# RodosVisor - an object-oriented and customizable hypervisor: The CPU virtualization

**Abstract:** RodosVisor is an object-oriented and bare-metal virtual machine monitor (VMM) or hypervisor designed for the aerospace industry, mainly to provide time and spatial separation to the NetworkCentric core avionics machine, Montenegro and Dittrich (2009). The NetworkCentric core avionics machine consists of several harmonized components working together to implement dependable computing in a simple way, with computing units managed by the local real-time operating system RODOS. To support partitioned software architectures such as AIR, Rufino et al. (2009), and MILS, DeLong, R. (2007), RodosVisor adapted the Popek and Goldberg's fidelity, efficiency and resource control virtualization requirements, Popek and Goldberg (1974), to the space application domain by extending them with extra ones, like timing determinism, reactivity and improved dependability. Another distinctive RodosVisor feature is the customized design based on generative programming techniques, such as aspect oriented programming and template meta-programming.

*Keywords:* Hypervisors; Safety-critical; IMA Architecture; Virtual Machine Monitor; Partitioned Software Architecture; Ring Decompression; Virtualization; Real-time; ARINC 653.

## 1. INTRODUCTION

Unlike past avionics systems, mainly developed using custom hardware and software, nowadays, due to the full life cycle costs of customized systems, it is noticeable a shift towards to the use of COTS-based (Commercial of-the-shelf) systems, forcing a migration from federated architectures towards Integrated Modular Avionics (IMA) architectures. In a federated architecture, individual subsystems perform dedicated functions, while in an IMA architecture, a single computing platform is used for distinct applications which are executed concurrent and independently on a single CPU. However, the integration of different COTS components with different levels of criticality and predictability, and from several suppliers, revealed new challenges such as reusability of existing software as well as prevention of safety hazards, leading to the adoption of TSP based (Time and Space Partitioning) architectures by the aerospace industry.

Virtualization-based approaches have emerged as candidates to build IMA architectures and, among several civil and military standards proposed to define an IMA architecture, ARINC 653, AEEC (2006), was adopted for the implementation of RodosVisor. A virtualization-based approach, implemented with deterministic hypercalls, efficient inter-partition communication, efficient partition switch, low overhead and low footprint, will provide significant cost savings through workload consolidation and BOM (Bill of Materials) cost minimization, while simultaneously retaining the ability to leverage the ecosystem of avionic applications through workload isolation, OS co-location, hot upgradeability and fault-tolerance into a single system.

There are three main approaches to carry out virtualization, namely, the VMM or hypervisor, the microkernel and the microvisor, Heiser and Leslie (2010), Iqbal et al. (2010); all designed as a thin software environment on which multiple applications and complete operating systems, each with a different objective, can run concurrently in fully protected and isolated execution environments or Virtual-Machines (VMs) as if they were running directly on the hardware. Regardless of the mechanism by which virtualization is implemented, one commonality among usage models is that the thin software abstraction layer must virtualize system resources such as CPU, memory and I/O devices by: (1) scheduling the VMs which share the hardware platform, (2) managing the resources assigned to each VM, and (3) saving/restoring the state when context switching between VMs.

The remaining of this paper is structured as follows. In section II related work is surveyed. In section III, an overview of RodosVisor architecture is presented. In section IV the CPU virtualization is introduced, namely execution modes, memory, interrupt and timer virtualization. Section V presents experiments with our hypervisor and, the paper finishes with conclusions and references.

## 2. RELATED WORK

IBM pioneered virtualization technology long time ago with CP/CMS, Creasy (1981), the first full-virtualization system,

to allow time-sharing a computer system among several operating systems. As mentioned above, virtualization requires a shift in thinking from physical to logical by treating IT physical resources as logical resources. When used in enterprise and on the desktop, it leverages load balancing, server consolidation, legacy code migration, cross platform interoperability and also security, Heiser (2008) (see Uhlig et al. (2005), for existing virtualization usage models).

To cope with the emerging trends in embedded systems such as increasing complexity and high demand for functional safety and security, as well as improved autonomy and usability, time-to-market pressure and determinism of operation and service, virtualization in embedded systems has been focused mainly on co-existence of a fixed set of vastly different operating systems (RTOS and GPOS), architectural abstraction and, increasingly security, see Rufino et al. (2009), Heiser (2008), Lei et al. (2008), Oikawa (2006), Aoyagi (2008), among others. Some of them, such as Rufino et al. (2009), Beltrame et al. (2010) and Crespo et al. (2009), are used in aerospace industry as they are ARINC-653 quasi-compliant.

## 3. RODOSVISOR ARCHITECTURE

ARINC 653 avionics standard was specified to meet the requirements of safety-critical systems which require strictly deterministic periodic processing. As such, it specifies the essential services for space and time segregation that are fundamental to build IMA architectures. RodosVisor is a type 1, Goldberg (1972), or bare-metal hypervisor, supporting full and paravirtualization, alongside with real-time capabilities, and, able to execute several applications according to the IMA concept. Like Beltrame et al. (2010) and Crespo et al. (2009), RodosVisor is an ARINC-653 quasi-compliant hypervisor as it implements ARINC-653 services but it doesn't follow standard ARINC-653 APIs (Application Programming Interfaces). Some virtualization implementations such as Rufino et al. (2009), Lynx Secure, Linx (online), and VxWorks 653, VxWorks (online), claim to be ARINC-653 complaint. Unlike these hypervisors however, RodosVisor is nearly 100% object-oriented implemented and to the best of our knowledge, only NOVA, from Steinberg and Kaue (2010), follows the same approach. However, NOVA presents different objectives and only supports hardware-assisted virtualization.

Fig 1 presents Rodosvisor's system architecture. Central to this organization is the core layer supported by RodosVisor. It virtualizes all main hardware components, such as CPU, memory and I/O, to implement the partition concept. RodosVisor guarantees system integrity by isolating spatial and temporally each VM from one another into separate memory areas and pre-allocated time quantum for runtime activity, respectively. Doing so, partition's software of different criticalities can be executed concurrently on the same processor, as presented in the upper layer of Fig. 1:

1. GPOS partition: composed by a general purpose operating system and associated applications;
2. RTOS partition: consisting of a real-time operating system and associated tasks;
3. OSless partition: that is a bare-metal compiled application.

RodosVisor achieves partition and hardware platform transparency through two software layers, the PAL (Partition Abstract Layer) and CAL (Core Abstract Layer), respectively. PAL defines a set of hypercalls used by partitions to request RodosVisor core services, while CAL defines a set of function calls to provide hardware services to the RodosVisor core layer, i.e., providing portability across different processor-based platforms.

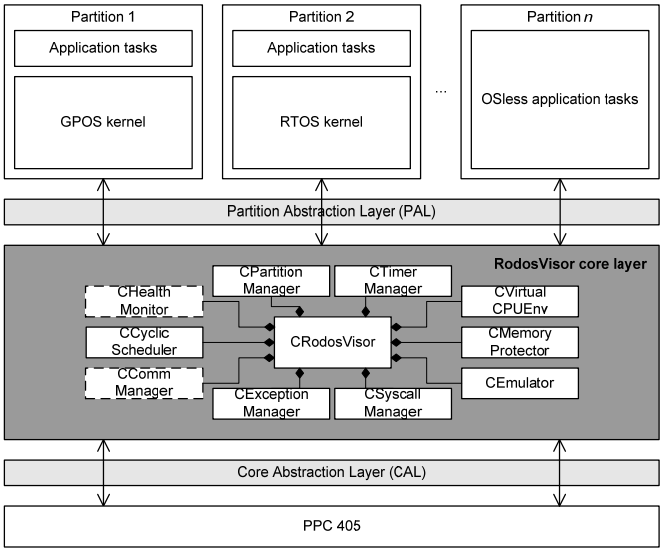


Fig. 1. Rodosvisor's system architecture supporting GPOS, RTOS and OSless partitions.

In order to integrate COTS components with different criticality in a virtualization-based IMA, the scheduling policy must provide desired real-time performance to the integrated system built from several VMs or partition subsystems. To be ARINC-653 compliant, at the hypervisor level, the main scheduling abstraction, the virtual CPUs (vCPUs), created for each partition, are scheduled in a strictly deterministic and periodic manner decided statically at system integration time.

The current tendency of SoC (System-on-Chip) along with FPGA's high integration and lower power consumption sparked a growing interest for CPU-based FPGA in space missions, and so, we chose a FPGA board as a platform for RodosVisor implementation and assessment tests. PowerPC was the chosen processor because it is a robust low power platform and also because Xilinx offers some space qualified FPGAs with built-in PowerPC core.

## 4. CPU VIRTUALIZATION

In order to impose spatial separation, RodosVisor provides an isolated virtual execution environment to each partition. As one of the consequences of running in a isolated execution environment is limited access to the processor resources, the challenge is how to present the execution environment for each VM as if it were the actual hardware with full access to it. From the various existing techniques, RodosVisor

implements full-virtualization, because it's advantageous in terms of compatibility and flexibility, and, paravirtualization, due to its lower virtualization overhead. The techniques used to virtualize execution modes, privileged level, timer and interrupt, are discussed in the following subsections.

### *4.1 Execution Modes Virtualization*

The PowerPC 405 (PPC405) has two execution modes: user and privileged. Only one of the two modes can be active at any time. If privileged mode is active, programs (i.e., privileged programs) can execute all instructions and access all registers. Instead, if user mode is active, programs (i.e., user programs) don't have access to the privileged feature set of the processor, thus, they can execute only a subset of all instructions and access only a subset of all registers. In user mode, the execution attempt of a privileged instruction is denied and an exception (i.e., a program exception) is generated instead. Other processors than PPC405 have a set of sensitive user mode instructions. These instructions, which change/expose privileged registers, must also be protected from a non-privileged execution mode in RodosVisor.

RodosVisor runs always in privileged mode, as only this mode provides complete control over the underlying hardware platform. This way it is able to manage and thus share the hardware platform among all VMs. On the other hand, each partition runs in user mode, but as execution environment of a vCPU expects full access to recourses, a mechanism must exist to provide, although indirectly, the lost functionality.

As the user mode execution of a privileged instruction generates an exception, this feature is used to catch attempts to execute privileged instructions from the VM, and then emulate their original functionality, as seen in Fig. 2.

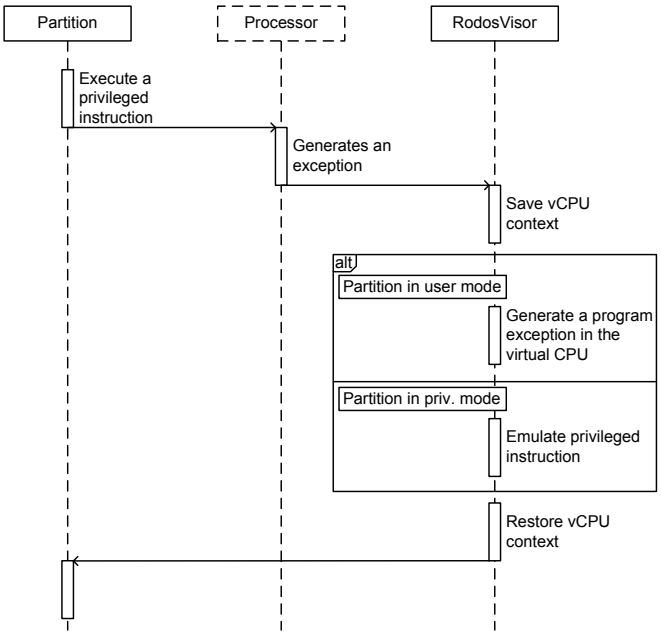


Fig. 2. Trap-and-emulate scenario in RodosVisor.

If the cause of the exception is the execution of a privileged instruction while in user mode but the vCPU being in privileged mode, the faulting instruction is emulated. As the processor registers are mapped in a PPC405 vCPU structure, it is possible to know the current execution mode of a vCPU by keeping track of its machine state registers (MSRs). If the cause of the exception was not the execution of a privileged instruction or if the vCPU is in user mode, the exception is forwarded back, as is, to the VM. This method is also referred to as full-virtualization. That is to say, although the CPU is in user mode, if a vCPU is in privileged mode it is fed with the illusion that it is in fact in privileged mode (virtually-privileged mode). Each privileged instruction has a corresponding emulation routine which will be called by the exception handler within the faulting VM.

Paravirtualization, on the other hand, requires VMs to collaborate with RodosVisor through a set of hypercalls specified at design time. This approach, unlike full-virtualization, requires the operating system, or partition code, to be modified but it is especially useful in architectures where full-virtualization is impossible or too demanding, as well as when extending the base functionality of a specific architecture. RodosVisor implements hypercalls through the user mode 'sc' instruction which generates a system call exception when executed, in a sequence of events similar to trap-and-emulate. When a system call exception is generated by the processor, the corresponding interrupt service routine is called (within RodosVisor's context). There, the state of the vCPU is saved and, if the vCPU is in privileged mode, the hypercall requested is executed, otherwise the vCPU can only be in user mode and the exception is forwarded back to the vCPU. Then, in either case, RodosVisor restores the vCPU, and execution of the VM is resumed.

### *4.2 Ring compression*

Most embedded processors, such as PPC405, provide only one unprivileged mode. In a simplistic approach, RodosVisor runs in the privileged mode (supervisor state) and VMs run in user mode (problem state). However if a VM runs a GPOS and respective applications, another intermediate mode is needed in order to isolate them from each other as in a non-virtualized environment. This way, the guest GPOS kernel is isolated from its applications and RodosVisor is isolated from guest GPOSes. To do so, a similar strategy to those adopted by EmbeddedXEN, ISYS (2010), IXIV VMM, Aoyagi (2008), xLuna, Beltrame (2010), and Xen-on-ARM, Seo (2008), will be followed to split the problem state in two logical modes: guest GPOS kernel mode and application mode (Fig. 3).

In the PPC405 processor, the execution modes are controlled by MSR[PR] (i.e., the bit field PR of the Machine State Register). Additionally, to distinguish between user mode and kernel mode in the VM, the PPC405 takes advantage of MSR[IR] and MSR[DR] fields together with write/execution attribute bits found in the memory management unit, MMU.

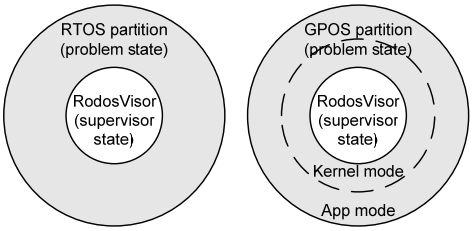


Fig. 3. Execution modes: native vs. RodosVisor.

A full implementation of ring compression in PPC405 takes advantage of its memory management mechanisms, but due lack of space it will not be detailed in this paper.

*4.3 Interrupt virtualization*

When a VM is running, any kind of interrupt, or exception, can occur. However, the destination of the interrupt can be RodosVisor or the VM itself. All interrupts are first handled by RodosVisor, and after dispatched to VMs accordingly, as presented in Fig. 4.

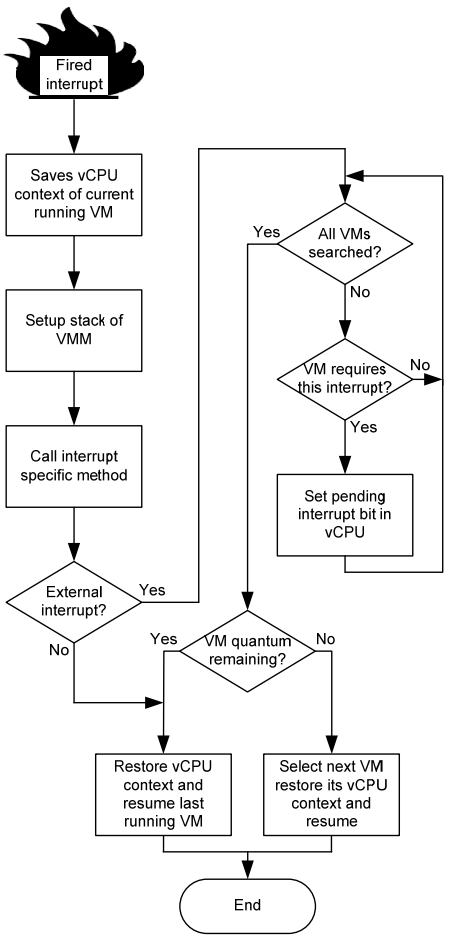


Fig. 4. RodosVisor default interrupt processing.

There are three classes of interrupts:

1. Real hardware interrupts which occur while a given VM is running and is intercepted by RodosVisor and handled bottom-up, i.e., from RodosVisor to guest OS;
2. Interrupts related to CPU state, such as trap and faults fired inside a VM that are handled top-down, i.e., from guest OS to RodosVisor and are delivered back to the running VM;
3. Virtual interrupts forward to those VMs containing the virtual devices.

By default unless changed at design time, RodosVisor doesn't discriminate between interrupts related to real-time activities and interrupts related to non real-time activities. Therefore, it intercepts and manages all hardware interrupts and delivers only virtual interrupts to the VMs. The interrupt will be delivered only to the running VM, or temporal separation would be lost. For all other VMs, the interrupt is enqueued, by setting the interrupt pending of the respective vCPU, so that it is handled when the VM resumes execution.

*4.4 Timer virtualization*

The RodosVisor timer can be seen as a special case of interrupt. This is due to the fact that the timer is the heart of the system, i.e., it multiplexes the hardware resources among partitions, and inside these, among applications. In the case of RodosVisor this is especially true in order to maintain temporal isolation between partitions.

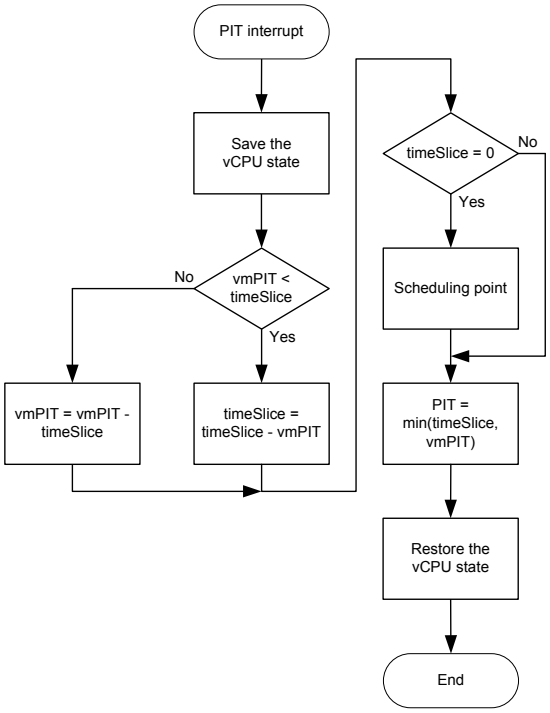


Fig. 5. RodosVisor virtual PIT state diagram.

As there are two levels of timing, RodosVisor level and partition level, two timers are required. As in other architectures PPC405 has just one PIT (Programmable Interval Timer), so RodosVisor implements a virtual timer to provide VM scheduling service as well as partially some VM inner timing services. In doing so, the internal timing

resolutions (i.e., clock tick) of all VMs must be known before hand, i.e., at design time. This way, if the PIT of the running VM expires before the next scheduling point, the PIT of the processor is configured according to the VM, otherwise, it is configured with remaining time to the next scheduling point, as presented in Fig. 5.

This is the default RodosVisor virtual PIT approach that only raises virtual timer interrupts to the running VM, but how to handle interrupts to inactive VMs? Dispatching virtual interrupt to appropriated VMs immediately after hardware interrupt firing, preserves timely delivery of interrupts but can break the temporal separation due to context switching cost. For inactive VMs, RodosVisor ignores the fired interrupts and only implements a virtual tickless timekeeping mechanism based on TB (Time-Base) unit to measure the passage of time and keep track of the wall-clock time. Therefore, when a VM is rescheduled for execution, its internal clock tick and all timing-related data structures are recalibrated to account for the time elapsed since its previous CPU execution quantum.

## 4. EXPERIMENTS

To assess the virtualization overhead of RodosVisor, several experiment have been conducted. More specifically, these experiments have measured: (1) RodosVisor's context saving overhead; (2) privileged instructions decoding overhead; (3) virtual instruction drift; (4) virtual instruction total execution time; (5) virtual PIT interrupt latency; (6) VM's context saving overhead. In these experiments the mtpit (move to PIT) instruction is used.

To conduct the experiments RodosVisor runs only one OSless partition, executing the procedure seen in Fig. 6. For a better understanding of the experiment the following explanation will start from (a).

In (a) the timestamp before the (virtual) execution of mtpit is saved. Then, when mtpit is executed, a Program exception is generated within RodosVisor's context, which saves the virtual context and which after that saves the timestamp in (b) in a special register visible from virtual (i.e. user) mode. Subtracting (b) to (a), RodosVisor's context saving overhead is obtained.

After saving the context, and the timestamp in (b), RodosVisor's decodes the faulting instruction (i.e. mtpit) and then the timestamp in (c) is saved in another special register visible from virtual mode. Subtracting (c) to (b), RodosVisor's privileged instruction decoding overhead is obtained.

After decoding the privileged instruction, the corresponding emulation is called, and there, before the execution of the true mtpit, the timestamp in (d) is also recorded. Subtracting (d) to (a) the virtual mtpit drift is obtained; that is, the time it takes from the execution of mtpit in virtual mode to the actual modification of the PIT in real mode.

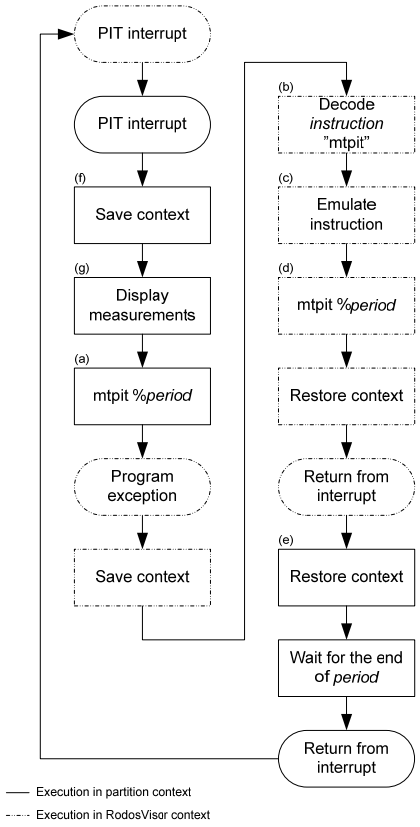


Fig. 6. Procedure used to assess RodosVisor's performance.

When the emulation routine completes and the virtual context restored, the system returns again to the virtual context. There, the timestamp in (e) is saved. Subtracting (a) to (e) the virtual mtpit total execution time is obtained.

Then, the VM returns from the interrupt and locks in a endless wait loop until the PIT expires again and a PIT interrupt is generated in RodosVisor's context, which redirects the interrupt to the VM, generating a virtual PIT interrupt. The VM, before saving the context records the timestamp in (f). With (f), (d) and period it is possible to determine the PIT interrupt latency through the formula:

$$latency = (f) - ((d) + period).$$

After saving the context, timestamp (g) is obtained. Subtracting (f) to (g) the virtual context saving overhead is obtained.

Finally the results are displayed; only raw data is presented, all calculations are made on a host computer. Because of this process's structure the first set of data is ignored, as well as the last set, if incomplete.

The results we obtained are summarized in Table I. With quantum of 42949672 CPU cycles 391 data sets were obtained and averaged using the arithmetic mean.

**Table 1. Rodosvisor's performance summary**

| Measurement | CPU cycles |
|---|---|
| RodosVisor's context saving overhead | 301 |
| RodosVisor's instruction decoding overhead | 351 |
| Virtual *mtpit* drift | 736 |
| Virtual *mtpit* total execution time | 1308 |
| PIT interrupt latency | 1393 |
| VM's context saving overhead | 2963 |

## 5. CONCLUSIONS

In this paper RodosVisor is presented. It is an object-oriented and bare-metal virtual machine monitor that enforces spatial and temporal separation, in order to meet the requirements of safety-critical systems defined in ARINC 653 avionics standards. Currently is running on a Xilinx FPGA board with built-in PowerPC core, thus demonstrating the main concepts of CPU virtualization that were presented in this paper.

The added value of object-oriented implementation will be visible and profitable at integration time, where generative techniques such as aspect-oriented programming (AOP) and/or template meta-programming will help designing fully customizable and optimized hypervisor. The implemented integration tool allows fine-grained selection of the functionality, as for example to replace the scheduling policy, as well as coarse-grained approach to remove complete block of functionality if it is not required.

RodosVisor, when fully implemented, will be able to provide system designers with the known benefits provided by a hypervisor-based system, together with compatibility with already existent software and with an augment feature set through paravirtualization, reducing drastically the overhead created by virtualization. The augmented feature will include I/O virtualization, inter-VM communication, UIO drivers, health monitoring, among other extensions that developers will be able to add alongside the already existent ones. Some features still need to be implemented; however, the major issues seem to have been already overcame which clears the way for a successful result.

## REFERENCES


AEEC, (2006). ARINC 653P1-3 Avionics Application Software Standard Interface. Part 1, Required Services. *Airlines Electronic Engineering Committee*.

Aoyagi, S. (2008). IXIV VMM: A VMM on 2-Level Ring Architecture. *IEEE 8th International Conference on Computer and Information Technology Workshops*, Sydney, Australia.

Beltrame, G., Fossati, L., Zulianello, M., Braga, P., Henriques, L. (2010). XLUNA: A Real-time, Dependable Kernel for Embedded Systems. *IP-SOC Conference*.

Creasy, R. (1981). The origin of the VM/370 time-sharing system. *IBM Journal of Research & Development*, Vol. 25, No. 5, 483-490.

Crespo A., Ripoll, I., Masmano, M., Arberet, P., and Metge, J. (2009). XTRATUM: an open source hypervisor for TSP embedded systems in aerospace. *Data Systems In Aerospace - DASIA*, Istanbul, Turkey.

DeLong, R. (2007). MILS: Protection our most Vital Systems. *Military Embedded Systems*.

Goldberg, R. (1972). Architectural principles for virtual computer systems. *Ph.D. Thesis, Division of Engineering and Applied Physics Harvard University*, Cambridge Massachusetts

Heiser, G. (2008). The Role of Virtualization in Embedded Systems., *1st Workshop on Isolation and Integration in Embedded Systems*, Glasgow, UK.

Heiser, G. (2008) Microkernel-Based Virtualization Meets Embedded Security Challenges. *http://rtcmagazine.com/articles/view/100958*.

Heiser, G., and Leslie, B. (2010). The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. *ACM SIGCOMM*, New Delhi, India.

ISYS Group. (2010). PENAR: Porting XEN on ARM-based Embedded Systems, Final technical report. *http://sourceforge.net/projects/embeddedxen/files/*.

Lei, J., Yang, X., Xiong, G., Jiang, W., Liao, Y. (2008). VMM-Based Real-Time Embedded System. *International Conference on Embedded Software and Systems Symposia*. Chengdu, Sichuan, China.

Lynx. (online). LynxSecure Embedded Hypervisor and Separation Kernel. *http://www.lynuxworks.com/virtualization /hypervisor.php*.

Montenegro, S., and Dittrich, L. (2009). Network Centric Core Avionics. *Data Systems In Aerospace - DASIA*, Istanbul, Turkey.

Iqbal, A., Sadeque, N., and Mutia, R. (2010). An Overview of Microkernel, Hypervisor and Microvisor Virtualization Approaches for Embedded Systems. *http://www.eit.lth.se/fileadmin/eit/project/142/virtApproaches.pdf*.

Oikawa, S. (2006). Linux/RTOS Hybrid Operating Environment on Gandalf Virtual Machine Monitor. *Embedded and Ubiquitous Computing, Lecture Notes in Computer Science*, Volume 4096/2006, pg. 287-296.

Popek, G. and Goldberg, R. (1974). Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17 (7), 412-421.

Rufino, J., Craveiro J., Schoofs, T., Tatibana, C., and Windsor, J. (2009). AIR Technology: a step towards ARINC 653 in space. *Data Systems In Aerospace - DASIA*, Istanbul, Turkey.

Seo, S. (2008). Research on System Virtualization using Xen Hypervisor for ARM based secure mobile phones. *Fifth IEEE Consumer Communications & Networking Conference – CCNC'08*, Las Vegas, Nevada.

Steinberg, U. and Kaue, B. (2010). NOVA: A Microhypervisor-Based Secure Virtualization Architecture. *The European Professional Society on Computer Systems (Eurosys) conference*, Paris, France

Uhlig, R., Neiger G., Rodgers, D., Santoni A., Martins, F., Anderson, A., Steven, B., Kägi, A., Leung, F., Smith L. (2005). Intel Virtualization Technology. *IEEE Computer*, vol. 38 no. 5, 48-56.