ROLE OF VIRTUALIZATION IN EMBEDDED SYSTEM[1]

Sreenivasan R

Lecturer in Electronics Engineering, Government Polytechnic College, Palakkad.

ABSTRACT

System virtualization, which is very popular in the business and personal, computing worlds, is recently getting a lot of attention in the embedded world. We will examine the differences in motivation for the use of system virtual machines, as well as the resulting differences in the requirements for the technology, beginning with a comparison of key characteristics of enterprise systems and embedded systems. We observe that these distinctions are very significant, and that virtualization can't meet the extraordinary prerequisites of inserted frameworks. Instead, virtualization as a special case necessitates more general operating system technologies. We argue that high-performance micro kernels, specifically L4, are a suitable technology for the needs of embedded systems of the next generation.

Keywords: System, virtualization, embedded, virtual machines, micro kernels.

INTRODUCTION

Virtualization has been a hot topic in the enterprise space for quite some time, but has recently become an important technology for embedded systems as well. It is therefore important for embedded-systems developers to understand the power and limitations of virtualization in this space, in order to understand what technology is suitable for their products.

This white paper presents an introduction to virtualization technology in general, and specifically discusses its application to embedded systems. We explain the inherent differences between the enterprise-systems style of virtualization and virtualization as it applies to embedded systems. We explain the benefits of virtualization, especially with regard to supporting embedded systems composed of subsystems with widely varying properties and requirements, and with regard to security and IP protection.

[1] How to cite the article:
Sreenivasan R., Role of Virtualization in Embedded System, *International Journal of Advances in Engineering Research*, April 2013, Vol 3, Issue 4, 82-93

IJAER

We then discuss the limitations of plain virtualization approaches, specifically as it applies to embedded systems. These relate to the highly-integrated nature of embedded systems, and the particular security and reliability requirements.

We present microkernels as a specific approach to virtualization, and explain why this approach is particularly suitable for embedded systems. We show how microkernels, especially Open Kernel's OKL4 technology, overcome the limitations of plain virtualization.

System virtualization has become a common computing tool, as evidenced by billion-dollar initial public offerings (IPOs) and startup company sales for hundreds of millions of dollars. The decoupling of virtual and actual registering stages by means of framework virtual machines (VMs) upholds different purposes, of which the most famous ones are:

• Consolidating services that were making use of distinct computers into distinct virtual machines that were running on the same computer. To achieve quality-of-service (QoS) isolation between servers, this makes use of the strong resource isolation offered by virtual machines;

• load-balancing across clusters by migrating live virtual machines or creating new virtual machines on demand on a host that is rarely used. This makes use of virtualization's platform abstraction;

• power management in clusters by moving virtual machines off of machines that are underutilized and allowing them to be shut down (in effect, load-balancing in reverse)

• firewalling administrations which have a high gamble of being compromised to safeguard the remainder of the framework. Isolation of resources is also used in this;

• running multiple operating systems (OSs) on the same physical machine, such as Windows, Linux, and MacOS, typically to run OS-specific applications This use is also made possible by resource isolation and is primarily applicable to personal computers (desktops or laptops).

The fact that all VMs typically run the same operating system (or, in the last scenario discussed above, "similar" operating systems in the sense that they offer roughly the same kinds of capabilities and similar abstraction levels) is one of the main characteristics of these usage cases. Similarly to physical machines, VMs communicate via virtual network interfaces (including network file systems) in these scenarios. This is in line with the VM view, which is, by definition, similar to the physical machine view. Clearly, most of the above use cases do not exist in today's embedded systems; however, some will become relevant as manycore chips become available. We need to examine the characteristics of contemporary embedded systems and identify similarities as well as distinctions between them and enterprise computing systems in order to comprehend why system virtual machines have recently received a lot of interest from developers of embedded systems.

IJAER

For quite some time, virtualization has been a hot topic in the business world, but it has recently emerged as an essential technology for embedded systems as well. Therefore, it is essential for developers of embedded systems to comprehend the advantages and disadvantages of virtualization in this field in order to determine which technology is best suited to their products. This white paper gives an overview of virtualization technology in general and focuses on how it can be used with embedded systems. We go over the fundamental differences between virtualization for embedded systems and enterprise-systems virtualization. We discuss the advantages of virtualization, particularly with regard to IP and security protection for embedded systems made up of subsystems with widely varying requirements and properties. Following that, we talk about the drawbacks of standard virtualization strategies, particularly in relation to embedded systems. These are related to the particular security and reliability requirements and the highly integrated nature of embedded systems. We discuss the advantages of microkernels as a particular virtualization strategy for embedded systems and present microkernels as an example of this strategy. We demonstrate how microkernels, particularly Open Kernel's OKL4 technology, circumvent plain virtualization's limitations. Then, we give a glimpse of this technology's future.

VIRTUALIZATION

Providing a software environment in which programs, including operating systems, can run as if they were on bare hardware is known as virtualization (Figure 1). Such a product climate is known as a virtual machine (VM). A VM of this kind is a reliable, isolated copy of the real machine [PG74].

The virtual-machine monitor (VMM) or hypervisor is the software layer that provides the VM environment. The VMM possesses three essential characteristics [PG74] in order to maintain the illusion that it contains:

1. Software is provided with an environment by the VMM that is nearly identical to the original machine;
2. At best, the speed of programs running in this environment slows down slightly;
3. System resources are completely under the control of the VMM.

Virtualization is highly practical because of these three important characteristics. The first (similitude) guarantees that product that sudden spikes in demand for the genuine machine will run on the virtual machine as well as the other way around. The second, efficiency, ensures that virtualization can be implemented in terms of performance. The third measure, resource control, ensures that software cannot exit the virtual machine. Language environments, such as the Java virtual machine, are also frequently referred to as virtual machines. A process virtual machine (VM) is referred to as this, while a system virtual machine (VM) is referred to as this and can run complete operating systems [SN05]. System VMs are the only topic of this paper.

IJAER

HOW IS IT DONE?

The majority of instructions must be executed directly by the hardware in order to meet the efficiency requirement: A single virtual machine instruction is substituted for multiple host hardware instructions by any interpretation or emulation. This necessitates that the virtual hardware and the physical hardware on which the VMM is hosted must largely be identical. It is possible for the virtual and physical machines to have minor differences. For instance, the virtual machine might have a few additional guidelines not upheld by the actual equipment. Compared to the virtual hardware, the physical hardware may have different devices or a different memory management unit. It's possible that the virtual machine will run older versions of the same fundamental architecture. Alternately, the virtual machine might be a new architecture version that hasn't been implemented yet. Virtualization can be about as effective as using the same hardware as long as there aren't too many differences and the different instructions aren't used a lot. Some instructions cannot be carried out immediately. All instructions that deal with resources must access virtual resources rather than physical ones because of the resource-control characteristic. As a result, the VMM must interpret these instructions in order to prevent virtualization from failing. The virtual machine is specifically required to interpret two types of instructions: control-delicate guidelines which adjust advantaged machine state, and in this manner impede the hypervisor's command over assets; instructions that are behavior-sensitive and can access and read privileged machine state. Although they are unable to alter resource allocations, they reveal the state of actual resources—specifically how it differs from virtual resources—breaking the virtualization illusion. Together, control-touchy and conduct delicate guidelines are called virtualization-delicate, or just delicate directions. There are two fundamental ways to guarantee that the virtual machine's code does not carry out any sensitive instructions: virtualization alone: ensure that the hypervisor is invoked rather than sensitive instructions executed within the virtual machine; unclean virtualization: Virtualization code should take the place of sensitive instructions in the virtual machine.

PURE VIRTUALIZATION

The traditional method is pure virtualization. It requires privileged access to all sensitive instructions. Favored directions execute effectively on the off chance that the processor is in a special state (normally called special mode, bit mode or boss mode) however create an exemption when executed in unprivileged mode (likewise called client mode), as displayed in Figure 2. The hypervisor's exception handler is the address at which an exception enters privileged mode.

The only requirement for pure virtualization is to execute all of the VM's code in the processor's non-privileged execution mode. The hypervisor will get a hold of any sensitive instructions in the VM's code. In order to keep the state of the virtual machine intact, the hypervisor interprets the

IJAER

instruction and "virtualizes" it. On nearly all current architectures, pure virtualization was previously impossible due to the presence of sensitive, non-privileged instructions that would access physical machine state rather than virtual machine state. Virtualization extensions, which enable the processor to be configured in a way that forces all sensitive instructions to cause exceptions, have recently been added by all major processor manufacturers. In any case, there are different motivations behind why options in contrast to unadulterated virtualization are generally utilized. One is the high cost of exceptions. A processing delay of one cycle per pipeline stage occurs when an exception drains the pipeline on processors with pipelines. When switching back to user mode, there is typically a similar delay. Additionally, exceptions and exception returns are branches that are typically unpredictable by a processor's branch-prediction unit, which increases latency. In high-performance processors with extensive pipelines, these effects typically add up to more than 10 to 20 cycles. The exception costs of some processors, like the x86 family, are significantly higher than this (many hundreds of cycles).

IMPURE VIRTUALIZATION

As depicted in Figure.3, impure virtualization necessitates the removal of non-privileged sensitive instructions from the virtual machine's code. A method known as binary code rewriting can make this clear: At the time of loading, the executable code is scanned, and any instructions that aren't working are changed with ones that make an exception or provide virtualization in another way, like keeping virtual hardware resources in user mode. Alternately, troublesome instructions could be avoided entirely from the executable code. Pre-virtualization, also known as afterburning, is a mostly automatic method that can be used to accomplish this at compile time [LUC+05]. Alternately, the source code can be manually altered to replace direct access to privileged state with explicit hypervisor invocations (hypercalls) instead. Para-virtualization is the term used to describe this strategy. The procedure remains unchanged from pure virtualization: The hypervisor is invoked whenever a virtualization event occurs, and the guest code executes in the processor's non-privileged execution mode. In addition to being able to deal with hardware that is not suitable for pure virtualization, para-virtualization and pre-virtualization offer another advantage: By replacing multiple sensitive instruction sequences with a single hypercall, they can reduce the number of costly unprivileged mode switches. Impure virtualization is therefore appealing even on fully virtualisable hardware because it has the potential to reduce virtualization overhead.

EMBEDDED SYSTEMS PROPERTIES

Historically, embedded systems were relatively straightforward, one-purpose devices. Hardware limitations (memory, processing power, battery charge) dominated. Additionally, their functionality was largely determined by hardware, with device drivers, a scheduler, and a little control logic serving as software. As a result, they had software complexity ranging from low to moderate. They were constrained by real-time constraints, which place unusual demands on

IJAER

operating systems for general-purpose computing. Additionally, traditional embedded systems are closed: Except for sporadic firmware upgrades, the entire software stack is loaded prior to sale and provided by the device manufacturer. However, general-purpose systems are increasingly being adopted by modern embedded systems. Their software's complexity and number of features are also increasing. Smartphone software stacks currently range from 5 to 7 Mloc and growing. Software occupies literally gigabytes in high-end automobiles, and it is said that loading the software takes longer than building the vehicle itself. Applications originally designed for the PC world are increasingly being run on embedded systems, such as the iPhone's Safari web browser, and new applications, such as games, are increasingly being written by programmers who lack experience with embedded systems. High-level application-oriented operating systems with common APIs (Linux, Windows, and Mac OS) are in high demand as a result. In addition, there is a significant movement toward openness [4, 17]. The owners of the devices want to install and use their own applications on the systems. This requires open APIs (and presents all the security challenges known from the PC world, including infections and worms). However, some of the previous distinctions from general-purpose systems remain. Implanted gadgets are still constant frameworks (or possibly some portion of the product is ongoing). Additionally, they frequently remain resource constrained: As a result of the slow rate at which battery capacity expands over time, mobile devices have limited energy budgets. Additionally, memory is frequently still a cost factor (in addition to being a consumer of energy) given that many embedded systems are sold for just a few dollars. At the same time, embedded systems, which are already present everywhere, are becoming an increasingly integral part of daily life to the point where it is becoming difficult to imagine living without them. They are increasingly being used in situations where a life or mission is at stake. As a result, there are increasing demands placed on security, dependability, and safety.

VIRTUALIZATION USE CASES

The ability to address some of the new challenges posed by them is what makes virtualization useful in embedded systems. One is addressing the conflicting requirements of highlevel APIs for application programming, real-time performance, and legacy support by providing support for heterogeneous operating system environments. Despite efforts to address this, mainstream application operating systems do not support true real-time responsiveness and are not suitable for supporting the large amount of legacy firmware running on current devices (for example, mobile phone baseband stacks alone can measure several Mloc). By allowing the simultaneous execution of an application OS (Linux, Windows, Symbian,...) and a real-time OS (RTOS) on the same processor, virtualization can assist in this regard (see Figure 1). The RTOS can continue to run the (legacy) stack that provides the device's real-time functionality if the underlying hypervisor is able to reliably and rapidly deliver interrupts to it. The application operating system can give the necessary item Programming interface and undeniable level usefulness appropriate for application programming. Keep in mind that the same result can be obtained by employing multiple processor

IJAER

cores, each of which runs its own operating system, provided that the hardware supports securely partitioning memory. The cost of a core is rising while the number of multicore chips is decreasing dramatically. Due to strong non-linearities in power management, two lower-performance cores, each of which can be put to sleep, are also likely to consume less power on average than one higher-performance core [22,23]. As a result, this usage scenario is probably only going to be relevant for a few years. However, since the same software architecture can be transferred between a multicore and a (virtualized) single core virtually unchanged, virtualization facilitates architectural abstraction. Virtualization will be used for a different, longer-term purpose with the upcoming manycore chips. With a lot of processors, embedded systems will probably have problems that are similar to the reasons virtualization is used in business today. By dividing the chip into several smaller multiprocessor domains, a scalable hypervisor could serve as the foundation for deploying legacy operating systems with poor scaling on a large number of cores. A hypervisor can either manage power consumption by removing processors from domains and shutting down idle cores, or it can dynamically add cores to an application domain, which requires additional processor power. Keep in mind that the application operating system must be able to handle varying CPU counts for this to work.) In a hot-failover configuration, the hypervisor can also be used to set up redundant domains for fault tolerance. Security is probably the most compelling case for virtualization. The likelihood of a compromised application operating system skyrockets with the shift toward open systems. As depicted in Figure 2, running such an operating system in a virtual machine that restricts access to the rest of the system can minimize the damage that results. Specifically, services that are accessible from a distance could be encapsulated in a VM, or downloaded code could only be run in a VM environment. If this security use case is true:

- critical functionality can be separated into VMs distinct from the exposed user- or network-facing ones, and
- the underlying hypervisor is significantly more secure than the guest OS (which, first and foremost, necessitates a significantly smaller hypervisor).

The hypervisor will simply increase the size of the trusted computing base (TCB) if those prerequisites are not met. This is not good for security. Lastly, widespread (and standardized) virtualization support paves the way for a novel distribution strategy for software applications: sending the application along with its own OS image. This gives the application developer a well-defined OS environment, which makes it less likely that the deployed software will fail because of configuration mismatches. Automatic removal of replicated page contents can lessen the impact of such a scheme, which has significant resource implications, particularly in terms of memory consumption [6].

IJAER

LIMITS OF VIRTUALIZATION

The aforementioned use cases demonstrate that embedded systems can benefit significantly from virtualization. However, there are significant restrictions on the use of system virtual machines (VMs) in embedded systems; in fact, these restrictions are directly related to the popularity of virtualization. By definition, a virtual machine operates on its virtual hardware as if it had exclusive use of physical hardware; virtualization is all about isolation. It is important to keep in mind that some vendors are not afraid to refer to (highly insecure) OS co-location as virtualization in an effort to capitalize on a well-known marketing buzzword. However, since this type of pseudo-virtualization only addresses the simplest heterogeneous-OS use cases, it will not be considered further.) The model of strongly isolated virtual machines does not meet the needs of embedded systems, as opposed to the server space. Because of their inherent high degree of integration, embedded systems require the cooperation of all subsystems in order to contribute to the system's overall operation. They disrupt the system's functional requirements by being isolated from one another. Effective sharing is necessary for embedded system cooperation between its various subsystems. A mobile phone may receive a video file over the cellular network (via the baseband OS), which is then displayed on the screen (by a media player running on the application OS). This is required for bulk data transfer. At least one additional copy operation is required to transfer this data "virtual-machine style" between the application VM and the real-time VM via a virtual network interface, resulting in a significant waste of processor cycles and battery life. A shared buffer and low-latency synchronization and messaging primitives are clearly the best options. The virtual-machine model, on the other hand, does not account for such operations. Scheduling demonstrates yet another gap between the virtual machine model and the requirements for embedded systems. The hypervisor schedules virtual machines as black boxes, and the guest operating system is in charge of scheduling activities within the VM. However, this is not suitable for embedded systems; As depicted in Figure 3, their integrated nature necessitates a global, integrated scheduling strategy. Background activities with a low priority will also be present in the RTOS domain, despite the fact that real-time activities running in an RTOS typically need to have the highest scheduling priority. These should not be able to override the application OS's user interface. Similarly, in the RTOS environment, the application operating system may be running some (soft) real-time activities, such as the media player, that may take precedence over other real-time activities. It is obvious that virtualization's decentralized, hierarchical scheduling model cannot address the characteristics of embedded systems. Similar to real-time scheduling, energy management is a system-wide problem that cannot be solved locally. Under pressure to meet deadlines, energy management in embedded systems frequently involves weighing energy consumption against performance (i.e., time). However, energy cannot be exchanged for virtual time because it is a global physical resource. This is in contrast to the server industry, where energy efficiency can be achieved through a hierarchical approach rather than energy savings [2-4]. Additionally, virtualization does little to address embedded systems' most pressing issue: the

IJAER

expanding programming intricacy, which takes steps to sabotage gadget heartiness and wellbeing. The acknowledged programming approach for addressing intricacy challenges is to utilize embodied parts [2-5] for shortcoming regulation. Encapsulation is provided by virtualization, but its granularity is insufficient to make a significant difference. After all, a virtual machine is designed to run an operating system that supports its software and emulates hardware. Because of this, virtual machines have a lot of weight, and embedded systems can't run more than a few of them (keep in mind that memory costs energy and money to make). A component framework that is lightweight in terms of memory overhead as well as the cost of inter-component communication and does not increase the system's overall complexity is needed to address the software complexity problem. Last but not least, control over information flow is becoming an increasingly significant issue. Although it may initially appear surprising, embedded systems are no longer single-user devices. For instance, a mobile phone handset has at least three distinct classes of stakeholders, or "users," because they each have distinct access rights:

1. The owner, or the person who purchased the gadget in some way.

2. The one who grants access to a wireless network is the wireless service provider. Similar to the DoCoMo/Intel OSTI proposal [7], which argues for a separation of enterprise and private service when the same physical device is used for both private and business purposes, we might see multiple concurrent service providers for the same physical device in the future.

3. Outsider specialist organizations who utilize the remote availability to offer types of assistance free of the remote specialist organization. Providers of multimedia content for information or entertainment fall under this category. Financial transactions like payments for random goods and services are becoming more and more common.

In general, these users distrust each other, and each has assets on the device they want to protect. The owner wants to keep the address books, emails, and documents that the other users do not have a legitimate interest in private and out of their reach. These would be further divided into private and enterprise data in the OSTI scenario, resulting in multiple logical owner-type users (or "roles") In order to comply with legal requirements and ensure correct billing, the service provider must ensure the network's integrity and that all network access is properly authenticated without interfering with others' legitimate use. It subsequently needs certainty that the gadget will stick to conventions. The content providers need assurance that their data will only be used in accordance with the owner's license (for example, only for a limited time and without being copied to other devices). Both the owner and the providers require assurance regarding the security of financial transactions with other third-party service providers, which includes strong access token protection. We along these lines have an entrance control issue not divergent based on what is seen as in conventional (time-shared) multi-client frameworks. Several agents have legitimate access rights to some data but no access rights to other data. A security monitor must be able to enforce access policies while information must flow between the agents. Importantly, many of the

IJAER

subsystems will be able to handle sensitive user data. The guest operating system must be trusted to enforce the information-flow policies if these subsystems are contained within a virtual machine. This would be a (potentially massive) expansion of the system's trusted computing base, which is clearly worse for security than simply trusting a single operating system. As previously stated, it would eliminate many of the potential security advantages of virtualization. The TCB should be reduced to a minimum for security reasons, and it should not be dependent on large application OSes (with a high risk of compromise).

CONCLUSIONS

Although virtualization has many features that are appealing to the embedded community, it is not a good fit for contemporary embedded systems on its own. Fine-grained encapsulation, integrated scheduling, and information-flow control require more general OS technology. All of the necessary features can be provided by high-performance microkernels, making it possible to switch from monolithic software stacks to componentized (fault-resilient) software stacks. The prospect of formal verification of the implementation of the microkernel opens up an exciting path to systems with unprecedented reliability. Hypervisors appear to be gaining a microkernel-like appearance, including the incorporation of microkernel-like primitives to address some virtualization drawbacks. However, the inherent advantage of microkernels is that they enable a smaller TCB . Additionally, due to the low scalability of formal verification methods, only a real microkernel will ever be able to provide the advantages of formal verification. We conclude that virtualization should be implemented in some form in future embedded systems for good reasons. System virtual machines' benefits, on the other hand, will only be fully realized if the operating system technology is changed to one based on high-performance microkernels.

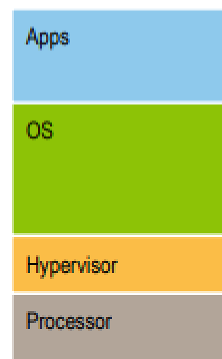

Figure 1. A virtual machine. The hypervisor (or virtual-machine monitor) presents an interface that looks like hardware to the "guest" operating system.

IJAER

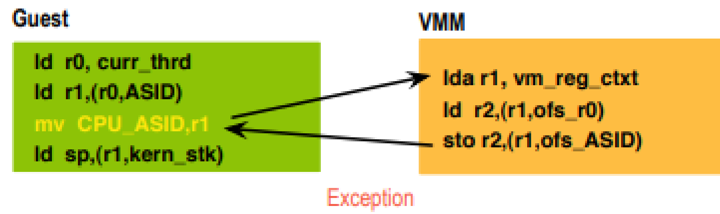

Figure 2. Most instructions of the virtual machine are directly executed, while some cause an exception, which invokes the hypervisor which then interprets the instruction.

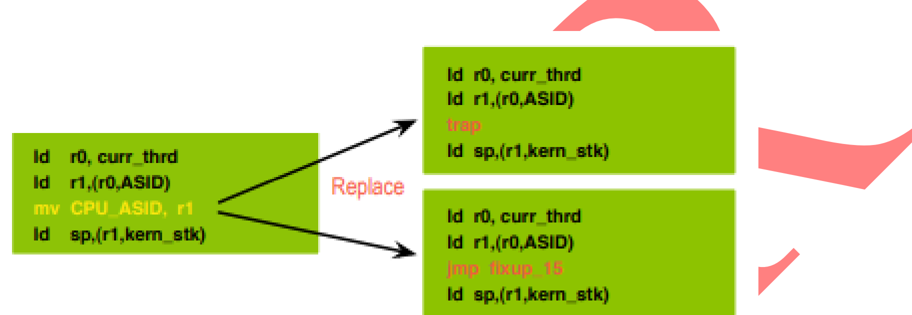

Figure 3. Impure virtualization techniques replace instructions in the original code by either an explicit hypervisor call (trapping instruction) or a jump to user-level emulation code.

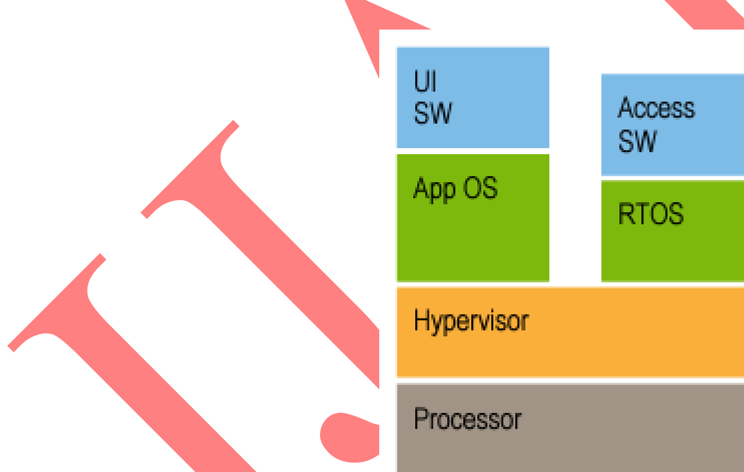

Figure 4: The primary use case for virtualization in embedded systems is the co-existence of two completely different OS environments, real-time OS and high-level application OS, on the same processor.

REFERENCES

1. Gilles, K., Groesbrink, S., Baldin, D. and Kerstan, T., (2012). Proteus hypervisor: Full virtualization and paravirtualization for multi-core embedded systems. *In Embedded Systems:*

Design, Analysis and Verification: 4th IFIP TC 10 International Embedded Systems Symposium, IESS 2011, Paderborn, Germany, June 17-19, 2012. Proceedings 4 (pp. 293-305). Springer Berlin Heidelberg.

2. Son, S.H., (2011). An MMU Virtualization for Embedded Systems. *In IT Convergence and Security 2012* (pp. 247-252). Springer Netherlands.
3. Patel, A., Daftedar, M., Shalan, M. and El-Kharashi, M.W., (2011), March. Embedded hypervisor xvisor: A comparative analysis. *In 2012 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (pp. 682-691). IEEE.
4. Muench, D., Isfort, O., Mueller, K., Paulitsch, M. and Herkersdorf, A., (2011, December). Hardware-based I/O virtualization for mixed criticality real-time systems using PCIe SR-IOV. *In 2012 IEEE 16th International Conference on Computational Science and Engineering* (pp. 706-713). IEEE.
5. Trujillo, S., Crespo, A. and Alonso, A., (2010, September). Multipartes: Multicore virtualization for mixed-criticality systems. *In 2012 Euromicro Conference on Digital System Design* (pp. 260-265). IEEE.
6. Armbrust, E., Song, J., Bloom, G. and Parmer, G., (2012, December). On spatial isolation for mixed criticality, embedded systems. *In Proc. 2nd Workshop on Mixed Criticality Systems (WMC)*, RTSS (pp. 15-20).
7. Paulitsch, M., Duarte, O.M., Karray, H., Mueller, K., Muench, D. and Nowotsch, J., (2011, August). Mixed-criticality embedded systems--a balance ensuring partitioning and performance. *In 2012 Euromicro Conference on Digital System Design* (pp. 453-461). IEEE.
8. Pinto, S., Oliveira, D., Pereira, J., Cardoso, N., Ekpanyapong, M., Cabral, J. and Tavares, A., (2012, September). Towards a lightweight embedded virtualization architecture exploiting arm trustzone. *In Proceedings of the 2013 IEEE Emerging Technology and Factory Automation (ETFA)* (pp. 1-4). IEEE.
9. Bardhi, B., Claudi, A., Spalazzi, L., Taccari, G. and Taccari, L., (2012). Virtualization on embedded boards as enabling technology for the Cloud of Things. *In Internet of Things* (pp. 103-124). Morgan Kaufmann.
10. Armbrust, E., Song, J., Bloom, G. and Parmer, G., (2011, December). On spatial isolation for mixed criticality, embedded systems. *In Proc. 2nd Workshop on Mixed Criticality Systems (WMC)*, RTSS (pp. 15-20).
11. Wildermann, S. and Teich, J., (2012, September). Self-integration for virtualization of embedded many-core systems. *In 2011 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems Workshops* (pp. 170-177). IEEE.
12. García-Valls, M., Cucinotta, T. and Lu, C., (2012). Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 60(9), pp.726-740.

IJAER