Method and Tools for Mixed-Criticality Real-Time Applications within PharOS

Matthieu Lemerre, Emmanuel Ohayon, Damien Chabrol, Mathieu Jan, Marie-Benedicte Jacques

Method and tools for mixed-criticality real-time applications within *PharOS*

Matthieu Lemerre, Emmanuel Ohayon, Damien Chabrol, Mathieu Jan, Marie-Bénédicte Jacques
CEA, LIST, Embedded Real Time Systems Laboratory
Point Courrier 94, Gif-sur-Yvette, F-91191 France
Email: firstname.lastname@cea.fr

***Abstract*—This paper provides an overview of some principles and mechanisms to securely operate mixed-criticality real-time systems on embedded platforms. Those principles are illustrated with *PharOS*, a complete set of tools to design, implement and execute real-time systems on automotive embedded platforms.**

The keystone of this approach is a dynamic time-triggered methodology that supports full temporal isolation without wasting CPU time. In addition, memory isolation is handled through automatic off-line generation of fine-grained memory protection tables used at runtime. These isolation mechanisms are building blocks for the support of mixed-criticality applications. Several extensions have been brought to this model to expand the support for mixed-criticality within the system. These extensions feature fault recovery, support for the cohabitation of event-triggered with time-triggered tasks and paravirtualization of other operating systems. The contribution of this paper is to provide a high-level description of these extensions, along with an analysis of their impact on the global system safety, in particular on the determinism property of the *PharOS* model.

I. Introduction

Over the last decades, the number of electronic equipments embedded in complex systems such as automotive vehicles or airplanes has increased in drastic proportions. At first, system architects used to embed one computing unit per functional system, which resulted in large distributed and heterogeneous real-time systems. The current number of equipments makes this approach no more sustainable, mainly due to its high costs. As a result the trend is to integrate several (possibly unrelated) applications on fewer computing units, resulting in what is called a mixed-criticality system.

This integration raises however a number of issues, such as:

- efficiently and securely sharing resources (and especially CPU time) between these applications;
- providing the same degree of isolation than achieved when using separate computing units;
- combining different real-time paradigms and APIs;
- handling multicore computing units, that are becoming the usual solution of the chip designers to increase performances.

This paper presents the formal and/or technical answers we provided to each of these questions. Each proposed solution is illustrated through *PharOS*, the system we have built following these methods and that addresses all the problems hereabove exposed.

PharOS features a rich model of tasks that accurately describes the task constraints, and admits an optimal dynamic scheduling algorithm. Moreover, it allows exact feasibility analysis, which avoids wasting CPU time by oversizing the time resources for the tasks. Its tool suite allows for automatic fine-grained memory partitioning, if the underlying platform comes with a Memory Protection Unit (MPU). At last, some additional extensions allow compliance with other real-time paradigms, or paravirtualization of other existing RTOSes and APIs, while taking advantage of the additional power provided by multicore computing units.

The article is divided as follows: Section II presents the *PharOS* formal task model. Section III proposes implementation mechanisms for temporal and spatial isolation, fault containment and recovery. Section IV focuses on cohabitation with the event-triggered paradigm and with another Operating System. In particular, we present here a paravirtualization use-case of *Trampoline* [1], a RTOS with an OSEK API, over *PharOS*. Section V presents related work, and Section VI concludes and suggest directions for future research.

II. Time-Constrained Automata (TCA): the task model for *PharOS*

At the heart of *PharOS* is a formal model of task, the time-constrained automata model [2], which is based on time. This section gives a short overview of this model and its implementation, which is necessary to understand the design of the temporal isolation in *PharOS*.

A. Short introduction to the TCA model

A *PharOS* application is composed of a set of tasks. The TCA model is a formal specification of all the possible temporal behaviors for each task. This specification is expressed using a directed graph, where arcs represent the successive jobs to be executed (one at a time), and the nodes bear the temporal constraints on the task. There are only four kinds of nodes:

- an `after(d)` (written $\blacktriangleright_d$) node constrains the execution of the following jobs to start after date d;
- a `before(d)` (written $\blacktriangleleft_d$) node constrains the execution of the preceding jobs to end before date d;
- an `advance(d)` (written $\blacklozenge_d$) node is a combination of an $\blacktriangleright_d$ and a $\blacktriangleleft_d$ nodes;

- a `no-constraint` (written ◉) node imposes no temporal constraints.

All constraint dates are expressed relatively as an increment to the previous $\triangleright_d$ or $\diamond_d$ node (i.e. $\triangleleft_d$ nodes do not change the "reference date"). Nodes with multiple outgoing arcs represent conditional execution (e.g. `if` statements).

Execution of an application can be seen as choosing one (possibly infinite) walk in the graph of each task, and choosing for each encountered job the intervals of time where it is actually executed. The execution is correct if, and only if:

- successive jobs of a task execute in order;
- the execution intervals for each job fulfill the timing constraints of all nodes.

B. An example using ΨC

The graph for each task need not be written manually: it can be automatically extracted from the control flow graph of an application. The language used to implement *PharOS* applications is ΨC, a variant of the C language with extensions for timing constraints. Figure 1 gives an example of a code excerpt for a task with the corresponding time-constrained automaton. Timing constraints statements are transformed into timing constraints nodes.

```
while(1) {a:
  after(1);
  b:
  if(...) {h:
    after(1);
   i: for(...)
       { advance(1); }
    }
  else {c:
    before(2);
    d: }
  e:
  advance(4);
  f:
}
```

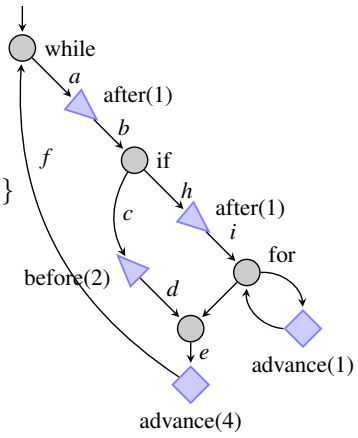

Fig. 1. A ψC code excerpt and corresponding automaton. An arc of label l represents the block of code to which label l belongs.

The beginning of a possible correct execution for this example is represented on Fig. 2. The code executes the `while` loop twice, and chooses the "else" part the first time, the "then" part the second time. The upper part of the figure represents a timescale, with the timing constraints and their date (expressed absolutely), and the blocks of code that are executed between these blocks. The lower part represents execution as a Gantt diagram. Execution is correct because blocks execute in order, and all timing constraints are fulfilled.

C. Implementation in PharOS

PharOS implements the previously described TCA formal model.

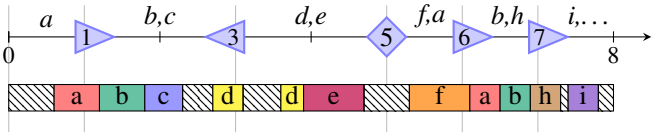

Fig. 2. Example of a correct execution for the automaton in Fig. 1.

1) ΨC compilation and scheduling: The automaton for each task is extracted from the ΨC code, and used for different timing analysis phases[1]. Eventually, ΨC code is compiled as regular C code, where timing constraints statement have automatically been transformed into system calls to the scheduler. Each system call indicates to the kernel the node that has been reached. The node contains the new start time, deadline and timing budget of the task, which the kernel uses to update the task parameters. These informations are used to implement the scheduling algorithm, which is a variant of EDF. Details can be found in [2].

2) Communication mechanisms: In addition to temporal constraints, ΨC provides different communication primitives. These primitives take advantage of the timing constraints to provide properties such as determinism, but do not have any impact on the TCA model (for instance, as they are non-blocking, they do not impact TCA scheduling).

In general, the actual *PharOS* implementation strictly follows the formal model, which does not impede it from addressing real-world applications.

III. Isolation mechanisms in *PharOS*

Isolation is the set of mechanisms that ensures that a misbehaving task cannot alter the execution of other tasks. In this section, we introduce the spatial and temporal isolation mechanisms for a safety-oriented implementation of the TCA model described in section II.

A. Determinism

An important principle for fault isolation is *determinism*. It means that the application behavior is independent of the scheduling or the execution time of the tasks (as long as timing constraints are met). It implies that variations in execution time of the tasks cannot impact the behavior of other tasks, and thus contributes to temporal faults isolation. In *PharOS*, memory segmentation imposes that all communications happen only through dedicated mechanisms that enforce determinism. This makes impossible for an application developer to write a non-deterministic application. Whatever the application, its behavior is always fully reproducible, even if spatial or temporal faults do happen.

1) Principles: Nondeterminism must be prevented in two ways:

- by complying with the *observability principle*: a message is formally sent only when the $\triangleleft_d$ node following the sending is reached, and can be formally received only

[1]In the actual version of *PharOS*, there are no $\triangleright_d$ nodes, only $\triangleleft_d$ and $\diamond_d$ nodes are allowed.

when the receiver has passed a $\blacktriangleright_{d'}$ with $d' > d$. This condition ensures timing consistency, i.e. that a message can be received only *after* it was formally sent.
- by ensuring that the order in which messages are received is independent of the actual date when the messages were sent (a shared FIFO being a typical *counter*-example). This property can only be reached with a careful implementation of the communication primitives.

2) Design and implementation: PharOS offers several communication primitives, such as temporal variables (i.e. periodic data-flow) or messages. More examples are given in Section IV-A3. Communications between tasks in *PharOS* follow a strict observability principle: only temporally visible data can be used in a code block, and already visible data can not be modified.

B. Spatial isolation and fault containment

1) Principles: Spatial isolation protects tasks against unauthorized accesses to code or data, either by the other tasks, by the system (i.e. the kernel) or even by the task itself. In our approach, an application is made of a set of tasks that are statically described. Tasks are protection units whose external communications are statically and entirely defined. Based on this static knowledge, and using the available hardware for memory protection (Memory Protection Unit, MPU, or Memory Management Unit, MMU) and privilege protection (CPU protection levels), the general principles to achieve spatial isolation are:

1) Separate the application into a set of memory contexts, following the separation of privilege principle [3]. It is more secure to have as many different memory contexts as possible. In particular, it is better to have more than one memory context per task, that changes depending on the actions it performs.
2) Generate for each memory context a constant table of the memory rights. Off-line generation of memory tables makes the kernel much simpler, and thus more secure. The generation is performed during the linking stage. Moreover tables should have as few rights as possible, following the least privilege principle [3].
3) Use access control checks and memory protection to control communication between tasks. Only tasks explicitly allowed to communicate should be able to do so, following the closed-system [4] and complete mediation [3] principles. Also, applications should be able to communicate only through the dedicated deterministic mechanisms (recall Section III-A).
4) Switch the memory rights when memory context changes (task switch, or switching of a memory context inside of a task). Memory rights switching should be performed by the kernel and the kernel only – this can be ensured with CPU protection levels.

Note that memory segmentation serves two separate purposes: spatial isolation and fault detection. For instance, forbidding write access to the stack of another task is spatial isolation and fault detection, while forbidding read access to another task is only fault detection. When coping with hardware limitations (e.g. limited number of MPU descriptors), early fault detection may be sacrificed to help providing full spatial isolation, as we are about to see in the next section.

2) Design and implementation:

a) Segments and layers: Our TCA-based system is made of three layers. The user layer hosts the functional code of the tasks. The system layer implements the communication mechanisms between tasks, as well as the management of state transitions within each task. Finally, the microkernel manages the time and the scheduling of tasks. Put together, the system layer and the microkernel make the kernel.

For these different layers, the memory is separated into segments with necessary and sufficient access rights depending on the execution context. That is for each physical memory segment, the memory access rights are defined for the following 4 contexts:

- the user layer;
- the system layer of the task that "owns" the segment, i.e. that may be allowed to write to it (there is at most one writer per memory segment);
- the system layer of the other tasks (multiple readers can be authorized);
- the microkernel.

This partitioning of each execution context is performed off-line by the toolchain of our system.

In the most general case, $2n + 1$ execution contexts are defined in an application made of n tasks[2], each context defining the access rights for several memory segments. This allows detection and confinement of operational anomalies (e.g. referencing fault) between the tasks, between a task and the kernel, inside the kernel and inside the microkernel.

Note that a memory address translation mechanism is not necessary to implement the principles exposed here. In other words, a MPU is sufficient, as long as it allows defining enough memory segments to partition a memory context. Indeed, the grain of the final segmentation depends ultimately on the flexibility of the memory protection hardware. When the MPU device defines only few segment descriptors (typically 8, on our *PharOS* experimentation platform), different memory segments can only be protected by common descriptors. For instance, code segments of different tasks are protected by a single descriptor, with therefore the same read-only memory rights for all tasks. In every case however, we never give write access to read-only memory segments. Thus isolation is fully achieved, and only fault detection is weakened.

C. Temporal isolation

1) Principles: Temporal isolation protects tasks against timing problems. In the case of a real-time system, this protection ensures that a task cannot affect the fulfilling of timing constraints (start times and deadlines) of other tasks. For instance, in a classical priority-based real-time system

[2]Per task, one context for its user layer and one context for its system layer, plus one dedicated context for the microkernel.

without temporal isolation, an infinite loop in a high priority task can deny CPU to less priority tasks, which will miss their deadlines.

The general principles to achieve temporal isolation are:

1) Give a timing budget to each task or group of tasks, that is replenished at some predefined points;
2) Ensure that given the timing budgets, all critical tasks will meet their deadlines in every possible case (ensure this for non-critical tasks if possible);
3) Ensure that a task or group of tasks cannot execute for longer than defined by its timing budget;
4) Ensure that scheduling cannot be outside the allowed cases.

Static scheduling as used in [5], is the simplest example of technique that ensures isolation. But it is also feasible to combine temporal isolation with dynamic scheduling.

2) Design and implementation:

a) Definition of timing budgets: We attribute a timing budget to every block of code between any two successive temporal constraints. Replenish occurs when the code executes one of the system call corresponding to a node. For instance on Fig. 1, there would be a budget for $b; h$, one for $b; c$, one for $f; a$, etc.

b) Feasibility analysis: We can perform an off-line feasibility analysis step using the automata of the tasks and their timing budgets. First, a synchronous product of the automata is computed, that shows every possible configuration of simultaneously executing blocks of code among all tasks. From this information, a linear programming problem is extracted. Solving it tells whether every task can reach its deadline with the given initial timing budgets. Details on feasibility analysis in *PharOS* can be found in [6], [7].

Note that feasibility analysis "only" provides the assurance that all *deadlines* of the systems are met, i.e. contributes to the security of the global application. Tasks can still lack some time to terminate execution within the given *timing budgets*. To avoid this situation, timing budgets should be set to an upper bound of the worst-case execution time (WCET) of the corresponding block of sequential code. But WCET analysis is outside of the scope of this paper.

c) Optimal scheduling: The microkernel is considered as a trusted piece of software, whose main role is to execute the scheduler. The scheduling algorithm it implements is a variant of EDF for time-constrained automata. We proved this algorithm to be optimal [2]. Optimality means that the proof of feasibility of a set of tasks ensures that EDF will always schedule tasks so that they meet their deadline (provided that they execute for no longer than their allocated timing budgets). We also have prototyped optimal scheduling algorithms for multiprocessor [7].

d) Timing budgets and deadlines monitoring: The microkernel also monitors timing budgets online using a hardware timer: whenever a task switch occurs, a hardware timer is set to the value of the remaining budget. Upon preemption, the budget is decreased by the amount of time spent executing the task, and the hardware timer is set to monitor the new task. Thus, the timer is triggered only when a task exhausts its timing budget.

A similar mechanism is used to monitor deadline overruns. This monitoring is not strictly necessary, since timing budgets are monitored, and they should have been checked against feasibility analysis. However, it is used as a defensive programming measure.

e) Node transitions monitoring: When a system call is made because a node has been reached, the system layer checks if the transition between the previous node and the current one is allowed. This design prevents a task from following a forbidden path on its execution graph. Indeed, should such an erroneous case go undetected, the set of currently executing blocks for all the tasks would be different from the ones checked by the synchronous product. This design also protects the microkernel from system calls flooding.

Furthermore, the communication design with statically limited sending/reception rate prevents tasks from flooding each other. The set of defensive programming mechanisms implemented by the kernel ensures that undetected denial of service attack can never occur within applications.

f) Non-blocking design: A task can affect scheduling only in two ways:

- by the path choosed on conditional nodes;
- by the execution time used to reach a temporal node.

There is no other way for a task to affect scheduling, for instance by yielding the CPU between nodes by blocking on a semaphore[3]. Combined with node transition and timing budget monitoring, this ensures that scheduling always happens as foreseen by the synchronous product.

g) Summary: how temporal isolation principles are applied: PharOS is a strict implementation of the temporal isolation principles based on timing budgets that we gave previously:

- Step III-C2b and III-C2c ensure that all critical tasks will meet their deadlines if, and only if, scheduling follows the cases anticipated in the feasibility analysis and timing budget is sufficient;
- Step III-C2d, III-C2e and III-C2f ensure that scheduling always happens as expected by the feasibility analysis, and that tasks cannot execute for longer than specified by their timing budgets;
- Ensuring that timing budgets for critical tasks are sufficient has to be proved separately, and is outside the scope of temporal isolation.

D. Group of tasks and error recovery

PharOS incorporates the concept of "groups of tasks", which is a first step towards mixed-criticality.

Tasks are partitioned into several groups. Local failure of one task (be it timing budget overrun, segmentation fault, illegal instruction, etc.) makes every task in its group to be stopped and recovered by restarting at some predefined point

[3]Actually, the only provided mean of synchronization in the TCA model is time: *PharOS* provides no semaphores or mutexes whatsoever.

later in time. Failure of a task has no impact on the behavior of other tasks belonging to other groups. The time needed for recovery can be modeled as special transitions in each task graph, and can be accounted for in the feasibility analysis.

Closely-related tasks should be put together in a group. A typical example are tasks that concur together to provide a single high-level functionality, and for which there is no advantage to allow them to fail individually. All tasks in a group are guaranteed to fail and restart consistently (e.g. without time shifts).

Groups of tasks and error recovery, on top of the previous isolation mechanisms, allow for a first level of mixed-criticality. Indeed, if non-critical tasks (i.e. tasks likely to fail) and critical tasks are separated in different groups, the failure of a "non-critical group" would have no incidence on a "critical group".

IV. Cohabitation with other execution models

A. Cohabitation of time-driven and event-driven applications

In embedded environment such as automotive domain, many activities (e.g. signal capture) require very tight timing constraints. Hardware performance does not allow addressing such activities with time-constrained automata. Therefore, *PharOS* allows coexistence of time- and event-driven tasks.

In such systems, two execution paradigms are active at the same time:

- the time-triggered paradigm where the behavior of the tasks (called TT tasks) is specified according to time-constrained automata;
- the event-triggered paradigm where the behavior of the tasks (called ET tasks) and their activation depends on the occurrence of the associated events (I/O interrupts).

In a system that provides both paradigms, an important point is to ensure noninterference among them while preserving the characteristics of the previously described model: safety, dependability, temporal determinism, spatial and temporal isolation.

PharOS addresses both time-triggered and event-triggered paradigms on dual core architectures. To reach the previously described objectives *PharOS* provides some isolation mechanisms that ensure safe communications between ET and TT tasks, with accurate temporal and spatial isolation. Most of these properties are automatically extracted from software design and automatically generated for the runtime.

1) Spatial partitioning and isolation: The partitioning mechanisms previously described are reinforced in order to maintain protection between the ET and TT domains. Figure 3 gives a graphical description of the memory protection mechanism.

Moreover, as *PharOS* can run on dual-core hardware architectures, a high degree of spatial and temporal isolation between the TT and ET domains is achieved by running TT tasks on one core and ET tasks on the other core. Nevertheless, mixed execution of ET and TT on a same core is currently under study. In addition, multi-cores architectures are also supported for TT tasks and TT Tasks are dynamically assigned

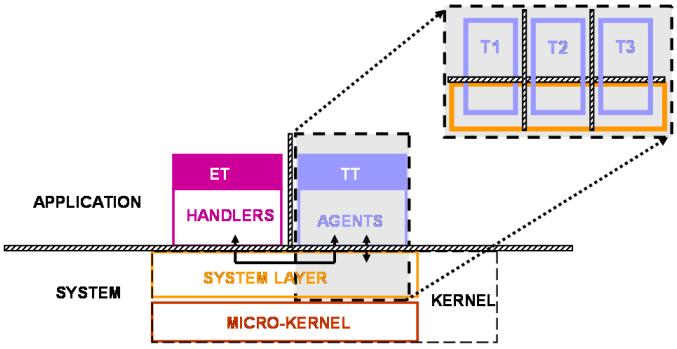

Fig. 3. Spatial isolation between ET and TT domains

to available cores at runtime. TT and ET tasks could therefore also be active at the same time on multi-cores architectures (with more than 2 cores), even though this has not yet been performed.

2) Temporal partitioning and isolation: We have seen in section III-C that all *PharOS* TT tasks are carefully monitored during execution. For the event-triggered tasks, the temporal control is performed by an interrupt configuration monitoring set by the following parameters:

- a time interval Δ_0;
- a maximum number N of authorized interrupt occurrences during the interval Δ_0.

When the number N is reached for a given source of interrupt, any new attempt to handle the same interrupt is detected by the system. Then, the corresponding interrupt source is disabled. As a consequence, the whole *PharOS* application cannot be disturbed by a temporal overflow of the ET domain. Note that N and Δ_0 are specification parameters of the behavior of event-triggered tasks, i.e an ET task is considered to be correct if and only if its number of occurrence during Δ_0 is equal or less than N. Therefore this mechanism protects the software against external or hardware errors, and cause no harm when the system behaves normally.

3) Communications: ET tasks do not communicate with each other, but can send and receive messages from the TT tasks.

The *ET→TT* interface allows point-to-point communication from one ET task (the sender) to one TT task (the recipient). Data is explicitly pushed by the sender and is made visible for the recipient according to the temporal observability principles (recall section III-A). Figure 4 shows an example of *ET→TT* communications. An ET task pushes some data when processing $a2$. This data will be available for the TT task after its first activation following the "push", i.e. at $T2$. Determinism of the TT tasks is preserved because a) FIFO do not receive information from multiple ET tasks, and b) the temporal observability principle ensure that behavior is not impacted by the concurrent execution of the ET and TT tasks.

The *TT→ET* interface allows point-to-point communication from one TT task (the producer) to one ET task (the consumer). Data are regularly updated by the system according

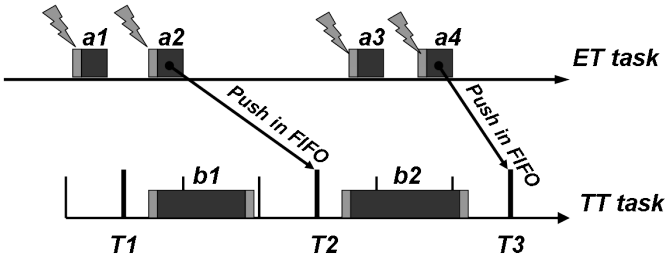

Fig. 4. Flow from ET to TT tasks

to temporal behavior of the producer. On event occurrence, the consumer accesses the latest value. Figure 5 shows such an example of *TT→ET* communications. A TT task produces a temporal flow X. When the event $e1$ occurs, the ET task observes that the value of X is 0. However, when $e3$ and $e4$ events occur, the ET task observes that the value of X is 4 in both cases.

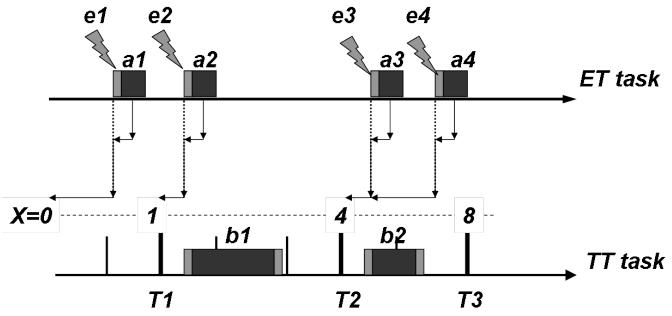

Fig. 5. Flow from TT to ET tasks

B. Virtualization support

The main goal of virtualization is to run safely heterogeneous applications on the same platform, with reduced development costs (e.g. applications using different APIs, mixed-criticality applications, etc.). We present here the main technical challenges that OS virtualization implies on embedded platforms, then we briefly describe a prototype of a virtualized OS running under the supervision of *PharOS*.

A lot of techniques exist to run multiple OSes on a same CPU. We will only focus on how to run one virtualized OS (or *guest*) under the supervision of a *host* OS that ensures the execution of real-time critical tasks, such as *PharOS*. We will only consider the case where the guest executes non-critical tasks, or at least less critical than the ones executed by the host[4]. In this context, we use the *paravirtualization* technique, which means we modify the guest OS to make it runnable by the host. Obviously, the host OS also require some modifications to execute and schedule the guest: these modifications form a minimal *hypervisor*.

[4]This model complies to a intuitive and simple rule: the OS that is able to execute the most critical tasks should be the one in full control of the hardware.

We enumerate here the key-points that ought to be carefully considered when implementing such a paravirtualization solution, then we illustrate them with a practical case based on *PharOS*.

1) CPU execution mode: if the CPU supports protected execution, the guest OS should *not* be able to run in privileged mode. That way, hardware protection ensures that the guest is not able to execute sensitive instructions that would compromise the host OS functionalities (interrupts masking, CPU halting, etc.). Instead, as a paravirtualized OS, the guest "asks" for the corresponding service to the host OS using dedicated system calls, or *hypercalls*. The handlers of these hypercalls are implemented as part of the hypervisor within the host kernel.

It is formally impossible to ensure any isolation between the guest and the host without the support of a protected execution mode. Likewise, it is impossible to ensure isolation between a regular OS and its tasks.

At last, note that some CPUs provide a "harware-assisted" virtualization feature [8] – especially the Intel x86 architecture. It basically consists in an additional protection level that ultimately allows the kernel and the tasks of the guest system to be isolated from each other. This separation ensures that a failing guest task can not crash the whole guest system – assuming that the guest kernel implements such protection mechanisms. Hardware-assisted virtualization also greatly helps in virtualizing un-modified OSes (full virtualization).

2) Guest scheduling - CPU resource sharing: we propose three scheduling policies for executing a guest OS on a real-time host OS.

The first policy consists in using the "idle-time": the guest is scheduled only once all real-time tasks of the host have been executed and completed. The guest can run as long as no real-time task reaches its earliest starting date. This policy dedicates all the "idle-time" of the CPU to the execution of the guest, thus providing an optimal utilization of the CPU resource. However, this is also a drawback: the CPU time allocated to the guest depends on the real-time behavior of the host application, and can potentially be very irregular. This policy is suited for executing "best-effort" guest OSes, e.g. *Embedded Linux*.

Another solution consists in executing the guest OS as a special real-time task of the host OS. This task is executed with a period p within a time window of duration $\Delta \leq p$, and is given a CPU time budget $b \times \Delta$, where $b \in [0, 1]$ is a percentage of the temporal window width (see figure 6). The guest task is scheduled as any other real-time task, but when its CPU budget is entirely consumed, the task is considered to be completed, and must wait for the next temporal window to be schedulable again. It is also possible to allow the guest OS to release the CPU before its timing budget is consumed, using a specific hypercall.

This policy gives the developer a precise control of the CPU share given to the virtual machine, equals to $\frac{b \cdot \Delta}{p}$. As it turns out, it is very well suited for virtualizing a real-time guest OS on a real-time host OS. In that case, p can be wisely chosen

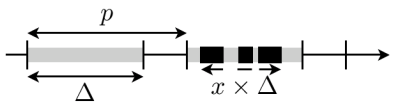

Fig. 6. Temporal parameters of the execution of the guest OS. The light gray time zone is the temporal execution window; the black zones are actual execution times. On this example, the CPU time consumed is $x \times \Delta$, with $x \leq b$.

to simulate a regular clock interrupt to the guest OS.

A last solution consists in mixing the two policies above to handle mixed-criticality among the tasks of the guest. For instance, guest tasks with soft real-time requirements can be executed within a dedicated host real-time task, but can also be given the remaining idle CPU time that will help reaching their deadlines.

3) Guest interrupts handling: when the guest is running, it may handle its own interrupts. However, when it is idle, the host must handle them on behalf of the guest, while still ensuring safeness and determinism. Once again, two policies are possible:

- Interrupts channels of the guest are active only when the guest is actually being executed. Interrupts received while the host is executing any critical task are lost. This solution is simple and safe, but the loss of interrupts is a drawback.
- Interrupts for the guest are received by the host even if the guest is "asleep", but under certain conditions. A mechanism similar to the one described in section IV-A should be implemented to prevent the host from an "Interrupt Storm". This would also guarantee a bounded overhead due to guest interrupts handling. The interrupts received should be buffered and delivered to the guest when it is executed again. This policy theoretically avoids interrupts lost for the guest, however its implementation is a bit more complex than in the first policy. Note also that this solution can simulate a periodic interrupt (e.g. clock interrupt) to the guest. A dark spot though: waking-up the guest is delicate. Indeed, it will probably not endure well to be flooded by all the interrupts missed while it was asleep, and a "temporization" policy would be more appropriate. This question deserves further study and experimentation and is part of our future work.

4) Memory isolation: this feature *cannot* be implemented without a hardware support, *i.e.* without a MMU or at least a MPU device. When such a hardware memory protection unit is available, the host OS must ensure that it remains the only one able to modify memory access rights. If the guest needs to modify those rights, it should do so through a dedicated hypercall, that will check the sanity of its request. Moreover, the guest OS should only be able to access to a dedicated part of the physical memory, but *not* to the memory of the host or of its critical tasks.

5) Communication between the host and the guest: the ultimate integration step of the guest OS is to allow him to communicate with the host. We voluntarily will not go into the details here, because *a lot* of solutions can be considered: shared memory buffers, dedicated hypercalls, virtual network devices, etc. In addition, these solutions depend on the requirements set on data communications between the guest OS and the host as well as the performance of the underlying hardware.

6) Use-case – Trampoline *over* PharOS*:* A first prototype of mixed systems has been realized with both *PharOS* and *Trampoline* [1] OSes, used respectively for high and low critical functions on a Freescale S12XE architecture. *Trampoline* is an open source Real-Time Operating System, that implements the *OSEK/VDX* API, a standard in automotive embedded software. This prototype is therefore a rather straightforward solution to feature *OSEK/VDX* tasks on a *PharOS* driven platform.

The paravirtualization of *Trampoline* is ensured by the *PharOS* microkernel. In accordance with the principles exposed previously, *Trampoline* is encapsulated in a time-triggered (TT) task and executed in protected mode. This guest TT task has a period of 10 ms, which corresponds to the elementary time quantum of the *Trampoline* real-time kernel. Using the notations from figure 6, the parameters of the guest task are: $p = 10\text{ms}, \Delta = 10\text{ms}, b = 10\%$. In other words, *Trampoline* is given 10% of the total CPU resource. Every time the *Trampoline* task is awaken by the scheduler, the mini-hypervisor implemented in the *PharOS* kernel simulates a clock interrupt for *Trampoline*. This regular "tick" indicates that 10ms have elapsed in the real world.

PharOS also provides hypercall services to *Trampoline* to manage its basic functionalities. Communications between host and guest OSes are not implemented yet in this prototype. However, a solution based on a specific ET task interface, as presented in section IV-A, is currently being studied.

The prototype is designed to execute safely two representative automotive applications. The first one is executed by *PharOS*: six real-time tasks put together a subset of a body controller. That controller includes some system output commands, communication mechanisms through a CAN-bus and sensors signal measurements. The second application, hosted by *Trampoline*, is basically a diagnostic function (aliveness monitoring) made of two real-time tasks based on the *OSEK* API.

The *Trampoline* task has been designed as a single *PharOS* task "group" (made of only one task), thus able to recover from an error and restart (see section III-D). Various error scenarii are used to stress *PharOS* safety mechanisms. The same tests were transposed to the *Trampoline* application, all resulting in successful detection and isolation by *PharOS*. The two main tests are:

- *Illegal memory access*: an attempt to read or write to *PharOS* memory space from *Trampoline* is detected and *Trampoline* is restarted.
- *Infinite loop*: an erroneous *Trampoline* task entering an infinite loop does not interfere with *PharOS* tasks scheduling. It is the responsibility of the *Trampoline* kernel however to detect this fault, and possibly restart

the task.

As a conclusion, let us stress that we have focused on one paravirtualization technique through a practical use-case, but a lot of alternatives can be considered. The choice of a solution should depend on the type of the virtualized OS, and mostly on the underlying architecture. Typically, multi-core architectures offer many different ways to solve the CPU-sharing problem – the simplest but clearly suboptimal solution being to dedicate one core for each guest.

V. Related work

Spatial and temporal isolation mechanisms have been introduced to support the virtualization of the execution of multiple operating systems on a single platform [9], [10], a long time ago. Since then, protection of information within operating systems has been the subject of numerous work, such as [4], [3] to name a few. Lately, the avionic domain has re-popularized and defined the principles of spatial and temporal partitioning [11], while Xen [12] has also re-popularized research work in the area of virtualization. However, fewer work have focused on issues of virtualization within real-time constraints [13], especially in highly-constrainted memory environments.

In [11], underlying mechanisms and issues to implement spatial and temporal partitioning have been shown. Classically, spatial partitioning relies on hardware memory protection units. However, a software approach has also been proposed called Software Fault Isolation [14] (SFI). In this approach, memory references are statically checked and, when it is possible, additional code is embedded in order to check accesses during execution. Proof-carrying code (PCC) [15] can be seen as a generalization of SFI, in which a safety policy is expressed using first-order logic that is certified and verified by a kernel prior to execution. But the problem of these static-checking methods is that they do not provide any protection against hardware bugs, nor do they allow execution of unchecked third-party code, which *PharOS* does.

Currently, several operating systems implements spatial and temporal partitioning such as DECOS [16] and successors or LynxOS-178 [17], an ARINC 653 compliant OS [5]. However, in both cases, temporal partitioning is ensured by static scheduling of partition time slots.

VI. Conclusion

The *PharOS* model, method and tools provide a strong basis for real-time systems isolation. We have explained how we combined temporal isolation with optimal scheduling and dynamic task behavior, using offline compilation and analysis, and online monitoring techniques. We showed how we achieved spatial isolation by automatic generation of fine-grained memory tables at compile time, and control of communication using a trusted "system layer" in a separate address space. Using these isolation principles as a basis, we have presented many different ways of extending *PharOS* to support several levels of mixed-criticality:

- error confinement and recovery with groups of tasks;
- interrupts processing with appropriate monitoring and physical core separation;
- paravirtualization of external OSes

Future work on *PharOS* will mainly focus on improving resource-sharing techniques for paravirtualization (CPU cores, interrupts channels, peripherals), and on new scheduling algorithms on multi-core embedded platforms, based on our previous work [7]. In addition, we plan to improve the safeness of our approach by making the system layer untrusted. The node transitions checking would be handled by the microkernel (as critical activities), and the system layer would ultimately be allowed to fail without impacting other tasks. Such a solution would also allow much more flexibility and facilitate the virtualization of communicating tasks.

Acknowledgment

The authors would like to thank Vincent David (CEA LIST), for its pioneer works on *OASIS* [6], that lead to the current implementation of *PharOS*.

This research on *PharOS* is supported in part by *Agence Nationale de la Recherche* (ANR), through the SCARLET project. The paravirtualization prototype of *Trampoline* over *PharOS* was especially developed within this project.

References

[1] J.-L. Bechennec, M. Briday, S. Faucou, and Y. Trinquet, "Trampoline An Open Source Implementation of the OSEK/VDX RTOS Specification," in *Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference on*, September 2006, pp. 62 –69.

[2] M. Lemerre, V. David, C. Aussaguès, and G. Vidal-Naquet, "An introduction to time-constrained automata," *EPTCS: Proceedings of the 3rd Interaction and Concurrency Experience Workshop (ICE'10)*, vol. 38, pp. 83–98, June 2010.

[3] Saltzer and Schroeder, "The protection of information in computer systems," *Communication of the ACM*, vol. 7, 1974.

[4] P. J. Denning, "Fault tolerant operating systems," *ACM Computing Survey*, vol. 8, no. 4, pp. 359–389, 1976.

[5] ARINC, "Avionics application software standard interface. arinc specification 653," January 1997.

[6] C. Aussaguès and V. David, "A method and a technique to model and ensure timeliness in safety critical real-time systems," in *Fourth IEEE ICECCS'98*, Los Alamitos, CA, USA, 1998, pp. 2–13.

[7] M. Lemerre, V. David, C. Aussaguès, and G. Vidal-Naquet, "Equivalence between schedule representations: Theory and applications," in *Proc. of the 14th Real-time and Embedded Technology and Application Symposium (RTAS'08)*, Los Alamitos, CA, USA, 2008, pp. 237–247.

[8] J. Fisher-Ogden, "Hardware support for efficient virtualization," *San Diego, USA*, 2006.

[9] R. A. Meyer and L. H. Seawright, "A virtual machine time-sharing system," *IBM Syst. J.*, vol. 9, pp. 199–218, September 1970.

[10] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk, "Multiple operating systems on one processor complex," *IBM Syst. J.*, vol. 28, pp. 104–123, March 1989.

[11] J. Rushby, "Partitioning in avionics architectures: Requirements," Jun. 1999.

[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of ACM SOSP '03:*. ACM Press, 2003, pp. 164–177.

[13] A. L. Henning Schild and A. Warg, "Faithful virtualization on a real-time operating system," in *Eleventh Real-Time Linux Workshop 2009*, Dresden, Germany, September 2009.

[14] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," *SIGOPS Oper. Syst. Rev.*, vol. 27, pp. 203–216, December 1993.

[15] G. C. Necula and P. Lee, "Safe kernel extensions without run-time checking," in *Proceedings of the second USENIX symposium on Operating systems design and implementation*, ser. OSDI '96. New York, NY, USA: ACM, 1996, pp. 229–243.

[16] R. Obermaisser, P. Peti, B. Huber, and C. El Salloum, "Decos: an integrated time-triggered architecture," *e & i Elektrotechnik und Informationstechnik*, vol. 123, pp. 83–95, 2006.

[17] LynuxWorks, "LynxOS-178. Certifiable RTOS for safety-critical computing," 2003.