# Integrating Linux and the real-time ERIKA OS through the Xen hypervisor

Arianna Avanzini
and Paolo Valente
Università di Modena e Reggio Emilia
arianna.avanzini@unimore.it
paolo.valente@unimore.it

Dario Faggioli
Citrix System Italia Srl
dario.faggioli@citrix.com

Paolo Gai
Evidence Srl
pj@evidence.eu.com

***Abstract*—Modern user interfaces grow more and more complex and cannot be possibly handled by the same software components in charge of the timely execution of safety-critical control tasks.**
**Evidence Srl recently proposed a single-board dual-OS system aimed at combining the flexibility of the Linux general-purpose operating system, which is able to produce any complex user interface, and the reliability of the automotive-grade ERIKA Enterprise operating system, a small-footprint real-time OS suitable for safety-critical control tasks and able to execute commands triggered by Linux.**
**The operating systems run on dedicated cores and, for efficiency reasons, they share memory with limited support for memory protection: although the system allows running two operating systems, from a safety certification point of view it suffers from the fact that safety-critical and non-safety-critical components should be isolated from each other.**
**In this paper we present, as an improvement to the initial implementation, again a double-OS system running, on a dual-core platform, ERIKA Enterprise and a full-featured Linux OS, but using the Xen hypervisor to run the two operating systems in two isolated domains. In the proposed setup, each of the domains runs on a dedicated core, assigned statically by the hypervisor. Linux runs as the control domain, and is therefore able to execute any of the components of the Xen toolstack; it is also able to grant to the real-time operating system access to any I/O-memory range needed for control tasks.**
**The described system also provides a simple, safe communication mechanism between the two operating systems, based on Xen's inter-domain event notification primitives and explicit sharing of a dedicated set of memory pages by the real-time operating system.**

## I. Introduction

Modern cars, as well as aircrafts, are equipped not only with more and more complex control systems, but also with increasingly advanced user interfaces and infotainment systems. Control software components, in charge of the execution of safety-critical tasks, must be complemented with more flexible, general-purpose ones, able to provide whatever infotainment services is required. As of the latter, the growing computational demand of modern user interfaces and complex infotainment applications can now be met only with multi-core systems, which are actually supplanting single-core ones. The emerging trend in software industries is to complement real-time operating systems, in charge of performing safety-critical tasks, with general-purpose ones, able to provide any infotainment service and to handle sophisticated user interfaces.

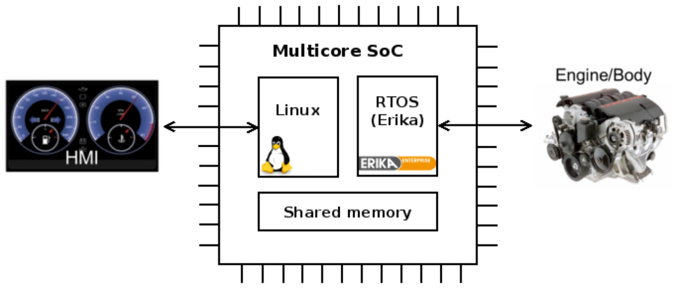


Figure 1: Outline of the initial design

In the context of integrating safety-critical and non-safety-critical software components, the company Evidence Srl has recently proposed a dual-operating-systems design based on a Freescale dual-core ARM board [13]. Such a design allows to run in parallel a real-time operating system and a general-purpose operating system. As of the latter, the selected general-purpose operating system, in this pilot design, is Linux, a very commonly used open-source solution which is commonly used in embedded applications. Despite its widespread usage, Linux has not achieved any certification for real-time or automotive, due to its complexity, to the small amount of documentation available for some of its core components, and to the huge effort needed by the process of certifying an operating system so large.

As of handling safety-critical tasks, the software component of choice is ERIKA Enterprise [11, 12], a low-footprint automotive-grade operating system. ERIKA Enterprise is an open-source operating system with hard real-time support which obtained the OSEK-VDX certification for automotive applications [24]. ERIKA Enterprise has a very small footprint (1-4 KB) and is therefore suited for most embedded applications; it also supports real-time task scheduling: its task scheduler implements hard real-time scheduling algorithms such as Fixed Priority Scheduling [5], Immediate Priority Ceiling and Earliest Deadline First [22]. In the solution initially proposed by Evidence Srl, each of the operating systems is exclusively assigned a core, so as to (a) reduce as much as possible temporal interference from the general-purpose operating system which could affect the response latencies of the real-time operating system; (b) allow the real-time operating system to be booted in parallel to the general-purpose operating system.

Also, the two running operating systems share memory for

simplicity purposes and implement a very efficient message-passing communication based on shared memory and interrupts. A consequence of this design aspect is that it provides **limited support to isolation** of software components that are responsible for the correct and timely execution of safety-critical tasks from tasks executed by the non-safety-critical software components.
In more detail, two problems might arise from such an interaction between safety-critical and non-safety-critical components.
(a) A **malfunction of the general-purpose operating system** might pollute the memory area of the real-time operating system, therefore leading to the incorrect execution of safety-critical tasks; this is not very likely to happen, as it would mean that Linux attempted to write outside of its addressing space, but is still possible and, in the event of its happening, would possibly cause catastrophic damage to users.
(b) Conversely, a **malfunction of the application running on the real-time operating system** could have memory segment descriptors clobbered with, and as a consequence of this any kind of memory pollution in the Linux memory area could happen.

The main idea behind the proposed work is to replicate the structure of the previous solution, but **adding a hypervisor as an extra abstraction layer** to isolate safety-critical and non-safety-critical software components by mediating accesses to shared memory and interrupts. The hypervisor we selected and exploited for this new solution is *Xen* [6]: the rationale behind this choice included its being widely used in embedded applications, its very low footprint and its already finalized and widely-tested ARM support.

Finally, the result of this work has been integrated in a virtual machine to ease the possibility to replicate the current design environment [35].

The content of the paper is organized as follows: Section II outlines related works relevant to this paper's topic; Sections III and IV contain references to the internal implementation of the Xen hypervisor and its communication mechanisms, including the design choices made during the development. Section V provides an evaluation of the results obtained, and finally Section VI states our conclusions.

## II. Related work

Hypervisors are becoming more and more common in nowadays embedded computing. Among the reasons of their popularity we can cite the increasing complexity of mixed-criticality embedded applications together with the need to reuse existing software stacks. In particular, the advent of functional safety standards such as the automotive standard ISO26262, or similar standards in the avionics and industrial domains, makes important the need to guarantee the highest safety standards of the most critical software subsystems. On the other hand, the rest of the system should continue to work reusing existing legacy software stacks, including complete embedded Linux subsystems.

In particular, we can highlight commercial solutions as Integrity [25], PikeOS [26] and WindRiver Hyperscan [27], which are likely the hypervisors most used in safety-critical avionics systems. Our aim was to exploit an open-source product to obtain an equally open-source solution, and for that reason we did not consider them.

On the open-source arena, we can highlight various type of approaches. Some are linked to the developments made in virtualization and cloud environments, such as Xen [1] and KVM [8]. Those are quite mature projects, with an established community and ARM hypervisor support. Other solutions instead tend to implement a minimal hypervisor which is small and amenable to mixed criticality systems. On this category we would like to cite QuestOS [28, 30], NOVA [29], and GMV Air [31]. Unfortunately they are not available for the ARM architecture, and therefore they have not be considered for our work. There is also a line of development of hypervisor for constrained architectures without MMU. In this case the hypervisor is used to para-virtualize interrupts, as in the work performed on the Aramis Project on Infineon Tricore [33]. Other solutions also include the possibility to use multi-core platforms to have multiple OS on different cores. In addition to the implementation performed by Evidence [13], we would like to cite TI Concerto [34], which integrates on the same chip a DSP and a Cortex M3 (not running Linux), and the FreeRTOS integration with Linux available as a demo for Altera/Xilinx FPGAs [21]. Finally, we would like to cite the CodeZero hypervisor [32], available on the ARM architecture, and approaches using ARM TrustZone, such as those proposed by Mentor Graphics.

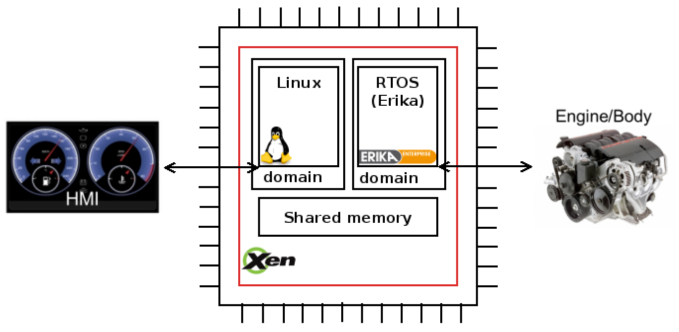
The hypervisor which has been selected for this work is Xen [6] because of its low footprint, support for ARM hypervisor extensions, growing open-source community, independence on the Linux kernel and mainly because of the safe resource management system it provides[1].

While exploiting a new abstraction layer to improve what was lacking in the previous design, the new solution must be able to fulfill the requirements that had the previous design be considered already feasible: (a) it must **prevent as much as possible temporal interference** between the running operating systems; (b) it must provide a communication channel that allows the real-time operating system and the general-purpose operating system an **efficient interaction**. The way in which this challenge was tackled by exploiting some of Xen's natively available features will also be set out in the following.

[1]For a more detailed explanation of the reasons behind this choice please refer to [1, 2, 3, 4].

## III. Hypervisor-based design

Figure 2: Outline of the hypervisor-based design



This Section outlines the new design proposed in this paper, which basically replicates the one already set out by Evidence Srl and described in Section I, but also adds the a hypervisor as an extra layer to improve isolation between concurrently-running operating systems; this Section will also provide a detailed insight in its proposed implementation, therefore evaluating the benefits and disadvantages of the used mechanisms.

The Xen hypervisor makes use of two types of guest operating systems. The *dom0* (also referred to as *control domain*) is a privileged domain which is started during boot, has direct access to hardware, and manages unprivileged domains. A *domU* (or *unprivileged domain*) is a domain which, by default, doesn't have any direct access to the underlying hardware, but instead uses peripherals by exploiting abstractions of hardware components as exposed by the control domain.

Being a proof-of-concept prototype, the new design initially sets up the Linux general-purpose operating system as a dom0 and runs the ERIKA Enterprise real-time operating system as a domU. The hypervisor-based design proposed in this paper was implemented on a cubieboard2 [15], an ARMv7-based board featuring a dual-core Allwinner A20 processor with virtualization extensions. The board was chosen both for its compliance with common requirements of an embedded system, and for its compliance with the requirements of the design itself, which included having a multi-core platform to run each operating system on a dedicated core; it also already had full compatibility with the ARM port of the Xen hypervisor, which explicitly provides support for ARM processors with virtualization extensions.
The ERIKA Enterprise operating system already supported numerous multi-core ARM-based boards; the generic structure of the code was extended to port it to the selected platform. The ERIKA operating system was moreover modified to avoid executing machine instructions which would potentially allow it to interfere with the execution of other domains and/or give a domain access to the addressing space of other domains, which would have been aborted by the hypervisor. Also, the build system had to be modified as, at the time, the domain loader included in the Xen-on-ARM toolstack only supported a non-compressed zImage format.

Finally, ERIKA Enterprise supported interrupt handling and distribution in a common fashion: it set up multiple interrupt handlers, each taking care of a single physical interrupt line. The interrupt distribution system provided by the Xen hypervisor, instead, is based on *event channels*, and basically has all emulated interrupt signals piggy-backed by a single physical per-processor interrupt (PPI 31). A driver for the Generic Interrupt Controller (GIC) provided by Xen has been successfully integrated in the code of the ERIKA Enterprise operating system.

As soon as ERIKA Enterprise was able to run as a Xen-on-ARM guest, we chose the GPIO controller as a reference for a simple peripheral to be handled in the restricted domain. Given that ERIKA Enterprise runs as an unprivileged domain, it is not allowed as a default to access I/O-memory areas and interrupts related to external peripherals; at the time of this work, Xen did not support granting direct access to I/O-memory regions to domUs (*I/O-memory pass-through*) running on ARM: it was therefore necessary to port the related hypercall (`XEN_DOMCTL_memory_mapping`), which existed only for the x86 architecture, to ARM[2].

One of the most relevant requirements set during the development of the initial dual-operating-system design is to be able to reduce as much as possible the interference between the real-time operating system and the general-purpose operating system. To such a purpose, the previous design included having each operating system running on a separate core of a multi-core platform. The new design achieves the same goal, as the Xen hypervisor allows to statically assign cores to running domains (*CPU pinning*).

## IV. Implementation of inter-domain communication

This Section aims at describing the communication mechanism which has been implemented in the hypervisor-based dual-operating-system design; the target use case of the interaction is to enable the general-purpose operating system to trigger the execution of a command by the real-time operating system as it would happen, e.g., in an advanced user interface able to trigger the execution of a task on the hardware facilities which ERIKA is in charge of controlling. The goals that were set for this implementation step are briefly outlined in the following.
**Efficiency**: communication must be as fast as possible and involve the hypervisor as little as possible.
**Safety**: communication must be performed in a way that prevents operating systems from interfering with each other. This is achieved by fully exploiting Xen's mediation between the two operating systems.
**Asynchrony for the real-time operating system**: a notification triggered from the general-purpose operating system must not preempt any high-priority task running on the real-time operating system. The real-time operating system must be able to define priorities and determine which of the tasks running in its container are at a higher priority than interrupts.
**Synchrony for the general-purpose operating system**: the general-purpose operating system must be notified as soon as possible of the completion of a command. The thread of execution of the general-purpose operating system interacting

[2]The code performing the port has been merged into the Xen hypervisor shortly after its proposal on the development mailing list and is currently available in Xen 4.5.

with the real-time operating system must be able to block until the command has been completed.

Communication has been implemented with two interacting and interdependent software components, the first residing in the general-purpose dom0 and the second hosted by the real-time domU. The communication driver has been implemented with a simplified split device driver model, based on a shared memory region and an interrupt line handled by the hypervisor (*event channel*). The access of the domains to both memory and interrupts is mediated by Xen.

### A. ERIKA Enterprise communication driver

As soon as it is initialized, the ERIKA Enterprise instance needs to perform tasks that enable the general-purpose Linux operating system to enumerate the commands and related resources it makes available. By having ERIKA Enterprise explicitly performing such operations, we make sure that it can assign them the priority it deems necessary with respect to other running tasks. Also, by letting it explicitly reserve resources for inter-domain communication, we ensure that ERIKA Enterprise is aware of the fact that such resources are actually used for communication and are as a consequence potentially subject to concurrent access.

Listing 1 shows the pseudo-code for the operations performed by ERIKA Enterprise in order to setup resources reserved for inter-domain communication. Firstly, ERIKA Enterprise requests to the Xen hypervisor an unbound event channel to perform communication. This is achieved by using the proper hypercall, which returns the event channel number reserved by Xen for the domain. While issuing the hypercall, ERIKA Enterprise will need to specify that the event channel is intended for inter-domain communication, and therefore any other domain can bind to it. Next, ERIKA enterprise needs to get the start physical address of the memory region it intends to use for the communication. After obtaining such a direct reference to the memory pages used for communication, ERIKA Enterprise must interact once again with the Xen hypervisor to set them as mappable by other domains; for security reasons, it will grant access privileges just to the dom0, which in our proposed setup is the general-purpose operating system: the dom0 will therefore be able to map such a memory region in its own addressing space and use the memory directly. Last, ERIKA Enterprise needs to pass the references to both the event channel and the memory area to the dom0 in some way: it therefore exploits the XenStore by writing the two needed pieces of information as values to conventional XenStore keys.

### B. Linux communication driver

To implement the other half of the communication driver, the Linux kernel has been extended with a specific device driver; its structure and goals quite resemble those of the previously-proposed implementation, but while the interface is kept as homogeneous as possible, the core of the implementation is adapted to make use of Xen's facilities. The user interface is a set of `sysfs` tunables used to provide to the kernel the parameters of a command to be executed by the ERIKA domU. In the case of the proposed proof-of-concept, commands concern the GPIO controllers and simply allow to set a particular GPIO pin to a particular value.
Listing 2 shows the simple algorithm used in the Linux half of the communication driver. The code implementing sysfs hooks has been omitted as it doesn't directly concern the design proposed in this document; also, the mapped memory region is treated as the container of a message memory area whose specific structure is of little or no relevance to understand the implemented policy.

The communication algorithm is implemented with the specific purpose of avoiding as much as possible any interference by the general-purpose operating system on the real-time operating system. In fact, each command to be sent to the real-time ERIKA operating system is seen as a self-contained transaction and the event channel is actually bound only for the minimum indispensable time to perform the needed operation. This also allows for a fine-grained identification of a valid state versus an invalid state of the shared memory area.

### C. Command delivery and handling

The core communication mechanism works on top of the shared memory region and event channel set up by the two operating systems. As shown in Listing 2, as soon as the `write_command()` function is invoked, the shared memory area is populated with data which is relevant to the execution of the command itself, in this case the pin number and pin value. Afterwards, an event-channel-related hypercall is invoked to notify the other domain about the changes performed on the shared memory region.

If the event channel is masked from the ERIKA Enterprise operating system's side, the signal will not be delivered to the unprivileged domain until the channel is unmasked: this allows that high-priority tasks are not preempted by the execution of communication primitives; also, the asynchrony of the communication, in fact, allows to set a priority to event channel handlers with respect to ERIKA Enterprise tasks which are ready for execution. If instead the event channel is not masked, as soon as the notification is received ERIKA Enterprise will handle the interrupt with a dedicated routine. The pseudo-code of such routine is shown in Listing 3, which instead omits all device-specific code.

## V. Evaluation of the results and future work

This Section evaluates benefits and disadvantages brought by the dual-operating-system design proposed in this paper. The evaluation focuses on functional aspects of the design itself and does not provide any information about the performance of the new design as compared to the performance of the old design. This choice derives from the fact that, even if already able to provide all the features required by the use case, the prototype is still at the stage of a *proof of concept*: the development of such a system focused on design aspects more than on performance.

### A. Evaluation of the communication mechanism

The communication setup routine involves issuing two hypercalls from the side of the real-time operating system; after such a setup is completed, however, the general-purpose operating system is able to make direct use of the shared

Listing 1: ERIKA Enterprise communication driver internals

```
global comm_area;

allocate_unbound_event_channel:
  return XEN_EVTCHN_OP(alloc_unbound, handling_routine);

reserve_comm_area:
  return get_machine_address_of_communication_pages();

register_comm_area_in_grant_table(comm_area, dom):
  XEN_gnttab_op(comm_area, dom);

advertise_channel_and_memory(evtchn, ref):
  XenStore_write("erika_evtchn", evtchn);
  XenStore_write("erika_memory", ref);

setup_interdom_comm_structures:
  evtchn = allocate_unbound_event_channel();
  comm_area = reserve_comm_area();
  register_memory_pages_in_grant_table(comm_area, dom0);
  advertise_channel_and_memory(evtchn, comm_area);
```

Listing 2: Linux communication driver internals

```
global evtchn = INVALID, comm_area = INVALID;

bind_with_erika_evtchn:
  return XenStore_read("erika_evtchn");

get_erika_memory_reference:
  memref = XenStore_read("erika_memory");
  return XEN_gnttab_op(map, memref);

signal_on_evtchn(evtchn):
  XEN_evtchn_op(NOTIFY, evtchn);

write_command:
  comm_area.pin_number = num, comm_area.pin_value = val;
  signal_on_evtchn(evtchn);

get_return_from_memory:
  return comm_area.ret;

on_pin_value_change:
  val = get_pin_value(), num = get_pin_number();
  evtchn = bind_with_erika_evtchn();
  comm_area = get_erika_memory_reference();
  write_command(ref, val, num);

on_erika_interrupt:
  unbind_erika_evtchn(evtchn);
  ret = get_return_from_memory(comm_area);
  /* handle return value */
  evtchn = ref = INVALID;
```

memory region, needing no further mediation from the hypervisor's side to access the memory area used for inter-domain communication.

Each notification sent to the partner operating system to signal about the presence of a new command or to notify the completion of a command, however, involves the Xen hypervisor, which might constitute a performance bottleneck. In fact, such need for constant participation of the hypervisor in the communication between the two operating systems could limit the capabilities of the real-time operating system, as an interrupt storm could keep the machine stuck in hypervisor mode for too long.

## B. *Evaluation of the design*

The new design provides an **improved support to isolation** of concurrently-running operating systems, adding the Xen hypervisor as an extra layer which mediates all accesses of concurrently-running operating systems to shared memory and interrupts. It also allows to highly reduce temporal interference by statically assigning a CPU to each domain for exclusive use. Finally, it implements a simple, efficient communication mechanism based on shared memory accessed through the hypervisor's mediation and that involves the hypervisor the least possible, exploiting hypercalls only when strictly necessary.

The proposed design, however, provides **no guarantees on**

Listing 3: Interrupt handling in ERIKA Enterprise

```
global num, val, comm_area;

get_pin_number_and_value:
  num = comm_area.pin_number; val = comm_area.pin_value;

execute_command:
  return gpio_set(num, val);

signal_on_evtchn(evtchn):
  XEN_evtchn_op(NOTIFY, evtchn);

notify_result(ret):
  comm_area.ret = ret;
  signal_on_evtchn(evtchn);

handling_routine:
  get_pin_number_and_value();
  ret = execute_command();
  notify_result(ret);
```

**the boot response times** of the real-time operating system, as it is run as an unprivileged Xen domain and therefore it must fully wait for the general-purpose operating system to fully boot before being able to start. Actually, the proposed solution does not even guarantee that the real-time operating system boots at all, as any malfunction of the general-purpose operating system could prevent it from being correctly initialized and therefore would prevent it from being able to boot the real-time operating system. Also, the Xen hypervisor, which constitutes a very critical component of the proposed design, is not certified for real-time applications; as of the current state of the project, it should be easily certifiable according to Design Assurance Level (DAL) E/D standards, as documentation exists for the most relevant features it provides, but cannot be brought as it is now to a DAL-B/A certification level, which would allow it to be exploited for real-time applications, or to meet an automotive-grade standard such as the Automotive Safety Integrity Level (ASIL). Also, the **support to isolation can definitely be improved**, as having the two software components running as differently-privileged domains could still allow for interference between the execution of the two operating systems.

## C. Future work

The roadmap which has been set up for the project currently involves three parallel threads: (a) investigate a possibility to boot the real-time operating system as fast as possible; (b) investigate a way to actually provide complete isolation between the safety-critical component and the non-safety-critical one; (c) verify whether the Xen hypervisor can be certified for real-time applications at any level. As of the first two threads, one possibility includes having the real-time operating system running as the control domain and the general-purpose operating system running as unprivileged domain. This would allow to reduce the possibility of a system-wide malfunction due to interference, but it would not entirely remove it; also, it would be necessary to port at least a subset of the features provided by the simplest toolstack provided by Xen (most probably `libxl`) to ERIKA Enterprise. Another possibility, which was just recently outlined, would exploit a new design based on complete disaggregation of the safety-critical and non-safety-critical components: a dom0 based on a Mini-OS microkernel, able to boot in a few milliseconds, would in this case start both ERIKA Enterprise and Linux as unprivileged domains and allow them safe interaction and access only to the resources which are strictly necessary to each domU. This design, although more complex and difficult to set up, would finally provide the required isolation level; it would be however necessary to switch to a more suited platform, e.g., a board providing more than two cores which would be able to run the dom0 and the two domUs on dedicated cores.
As of the second issue concerning certification of the Xen hypervisor, there is ongoing effort to certify at least the core subset of the hypervisor's features for real-time applications [20]. A different line of action would instead include considering the possibility to exploit another hypervisor that has already achieved some kind of certification, such as the Jailhouse partitioning hypervisor [18].

## D. Porting the solution to other RTOS

We believe that the proposed solution can be easily ported to other (small) operating systems. In particular, ERIKA Enterprise has a rather limited set of requirements on the target core instruction set and architecture. Every tiny RTOS making usage of interrupts and having a rather limited hardware abstraction layer can offer a similar porting. However, we think the choice of ERIKA Enterprise is currently justified by the fact it implements an automotive API making it a good candidate for the integration in next-generation automotive infotainment solutions.

## VI. Conclusions

In this paper we described the work performed for porting the ERIKA Enterprise kernel as a domU in XEN. The paper highlighted the main design choices, and the current limitation of the approach.

The result of this work is available as open-source, and has generated high interest in the Xen development community, resulting in the developed code being accepted in mainline Xen and available since Xen 4.5.

The complete procedure to obtain the described setup has also been described on the ERIKA Enterprise wiki to allow easy installation and deployment, while a complete example of the setup has been integrated in a publicly accessible virtual machine [35].

As of future developments of this proof-of-concept work, we plan to continue the porting to provide a complete virtualization-based system on ARM-based microcontrollers.

## REFERENCES

[1] P. Barham and B. Dragovic and K. Fraser and S. Hand and T. Harris and A. Ho and R. Neugebauer and I. Pratt and A. Warfield, *Xen and the art of virtualization* ACM SIGOPS Operating Systems Review, vol. 37, n. 5, pages 164-177, 2003.

[2] K. Adams and O. Agesen, *A comparison of software and hardware techniques for x86 virtualization* ACM Sigplan Notices, vol. 41, n. 11, pages 2-13, 2006.

[3] P. Dewan and D. Durham and H. Khosravi and M. Long and G. Nagabhushan, *A hypervisor-based system for protecting software runtime memory and persistent storage* Proceedings of the 2008 Spring simulation multiconference, pages 828-835, Society for Computer Simulation International, 2008.

[4] V. Chaudhary and M. Cha and J. P. Walters and S. Guercio and S. Gallo, *A comparison of virtualization technologies for HPC*, pages 861-868, 22nd International Conference on Advanced Information Networking and Applications, 2008.

[5] J. P. Lehoczky, *Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines*, vol. 90, pages 201-209, RTSS 1990.

[6] Citrix, *The Xen Project, the powerful open source industry standard for virtualization*, http://www.xenproject.org/, 2003.

[7] Qumranet, *QEMU, Open-source processor emulator*, http://wiki.qemu.org/Main_Page, 2007.

[8] Qumranet, *KVM, Kernel-based virtual machine*, http://www.linux-kvm.org/page/Main_Page, 2007.

[9] Oracle, *Oracle VM VirtualBox*, https://www.virtualbox.org/, 2007.

[10] VMware, *VMware Virtualization for Desktop and Server, Application, Public and Hybrid Clouds*, http://www.vmware.com/, 1999.

[11] P. Gai, *ERIKA Enterprise, Open-source OSEK/VDX-certified RTOS*, http://erika.tuxfamily.org, 2002.

[12] P. Gai and E. Bini and G. Lipari and M. Di Natale and L. Abeni, *Architecture for a portable Open Source Real Time Kernel Environment*, Proceedings of the 2nd Real time Linux Workshop, Orlando, Florida, December 2000.

[13] P. Gai and C. Scordino and B. Morelli, *A fully Open-Source platform for automotive systems*, http://www.evidence.eu.com/embedded-linux-osekvdx-erika-enterprise-dual-core-automotive-cpu-without-hypervisor.html, 2013.

[14] Denx, *Das U-Boot, the Universal Boot Loader*, http://www.denx.de/wiki/U-Boot, 1999.

[15] Cubietech, *Cubietech cubieboard2*, http://linux-sunxi.org/Cubietech_Cubieboard2, 2013.

[16] ISO, *ISO 26262*, http://www.iso.org/iso/catalogue_detail? csnumber=43464, 2011.

[17] Continental Automotive GmbH, *OSEK-VDX Portal*, http://www.osek-vdx.org/, 2007.

[18] J. Kiszka, *Jailhouse: A Linux-based Partitioning Hypervisor*, http://lwn.net/Articles/574273/, 2013.

[19] The Xen Project, *RT-Xen: Real-Time Virtualization based on Xen*, https://sites.google.com/site/realtimexen/, 2011.

[20] N. Studer and R. VanVossen, *Xen and the Art of Certification*, http://www.slideshare.net/xen_com_mgr/art-certification, 2014.

[21] Xilinx, *Zynq All Programmable SoC Linux-FreeRTOS AMP Guide*, http://www.xilinx.com/support/documentation/sw_manuals/petalinux2014_2/ug978-petalinux-zynq-amp.pdf, 2014.

[22] G. Lipari, "Earliest Deadline First", Scuola Superiore Sant'Anna, Pisa, Italy, 2005.

[23] N. C. Audsley and A. Burns and A. J. Wellings, *Deadline monotonic scheduling theory and application* Control Engineering Practice, vol. 1, n. 1, pages 71-78, Elsevier, 1993.

[24] OSEK-VDX Consortium, *ERIKA Enterprise Cerification. Follow Project Status / Certifications / Binding Index CB 4.5*, http://www.osek-vdx.org, 2014.

[25] Green Hills Software, *INTEGRITY Hypervisor*, http://www.ghs.com/products/rtos/integrity_virtualization.html, 2015.

[26] Sysgo, *PikeOS Hypervisor*, https://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/, 2015.

[27] WindRiver, *HyperScan Hypervisor*, http://windriver.com/products/ hyperscan/, 2015.

[28] Y. Li and R. West and E. Missimer, *A Virtualized Separation Kernel for Mixed Criticality Systems* Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Salt Lake City, Utah, March 2014.

[29] U. Steinberg and B. Kauer, *NOVA: A Microhypervisor-Based Secure Virtualization Architecture* Eurosys, 2010.

[30] R. West et al., *QuestOS*, http://www.hypervisor.org, 2015.

[31] GMV, *GMV Air Hypervisor*, http://www.gmv.com/en/aeronautics/products/air/, 2015.

[32] CodeZero, *CodeZero Embedded ARM Hypervisor*, http://www.l4dev.org/, 2015.

[33] D. Reinhardt and G. Morgan, *An Embedded Hypervisor for Safety-Relevant Automotive E/E-Systems* Proceedings of SIES 2014, Pisa.

[34] Texas Instruments, *TI Concerto*, http://www.ti.com/lit/wp/spry174a/spry174a.pdf, 2015.

[35] P. Gai and C. Scordino and B. Morelli and P. Valente and A. Avanzini, *Xen hypervisor porting*, http://erika.tuxfamily.org/wiki/index.php?title=Xen_Hypervisor, 2015.