A Linux container-based architecure for partitioning real-time tasks sets on ARM multi-core processors.

Cédric Cazanove
IRIT - INP -ENSEEIHT
Toulouse, France

Frédéric Boniol
ONERA
Toulouse, France

Jérôme Ermont
IRIT - INP - ENSEEIHT
Toulouse, France

ABSTRACT

Temporal and space partitioning is the main principle of Integrated Modular Architectures found in automotive and avionic domain. This concept was theoretically proposed in 2000 by John Rushby, showing that strict isolation between safety-critical functions is a solution to guarantee real-time requirements and to offer modular capacities. This principle is today implemented by commercial hypervisor such as Xtratum, PikeOS, Asterios and VxWorks653. These hypervisors are often efficient and qualified for the most safety-critical levels. However, these commecrial solutions are often very expensive. Recently, a new open-source solution called RT-CASEs has been proposed by Cinque et al. This solution is based on Linux and on a container mecanism (Docker). It ensures partitioning properties. However, this solution requires the Linux patch RTAI, which is not compliant with Docker on ARM multi-core processors. This short paper proposes to adapt the RT-CASEs principles on the Xilinx MPSoC processor composed of 4 ARM A53 cores.

1. INTRODUCTION

Hardware (HW) and software (SW) systems play an increasingly important role in modern vehicles. They implement more and more safety-critical functions, and at the same time, they must be ever lighter and cheaper. To face the safety-critical constraints, automotive and avionic domains have developed the Integrated Modular Architecture (IMA) paradigm. IMA allows resource sharing and time partitioning between SW applications on single processors in order to ensure strict isolation, i.e., to ensure that a faulty functional or temporal behavior occurring in a given application does not affect the other applications although they share the same HW resources. For that purpose, the automotive and avionic domains have developed respectively the AUTOSAR [3] and ARINC653 [1] standard. To face the cost constraint, these two domains try to embed more and more multi-core processors to increase the on-board computing power and to reduce the number of embedded computers. Following these trends, the challenge is now to implement the IMA paradigm on multi-core processors.

To answer this challenge, several commercial solutions have been proposed. Most of them are based on proprietary virtualization approaches. PikeOS, Asterios and VxWorks653 are examples of such solutions. However, they are not available on all the possible processors targeted by the embedded domains, and adapting them to new processors could lead to high extra-costs.

To face the cost constraints, several works explore the use of Linux-based solutions in safety-critical embedded contexts. That is the case for the industrial companies Planet and SpaceX which chose Linux as operating system for some embedded computers in their space launchers. Similarly, NASA also experimented Linux for the navigation computer of the Ingenuity Mars helicopter [4]. One of the most powerful advantages of Linux is its re-usability. The open-source philosophy allows having a lot of libraries and drivers developed for Linux on different hardware. All these advantages make Linux useful for embedded systems.

However, combining the IMA principle with Linux on multi-core processors remains a difficult challenge. Recently, Cinque et al [2] have proposed a new open-source solution, called RT-CASEs, based on Linux and on a container mechanism (Docker). Docker ensures spatial partitioning. RT-CASEs uses the Linux patch RTAI to monitor and control the real-time behavior of the SW applications. However, RTAI is not compliant with Docker on ARM multi-core processors (which are now widely used in the embedded domains).

The objective of this short paper is to explore another Linux-based architecture, similar to the RT-CASEs solution, for multi-core ARM processors. The proposed approach is based on the PREEMPT-RT patch to take into account real-time requirements, and on Docker to provide spatial isolation capabilities. We have experimented our solution on the Xilinx UltraScale+ processor (composed of 4 ARM Cortex A53 cores).

2. RELATED WORK

The principle of IMA has been introduced in 2000 by J. Rushby [6]. He laid the groundwork of SW partitioning showing that a strict isolation is a solution for achieving safety-critical requirements. Following that seminal work, several proposition have been done for implementing the IMA principle. Most of them are based on virtualization mechanisms. Xtratum, PikeOS and ASTERIOS belongs to this familiy. They can be considered as type 1 hypervisors (as they are running directly on the HW layer). The main advantage of a type 1 hypervisor is that it can execute with a higher security level on the hardware. It creates partitions and allocates time and memory to each partition. A partition contains either an operating system or a bare-metal application. To reach full isolation, a hypervisor can reset all the memory-related caches (L1, L2, L3, TLB, ...) during each partition switch. This ensures that an application in partition A cannot access data of partition B that would have stayed in the caches. However, this solution increases significantly the partition switch time and does not easily hold on multicore processors. One drawback of hypervisors is that the scheduling is mostly static between partitions. The hypervisor scheduler has to schedule partitions and not processes, it does not have much information on the needs of the partitions. In particular, a partition that has nothing to execute at a moment will still be scheduled during its reserved time. Another drawback of current commercial hypervisors is that most of them are proprietary frameworks.

Other solutions have been proposed based on the notion of containerization. A container is another type of virtualization mechanism. The difference between a container and a virtual machine is that a container only virtualizes SW layers including operating system services and not HW. The isolation provided by a container is not as strong as the one provided by a hypervisor since the code is running on the same HW and with the same level of execution. First of all, a container does not have its scheduler and thus it cannot control what happens during a context switch. Second of all, the applications inside a container are executed on the same operating system. However, a container provides virtualization of operating system services and the different processes cannot interact with other processes through these services.

Following the containerization idea, the authors of [2] proposes RT-CASEs as a useful alternative to virtualized partitioned systems. The solution developed in RT-CASEs is to allocate safety-critical applications into containers, characterized by stringent timeliness and reliability requirements, and to allow these containers to cohabit with non real-time containers on the same HW. The approach requires the Linux RTAI patch to use specific services (provided by RTAI) to control the execution time of each task. The experiments done in [2] seem to show that RT-CASEs can be seen as a lightweight open-source solution. However, the RTAI Linux patch does not work on ARM processors, making RT-CASEs no more available on ARM platforms.

To face this last difficulty, the the Linux PREEMPT-RT patch can be used. It has been created to adapt the kernel to a real-time context [5]. This patch is actually merging into the upstream Linux stable version. The important aspect of the Linux PREEMPT-RT patch is the preemption model. In the mainline kernel, plenty of the code is non-preemptible which can delay the tasks execution time. To reduce the impact of this, it is possible to make the kernel in a fully preemptible mode. It means that all kernel code is preemptible (except for a few critical parts).

In this paper we extend the RT-CASEs approach with PREMPT-RT (instead of RTAI) on a multi-core processor.

3. APPROACH

3.1 Application model

We assume a system composed of SW applications, each application is characterized by a priority level and is composed of periodic SW tasks. We suppose the applications are independent, that is, there is no communication and synchronization constraints between applications. So they can run in parallel.

Definition 1 (System). *A system S is a set of SW applications possibly running in parallel:*

$$S = A_1 \parallel \ldots \parallel A_n$$

Definition 2 (Application). *A SW application is characterized by a priority level and is composed of a set of periodic tasks. Let $A_i \in S$:*

$$A_i = < prio, Tasks >$$

with

- $A_i.prio \in \textsc{Nat}$ *is the priority level of* A_i,
- $\forall i \neq j, A_i.prio \neq A_j.prio$ *(two different applications have different priority levels),*
- $n_i = card(A_i.Tasks)$ *is the number of tasks of the application,*
- *and* $A_i.Tasks = \{\tau_i^1, \ldots, \tau_i^{n_i}\}$ *is the set of SW tasks running in* A_i.

The execution order of the tasks inside the application scope is defined by two attributes: a priority and a period.

Definition 3 (Task). *A SW tasks is characterized by a priority level and a period. Let $\tau_i^j \in A_i.Tasks$:*

$$\tau_i^j = < prio, P >$$

where

- $\tau_i^j.prio$ *is the local priority level of* τ_i^j *w.r.t the others tasks of* A_i,
- $\forall j \neq k, \tau_i^j.prio \neq \tau_i^k.prio$ *(two different tasks have different local priorities),*
- *and* $\tau_i^j.P$ *is the period of* τ_i^j.

Note that this definition assumes that the underlying scheduling strategy is based on a static priority policy. The second observation is that we have only associated a task with its local priority inside the application. We have to extend this definition to define the global priority model, that is, the priority order between tasks belonging to different applications. For that purpose, we follow the choice made by Cinque et al in [2].

Definition 4 (Global priority model). *Let S a system, let two applications $A_i, A_j \in S$ and let two tasks $\tau_i^k \in A_i.Tasks$ and $\tau_j^l \in A_j.Tasks$. The global priority order is a function Prio such that:*

$$Prio(\tau_i^k) < Prio(\tau_i^l) \Leftrightarrow \begin{cases} i = j \text{ and } \tau_i^k.prio < \tau_j^l.prio \\ i \neq j \text{ and } A_i.prio < A_j.prio \end{cases}$$

In other terms, tasks belonging to high criticality applications have a higher priority than tasks belonging to low criticality applications, and tasks belonging to the same application are ordered by the local priority order.

3.2 Static architecture

Applications are the partitioning units. That is, we want to execute them in such a way that faulty executions in an application (in one or several tasks of the application) can not cause faulty executions in other applications.

For that purpose, we propose the architecture principle depicted in Figure 1. Let us consider a system $S = A_1 \parallel \ldots \parallel A_n$. Let suppose that S run on a multi-core processor composed of N cores ($N > 1$). The main ideas are the following:

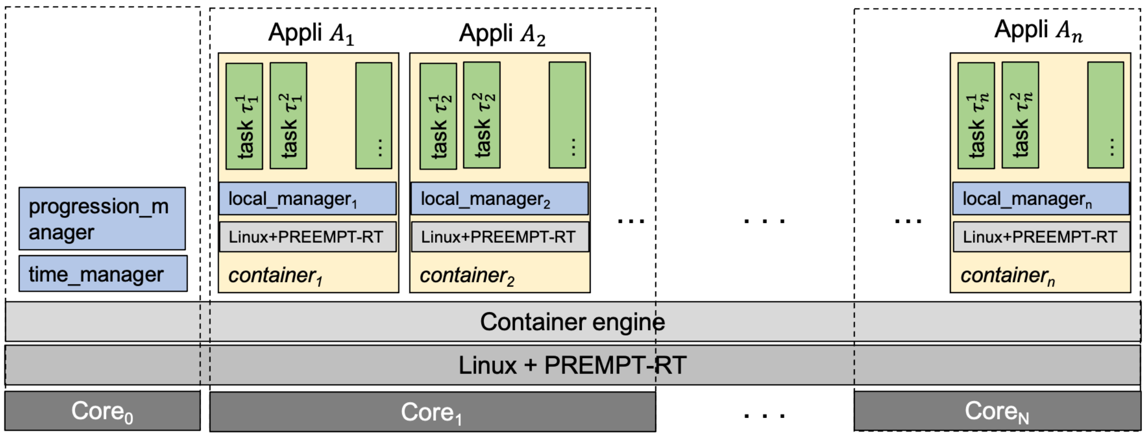

Figure 1: High-level architecture

- The whole platform is managed in a SMP way (Symmetric Multi Processor), that is, all the cores (and the processes running on them) are managed by a global Linux system.

- To allow real-time execution, the global Linux system is extended with the PREEMPT-RT patch, which provides real-time preemptive scheduling capabilities.

- Each application A_i is associated with a dedicated container (called $container_i$ in the Figure) provided by a container engine. All the tasks τ_i^j of application A_i run in the same $container_i$. Conversely, two tasks from two different applications run in two different containers.

- The content of each container is managed by a local lightweight Linux version. As in [2], the "containerization" of the applications allows spatial partitioning. It ensures that different applications do not share the same services and the same memory space. This spatial partitioning prevents thus fault propagation through service calls and shared addresses.

- The HW cores 1 to N of the multi-core processors host the SW applications, while core 0 is dedicated to the monitor process. Reserving the core 0 allows the monitoring process (that is, the process that controls the execution of each application) to be physically isolated from the SW applications, and thus to not suffer from interferences by the applicative part.

- Each application A_i (and its associated container) is statically allocated to a single core (from 1 to N). There is no migration between the cores (static allocation), and there is no parallelism inside an application. However, several applications can be hosted by the same core.

- The monitoring part is divided into three threads: two running on core 0 (*progession* and *time manager*) and one running in each applicative container. These thread are in blue in Figure 1.

3.3 Execution rules

As such, each core (from 1 to N) hosts several tasks possibly belonging to different applications. The execution of these tasks is managed under the preemptive fixed priority scheduling method allowed by PREEMPT-RT (the task ready for execution with the highest priority is elected by the scheduler). The question is then how to ensure timing isolation between tasks from different applications. As mentioned in the introduction, commercial solutions such as PikeOS, Xtratum, and Asterios associate each task (or each application) with a static time window. At the end of the time window, the tasks associated with it are suspended to launch the tasks of the next time window. These solutions require the definition of a time frame for each core, which must be compliant with the applicative requirements. Designing such a time frame is a difficult job, particularly when applications change.

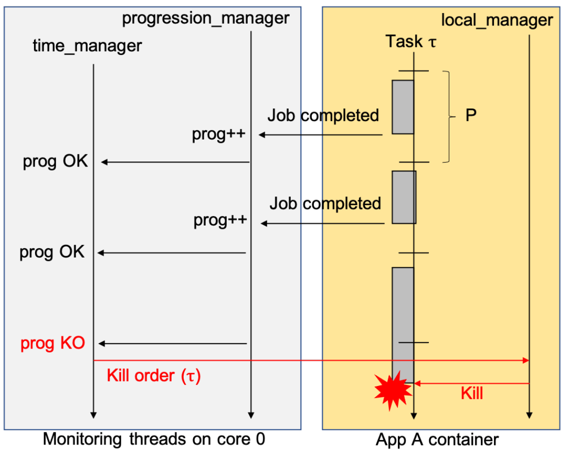

Figure 2: Execution rules

In our approach we choose a more flexible solution, based on three rules:

1. Tasks are scheduled on each core according to the global priority function *Prio* (definition 4).

2. The thread *progression_manager* (core 0, belonging to the monitoring part) stores the progression counter of each tasks at each instant. The progression of a task at a time t is the number of executions it has completed at this time. Let us consider a task τ of period P. Let us consider a time t and let us consider an integer k such that $kP \leq t < (k+1)P$. In normal case, the progression of τ, denoted $prog(\tau, t)$ can be:

   - $prog(\tau, t) = k$ if the current job of τ (i.e., the one which must start at kP) is not completed,
   - $prog(\tau, t) = k + 1$ otherwise.

   Concretely, each time a task completes its execution (that is, its current job), it sends a message through a dedicated queue to *progression manager*, which increments the progression counter of τ.

3. Consequently, for each task τ of period P a normal progression at time t is characterized by:

$$prog(\tau, t) - 1 \leq \lfloor \frac{t}{P} \rfloor \leq prog(\tau, t) \qquad (1)$$

   At each time unit, *time_manager* (core 0) checks the condition 1 for each task τ. If the condition is met, then the schedule seems to correct. Else, the task is late, meaning that it may steal computing resources to other applications. In that case, *time manager* sends a message to the *local manager* in charge of the task to kill it.

These rules are illustrated in Figure 2.

3.4 Discussion: IMA or not IMA?

The consequences of the previous rules are:

- If the system S is schedulable, that is, if all tasks complete before the end of their period, then *time manager* does not send any kill order.
- If for some reasons the system becomes no more schedulable, then *time manager* will kill the delayed tasks.

However, it is important to note that this sanction does not implement the IMA principle presented in the introduction. Indeed, the task which has been killed is not necessarily the faulty task. It could be a victim task that has been delayed because a task with a higher priority lasted more than its WCET. In that case, the faulty task is the task with the high priority and the fault cannot be seen by the *time manager* because the faulty task completes its job before the end of its period.

As a consequence, the three previous rules are not sufficient. To implement the IMA principle it is necessary to implement a more precise monitor. This monitor must be based on the worst-case scenario encountered by each task (based on the WCET of each task). As there is no migration and no dependency between the tasks, there is no timing anomaly. That means that if at time t a task τ is the first one to exceed the predicted worst-case scenario (for this task), then τ is the first faulty task, and a sanction can be applied on it.

App & Container	Core	Tasks		
		id	prio	period (s)
0	Core 1	τ_1	82	5
		τ_2	87	10
1	Core 1	τ_3	88	15
2	Core 2	τ_4	83	5

Table 1: Configuration of our first experiment

Tasks	Comput. time (s)	Progression
t1	0.4	OK
t2	0.4	OK
t3	0.4	OK
t4	0.4	OK
t1	3	KO
t2	4	KO
t3	1	OK
t4	1	OK

Table 2: Execution results

4. FIRST EXPERIMENT

As this work is still in progress, we have implemented the architecture in Figure 1 with the three rules in section 3.3 (and not the full IMA rules discussed above). We targeted the HW platform Zynq Ultrascale+ made by Xilinx which is one of the COTS hardware platforms considered by the aeronautic and space industry. This processing unit involved in this platform is composed of four ARM Cortex-A53 cores with a frequency going up to 1.5 GHz. Each container implements Alpine Linux 3.17.3. And the global Linux for the four cores is from the Xilinx Yocto branch for the UltraScale+ processor.

The implementation is composed of 3 containers and 4 tasks. The configuration is given in Table 1. We performed 2 simple tests. The first one considers a schedulable system, with a computational duration small compared to the period duration. All tasks respect the progression constraint (1). By increasing the execution time, the second test shows that tasks with smaller periods do not respect the progression constraint. The calculation of the core load confirms the unfavorable situation. But the progression problems occur on the small period tasks for which a higher priority has been allocated (in a RM way). This behavior should be investigated.

5. CONCLUSION AND FURTHER WORKS

This work is a first attempt to reproduce the RT-CASEs principles on an ARM-based processor. Like RT-CASEs we use Docker. However, we develop another monitor strategy based on PREEMPT-RT and on the architecture depicted in Figure 1. The next work will be to implement the IMA rules discussed in section 3.4. We think that we will face two difficulties: the first one is to generate the expected worst-case scheduling of each task and to memorize it in the *time manager* to decide to send a sanction to a faulty task. The second expected difficulty will be about the required precision of time to be able to sanction fastly enough a faulty task in order to protect the other applications.

6. REFERENCES

[1] Aeronautical Radio Inc. ARINC Specification 653 P1-3. Avionics application software standard interface: Required services, 2013.

[2] Marcello Cinque, Raffaele Della Corte, Antonio Eliso, and Antonio Pecchia. RT-CASEs: Container-Based Virtualization for Temporally Separated Mixed-Criticality Task Sets. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:22, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[3] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. AUTOSAR–A Worldwide Standard is on the Road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*. Citeseer, 2009.

[4] Håvard Grip, Johnny Lam, David Bayard, Dylan Conway, Gurkirpal Singh, Roland Brockers, Jeff Delaune, Larry Matthies, Carlos Malpica, Travis Brown, Abhinandan Jain, Alejandro Martin, and Gene Merewether. Flight control system for nasa's mars helicopter. In *AIAA Scitech 2019 Forum*, 01 2019.

[5] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The Real-Time Linux Kernel: A Survey on PREEMPT_RT. *ACM Computing Surveys*, 52(1):1–36, feb 2019.

[6] John Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB, 2000.