embeddedworld2018
Exhibition&Conference
... it's a smarter world

Challenges in Virtualizing Safety-Critical Cyber-Physical Systems

Alessandro Biondi, Mauro Marinoni, and Giorgio Buttazzo
Scuola Superiore Sant'Anna
Pisa, Italy
{alessandro.biondi, mauro.marinoni, giorgio.buttazzo}@santannapisa.it

Claudio Scordino and Paolo Gai
Evidence SRL
Pisa, Italy
{claudio, pj}@evidence.eu.com

***Abstract* — Embedded computing platforms are evolving towards heterogeneous architectures that require new software support for simplifying their usage, optimizing the available resources, and providing a predictable runtime behavior for managing concurrent safety-critical applications. This paper describes the main challenges in providing such a software support through virtualization techniques, while taking into account safety requirements, security issues, and real-time performance. An automotive application is considered as a case of study to illustrate some of the presented concepts.**

Keywords — Heterogeneous platforms, embedded computing, real-time systems, virtualization, hypervisor.

I. INTRODUCTION

The design of computing infrastructures for modern cyber-physical systems is facing with two major trends that are significantly steering the development process of embedded software. On one hand, the last years have been characterized by a continuous increase of the software complexity to meet more and more richer functional requirements and to support new technologies. At the same time, computing platforms are evolving toward heterogeneous designs that integrate multiple components such as multicore processors, general-purpose graphic processing units (GPGPUs), and field programmable gate arrays (FPGAs), which allow power-efficient parallel execution of multiple software systems at the cost of a paradigm shift in their development.

These two trends are increasingly pushing software designers to integrate a higher number of functions in the same hardware platform, typically resorting to methodologies such as component-based software design (CBSD) and also facing with the problem of incorporating legacy software. Furthermore, in many industrial fields, integration is considered the most affordable solution to problems related to space, weight, power, and cost (SWaP-C).

Virtualization of computational resources established as a de-facto technique to address these needs while efficiently exploiting the processing power of modern platforms. Virtualization is typically achieved via *hypervisors* (also called virtual machine monitors), which allow executing multiple software domains upon the same platform, each of them possibly executing a different operating system (OS). The domains benefit from the illusion of disposing of a dedicated computing platform, while in reality the access to the shared computational resources is regulated by the hypervisor, which typically offers to the domains sets of virtualized memory address spaces, CPUs, and possibly peripherals. Nowadays, this technology is increasingly adopted to realize multi-OS solutions [22] for mixed-criticality systems, integrating a mission-critical real-time operating system (e.g., to perform sensing, control, and actuation tasks), with rich, non-critical operating systems such as Linux, which exploit a large availability of drivers, libraries, and connectivity stacks. Realistic designs possibly also include the integration of legacy software systems as-a-whole, i.e., with their original operating system, drivers, and configurations, thus favoring the evolution of cyber-physical systems towards centralized schemes with few but powerful computing platforms.

Orthogonally to such major trends, designers of new-generation embedded software cannot neglect *safety* and *security* needs, which inevitably affect the functionality provided by virtualization stacks. The former are driven by increasingly stringent legal regulations and certifiability requirements, while the latter are becoming of paramount importance due to the exposure of embedded computing platforms by means of network connections. The integration of components with different safety and security levels (also known as MILS systems) may pose hazards in guaranteeing key requirements of the critical software such as timing constraints and data integrity and confidentiality. For instance, if no proper *isolation* mechanisms are provided by the hypervisor, a malfunctioning or an attack interesting a low-critical domain may arbitrarily delay the execution of critical tasks, thus compromising the system behavior or strongly jeopardizing its performance.

The joint consideration of all such a kind of aspects poses several challenges in the development of suitable virtualization layers. The scope of this short paper is to discuss some of such challenges, with a particular focus on temporal and spatial isolation of software domains, timing predictability, resource

contention, and the management of hardware-based security technologies.

II. BACKGROUND

A. Hypervisors

The concept of Hypervisors dates back to the 60's [13], but it became significant in the last decade as a fundamental solution to harness the complexity of the modern hardware platforms, and the multiple applications executing concurrently on top of them. This need for isolation could be declined in different ways depending on the specific application requirements and the underlying platform executing it.

A platform on which the hypervisor executes is denoted as the *host machine*, and each virtual machine managed by the hypervisor is called a *guest*. The two main features upon which is based the classification of a hypervisor concern the type of implementation and the abstraction provided to the guest virtual machine. There are two types of hypervisor:

- Type-1, also called native or bare-metal, which directly run on the hosting hardware to control it and to handle guest operating systems;
- Type-2, also called hosted, where the hypervisor is provided as an extension to an operating system that is executed on the host while the guests run as tasks..

Another element of distinction comes from the API exposed by the host to the generic guest OS:

- In *fully virtualized* solutions the guest executes in a transparent manner and without software modifications, while the hypervisor provides the API to emulate the underlying platform;
- In a *paravirtualized* implementation the guest is aware of the presence of virtualization. Thus it uses an API similar, but not identical, to that of the underlying hardware. This allows to create specific solutions and reduce the overhead.

Due to the advantages of higher flexibility and no modifications required in the guest domains, the hardware manufacturers started providing virtualization extensions to support full virtualization, which allow minimizing the overheads resulting from the emulation of the underlying platform.

B. Existing solutions

The wide range of application scenarios and platforms fostered the creation of a significant number of hypervisors, each of them with a focus on a subset of the several issues concerning virtualization. Moreover, the profound interaction between the hypervisor and the hardware platform leads to a considerable effort when porting the hypervisor to a new architecture, also due to the extensive use of specific platform features to improve performance. The result is a reduced set of hypervisors available for each particular platform.

Since some application fields, like mainframes, cloud infrastructures, and virtualized network infrastructures highly benefit from virtualization and massively relies on Linux, several hypervisors pivoting on the latter have been developed. Among the firsts and one of the most famous is Xen [14], which executes Linux in a privileged *domain* called *dom0*. The wide range of supported platforms is considered one of its main advantages, but also as a drawback because it has lead to a considerable codebase. A similar approach is followed by KVM [15], which is a virtualization infrastructure available in the mainline kernel that turns it into a type-1 hypervisor. Jailhouse [16] is a type-1 partitioning hypervisor, more concerned with isolation rather than virtualization, aiming at creating a small and lightweight hypervisor targeting industrial-grade applications. Like Xen, Jailhouse requires Linux to provide the management interface, which allowed keeping the size of source code small. Like KVM, it is loaded from a regular Linux system, but when started, it takes full control of the hardware and splits the hardware resources into isolated compartments (called *cells*) that are entirely dedicated to guest software programs (called *inmates*). One cell runs the Linux OS and is known as the root cell, that is similar to the dom0 in Xen, but the root cell doesn't assert full control over hardware resources as *dom0* does.

When dealing with embedded systems and their possible requirements regarding safety and security, it is essential to exploit solutions characterized by a small codebase both for SWaP and certification issues. Xvisor [17] is a type-1 hypervisor, aiming at providing an entirely monolithic, light-weight and portable virtualization solution. The most appealing characteristic of Xvisor is that it provides full virtualization, and therefore supports a wide range of unmodified guest operating systems. NOVA [18] is an academic hypervisor designed at TU Dresden. It follows the micro-kernel approach, and it has been developed using the C++ programming language. Another significant feature is the fixed-priority preemptive scheduler with execution time budgets and priority inheritance. XtratuM [19] is a hypervisor specially designed for real-time embedded systems, providing fixed priority scheduling, and relying on paravirtualization. Fiasco [20] is a hypervisor based on the L4 ABI and is implemented using the C++ programming language. The Fiasco kernel is enriched by a broad set of user-space components, collectively called L4 Runtime Environment (L4Re). Attempts have been made to exploit the TrustZone security features available on modern ARM processors into hypervisors. An example is the SierraVisor [21] hypervisor.

Despite all the effort from these and other projects, there are still significant issues to be addressed before being able to provide a considerable level of isolation and virtualization for modern heterogeneous platforms. The next section outlines some of the more significant ones.

III. MAJOR CHALLENGES

A. Achieving effective isolation on multicores

Isolation capabilities are of paramount importance for an hypervisor to be used within a mixed-criticality system. Two types of isolation can be identified: *spatial* and *temporal*. Most (if not all) solutions provide support for spatial isolation of memory spaces, which is typically achieved by means of memory virtualization leveraging memory management units (MMU). Temporal isolation is generally realized by reserving

dedicated CPUs to a domain, or by implementing bandwidth reservation schemes for the CPU time, e.g., by reserving a budget of execution time that is periodically provided to a domain by the hypervisor scheduler.

Although these features are primary, and in fact are widely supported by open-source and commercial hypervisors, they are not enough to guarantee an effective isolation on commercial off-the-shelf (COTS) multicore platforms. Indeed, even if the domains access separate memory regions, and execute upon disjoint sets of CPUs, mutual interference is still possible due to the implicit contention of architectural resources such as caches and memory banks. These resources are typically not under the control of the hypervisor, but rather they are transparently managed by chip subsystems (e.g., the memory controller) that in most cases are not conceived to enforce isolation nor to guarantee timing predictability [5][6].

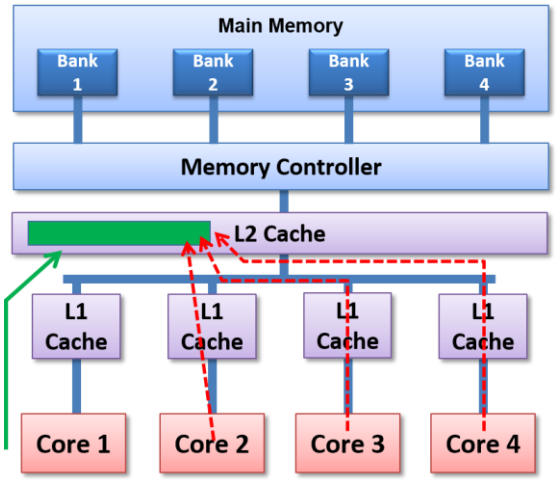

Figure 1 - Inter-core interference in accessing a shred level of cache

For instance, consider a quad-core platform with private level-1 caches for each core and a shared level-2 cache, as it is illustrated in Figure 1. Suppose that a critical real-time operating system is executing upon the first core, while the remaining three cores are dedicated to execute a general-purpose Linux domain. The execution of the critical domain results in fetching data and code from the main memory, consequently populating the level-2 shared cache (green box in the figure). In parallel, the Linux domain can also populate the same cache, with the result that the content stored by the critical domain can be *evicted*, hence provoking cache misses at the next access. This phenomenon may generate large and unpredictable interference across domains, thus breaking isolation by introducing a strong coupling of their timing properties. Conversely, if the Linux domain is subject to an attack or a malfunctioning such that it floods the system with memory transactions, proper isolation mechanisms should shield the critical domain.

To further complicate the problem, inter-domain interference can also arise when accessing the main memory, e.g., in correspondence to cache misses. The access to DRAM memories is subject to highly variable delays that depends on the actual memory location to be accessed and simultaneous pending memory transactions. Furthermore, DRAM memory controllers generally resort to scheduling algorithms that re-order memory accesses with the aim of improving throughput. While these algorithms provide benefits in the average-case, they leave room for pathological scenarios that lead to high worst-case latencies, hence harming the system predictability.

In the literature, several clever solutions have been proposed to solve this kind of issues in non-virtualized multicore systems. Software-based approaches such as *cache coloring* or *cache lockdown* [7] can be employed to partition the amount of cache used by a core, or more in general by a set of software tasks. Reservation of memory bandwidth [5] and bank-aware memory allocators [7] have also been proposed to control the contention in accessing the main memory. Nevertheless, to the best of our records, adequate support for such techniques is limited in commercial hypervisors.

Modica et al. [8] realized effective isolation mechanisms for shared caches and main memories in an open-source hypervisor targeting ARM platforms. The authors developed a new virtual memory allocator that employs cache coloring to statically isolate the amount of shared cache reserved to each domain. Furthermore, a bandwidth reservation mechanism to access the main memory has been integrated with the hypervisor scheduler. Their experimental results showed that inter-domain interference can increase the execution time of state-of-the-art benchmarks up to the 50%, while the realized mechanisms can restore isolation at the price of degrading average-case performance.

B. *Virtualization of FPGAs and GPGPUs*

Heterogeneous platforms that include FPGAs and/or GPGPUs represent very attractive and powerful solutions to implement modern cyber-physical systems, but at the same time they introduce new problems in terms of resource management. Concerning virtualized systems, FGPAs and GPGPUs should also be controlled by the hypervisor and made available to domains in a controlled manner.

Modern FPGAs dispose of dynamic partial reconfiguration (DPR) capabilities, which allow reprogramming a portion of the FPGA area while the rest continues to operate. This interesting feature may be used to virtualize the FPGA area supporting several hardware modules and accelerators in time sharing, whose overall area consumption exceeds the one that is actually available in the platform. A framework [11] has also been proposed to ensure that the reconfiguration and area contention delays are predictable, thus making realistic the adoptance of this technique in the context of critical systems. Static FPGA virtualization is also possible by controlling its configuration phases. Unfortunately, no integration within a hypervisor is today available.

Dually, work has also been dedicated to the development of software mechanisms to integrate the advantages of GPGPU into the virtualization paradigm. Hong et al. [23] provided an overview of the state-of-the-art of virtualization techniques, hardware supports, and scheduling mechanisms for multiple concurrent requests. They also outlined a list of challenges that still require being addressed to improve the exploitation of

GPGPUs, ranging from overheads reduction to energy management, from scalability and space optimization to security.

Another issue consists in the fact that modules deployed onto the FPGA and GPGPUs can typically act as memory masters on the system bus, hence (i) generating additional memory interference (e.g., see [10]) that complicates the problems discussed in the previous section, and (ii) potentially exposing memories to an uncontrolled access that may bypass the spatial isolation. The first problem needs to be addressed with adequate support, such as specialized software-based memory bandwidth controllers, or in the case of FPGAs with the development of hardware bandwidth controllers deployed onto the FPGA and managed by the hypervisor. The second problem requires dealing with virtualization techniques and components such as I/O MMUs.

C. *Supporting hardware-based security technologies*

Due to the external exposure by means of network and bus connections, security issues became central aspects in the design and development of modern embedded computing systems. Although a rich set of software-based techniques have been developed to increase the security level of a software system, cyber attacks are also increasingly becoming more and more complex, defeating most attack mitigation techniques and/or exploiting wrong software configurations. With the intent of providing a robust support to implement security features, chip makers are moving towards architectures that offer hardware-based solutions to realize trusted execution environments (TEEs). TEEs must be strictly isolated for the normal execution environment and should also dispose of dedicated computing resources.

One of the most popular of such technologies is TrustZone developed by ARM. TrustZone provides hardware-based isolation of two execution worlds: *secure*, conceived to support the execution of a TEE, and *non-secure*, which is provided to host the execution of a rich (classical) operating system. TrustZone-enabled chips may also include support for secure boot, i.e., cryptographic validation of the firmware to be executed, and cryptographic hardware accelerators. The introduction of such features poses new challenges when realizing a security-aware virtualization stack.

First, there is the need to virtualize such hardware-based security technologies to allow the coexistence of multiple domains each potentially comprising a TEE running in a virtualized secure world. Initial attempts in this direction have been made by Cicero et. al [9], which proposed an open-source dual-hypervisor solution where two jointly-configured hypervisors are employed to virtualize secure and non-secure worlds, respectively, both orchestrated by a monitor firmware that handles world switches and dispatches interrupt signals. This solution avoids the existence of a single point of failure and aims at containing the run-time overhead. Remarkable efforts have also been spent by Hua et al. [12], which proposed a centralized solution to virtualize TrustZone by building upon the Xen hypervisor.

Second, hypervisors should offer the virtualization of cryptographic hardware resources, possibly guaranteeing strict integrity and confidentiality of data even in the presence of side-channel attacks. Built-in support for software-based attack mitigation techniques such as data execution prevention (DEP), address-space layout randomization (ASLR), and control flow integrity (CFI) are also desirable. The latter require careful attention when integrated with virtualization mechanisms.

Third, to the end of supporting component-based software design and possibly open environments, hypervisors should provide software authentication mechanisms also at the level of domains, paying particular attention at rollback-based attacks. The authors believe that list is not limited to the above-mentioned challenges and that security-related aspects will likely steer the design of future virtualization software.

IV. THE AUTOMOTIVE CASE

As a proof of concept, this section describes a realistic scenario related to the automotive domain in which virtualization is applied. The described solution, from the RETINA project [1], aims at providing an AUTOSAR-compliant software stack for next-generation automotive systems. The stack allows the integration of components with different criticality levels onto modern multi-core SoCs, reducing the overall time-to-market and manufacturing costs.

At the lowest level, the stack consists of an hypervisor to enforce isolation (thus, reliability and safety) between the guest operating systems. The RETINA project relies on Jailhouse [2], a small and lightweight type-1 hypervisor developed by Siemens and released as Open-Source software. The hypervisor supports both x86-64 and ARM-based platforms, provided the availability of hardware virtualization instructions. Rather than providing resource virtualization and scheduling (like e.g. the Xen hypervisor), Jailhouse focuses on isolation and resource partitioning. For this reason, there is no intra-core scheduling (i.e., each core cannot run more than one guest OS) and resources are statically assigned to only one guest. This static approach allows to:

- provide average latencies and jitters similar to bare-metal solutions, due to the low run-time overhead;
- ease potential certification processes in the future, thanks to a very small codebase.

On top of the hypervisor, the RETINA project runs two guest OSs with different criticality levels. The real-time and safety-critical tasks are run by the ERIKA Enterprise RTOS [3]. ERIKA Enterprise is a tiny RTOS (i.e. a few KBs of footprint) designed and certified for the automotive market. It is developed by Evidence Srl and released as Open-Source software under a dual licensing model.

The less critical tasks (e.g., HMI, logging, etc.), instead, are executed on a Linux guest, improved through the PREEMPT_RT real-time patch [4] when needed. The communication between the two OSs is done by means of a library exposing an API similar to the one specified by the AUTOSAR COM standard. The library is meant to be used by an AUTOSAR Run-Time Environment (RTE) generator developed by Evidence Srl for its RTOS. Most critical tasks are run using the SCHED_DEADLINE Linux scheduler [17].

Figure 2 summarizes the main components of the automotive software stack described above.

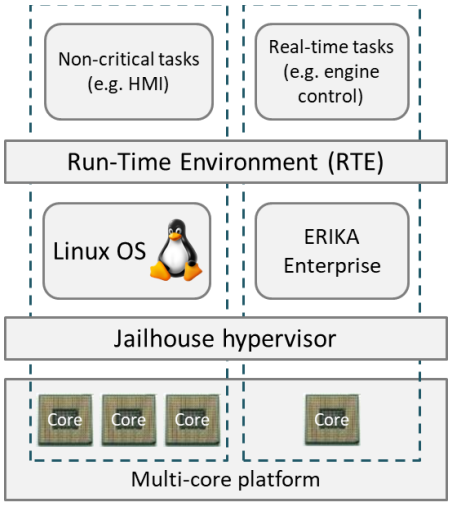

Figure 2 - Multi-OS automotive software stack developed for the RETINA project.

V. CONCLUSIONS

This paper presented some of the major challenges in providing a software support for exploiting modern heterogeneous platforms for complex safety-critical systems consisting of several interacting components with real-time requirements. Virtualization techniques, successfully used to isolate the behavior of software components running on the same processor, are considered to be extended for managing other architectural resources, such as shared memories, and other computational units, such as FPGAs and GPUs. Issues concerning safety, security, and real-time performance are also discussed and illustrated using a case of study taken from the automotive domain.

REFERENCES

[1] RETINA EUROSTARS project, http://retinaproject.eu/

[2] Siemens, Jailhouse hypervisor, https://github.com/siemens/jailhouse

[3] Evidence Srl, ERIKA Enterprise RTOS, http://www.erika-enterprise.com/

[4] The Linux Foundation, Real-Time collaborative project, https://wiki.linuxfoundation.org/realtime

[5] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013, pp. 55–64

[6] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), April 2014.

[7] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," ACM Comput. Surv., vol. 48, no. 2, Nov. 2015.

[8] P. Modica, A. Biondi, G. Buttazzo, and A. Patel, "Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms," in Proceedings of the 18th IEEE International Conference on Industrial Technology (ICIT 2018), Feb. 2018.

[9] G. Cicero, A. Biondi, G. Buttazzo, and A. Patel, "Reconciling Security with Virtualization: A Dual-Hypervisor Design for ARM TrustZone," in Proceedings of the 18th IEEE International Conference on Industrial Technology (ICIT 2018), Feb. 2018

[10] B. Forsberg, A. Marongiu and L. Benini, "GPUguard: Towards supporting a predictable execution model for heterogeneous SoC," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, Lausanne, 2017, pp. 318-321

[11] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, "A framework for supporting real-time applications on dynamic reconfigurable FPGAs," in Proc. of the IEEE Real-Time Systems Symposium (RTSS 2016), December 2016, pp. 1–12

[12] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM trustzone," in In Proc. of the 26th USENIX Security Symposium, 2017.

[13] R. Adair, R. Bayles, L. Comeau, and R. Creasy, "A virtual machine system for the 360/40," Technical Report 320-2007, IBM Corporation, Cambridge Scientific Center, May 1966.

[14] Xen project, https://www.xenproject.org/

[15] Linux Kernel Virtual Machine, http://www.linux-kvm.org/page/Main_Page

[16] Jailhouse project page, https://github.com/siemens/jailhouse

[17] J. Lelli, C. Scordino, L. Abeni, D. Faggioli, "Deadline scheduling in the Linux kernel", Software: Practice and Experience, 46(6): 821-839, June 2016.

[18] Nova hypervisor, http://www.hypervisor.org

[19] XtratuM project page, http://www.xtratum.org

[20] Fiasco project page, https://l4re.org/fiasco/

[21] SierraVisor, http://www.openvirtualization.org

[22] PikeOS hypervisor, https://www.sysgo.com/products/pikeos-hypervisor/

[23] Hong, Cheol-Ho & Spence, Ivor & Nikolopoulos, Dimitrios, "GPU Virtualization and Scheduling Methods: A Comprehensive Survey". ACM Computing Surveys. 50. 1-37, 2017.