2017 NDIA GROUND VEHICLE SYSTEMS ENGINEERING and TECHNOLOGY SYMPOSIUM
Vehicle Electronics and Architecture (VEA) Technical Session
August 8-10, 2017 – Novi, Michigan

PORTING OPERATING SYSTEMS TO RUN IN XEN VIRTUAL MACHINES

Jarvis Roach
DornerWorks, Ltd.
Grand Rapids, MI

ABSTRACT

Semiconductor manufacturers are creating new System on Chips that allow embedded system developers to design consolidated architectures to reduce size, weight, power, and cost. However, combining software functions onto a single computing resource creates safety and security concerns due to reduced fault containment and increased coupling between software components. Safety and security-conscious industries use various software separation solutions to isolate software functions logically in order to achieve a comparable level of decoupling and fault containment that distributed/federated systems enjoy as a by-product of their system architecture. This paper will assess the suitability of common separation solutions for use in embedded systems and explain our preference for Xen, an open source Type I hypervisor. This paper will also examine reasons for porting operating systems to run in virtual machines, also known as paravirtualization, and evaluate how certain properties of operating systems can impact this task. Finally, we will conclude with lessons learned from our efforts paravirtualizing a number of operating systems.

INTRODUCTION

A System on Chip (SoC) is a powerful and versatile microchip that incorporates previously separate electronic components into a single integrated circuit. SoCs typically include several processing cores, often with multiple clusters of heterogeneous processors, along with dedicated I/O controllers, memory blocks, timers, voltage/power regulators, memory bus controllers, and various other peripherals. The available processing power, availability of I/O, and level of integration makes SoCs an ideal target platform onto which numerous software functions (SWFs) can be consolidated, enabling an overall reduction of size, weight, power, and cost – a key goal for many ground vehicle systems.

However, combining software functions on a common computing platform creates safety and security concerns. In non-consolidated systems (also known as distributed or federated), each discrete microprocessor provides a natural zone of fault containment as a by-product of the hardware architecture. Distributed and federated architectures also impose loose coupling between software functions executing on different microprocessors since interactions between them must be facilitated by some manner of external communication. These benefits of fault containment and enforced loose coupling can be lost when software

functions are consolidated onto a common SoC, which is a concern for industries where safety and security are of paramount importance, such as aviation and defense.

In this paper, we review the history of separation solutions and the need for them in modern systems. Next, we assess software methods used in various industries to logically isolate, or partition, software functions in order to achieve increased fault containment and loose coupling, enumerating the benefits and limitations of each method. We make the case for using virtualization provided by a Type I hypervisor as a separation solution well suited for most embedded systems with explicit safety or security requirements. We conclude the section explaining our preference for the Xen Project hypervisor as a separation solution for embedded projects.

The next section of the paper focuses on the operating system (OS) running in a virtual machine (VM) created by the hypervisor. We list typical reasons for porting an operating system to run in a virtual machine, also known as paravirtualizing. Then we discuss properties of potential operating systems and how they may be impacted by virtualization. We end that section with a description of three licensing models – permissive, copyleft, and proprietary – and their implications in order to highlight how licensing can affect the decision to paravirtualize an operating system.

In the final section of this paper we describe our experiences and share lessons learned from paravirtualizing the following operating systems: the Xilinx standalone library, Micrium μC/OS-III, Real Time Engineers FreeRTOS, and Wind River VxWorks 7.0. While our experiences are with paravirtualizing to a specific virtual environment, VMs created by Xen, these lessons learned can be generalized and applied to paravirtualization efforts for other hypervisors.

THE NEED FOR SEPARATION IN MODERN SYSTEMS

Moore's Law, the doubling of components on an integrated circuit every two years, coupled with market forces ultimately drives the need for separation in modern systems. As chips become more powerful, system developers are presented with the challenge of effectively using them. The processing requirements for a single software function do not follow the same exponential growth that processor performance enjoys. It is natural to conclude that system developers would want to consolidate existing SWFs in the system onto a single chip in order to avoid wasting any available processing power. It is also natural to conclude that developers would also want to take advantage of any remaining processing power to add new software functions to support new features to provide market advantage over competitors. The days of running a single SWF on a processor are over; it is not economical nor sustainable for the vast majority of business cases.

As a single hardware resource becomes responsible for hosting multiple SWFs, separation solutions become necessary to restrict unfettered growth in complexity and to mitigate the impacts of concurrent execution of those software functions. Without separation solutions, the number of possible interactions between different pairs of software functions grows geometrically with the number of functions consolidated, dramatically increasing the effort to analyze and describe the interactions and side effects.

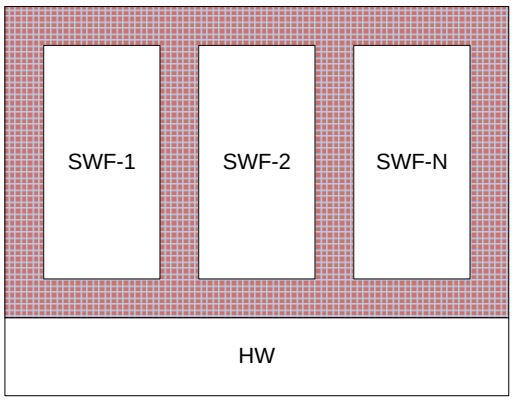

Figure 1: Separation of SWFs

Separation solutions are useful for providing fault containment by preventing erroneous or malicious behavior in one software function, or application, from affecting another. Early literature [1] identifies the need for spatial separation for fault containment,

which is commonly achieved by restricting the memory locations a SWF can access, thus protecting the data and instructions of the other SWFs in a consolidated system. ARINC 651 also identifies the need for temporal separation, later expanded upon in ARINC 653, where a SWF in one partition cannot affect the execution time or time to access a resource of a SWF in another partition [2]. This goal has become more difficult to achieve with increases in hardware complexity.

Separation solutions are also useful for enforcing greater decoupling between software components. Coupling between software components leads to various issues with development, integration, maintenance, and future migrations. This is because coupling leads to complex dependencies between software components, often implicit or unknown, such that a change to, or addition of, a software component often has a wide-reaching and unexpected ripple effect throughout the system. Separation solutions can be used to enforce strong decoupling where any dependencies between software functions are made explicit, making it easier to understand and eliminate unintended or unexpected interactions. Strong decoupling also allows greater freedom to develop software functions in parallel with a higher confidence that the different pieces will not interfere with one another when integrated together. These properties allow development and integration of SWFs to different safety and security criticalities. This can be a big cost savings as lower criticality SWFs do not have to be developed with the same design assurances as the higher criticality SWFs. Strong decoupling also reduces the effort of re-using existing software and the risk that adding new software functions will destabilize the system, resulting in an overall reduction of non-recurring engineering costs.

Table 1: Benefits of Separation

Summary of Separation Benefits
Simplifies management of multiple SWF on a hardware target • Increases utilization of hardware resources
Improves containment • Fault • Exploit
Enforces decoupling • Easier integration • Easier re-use • Enables mixed criticality

Software separation solutions have seen wide spread use in the following markets: cloud servers, security, and safety.

Separation and the Cloud

Consolidation of software functions to fewer processors has been seen before: in the server market. Initially most businesses hosted a single application on a server due to the relative parity between an application's processing requirements and the server's processing power, and also due to early limitations of the server's OS when running multiple applications simultaneously. Over time, increases in computing power resulted in excess capacity, creating demand for a means to consolidate applications in a way that was also easy to manage them. Demand for a solution to provide backwards compatibility for obsolescent operating systems also grew during this time, as this enabled businesses to continue using legacy applications dependent on those outdated operating systems. VMWare provided the first commercially available hypervisors for x86 computers in 2001 that met both of these needs. An initial consolidation ratio of 5:1 allowed businesses to eliminate four out of five servers [3]. Recent VMWare reports show consolidation ratios of 15:1 [4]. As

Porting Operating Systems to Run in Xen Virtual Machines, Roach.

more of a company's services were migrated to run in virtual machines, it became possible to outsource data center services entirely, and infrastructure-as-a-service and platform-as-a-service in the cloud became a viable business model, allowing companies to host their applications on remote virtual servers.

Embedded SoCs have reached a similar point in their evolution, as their processing power is surpassing the processing requirements of the SWFs that can easily be managed to run on it. If the market follows the same pattern, we will see hypervisors emerge as the separation solution of choice, followed by a period of about ten years of consolidation of software functions. One conjecture is that this period will be followed by mass migration of those embedded applications/software functions to more remote servers, and indeed this is the paradigm seen evolving with the Internet of Things (IoT). However, another possible evolution path is one where embedded applications/SWFs are kept boxed within the boundaries of an embedded system, but are allowed to migrate between different processing nodes within the system in order to mitigate hardware failure or to achieve greater performance [5].

Separation and Security

Intuitively, separation supports cybersecurity as it provides a means to enforce the isolation principle. Use of various separation technologies for improving security by way of the isolation principle in computing systems can be found in the literature as early as 1974 [1]. Rushby argues for the use of specialized virtual machines, a separation solution described in more detail later in this paper, for security systems, coining the term and concept of a *separation kernel*, a component used to create virtual environments for distributed security functions [6]. Separation kernels evolved into a component of the Multiple Independent Levels of Security (MILS) architecture [7] presented by Mark Vanfleet at the Open Group Security Forum in 2002 [8]. The MILS architecture was realized by Green Hills, LynxWorks, and WindRiver in their security products. More recently, VanderLeest [9] has suggested use of a more general purpose hypervisor for security in embedded products and McDermott, et al. [10] also suggest use of virtualization provided by hypervisors as an alternative to separation kernels. Assured Information Systems' SecureView product, based on a Type I hypervisor separation solution, is used to enforce cross domain separation, allowing a reduction of the number of desktop computers needed in offices with secure and non-secure networks [11].

Separation and Safety

The ARINC 651 standard describes the Integrated Modular Avionics (IMA) approach to consolidate federated software functions onto a single box to, among other things, reduce cost and improve safety in avionic products [2]. The standard stresses the use of partitioning to "limit fault propagation… and therefore reduce the probability of system failure." ARINC 651 assigns this responsibility to the operating system, requiring it to maintain "robust partitioning between software modules for even the most critical functions." The ARINC 653 standard further defines a standardized partitioning operating system that attempts to provide fault containment and decoupling through spatial separation based on memory spaces, and temporal separation based on a time sliced scheduling algorithm [12]. Rushby defines his Alternative Gold Standard for Partitioning as "The behavior and performance of software in one partition must be unaffected by the software in

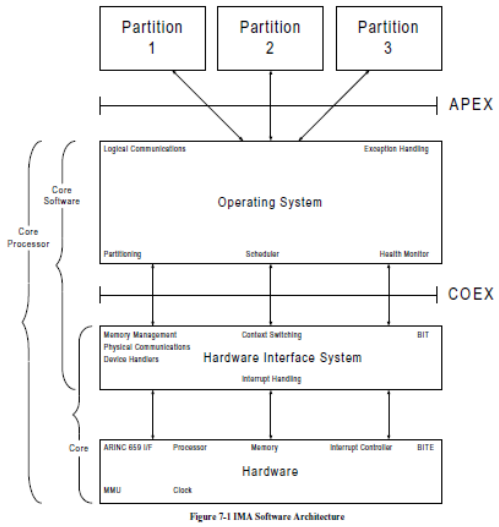

Figure 2: ARINC 651 IMA SW Arch [2]

other partitions." [13]

SOFTWARE SEPARATION SOLUTIONS

Separation can be provided by hardware, software, or both. In this paper, we focus on software solutions, using at most standard hardware features, broadly divided into three categories: through use of general purpose operating system (GPOS) features, through use of specialized operating systems designed to provide separation, and through use of virtualization services capable of running operating systems in their own virtual machines.

GPOS Features

The simplest approach to achieving some degree of separation is to use a modern general purpose operating system, such as Microsoft Windows or Linux, to run software functions as processes in their own protected memory spaces. In this case, the underlying computer hardware must provide a memory paging or memory protection system that can trap any attempts of a process to access memory outside of its assigned memory space. Some operating systems can provide an additional degree of spatial separation by running processes on their own processor cores using CPU affinity. With proper software development techniques, software functions running as separate processes can be decoupled through use of inter-process interfaces exercising services that the GPOS provides.

There are a number of issues with this approach. First, access to I/O devices and drivers is not typically controlled or regulated, so faults or exploits of the commonly accessible I/O functions and software stacks could easily propagate to other software functions. Second, the typical GPOS is not designed with fault isolation and decoupling in mind, resulting in data and control coupling between processes through the use of shared data structures in the OS layer itself. Third, while a GPOS may be capable of providing spatial separation by running software functions in their own memory spaces, they typically do not have means to provide temporal isolation to prevent multiple software functions from executing on different cores at the same time, which could cause execution jitter due to cache or system bus contention. Fourth, the decoupling of processes relies on adherence to good software development techniques, which can be difficult to ascertain after-the-fact with previously developed software components, especially if source code is not available. Also, on most general purpose operating systems it is trivial for software developers to create control or data couples between software functions in their pursuit of an optimal solution [14], and a GPOS does not typically safeguard against this by default. Some operating systems have security features, such as seLinux's mandatory access controls, that when enabled can protect against this [15]. Fifth, using a single OS to satisfy all of the various requirements could require compromises of capabilities, features, or performance, resulting in a non-optimal system. A common place where this compromise happens is when designers are forced to make tradeoffs between the need for a rich, graphical user interface (GUI) OS like Linux versus the need for a real-time operating system like FreeRTOS or VxWorks. Sixth, when consolidating multiple existing SWFs it may be discovered that some of those functions were developed with dependencies on a different OS than the one the SWFs are being consolidated to, requiring those SWFs to be ported to use that new operating system. Finally, operating systems, especially feature rich general purpose operating systems like Linux, have millions of lines of code, exposing a large exploit attack surface and defect cross-section.

Some operating systems like Linux also provide operating-system-level virtualization, also known as containers. Examples include Docker and chroot jail. OS-level virtualization uses permissions or name spaces to provide multiple isolated user-space instances [16]. This approach can offer better isolation of

drivers (compared to using memory spaces alone), along with quota or bandwidth limiting of I/O, CPU, and memory. This separation method also enforces better decoupling by preventing software developers from dynamically creating data or control couples. Additionally, OS-level virtualization does not need specific hardware virtualization extensions as do forms of virtualization described later in this section. OS-level-virtualization has been enjoying significant growth in the cloud-server market in recent years [17].

However, containers solutions still rely on a single operating system, potentially requiring porting efforts and/or compromises in OS selection, with the concomitant problems of large attack surface and large defect cross-section. Unintentional coupling resulting from the use of shared OS data structures could still occur as well. Finally, OS-level virtualization does not provide temporal separation between the SWFs.

Separation Kernels/Partitioning Operating Systems

As previously mentioned, the idea of a separation kernel was first proposed by Rushby to address issues discovered during the development of complex security kernels [6]. The task of the separation kernel is to provide an isolated execution environment for each component and to handle the communications between them. This first definition of separation kernel was strong on spatial separation, but only briefly touched on the need for temporal separation. Temporal isolation begins to be addressed in ARINC 651, and the ARINC 653 specification defines a specific algorithm for partition scheduling that ensures temporal isolation.

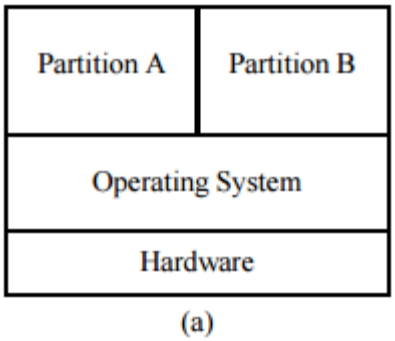

Figure 3: Separation Kernel (OS) [13]

Separation kernels and partitioning operating systems adhering to specifications like ARINC 653, such as Wind River VxWorks 653 – a specialized variant of the popular VxWorks RTOS – [18] and Green Hills Integrity-178 [19], provide spatial and temporal partitioning as core functions. These types of operating systems are also typically developed with design assurances showing how coupling is avoided and/or managed in the operating system layer in order to meet stringent certification standards. Design assurances also help reduce number of exploits and defects. Furthermore, I/O access is more tightly controlled than by a typical OS. As each software function/application is run in its own memory-bounded partition, decoupling is more strongly enforced as dependencies have to be explicitly created between software units.

However, separation kernels and partitioning OS solutions still rely on a single operating system, potentially requiring porting efforts and/or compromises in OS selection. Another concern with this class of separation solution is their specialized nature, with restrictions and additional overhead necessary for supporting the partitioning, coupled with higher cost, necessary for maintaining evidence of design assurances, make it unlikely that these solutions will see wide-spread outside their specific problem domains.

Virtualization

The idea of abstracting hardware and virtualizing it for software is not new, it has been in the literature since the 1960's. In 1974, Popek and Goldberg formalized the definition of the virtual machine monitor (VMM), also known as a hypervisor, as a piece of software with the following three essential characteristics:

1) The VMM provides an environment for programs which is essentially identical with the original machine;
2) Programs run in this environment show at worst only minor decreases in speed;
3) The VMM is in complete control of system sources. [21]

The virtual machine is the environment created by the VMM. In the same paper, Popek and Goldberg state that "isolation, in the sense of protection of the virtual machine environment, is meant to be implied as a result of the third characteristic defined above." Rushby proposes the idea of using a VMM, instead of an OS, as the basis for separation and partitioning kernels [6] [13]. Like the other solutions, the virtual machines that the hypervisor creates and manages each run within their own virtual memory spaces, and like with partitioning kernels, the hypervisor can use scheduling algorithms to ensure that VMs run at times separate from other VMs.

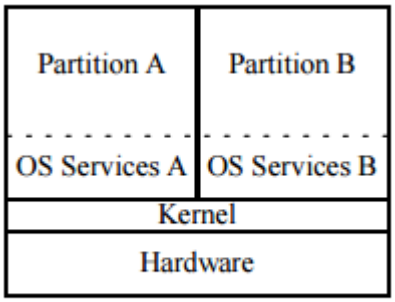

Figure 4: VMM Separation [13]

In his 1973 thesis, Goldberg [22] makes a distinction between VMMs that run directly on the bare metal and VMMs that run as an application on top of an OS. He classified these as Type I and Type II respectively. Virtualization services provided by a Type II hypervisor are useful for running an OS different from the one running on the desktop, such as Linux running in a VM on a Windows desktop. However, VMs created by Type II hypervisors have poor performance because of the reliance on the host OS to provide access to the hardware. Since Type II hypervisors are dependent on the singular host OS, the solution suffers from many of the same problems as OS-level virtualization described previously.

Virtualization services provided by Type I hypervisors, however, can provide partitioning below the OS layer, allowing a different operating system to be used in each virtual machine. This allows easier consolidation of legacy software functions, enabling greater re-use of source code, designs, requirements, tests, and other software life cycle artifacts. Because Type I hypervisors are in direct control of the hardware, there is no dependency on a host OS, and Type I hypervisors do not suffer the same performance penalties as do Type II hypervisors. This class of VMMs are also smaller than the typical general purpose OS because they follow microkernel design principles, focusing solely on key functionality such as memory management, CPU scheduling, and interrupt dispatching. The smaller footprint makes them easier to certify and presents a smaller attack surface and defect cross-section.

An additional benefit of using virtualization is that it allows the number of physical processor cores to be abstracted, allowing for many possible configurations of VM-to-core allocations. This can prove useful when migrating from a single core to multicore hardware target. In this scenario, the VM the SWF is running in could be configured with a single virtual CPU (vCPU), which could initially be mapped to a single physical core (pCPU). Then later, when newer hardware is available or the behavior on the single physical core has been prototyped satisfactorily, the VM can be reconfigured to map that single vCPU to multiple physical CPUs (pCPUs), allowing the SWF to take advantage of additional processing resources on the new target. A converse example is where a SWF designed to take advantage of parallelism of multiple cores from a previous hardware generation is migrated to a newer target platform with much faster CPUs. In this case, the SWF's VM could be configured to run with multiple vCPUs, to support the multi-threaded algorithms, but which are mapped to a single high speed pCPU, leaving the other pCPUs on the SoC to be dedicated to other SWFs.

A real-world example of how these features could be used is a system configuration based on CAST-32a, an aviation guidance paper that recommends the system should only allow one partition to run at a time to prevent interference due to concurrent access to shared hardware resources, such as pCPUs, cache, and system busses [23]. In a CAST-32a compliant configuration, each partition would be deployed to its own VM, each VM would be configured with access to the shared resources it needs, and the system would be

configured such that the hypervisor uses an ARINC 653 scheduler to ensure each VM runs in a different time slice.

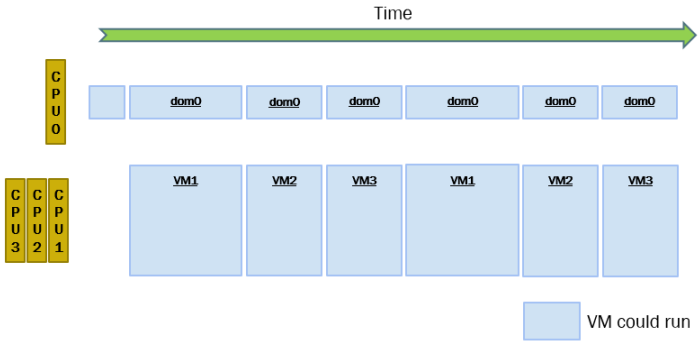

Figure 5: CAST-32a Scheduler Configuration

Another example configuration would be to support a security-centric solution with requirements that each SWF be restricted to run on its own assigned subset of available cores. Doing so would limit the potential of a compromised or malicious VM to violate information security with side channel exploits that glean information based on the underlying processor core's hardware properties. In this configuration, each SWF is deployed on its own VM, and each VM is configured with sole, exclusive access to one or more physical CPU cores.

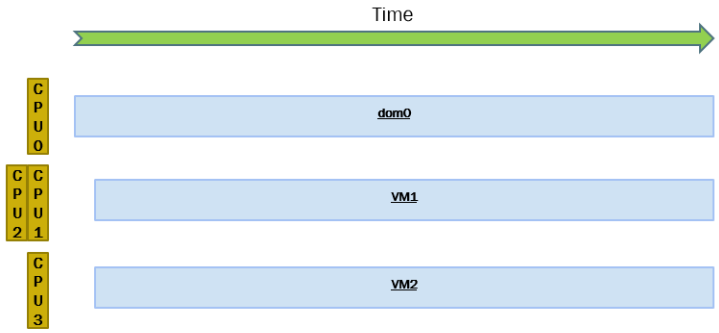

Figure 6: Security-Centric Scheduler Configuration

Xen for Embedded Systems

Xen is an open source Type I hypervisor that supports popular architectures such as x86_64, ARMv7-A, and ARMv8-A, and provides separation on many embedded products today [24]. Started in 1999 as an academic project, Xen has since grown to wide spread acceptance in the cloud server market. It is supported by an active community of professional and academic software developers with a common interest in adding new features and correcting defects. Many large companies, such as Amazon, Citrix, AMD, ARM, Intel, and Oracle, are members of the Xen Project Advisory board [25], and many other notable companies like IBM [26] and Google make use of Xen to power their services. This large user base on a wide variety systems improves reliability and security because the broad exposure quickly reveals any underlying problems that can be rapidly corrected by the community. This also allows the identification, and rectification, of security issues, realizing the National Institute of Standards and Technology (NIST) information security principle of

Open Design, which states that "System security should not depend on the secrecy of the implementation or its components." [27]

Below is a diagram of the Xen Project architecture. The Xen Project hypervisor runs directly on the hardware and is responsible for handling CPU, memory, and interrupts. It is the first software component to run after the bootloader(s). The hypervisor creates a privileged virtual machine, called *dom0*, which runs the control stack for managing VM configuration, creation, and destruction. In the Xen architecture, driver support for system devices is provided by software running in the VMs. Device virtualization is achieved by software in privileged VMs, like dom0, providing back-end drivers which control the underlying physical devices and which provide an interface for front-end drivers running in other VMs. To the OS in the other VMs, the front-end driver is controlling a physical device. Instead, the front-end driver actually passes the request to the back-end driver in the privileged VM, which arbitrates between multiple requests and performs the desired operations on the actual hardware.

Key features of Xen include:

- Small footprint and interface (around 1MB in size).
- Operating system agnostic: Xen supports a wide variety of proprietary and open source guest operating systems. The OS used for dom0 is a bit more restricted, with most installations using Linux. However, other operating systems can be used instead.
- Driver Isolation: In the Xen Project hypervisor, device drivers are run inside of a virtual machine. If the driver crashes, or is compromised, the VM containing the driver can be rebooted and the driver restarted without affecting the rest of the system. By default, dom0 provides all of the back-end drivers, but this functionality can be distributed to other VMs to isolate and protect dom0's control stack function.

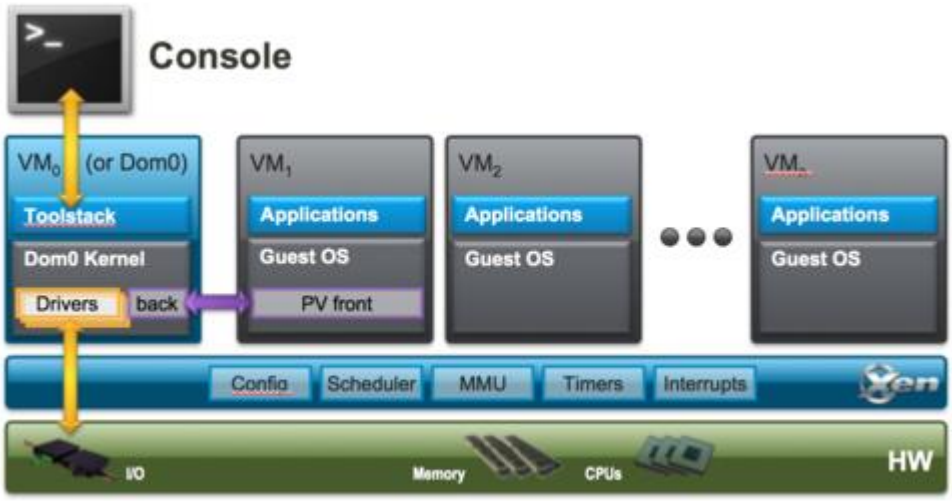

Figure 7: Xen Architecture [28]

As a Type I hypervisor, Xen offers the previously described benefits as a separation solution for embedded systems. Additionally, Xen is open source, providing the community a quick way to try out concepts and share results, as can be seen by its use in many research projects and papers. Because Xen is supported on the x86_64 as well as on the ARMv7-A and ARMv8-A architectures, it is possible to develop on personal

computers while having an obvious path to deployment on embedded targets. Xen also has flexible I/O allocation strategies, where an I/O device can either be virtualized through front-end/back-end drivers, or it can be *passed through*, meaning exclusively allocated, to a single VM. Finally, because of public and commercial engagement, it is safe to assume that Xen will continue to enjoy community support and remain available as open source for the foreseeable future. This combination of benefits makes Xen one of the best choices for a separation solution in embedded systems.

PARAVIRTUALIZING OPERATING SYSTEMS

This section will focus on aspects of porting operating systems to run in virtual machines created by the Xen hypervisor. There are common reasons that might pertain to your situation for paravirtualizing a specific operating system despite the existence of "Xen-ready" operating systems, some of which are explored in the following subsection. Before paravirtualizing an OS, its properties should be evaluated in the context of the virtual environment in order to make informed choices during the process For example, if a real-time operating system (RTOS) is under assessment, then it is important to understand the responsiveness and jitter requirements in order to make intelligent tradeoffs while modifying the OS. Also, the implications of the OS's licensing model can affect the decision whether to paravirtualize an operating system, since it results in a derivative work subject to the terms of the license for the original OS.

Reasons for Paravirtualizing an Operating System

Operating systems typically require some modification to run in the virtual machines that Xen creates on ARM platforms. While it is possible to run unmodified operating systems on Xen's Hardware Virtualization Machine (HVM) type virtual machines on x86 platforms, doing so raises concern about the safety and security of the resulting system. Aside from suitability concerns of using x86 microprocessors in power restricted embedded products, this family of processors typically have features, such as bursting and hyper-threading, which are inherently non-deterministic, making real-time guarantees extremely difficult to prove. Furthermore, it is also often difficult to get the necessary hardware design documentation from the manufacturer to perform the required analyses for system safety and security. Additionally, the code size of Xen is six times larger for x86 than for ARM [29], exposing a larger attack surface and defect cross-section. There is some evidence validating this concern found by comparing the number of security warnings from the Xen Project specifically for x86 versus the warnings that are for all Xen variants or specifically for ARM [30]. Finally, running unmodified operating systems on x86 is less efficient than running a modified version on ARM due to increased exception handling done by the hypervisor for HVM virtual machines.

To date, several operating systems have already been paravirtualized to run in a virtual machine created by Xen/ARM, such as Linux, FreeRTOS, µC-OS III, and VxWorks 7.0. However, even with the availability of already paravirtualized operating systems and the availability of Xen-specific operating systems, such as MiniOS and MirageOS, there are several reasons that would drive a development team to undertake the effort of paravirtualizing a new operating system.

One common reason for paravirtualizing an OS is when using an existing SWF which is only available in a binary form and the source code not available. This could be the situation with legacy applications, but it could also be the case on collaborative projects where the participants involved do not wish to share the source code of their SWFs. Another compelling reason is when a SWF has a dependency on a specific OS. This could be a direct dependency, where the SWF uses a capability or interface uniquely provided by the OS, or an indirect dependency, where the SWF depends on another software component, such as 3rd party library, which itself has a direct dependency on the OS. A third reason is when it would be more economical

to paravirtualize a single OS than to port multiple existing SWFs to an OS that can already run in a Xen VM. This calculation should include savings from those development and certification artifacts that can be re-used when the SWF is left untouched, which may include requirements, design, source code, unit tests procedures, unit test results, and, for some certification standards, work history. A fourth reason would be to leverage a software development team's familiarity and proficiency with an OS and corresponding ecosystem of tools and utilities when developing many new SWFs; it may be more cost-effective to paravirtualize an OS to allow the development team to exercise their existing expertise.

Operating System Properties Relevant to Paravirtualization

Decky classifies operating systems based on various characteristic properties of relevancy to the problem space [31], which is the approach taken to discuss paravirtualization impacts. Three OS properties found to be most relevant when considering virtualization are: OS purpose, multiprocessing support, and user interfaces support. Each is discussed in further detail below.

In addition to operating systems, application frameworks can run in virtual machines created by Xen. In our terminology, an *application framework* is a software component short of a full operating system, such as Xilinx's collection of standalone libraries for bare metal applications, which provide some subset of the OS services and are useful to consider in the same context as "operating systems". A library providing routines to manage low level processor features and perform simple scheduling, such as with a cyclic executive, would fit in this description.

Property of OS Purpose

The purpose of the OS can impact our paravirtualization approach. Below, we discuss general purpose, real-time, distributed, and separation purposes.

Many operating systems, such as Microsoft Windows and Linux, are designed for general purpose use, that is, to provide the typical and expected operating system services of abstracting the hardware and making it easy to run multiple applications. These kinds of operating systems are commonly run in Xen VMs, as evident by the number of instances of general purpose operating systems hosted by cloud-based servers running Xen. One common abstraction that operating systems provide is to effectively time share access to computing resources using some scheduling algorithm. This basic purpose is common between practically every microkernel, monolithic kernel, and all but the most basic exokernel and application framework. This is also a service provided by Xen, which has a variety of different scheduling algorithms and CPU affinity controls. In fact, Xen's time sharing management can surpass that of the operating system under consideration for paravirtualization. If no other services are needed from the OS other than process scheduling, then it may be more economical to split apart the tasks into their own VMs and let Xen schedule them directly instead of paravirtualizing the OS.

Some operating systems, such as VxWorks and FreeRTOS, are designed for real-time operations, otherwise known as real-time operating systems (RTOS). *Real-time* means that events must be handled within a specific amount of time. To ensure that timing deadlines are met, an RTOS is designed for low interrupt latency and high determinism. Balancing these two aspects is a difficult task since improving one often comes at the cost of the other. In IMA safety systems, typically all interrupts except for the one for periodic timer are disabled for reasons summarized by VanderLeest [32]. Such approaches negatively impact interrupt latency but ensure determinism. The impact to these real-time constraints must be understood when considering a virtualization solution. For example, when running on an ARM using the GIC-500v2, such as the Zynq® Ultrascale+™ MPSoC, the Xen hypervisor intercepts all incoming interrupts and, after some

processing, generates virtual interrupts to the appropriate VM. This adds interrupt latency for the OS running in a VM as compared to that OS running directly on hardware. Similarly jitter, or variation in execution time, can be induced by other VMs running simultaneously on other cores due to contention for cache and other shared resources. It is for this reason that the FAA recommends in CAST-32a that only one partition be allowed to run at a time (but that one partition can use multiple cores) [23]. In general, a SWF's timing and jitter requirements need to be understood in order to inform tradeoff decisions that will need to be made while paravirtualizing the OS. Paravirtualization could require the following tradeoffs:

- determinism vs efficiency: for example, trade-off between a non-work-conserving scheduling algorithm, like ARINC-653, that helps ensure determinism with a work-conserving scheduling that optimizes core utilization;
- determinism vs performance: for example, trade-off between avoiding or flushing caches to reduce jitter or to allow full cache use to improve performance;
- determinism vs response latency: for example, trade-off between polling for I/O to avoid timing variation caused by interrupt handling, or using interrupts to minimize interrupt latency.

Distributed operating systems, such as LOCUS and MICROS, are another specialization of purpose. In a specialized OS of this type, OS services are separated and allocated to a number of processing nodes. While distributed operating systems are not commonly used in embedded systems, virtualization using Xen would provide an ideal platform for prototyping such a system architecture on a single chip, as processing nodes can be assigned to VMs and created or destroyed as needed. Connectivity through virtual network interfaces is also well support on Xen.

A final OS purpose to consider is that of providing separation and partitioning, such as provided by VxWorks 653 and Integrity-178. Although the separation features provided by these operating systems would be redundant to those provided by Xen, it may be beneficial to run such an OS in a VM in order to re-use an existing SWF, in which case the OS would need to be paravirtualized. However, if it is the case that the existing SWF is just dependent on a specific interface provided by the separation/partitioning OS, then another option to consider is to add the required interface to an already paravirtualized OS [9]. When the separation/partitioning OS provides an unrelated capability that the SWF is dependent on, then paravirtualization is likely the most effective course of action. In this case, it is likely the OS will attempt to exercise control over the system's resource that provide partitioning or other security/safety related feature. If this responsibility is already handled, or can be handled, by some combination of Xen, dom0, or other VMs already implemented or envisioned for the system, then the functionality should be removed from the separation/partitioning OS as part of the paravirtualization process. If not, then the choices are limited to either passing those resources through to the OS's VM, or to virtualize the resources. In the first case, only minimal paravirtualization changes would be needed to control the resource, but software in other VMs would be prohibited from accessing them. The second case requires changes to the hypervisor and/or OS providing back-end drivers in addition to creation of front-end drivers for the OS being paravirtualized.

Property of Multiprocessing Support

Another defining property of operating systems is whether or not they support multicore processing. Application frameworks, due to their simplicity, typically do not provide multicore processing. For the same reason, some exokernels and microkernels may not support multicore processing either. Some monolithic kernels, particularly older ones, only support single core processing, but most modern ones like Linux do. If the OS to be paravirtualized only supports single core processing, then it is pointless to configure its VM

with multiple virtual CPUs. However, a single vCPU could be mapped to multiple physical CPUs, which would be useful if the pCPUs were being time-shared between multiple VMs. Otherwise, the most efficient use of resources would be to configure the single core OS VM to use one vCPU that maps to one pCPU. If the OS supports multicore processing, then assigning multiple vCPUs to the VM in which it will run will be beneficial as the OS will treat each vCPU as a real core with respect to thread scheduling. The VM configuration determines the number physical cores that VM actually ends up using, examples of which were previously provided. There is a small effort required to paravirtualizing multicore operating systems for Xen/ARM.

Property of User Interface Support

Another defining property of operating systems are the user interfaces (UI) which it supports. For embedded systems, the UI options typically consist of none, command line interfaces (CLI) via serial or network, and/or one or more graphical user interfaces (GUI). At best an application frameworks may support a CLI, or but sometimes might support no user interfaces at all. Microkernels and exokernels will typically support a CLI and may also have in their ecosystem an external component which provides a GUI. Monolithic kernels typically support both a CLI and a GUI. Xen has good support for virtual serial and network, but virtualizing and sharing graphics acceleration hardware, such as a Graphics Processing Unit (GPU), is a non-trivial task. One approach may be to pass the GPU to a single VM to support that OS's GUI. The i.MX8 from NXP supports a variant of this approach, where it has a GPU that can be partitioned by the firmware into two independent GPUs that can each be passed through to different VMs.

Licensing

Another aspect to evaluate when considering an OS for paravirtualization is its licensing terms. License to use the binary form does not imply permission to modify the source, or even access to the source code, which is necessary when paravirtualizing the OS. The two broadest categories of licensing are proprietary and open source, and each come with their own set of concerns.

With proprietary licenses, one has to come to an agreement with the owner of the copyright for permission to use and modify the software. While a vendor is usually more than willing to sell permission to use the software, they usually place restrictions on modifications. As an additional complication, any changes made to paravirtualize the OS to run on Xen would be outside the vendor's development line, and could potentially end up unsupported in future versions of the OS. It is usually in all parties' interest to come to an agreement that any modifications be added back to the main line distribution for future support. An alternative approach would be to pay the vendor to paravirtualize the OS. Unfortunately, either approach can run into difficulties if the vendor of the OS also offers a hypervisor solution, as they might see it against their business interest to make their OS product run on a competing solution. However these problems are not insurmountable, as evidenced by the WindRiver/DornerWorks collaboration to port VxWorks 7.0 to Xen/ARM and apply the changes to future VxWorks product lines [33].

Open source licenses come primarily in two groups, *copyleft* and *copycenter* [34]. Both grant permission to use and modify the software without compensation, but copyleft licenses includes a clause requiring that the source code *and derivative works* be made available to anyone receiving the binary directly or indirectly by owning a product on which the binary runs. Since a paravirtualized OS would be a derivative work, to comply with the license agreement it would also need to be made copyleft, requiring the changes made to the source code be provided to anyone receiving the binary. Copycenter does not have this proliferation clause, allowing free use of the software, resulting binaries, and any derivative works, without the requirement to

make the source code publicly available but with a minimum requirement of maintaining the original copyright notice. GNU Public License version 2 (GPLv2), which Linux and Xen use for most of their components, is an example of copyleft, while the FreeBSD clause is an example of copycenter.

Another caution is that even though open source software can be used and modified without compensating the owner of the copyright, it does not mean it is entirely free. Open source solutions typically do not have support options guaranteed to help resolve issues when using the software. However, this can be overcome if 3rd parties exist who offer professional support services, such as Red Hat for Linux [35] and DornerWorks for embedded Xen [36].

PARARVIRTUALIZATION LESSONS LEARNED

DornerWorks has paravirtualized a number of different operating systems and application frameworks for Xen on the ARM Corex-A53, an ARMv8-A microarchitecture, as implemented on the Zynq® Ultrascale+™ MPSoC. These include Xilinx's standalone libraries, µC/OS-III, FreeRTOS, and VxWorks 7.0, as well as working extensively with previously paravirtualized operating systems and frameworks like Linux, MiniOS, and MirageOS. From these experiences, we have derived a scheme for classifying the changes we found that were required for the operating system to function correctly in the virtualized environment. Generally speaking, these changes can be classified as follows: changes due to the system architecture, changes required to operate in the environment created by a specific hypervisor, changes required to comply with the processor architecture theory of operation, changes required by the SoC being targeted, and changes required by the specific hardware target, development board or system on module (SOM) card. Note that some changes may fall into multiple classes, however we do not feel that this would hinder the primary goal of identifying and making all the necessary changes to port the operating system to the virtual environment.

Table 2: OS Paravirtualization Change Classes

Class of Change	Changes needed because …
System Allocation	OS not in control of all resources.
Hypervisor	Hypervisor specific implementation/virtualization details.
Processor Architecture	Restrictions/ programming guidelines for processor.
SoC	SoC specific implementation details.
Hardware Target	Board/SoM specific implementation details.

System Allocation Changes

Changes in this class are required due to the fact that an operating system is often designed with the implicit assumption that it would be in full control of the target platform resources. When evaluating an OS to paravirtualize, its interactions with shared resources, power states, security states, and platform level controls must be understood in the context of the virtualized system. The hypervisor may virtualize those resources in a way that is transparent to the OS; for example, Xen virtualizes the generic ARM hardware time and CPU state. For those resources that are not virtualized, an architectural plan is needed to allocate to the VMs those resource which it will manage on behalf of the entire system. If a resource an OS needs to access has been exclusively allocated to its VM, then minimal or no effort is needed to paravirtualize the OS with regards to using that resource; i.e. drivers for that device can remain unmodified. If the resource an OS needs to access is allocated to a different VM, then access to the resource needs to be removed from that OS. If multiple operating systems in different VMs must access the same resource, then that resource must be virtualized. However, doing so will require modifications to software components in other VMs or in Xen itself, and it is best to avoid this situation if possible.

Some examples include changes to…:

- Avoid use of shared resource
  - Clocks, CPU state, GPIO
- Avoid platform state changes
  - Power control and coordination
  - Security states

Hypervisor Changes

Even when a hypervisor virtualizes a resource, sometimes the behavior or interface to that resource is different from that of the physical resource. When that is the case, the OS to run in a VM needs to be modified to take those differences into account. For example, Xen masks interrupts in the Generic Interrupt Controller (GIC), and leaves them masked, which in the virtualized environment makes it seem like the GIC is automatically masking an interrupt where, based on behavior of the real GIC, the OS might expect the interrupt to remain unmasked. Additionally, each hypervisor defines its own set of interfaces that paravirtualized operating systems can use to gain access to the VMM's services. For example, the Xen/ARM provides an API based on the HVC opcode, which is used to trap to the Xen context from the OS; a device-tree with configuration information that is populated in the VM's memory; and support for the ARM Generic Timer virtual timers. Finally, the VM that the hypervisor creates may have specific characteristics that the OS has to be adapted for, equivalent to the changes required when porting an OS to a hardware target with a different memory map. For example, the current version of Xen/ARM configures each virtual machine to have a specific starting *intermediate physical address* (which to software running in the VM is the physical address, but which is really a form of virtual memory). This may require relinking an OS's object files to run in that address space. Some operating systems like Linux are capable of running from any address space as long as it can configure the memory management unit (MMU) to do the appropriate virtual memory translations. On ARM architectures with hardware virtualization, this is supported via a two stage MMU. In the second stage, the intermediate physical address to physical address translation is configured and handled by the hypervisor. The MMU's first stage interface is virtualized to allow each operating system running in a VM to configure and handle its own virtual address to (intermediate) physical address translation.

Some Xen/ARM specific examples include changes to…:

- Use the Power State Coordination Interface (PSCI). Xen expects the OS in the VM to use the HVC opcode for PSCI calls instead of the SMC opcode. This is needed for controlling of processor states, which is necessary for multicore support.
- Use the ARM Generic Timer virtual timers. Although Xen virtualizes the Generic Timer physical timers, it is more efficient to use the virtual timers instead.
- Account for differences in Xen generated device-trees. Xen/ARM creates a device-tree for each VM with specific semantics, which in at least one case was not shared by the OS being run in the VM.
- Account for differences in virtualized GIC behavior; Xen implicitly masks interrupts, requiring the OS to explicitly unmask the interrupt each time.
- Account for any differences in the virtual bootloading behavior.
- Use the starting intermediate physical address base.
- Use the virtual GIC distributor interface base address.

Processor Architecture Changes

Changes in this class are those that are required to make the OS work in a virtual machine as per the theory of operation described for the processor family. For example, the ARMv8 expects that an operating system would run at Exception Level 1 (EL1) [37], which has greater privileges and access than that of applications, which are expected to run at EL0. Ideally, the OS being ported would adhere to these guidelines already, but on more than one occasion we have found that to not be the case.

Some ARMv8-A specific examples include changes to…:

- Avoid use of operations requiring higher privilege levels:
  - System calls should be made with the SVC, trap to EL1, opcode and not the HVC, trap to EL2, or SMC, trap to EL3, opcodes.
- Avoid accessing privileged resources:
  - OS should not access of EL2 or EL3 reserved registers.
  - OS should not manipulate MMU stage 2 translation tables.
    - OS can manipulate MMU stage 1 translation tables to set up virtual memory for itself and its applications.

SoC Changes

Changes in this class are made to reconcile specific properties of the SoC design with the OS being paravirtualized, typically in regards to peripherals outside the processor cluster. One possible source of change is the boot sequence, as SoCs typically provide some kind of primitive first stage boot loading functional that an OS may have been implemented to be dependent on, which is replaced by the VM boot sequence. Changes may also be needed to use different I/O device instances. For example, an OS that is designed or configured to use the first of four Ethernet controller may need an update to use one of the other three instances. SoCs can also introduce additional platform resources that the hypervisor does not virtualize, which needs to be considered in the overall system architectural context and dealt with appropriately.

For example, the Zynq® Ultrascale+™ MPSoC from Xilinx is a powerful embedded computing processor, implementing a quad core Cortex-A53 application processor, a dual core Cortex-R5 real-time processor, a dual core Mail-400 graphics processor, all coupled with Xilinx's 14nm UltraScale+ FPGA fabric [38]. The MPSoC has several system-level peripherals that are accessible via register banks that Xen does not currently virtualize. Chief among these are the registers for programming and resetting the FPGA. These registers controlling the FPGA fall under the category of a single platform resource that Xen does not currently virtualize. One recommended option is to pass access of the resource that controls the FPGA to a single, privileged VM like dom0. Even though the FPGA control registers are not virtualized, a device instantiated in the FPGA can be virtualized if an appropriate back-end driver is available, or it can passed through to a VM.

Some examples include changes to…:

- Account for differences between SoC and VM boot sequences.
- Use different I/O instance.
- Take advantage of IP provided in extra-processor components, like FPGA.

Hardware Target Changes

The specific target board or SoM layout may complicate how I/O devices are passed through to VMs. It is often the case that an I/O device, like an Ethernet or Universal Serial Bus (USB) controller, is dependent on a simpler I/O device, such as general purpose I/O controller used to toggle reset pins or a Serial Peripheral Interface (SPI) or Inter-IC (I2C) bus used to control a physical layer chip. The simplest solution is to pass access to the simpler I/O controller through to the VM for the OS to access as well. The problem occurs if a different VM has an I/O peripheral passed through to it that is also dependent on the same simpler I/O function. For example, this would be a situation if both Ethernet and USB relied on the same I2C bus to control their physical layer chips but were passed through to different VMs.

Projects with the freedom to design the target hardware should try to avoid this sort of dependency. When that is not an option, one possibility is to re-allocate resources so either the OS is allocated an I/O device with no dependencies on shared hardware, or the OS is allocated to use a virtualized device. If neither approach is viable, because no dependency-free I/O instances are available and no virtualization exists for the desired I/O, then another solution to consider would be passing control of the simpler I/O function to a privileged VM, like dom0, that would drive it appropriately. This approach works well for simple one-time setup and initialization, but more complex and dynamic behavior requires the I/O resource to be fully virtualized.

Examples, changes to…:

- Resolve passed through I/O dependencies on shared hardware
    - Use different physical I/O instance or use a virtualized I/O instance instead.

CONCLUSION

System architects need to consider how to maintain or establish fault containment and loose coupling between software functions they wish to consolidate onto a common SoC. Different software separation solutions exist that can help isolate or partition software functions from each other, the choice of which will have far-reaching impacts. A properly selected software separation solution can help reduce risk to cost, schedule, and performance objectives. Embedded virtualization using Xen, an open source Type I hypervisor, on an ARMv7 or ARMv8 platform is a promising option worth considering, but may require porting of an operating system to run correctly in the virtual machine created by the hypervisor. We have classified the kinds of changes that may be needed and provided specific examples. Even though the examples are specific to Xen running on the Zynq UltraScale+ MPSoC, the lessons learned that were shared can be applied to paravirtualization efforts for other hardware/hypervisor combinations.

REFERENCES

[1] J. P. Anderson, "Systems Architecture for Security and Protection," in Approaches to privacy and security in computer systems: proceedings of a conference held at the National Bureau of Standards, Washington D.C., 1974.

[2] ARINC 651-1, "Design Guidance for Integrated Modular Avionics," Aeronautical Radio, Inc., Annapolis, 1997.

[3] M. Portnoy, Virtualization Essentials, Hoboken: Sybex, 2012.

[4] VMWare, "Server Consolidation," 2017. [Online]. Available: https://www.vmware.com/solutions/consolidation.html. [Accessed 27 June 2017].

[5] P. Bieber, F. Boniol, M. Boyer, E. Noulard and C. Pagetti, "New Challegnes for Future Avionics," *AerospaceLab*, no. 4, pp. 1-9, 2012.

[6] J. Rushby, "Design and Verification of Secure Systems," in *ACM Symposium on OPerating System Principles*, Pacific Grove, 1981.

[7] C. Boettcher, R. DeLong, J. Rushby and W. Sifre, "The MILS Component Integration Approach to Securing Information Sharing," in *27th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, St. Paul, 2008.

[8] P. Parkinson, "Multicore MILS," in *9th IET International Conference on System Safety and Cyber Security*, Manchester, 2014.

[9] S. H. VanderLeest, "ARINC653 Hypervisor," in *29th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Salt Lake City, 2010.

[10] J. McDermott and L. Freitas, "A Formal Security Policy for Xenon," in *Proceedings of the 6th ACM workshop on Formal methods in security engineering*, Alexandria, 2008.

[11] AIS, "SecureView," [Online]. Available: https://www.ainfosec.com/innovative-products/secureview/detail/ . [Accessed 17 July 2017].

[12] ARINC 653P1-3, "Avionics Application Software Standard Interfaces Part 1 - Required Services", 2010.

[13] J. Rushby, "Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance," NASA Langley Research Center, Menlo Park, 1999.

[14] A. J. Lattanze, "Coupling," in *Architecting Software Intensive Systems: A Practitioners Guide*, Boca Raton, CRC Press, 2008, p. 58.

[15] NSA, "Security-Enhanced Linux," [Online]. Available: https://www.nsa.gov/what-we-do/research/selinux/. [Accessed 27 June 2017].

[16] S. Hogg, "Software Containers: Used More Frequently than Most Realize," Network World, 26 May 2015. [Online]. Available: http://www.networkworld.com/article/2226996/cisco-subnet/software-containers--used-more-frequently-than-most-realize.html. [Accessed 17 July 2017].

[17] J. McKendrick, "Container technology market to grow 40% a year, analysts predict," Service Oriented, 17 January 2017. [Online]. Available: http://www.zdnet.com/article/container-market-keeps-growing/. [Accessed 17 July 2017]

[18] Wind River Systems, Inc., "VxWorks 653 Product Overview," 2017. [Online]. Available: https://www.windriver.com/products/product-overviews/vxworks-653-product-overview/. [Accessed 27 June 2017].

[19] Green Hills Software, "INTEGRITY-178 EAL 6+ certified, safety-critical RTOS," [Online]. Available: https://www.ghs.com/products/safety_critical/integrity-do-178b.html. [Accessed 27 June 2017].

[20] E. Brown, "Embedded Linux Keeps Growing Amid IoT Disruption, Says Study," 20 March 2015. [Online]. Available: https://www.linux.com/news/embedded-linux-keeps-growing-amid-iot-disruption-says-study. [Accessed 27 June 2017].

[21] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Communicatons of the ACM*, vol. 17, no. 7, pp. 412-421, 1974.

[22] R. P. Goldberg, "Architectural Principles for Virtual Computer Systems," National Technical Information Service, Springfield, 1973.

[23] FAA, *CAST-32A, Multi-core Processors,* Washington D.C.: U.S. DOT, 2016.

[24] Xen Project, "Xen Project Software Overview," [Online]. Available: https://wiki.xen.org/wiki/Xen_Project_Software_Overview. [Accessed 27 June 2017].

[25] Xen Project, "Project Members," [Online]. Available: https://xenproject.org/directory/project-members.html. [Accessed 27 June 2017].

[26] IBM SoftLayer, "CloudLayer® Computing vs. Amazon EC2," [Online]. Available: http://cdn.softlayer.com/PS_EC2vsCL.pdf. [Accessed 27 June 2017].

[27] K. Scarfone, W. Jansen and M. Tracy, "Guide to General Security," National Institute of Standards and Technology, Gaithersburg, 2008.

[28] Xen Project, "File:Xen Arch Diagram.png," 20 April 2015. [Online]. Available: https://wiki.xen.org/wiki/File:Xen_Arch_Diagram.png. [Accessed 27 June 2017].

[29] Xen Project, "Xen ARM with Virtualization Extensions whitepaper," [Online]. Available: https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper. [Accessed 27 June 2017].

[30] Xenproject.org Security Team, "Advisories, publicly released or pre-released," [Online]. Available: https://xenbits.xen.org/xsa/. [Accessed 17 July 2017].

[31] M. Decky, "Application of Software Components in Operating System Design," Charles University, Prague, 2015.

[32] S. H. VanderLeest, "Taming Interrupts: Deterministic Asynchronicity in an ARINC 653 Environment," in *33rd IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Colorado Springs, 2014.

[33] Wind River Systems, Inc., "DornerWorks, Ltd.," 2017. [Online]. Available: https://www.windriver.com/alliances/newdirectory/company.html?id=23170. [Accessed 17 July 2017].

[34] World Heritage Encyclopedia, "Copycenter," [Online]. Available: http://worldlibrary.net/articles/eng/Copycenter. [Accessed 17 July 2017].

[35] Red Hat, Inc., "Customer Portal," [Online]. Available: https://access.redhat.com/support. [Accessed 27 June 2017].

[36] DornerWorks, Ltd., "Xen Zynq Distribution Support Services," [Online]. Available: http://dornerworks.com/xen/xilinxxen/xen-zynq-distribution-support. [Accessed 27 June 2017].

[37] ARM, "Fundamentals of ARMv8," in *ARM Cortex-A Series Programmer's Guide for ARMv8-A*, ARM, 2015, pp. 28-36.

[38] S. H. VanderLeest and D. White, "MPSOC Hypervisor: The Safe & Secure Future of Avionics," in *34th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Prague, 2015.