time and do not mask interruptions. During the rest of this work we will choose the first method that consists in masking interruptions in addition to synchronize the access to the resource, this will assure a coherent state in the kernel at all times.

Protection for the three type of data structures presented needs to be assured *locally*: when the task gets preempted due to an interruption, and *externally*: protection from external contexts trying to perform modifications. The simplest way to do this consists in **adding a synchronization mechanism into each data structure**. This could be a reliable options, system calls are usually fast [22] and having active waiting in kernel is usually common in other kernels. For example in Trampoline [11], the multi-core branch of the kernel follows this approach: any shared data structure between the execution units has a spinlock assuring mutual exclusion. Developers need to chose if they want to mask the interruption and delay the handler until critical operations are done, or if they want to keep a fully preemptive kernel without masked interruptions.

We also need to assure that adding synchronization mechanisms in data structures does not include deadlocks, livelocks or other time anomalies. Usually critical sections do not perform a big nesting, *i.e.* system calls do not grab more than one or two locks at the same time, and when nesting is needed, we need to assure that locks are freed in the opposite order than when they were grabbed.

1.3 Assuring data's integrity when needed

Since the new version of the RTOS will run in a multi-core architecture different system calls can be executed at the same time. The approach that we would like to implement consists in allowing parallelism in system calls without compromising the data structures' integrity.

In order to determine if data structures can be exposed to data races, engineers can use the *call graph*. Obtaining it is straightforward, most compilers have an option to create an output file to draw a call graph. For GCC, using the `-fdump-rtl-expand` options for compilation creates a file (a `.166r.expand`) that can be used by `egypt` and then output the call graph for each file. Developers just need to make sure that all files needed are compiled: most operating systems have a dynamic compilation tool chain, *i.e.* only the files needed will be compiled.

```
$ egypt --include-external sched.c.166r.expand | dot -Tpng > out.png
```

Listing 1: Call graph script

```
#!/bin/bash

for f in *.c.166r.expand;
do
egypt --include-external $f | dot -Tpng > ${f%%.*}.png;
done;
```

Once the call graph is obtained, functions performing modifications on data structures can be identified. The idea is to proceed by data structures: for each function manipulating a shared data structure, we need to check the type of operations that are performed (reads or writes) and try to identify patterns that are used throughout the whole kernel code. The patterns usually consists in how a data structure is accessed and modified, how the kernel manipulates various data structures at the same time, and so on. The callgraph will allow developers to identify where to find the patterns and then apply the necessary changes.

A) Local protection

The following image shows the functions that interact with the task descriptor data structures: `pok_threads`. Arrows that go from the functions to the data structure represent write instructions, typically a modification on one of the fields, whereas arrows that go from the data structure into the function represent reads.

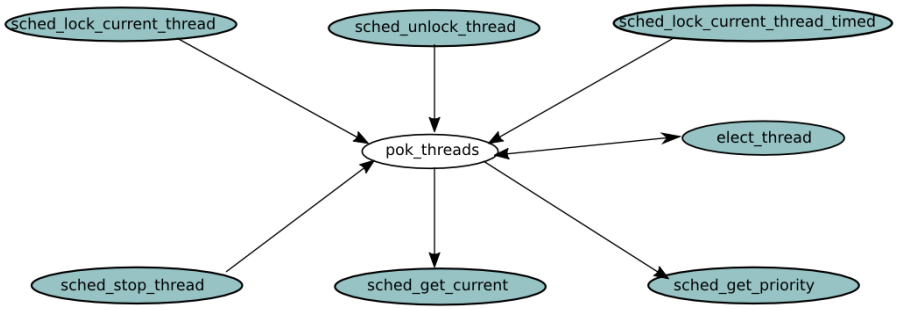

Figure 14: Functions interacting with `pok_threads`

In order to protect data structure modifications in functions, a pattern making data structure operations atomic (blocking interruptions and/or locking spin locks) can be used. Masking interruptions is essential to assure local coherence. However like it was explained before, an interruption might have a higher priority than the current task, *e.g.* a failure in the system needs to be treated immediately to prevent important loses.

B) External services interacting with the data structures

The following image shows the different functions calling the scheduling service through its various functions. This representation is used to illustrate that multiple execution units in the multi-core architecture might be in conflict at execution when they are after the same resource.

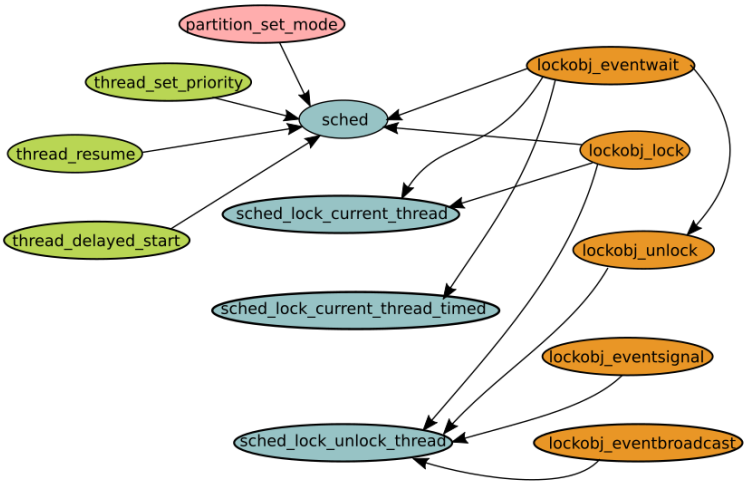

Figure 15: Call graph for the scheduling service in POK

The blue ovals represent the scheduling services. The orange entities represent the synchronization

mechanisms in POK, while the green is the thread service (only used at startup). Finally the partition service is presented in pink, like for the thread service this call is only performed when the system needs to be initialized or restarted. The arrows represent the function calls, for example `lockobj_lock` will call the set priority function, which in a single-core version will be the unique function able to access the thread data structure, as opposed to being potentially dangerous in a multi-core implementation (other system call might be performing changes in the thread).

In this section we explained how we can achieve parallelism in function calls. The first step was to identify the different data structures that were used by the scheduling service in order to duplicate them when it was needed and protect them when they were shared. Using the partitioning approach would allow developers to save a lot of tested code of their implementation. Nonetheless, it is clear that data's integrity is compromised when data structures need to be share between the execution units. The first approach used in some kernels is to add synchronization mechanisms protecting the critical sections. We decided that we would follow the same approach and use existing locking mechanisms to ensure mutual exclusion. Adding this feature to some of the kernel functions will need a new WCET analysis, but this is out of the scope for the internship.

2 Identifying recurrent transformation patterns

During the development period, I was comparing the POK kernel and the seL4 kernel to see if there were any recognizable patterns to protect shared data structures used by the system calls. And in fact some similarities were recognizable. Some changes made are applicable to any shared data structure but were first applied to scheduling data structures.

2.1 Protection rules for shared data structures

After describing the implementation changes that will affect the scheduling data structures and with the synchronization method explained, we can define *protection* rules for shared data structures, assuring mutual exclusion and data's integrity for them. These rules will be applied to a pattern in the code, for example during the experimentation I noticed that operations affecting the data structures were very similar in all the files that used them. This section will go through each pattern and explain what rules need to be applied to protect them against data races and memory incoherence.

A) Protecting sequential operations (reads and writes)

Rule 1: Functions performing sequential direct changes, or reads directly in the data structures need protection. The sequential operations will be wrapped by a synchronization mechanism.

B) Protecting iteration patterns

Rule 2: For data structures used as containers, *e.g.* the task container, operations usually use an iteration method to find a target. Therefore, protection should be introduced for the iteration loop.

C) Atomic operations for dynamic data structures

Rule 3: To protect dynamic data structures, its functions need to be transformed to become atomic, this will assure that concurrent access will not happen even with the presence of multiple execution units.

It is very likely that specific operations handling dynamic data structure are implemented in the single core version of the RTOS. The porting will consist in assuring that modifications will remain exclusive (if another execution context tries to call the same function at the same time, then synchronization will be needed), this can be achieved with the lock introduced in the data structure.

3 Automating changes for the source code

In this section, we will explain how automation can be achieved with the proposed rules/patterns. In the first subsection we will describe how patterns can be recognized. The second subsection describes the method used: sematic patches and the tool Coccinelle. We end this section by describing the tool created during this internship: a semantic patch generator able to recreate the patterns for other kernels programmed in the C language.

3.1 Identifying the patterns in the source code

A) Sequential operation in the data structures

The first rule that we have consists in protecting data structures when they are exposed to sequential read or writes (protecting when only one affectation is done might necessary as well). The following listings gives an example of the pattern that we would like to recognize in the source code.

Listing 2: Simple mutual exclusion on task descriptor

```
1  ...
2  td->field = new_data;
3  ...
```

Listing 3: Protected simple mutual exclusion on task descriptor

```
1  ...
2  mask_interruptions();
3  lock(td->lock);
4  td->field = new_data;
5  unlock(td->lock);
6  unmask_interruptions();
7  ...
```

Detecting modifications in data structures can be straightforward, using regular expressions can allow engineers to detect critical sections that need to be protected, *e.g.* if a method modifies the data structure being used, its call need to be wrapped by disabled interruptions and/or a synchronization mechanism. The same principle applies if a data structure is modified, the most common case of a data structure modification is the reference and affectation.

Listing 4: Setting the thread state

```
1  void pok_sched_lock_current_thread_timed (const uint64_t time)
2  {
3      pok_threads[current_thread].state = POK_STATE_WAITING;
4      pok_threads[current_thread].wakeup_time = time;
5  }
```

This example shows a function setting the state of a task to locked with a given timeout. The modifications in the data structure `pok_threads` are highlighted with the red dots. These modifications needs to

be protected because the thread data structure can be used by a different service, or it can get preempted between the two affectations.

In order to protect this change, we follow the pattern proposed in the Listing 3 and proceed to disable the preemption and add a spinlock between the modifications of the data structure.

Listing 5: Setting the state of a task with mutual exclusion

```
void pok_sched_lock_current_thread_timed (const uint64_t time)
{
    pok_preempt_disable();
    SPIN_LOCK(pok_threads[current_thread].spin);
    pok_threads[current_thread].state = POK_STATE_WAITING;
    pok_threads[current_thread].wakeup_time = time;
    SPIN_LOCK(pok_threads[current_thread].spin);
    pok_preempt_enable();
}
```

The rest of the rules will be explained in the following subsection, where we explain how we can achieve automation with Coccinelle and semantic patches.

3.2 Semantic patches and Coccinelle

We would like to automate the patching procedure, in fact many bugs can overcome in the porting phase. Some critical sections might not be seen and could be left out. For this reason we believe that having an adapted tool might be beneficial. Coccinelle [2], is a tool that sees some success in the Linux kernel and that I decided to use during my internship to see if I could achieve some automation in the porting phase of the kernel.

Using semantic patches can be beneficial to an operating system, the same type of changes be applied to different files. This assures that all the matching expressions will be treated. The following listing shows an example of a rule to protect double affectations in the state and wakeup time for a task.

Listing 6: Semantic patch protecting the state change in the task descriptor

```
@ rule2 @
expression id;
expression p1, p2;
identifier pok_threads;
@@
<...
+pok_disable_preempt();
+SPIN_LOCK(pok_threads[id].spin);
pok_threads[id].state = p1;
pok_threads[id].wakeup_time = p2;
+SPIN_UNLOCK(pok_threads[id].spin);
+pok_enable_preempt();
...>
```

Coccinelle just needs a simple command line using `spatch`. The following example shows how Coccinelle can apply the semantic patch in all the c files in a subdirectory of the POK kernel. The patch will be usable by a revision control software like Git.

```
$ spatch -sp_file patch1.cocci kernel/core/ *.c > thread.patch
```

The patch created can be tested and then emailed to a developer's mailing list for example. The kernels studied in this work have an open development very similar to the well known Linuk kernel mailing list.

Some patches can be transformed in order to have more hits when the source code is parsed, this is the main propose of having the semantic patch. For example the following listing protects sequential read or writes.

Listing 7: Sequential read/writes in POK

```
@ multiplewrites @
expression p1, p2;
identifier attr;
@@
<..
+pok_preempt_disable();
+SPIN_LOCK(pok_threads[id].spin);
attr->deadline = p1;
...
attr->stack_size = p2;
+SPIN_UNLOCK(pok_threads[id].spin);
+pok_preempt_enable();
..>
```

Another example of semantic patch is the pattern protecting iteration for container data structures. The following listing shows what pattern we want to detect (the iteration uses a pseudo-C syntax for space). The iteration that we present is actually the strongest pattern we have. One lock needs to assure coherence for the container while the iteration takes place while inside the loop we will have a synchronization mechanism for the contained data structures. In the POK kernel for example we would lock at the level of the partitions and then at the level of each thread, that way if a functions needs a specific thread while the loop is executing it might be able to perform changes. However the partition will have the same size.

Listing 8: Iteration in the task container

```
    ...
    task_descriptor *td;

    for(td: task_container) {
    td->field = new_data;
    }
    ...
```

Listing 9: Protected iteration in task container

```
    ...
    task_descriptor *td;
    mask_interruptions();
    lock(task_container->lock);
    for(td: task_container) {
    lock(td->lock);
    td->field = new_data;
    ...
    unlock(td->lock);
    }
    unlock(task_container->lock)
    unmask_interruptions();
    ...
```

The following listing shows a particular case when the iteration can be interrupted due to a test case (the code was looking for a particular data structure for example or it detected an error), a new function might be called or a goto forces the system call to terminate (the goto goes forward in the code). In this case, the lock needs to be freed before the function call or the goto tag.

Listing 10: Iteration with goto or function call

```
1   ...
2   task_descriptor *td;
3
4   for(td: task_cont) {
5   td->field = new_data;
6   ...
7   if (td->field == test)
8   goto end; / function();
9   }
10  ...
```

Listing 11: Protected iteration with goto or function call

```
1   ...
2   task_descriptor *td;
3   mask_unterruptions();
4   lock(task_cont->lock);
5   for(td: task_cont) {
6   td->field = new_data;
7   lock(td->lock);
8   ...
9   if (td->field == test) {
10      unlock(td->lock);
11      unlock(task_cont->lock);
12      unmask_interruption();
13      goto end; / function();
14  }
15  unlock(td->lock);
16  }
17  unlock(task_cont->lock);
18  unmask_interruption();
19  ...
```

Other patterns that can be recognized are presented in the next part of the work. Different forms of iteration with do-while, while loops can be used in the kernel as well and will need protection mechanisms if they manipulate data structures. For dynamic data structures that have a specific API that manipulates them we will have to make the functions atomic. We will explain the other patterns that are used in the next section, where we go through our contribution to the topic of adaptation of the safety-critical system: a semantic patch generator.

3.3 Generating semantic patches

A tool delivered with this work is a semantic path generator: this program is a shell script that generates sematic patches (used by Coccinelle) to protect data structures and the different patterns that manipulate them. In fact protection and synchronization mechanisms will be deployed in a very similar way for the different safety-critical systems implementations. Taking advantage of these similarities allow us to automate changes performed in the source code.

The iterative script will ask the developer how many data structures need to be protected. One file will be created for each data structure. After the user chooses the name for the file, the generator will need ask for general data that will be used throughout the whole generation process: the name of the locking and unlocking function (*e.g.* `SPIN_LOCK`, `lock_acquire`), the signature of the masking functions (*e.g.* `preempt_disable()`), the name of the synchronization field in the data structure that will be used (*e.g.* `spin`). The next step is to start choosing the patterns: currently the script can generate eight different protection rules for data structures, each rule can be generated more than once (*e.g.* sequential writes for the same data structure might be different at some parts of the kernel). Once the first semantic patch is generated, the shell script will proceed to create another file if the user wants to protect more than one data structure with the patterns.

Now that we have explained the different patterns that can be recognized in the source code of the RTOS' implementations we will go through the different results that we have obtained using the semantic patch generator and Coccinelle when we applied these tools to the seL4 and POK kernel.

PART VI

Validation of the contributions

This last part will explain the results obtained after using our adaptation method in the two kernels. For both implementations I went through the whole analysis: identifying the data structures and the modifications that will be needed (duplication, synchronization mechanisms), getting the call graph, identifying which functions and what files were performing changes in the data structures.

1 Traceability of changes

One of the main requirements in the validation process, is to have the ability to see where the changes are made in the source code. We do not want to apply a transformation rule in some file/function of the kernel that does not require any changes. To solve this problem we use the call graph: we can know which function has access to the variable that need changes. Once the files are identified, using the semantic patch generator will create the file for Coccinelle. Since `spatch` can be used easily in various files at the same time targeting the right files is not a problem.

In addition to the automatic changes, some manual modifications need to be performed: adding the multi-core support/initialization in the operating system is architecture specific and needs the direct intervention of engineers. Due to non-modularity of this code, our semantic patch generator does not create the booting code for a specific architecture. This could be something very interesting to provide, but in reality it takes a lot of time. Supporting a new architecture in an existing implementation would need more time and developers behind it.

Other manual changes will consist in adding the synchronization mechanism for each data structure. Having the synchronization mechanism in the data structure is the easiest method to assure that the function will have access to the right lock, using global ones could be error-prone and quite complicated, and like it was stated before, this method is used in other multi-core implementations in some RTOSes.

2 Results for the sempatch generator

For the two implementations we used the semantic patch generator to patch the data structures that are used in the `kernel` folder of each implementation. In those files is where we would like to assure parallelism and

Template name	POK	seL4
Multiple affectation	10	12
For 1	3	4
For 2	2	0
For 3	0	0
Switch	2	0
Switch return error	5	0
While	1	3
Do while	2	2
Atomic functions	0	12

Number of matches for every proposed template

In total for the POK kernel we have around 100 lines of code that were added by the semantic patches generated and 20 lines of code added by hand to duplicate the number of data structures (this takes place in the initialization function, where we introduce a new macro counting the number of cores). For the seL4 kernel we had around 120 insertions with the patterns.

3 Battery tests for POK

The final validation method to prove that our changes did not have an impact in the current implementation of POK would be to demonstrate that the kernel did not suffer any regression regarding *execution time*, *data's integrity*.

The kernel disposes of a battery test to check if any new feature implemented introduces regressions. Validating every test is the first step to check if the implementation did not break any mechanism or if any deadlocks were introduced. Additional tests need to be added as well, we need to make sure that true parallelism was obtained in the new implementation with system calls being able to run in parallel. Like it was stated before arriving to a complete multi-core version in one of the operating system studied would require a lot of time and a full team of developers taking care of different parts: validation of the implementation (very important for the seL4 kernel since the OS is formally proved and is correct against its implementation), code changes in the source code, architecture support and the external tools that use these kernels should be updated as well (Ocarina [5] and RAMSES [7] for code generation for POK).

A performance test needs to me performed as well to know if any time anomalies are introduced. This test will could also be used to estimate WCET, WCRT for each function call. However it will need external tools to calculate the time spent in each function.

With the state of the semantic patch generator I strongly believe that it can be extended and tuned to perform in many kernel implementations and that it is a viable option to apply pattern changes to protect shared data structures in a RTOS. The idea behind this contribution was to guide and help the developer in the porting procedure of the safety-critical system. While doing this we wanted to conserve as much as we could of the current implementation. In order to recycle code we use *duplication* if the data structures are binded to a duplicated hardware component (*e.g.* we need to create a new data structure for each core introduced), and *protection* of shared data structures.

PART VII

Conclusion

The main focus of this internship was to evaluate how multi-core architectures can affect the integrity of the current safety-critical systems and their operating system's implementation.

Off-the-shelf architectures are becoming dominant in the market and the real-time community would like take advantage for various reasons: on the execution level (improved performance due to the multiple execution units available, being capable of running various type of tasks in one multi-core processor, having backup units in the same chip, *etc*...). On the economy side there is an big interest as well: having off-the-shelf components are cheaper and more accessible than custom solutions, also deploying a multi-core platform instead of n single-core chips, will reduce the weight of the system (typically aircrafts deploy several processors) which can reduce costs significantly.

With the extensive literature review presented in this work, we showed what the main approaches and propositions of the real-time community are: **i)** adapt the current system implementations and make them multi-core capable without losing the robustness and the integrity of the code, or **ii)** start from scratch and try to overcome the difficulties behind multi-core architectures by proposing completely different algorithms/architectures than the ones used in single-core implementations. The second option however would need more effort than the first one and custom solutions could be very expensive in time and financially. Most mature systems available in the market now have been proved, tested, and were developed with a single-core perspective. All the effort that was dedicated to comply with strong standards and properties in the single-core implementations should be kept and assured in the multi-core implementation as well, for this reason we would like to follow the first approach and adapt the existing implementations conserving the properties already delivered.

Our main objective in adapting the existing RTOS implementation consisted in obtaining parallelism for system calls. The problem behind it is that system calls can share data, in particular data structures that represent either a hardware state or a logical state in the kernel. They will need to be accessed in mutual exclusion and this cannot be freely guaranteed with a multi-core implementation (it was a problem for uniprocessors but it becomes more complex with multiple execution units). To have a better idea of the problem and to be able to determine what changes are needed in the source we presented a language modeling the dependencies between clients (the new execution units that can request a given service of the kernel), the service that can be executed in parallel or not, and the data structures handled by each services representing either a pure logical state, a configuration of a single hardware confronted to the multiple execution units or representing duplicated hardware. The modeling allowed us to determine what data structures need to be added and in combination with the call graph what data structures need to be protected.

Having duplicated data structures that are not shared is ideal in a multi-core implementation because synchronization will not be needed at any time. In fact some multi-core adaptations suppose that no data can be shared between cores and that the kernel itself can be duplicated and deployed in each core. We estimate that this approach is not really satisfactory and that we would like to allow communication between CPUs with only one kernel. Therefore, we added synchronization/protection mechanisms and identified recognizable patterns that can be used in different operating systems. To assist engineers even further in the porting procedure a script generating sematic patches for Coccinelle was developed and can now detect nine

different protection patterns. The results presented are preliminary and I believe that they can be improved over time. Nonetheless they are relevant and show the possibility of automation in the porting procedure for a RTOS.

In conclusion, having a detailed modeling language and the right tools to transform and analyze code, can allow developers to have a precise idea of the changes required to obtain a multi-core safety-critical system, supporting true parallelism and complying to the same guarantees than the uniprocessor implementation.

References

[1] *Cheddar*. http://beru.univ-brest.fr/~singhoff/cheddar.

[2] *Coccinelle: A program matching and transformation tool for systems code*. http://coccinelle.lip6.fr/.

[3] *ERIKA Enterprise*. http://erika.tuxfamily.org/drupal/.

[4] *JabRef reference manager*. http://jabref.sourceforge.net/.

[5] *Ocarina*. http://www.openaadl.org/ocarina.html.

[6] *POK, a partitioned operating system*. http://pok.tuxfamily.org.

[7] *RAMSES*. http://penelope.enst.fr/aadl/wiki/Projects#RAMSES.

[8] *Safety and security related features in AUTOSAR*. http://www.automotive2010.de/programm/content_data/Bunzel-AUTOSAR.pdff.

[9] *SLR tool*. http://slrtool.org.

[10] *Specification OSEK OS 2.2.3*. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf.

[11] *Trampoline*. http://trampoline.rts-software.org/.

[12] *seL4*, aug 2014. http://dx.doi.org/10.5281/zenodo.11247.

[13] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr, *Basic concepts and taxonomy of dependable and secure computing*, IEEE Trans. Dependable Sec. Comput, 1 (2004), pp. 11–33.

[14] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, *An analysis of Linux scalability to many cores*, in 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings, R. H. Arpaci-Dusseau and B. Chen, eds., USENIX Association, 2010, pp. 1–16.

[15] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson, *Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?*, in IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, 2008, pp. 342–353.

[16] M.-I. Chen and K.-J. Lin, *Dynamic priority ceilings: A concurrency control protocol for real-time*, Real-Time Systems, 2 (1990), pp. 325–346.

[17] P. Chevochot and I. Puaut, *Experimental evaluation of the fail-silent behavior of a distributed real-time run-time support built from cots components*, in Dependable Systems and Networks, 2001. DSN 2001. International Conference on, IEEE, 2001, pp. 304–313.

[18] R. I. Davis and A. Burns, *A survey of hard real-time scheduling for multiprocessor systems*, ACM Comput. Surv, 43 (2011), p. 35.

[19] D. De Niz, K. Lakshmanan, and R. Rajkumar, *On the scheduling of mixed-criticality real-time task sets*, in Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE, IEEE, 2009, pp. 291–300.

[20] J. Delange and L. Lec, *POK, an ARINC653-compliant operating system released under the BSD license*, in 13th Real-Time Linux Workshop, 2011.

[21] G. Georgakoudis, D. S. Nikolopoulos, H. Vandierendonck, and S. Lalis, *Fast dynamic binary rewriting for flexible thread migration on shared-ISA heterogeneous MPSocs*, in ICSAMOS, IEEE, 2014, pp. 156–163.

[22] M. Gerdes, F. Kluge, T. Ungerer, C. Rochange, and P. Sainrat, *Time analysable synchronisation techniques for parallelised hard real-time applications*, in 2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012, W. Rosenstiel and L. Thiele, eds., IEEE, 2012, pp. 671–676.

[23] R. Hilbrich, J. R. van Kampenhout, and H.-J. Goltz, *Model-based generation of static schedules for safety critical multi-core systems in the avionics domain*, in Echtzeit, W. A. Halang, ed., Informatik Aktuell, Springer, 2011, pp. 29–38.

[24] X. Jean, D. Faura, M. Gatti, L. Pautet, and T. Robert, *Ensuring robust partitioning in multicore platforms for ima systems*, in Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st, IEEE, 2012, pp. 7A4–1.

[25] K. S. Khan, R. Kunz, J. Kleijnen, and G. Antes, *Five steps to conducting a systematic review*, Journal of the Royal Society of Medicine, 96 (2003), pp. 118–121.

[26] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, *seL4: Formal verification of an OS kernel*, in Proceedings of the 22nd Symposium on Operating Systems Principles (22nd SOSP'09), Operating Systems Review (OSR), Big Sky, MT, oct 2009, ACM SIGOPS, pp. 207–220.

[27] M. Panić, E. Quiñones, P. G. Zaykov, C. Hernandez, J. Abella, and F. J. Cazorla, *Parallel many-core avionics systems*, in Proceedings of the 14th International Conference on Embedded Software, EMSOFT '14, New York, NY, USA, 2014, ACM, pp. 26:1–26:10.

[28] P. Regnier, G. Lima, E. Massa, G. Levin, and S. A. Brandt, *RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor*, in RTSS, IEEE Computer Society, 2011, pp. 104–115.

[29] F. Reichenbach and A. Wold, *Multi-core technology – Next evolution step in safety critical systems for industrial applications?*, in 13th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD 2010, 1-3 September 2010, Lille, France, S. López, ed., IEEE Computer Society, 2010, pp. 339–346.

[30] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill, *Multi-core real-time scheduling for generalized parallel task models*, Real-Time Systems, 49 (2013), pp. 404–435.

[31] C. E. Salloum, M. Elshuber, O. Höftberger, H. Isakovic, and A. Wasicek, *The ACROSS MPSoC - A new generation of multi-core processors designed for safety-critical embedded systems*, Microprocessors and Microsystems - Embedded Hardware Design, 37 (2013), pp. 1020–1032.

[32] L. Sha, T. Abdelzaher, K. E. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok, *Real time scheduling theory : A historical perspective*, Real-Time Systems, 28 (2004), pp. 101–155.

[33] L. Sha, R. Rajkumar, and J. P. Lehoczky, *Priority inheritance protocols: An approach to real-time synchronization*, Computers, IEEE Transactions on, 39 (1990), pp. 1175–1185.

[34] A. Silberschatz, P. B. Gavin, and G. Gagne, *Operating Systems Concepts*, ninth edition ed., 2012.

[35] S. J. Upadhyaya, *Rollback recovery in real-time systems with dynamic constraints*, in Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International, IEEE, 1990, pp. 524–529.

[36] M. von Tessin, *Towards high-assurance multiprocessor virtualisation*, in VERIFY@IJCAR, M. Aderhold, S. Autexier, and H. Mantel, eds., vol. 3 of EPiC Series, EasyChair, 2010, pp. 110–125.

[37] J. Xu and B. Randell, *Roll-forward error recovery in embedded real-time systems*, in 1996 International Conference on Parallel and Distributed Systems (3rd ICPADS'96), Tokyo, June 1996, IEEE && IPSJ, pp. 414–421.

[38] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, *Memory-centric scheduling for multicore hard real-time systems*, Real-Time Systems, 48 (2012), pp. 681–715.

[39] Q. Zhao and Z. Gu, *A state-of-the-art survey on real-time issues in embedded systems virtualization*, Journal of Software Engineering and Applications, 05 (2012), pp. 277–290.