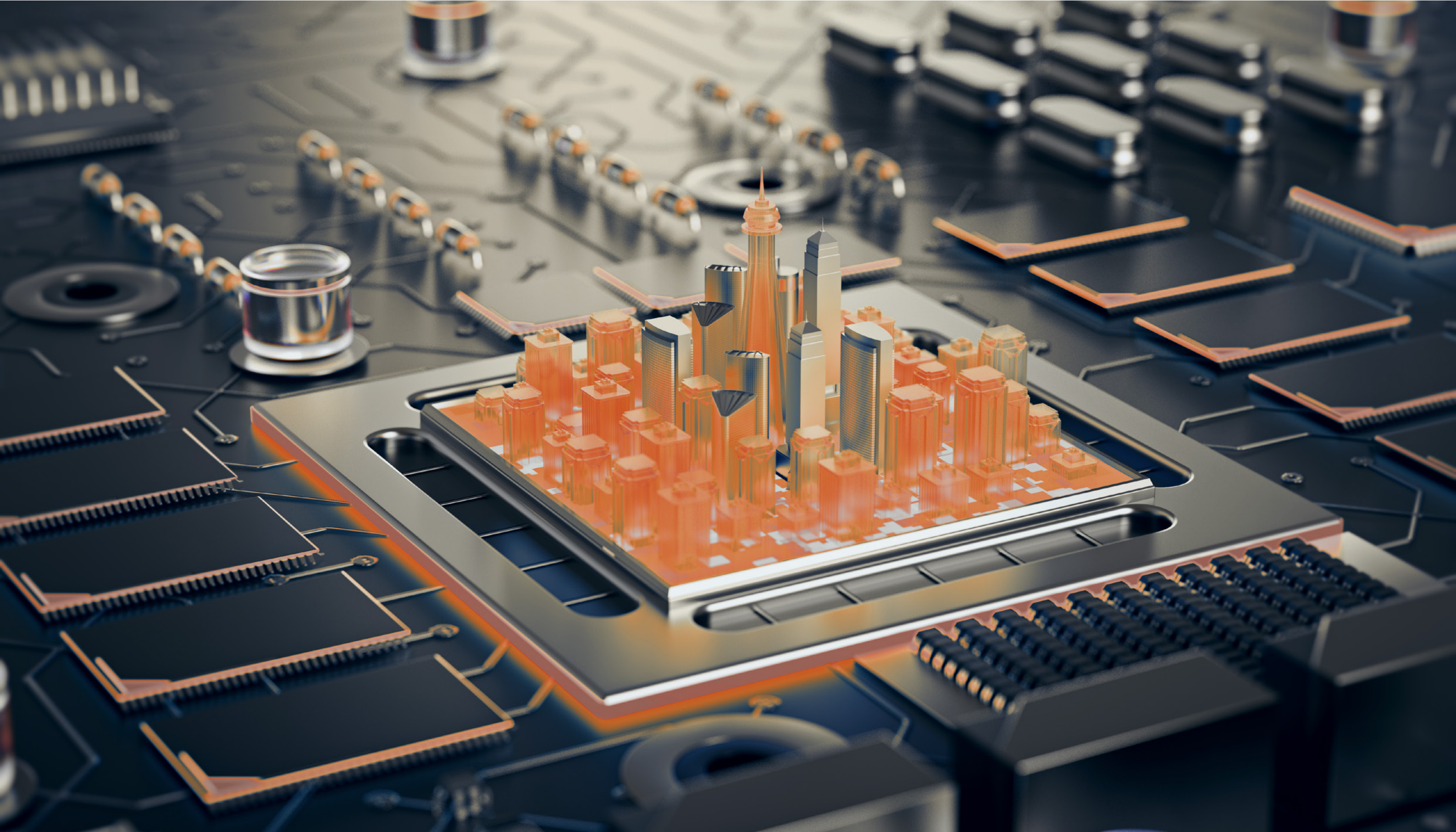
eSOL

Whitepaper



When and How to Use Virtualization in Embedded Systems Design (and How a Hypervisor Helps)

In application areas such as Industrial IoT and automotive, there is a strong need to combine more software-based functions in fewer hardware components. This is achieved through aggregation, putting all functions into the same device.

Here, virtualization is used for this software aggregation. For cloud server applications, containers are currently known as a relatively lightweight solution. However, when embedded systems have a safety and security aspect, one has to resort to other solutions.

Hypervisors, which have been used in the embedded area for years, offer a very safe and secure solution. In this white paper we discuss containers and the four different hypervisor types. Furthermore, we will present a new real-time hypervisor based on an innovative multikernel technology RTOS with POSIX interface and explain how to achieve better performance and freedom from interference (FFI) on multi/manycore hardware.

2022-12

Introduction	3
Virtualization Techniques	4
Hypervisor Choices	6
Type 1.5: eMCOS Hypervisor	7
Multikernel Architecture	8
Hypervisor Device Management	9
General advantages of eMCOS Hypervisor	10
eMCOS Hypervisor in Action	11
Conclusion	11

Introduction

Stability and robustness are almost always key requirements of embedded systems. However, as a design evolves, extra functionality is often added in a modular approach producing excessively distributed systems that impose increased cost and complexity.

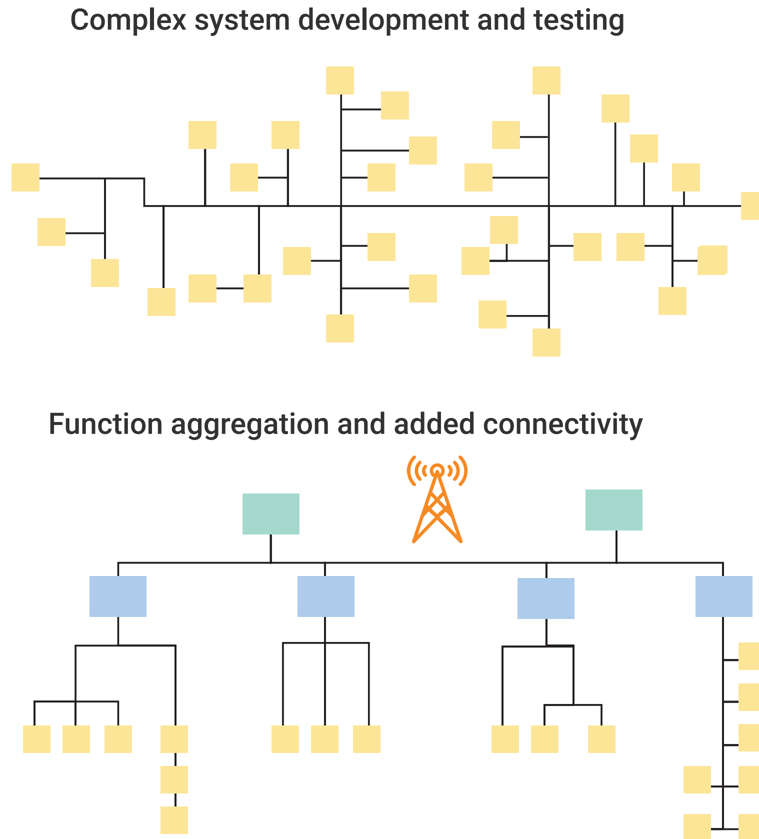


Figure 1. Evolution of complex embedded system architecture

After aggregation, consolidation describes introducing adaptations at the software level to combine functions on common hardware. This offers the opportunity to save the physical bill of materials (BOM) by reducing not only the number of processors but also passive components, connectors, wiring, PCBs and enclosures.

Virtualization is a powerful technique that can be used to consolidate functions on a single hardware platform. Recently, containers have also emerged in embedded systems as a popular and effective way to achieve consolidation. Another approach is to use a hypervisor. Each approach has its own strengths as well as some issues that must be considered.

The automotive industry provides a prime example of this tendency. As demands have grown for more and more user features and extra mandatory safety systems, the number of electronic control units (ECUs) distributed throughout the vehicle has grown enormously with over 100 ECUs present in some high-end models.

In this space and others, such as edge computing applications and general industrial electronics, there is a strong need to merge more functions into less hardware. This can be achieved through aggregation, by simply bringing the functions with minimal changes into the same box. The trend towards domain and, more recently, zonal architectures in automotive is an example of this approach.

Virtualization Techniques

Implementing virtualization can be complex, involving extensive development time, costs, and risk. As stated, robustness and stability are extremely important. If there is a safety aspect to the system's functionality, this can be difficult to manage in an embedded system. While it may be acceptable for the user interface, for example, to fail, it is critical that safety-related systems should offer continuous availability. However, when there are multiple safety-critical functions it is also important – and challenging – to assign a different and appropriate priority to each. Similarly, care must be taken to ensure adequate system security.

Category	Hypervisor	Containers
General	Hypervisor is a strong technique for virtualization: each environment runs in a full virtual machine (VM)	Container is a weaker virtualization technique: groups of applications share the same operating system services
Granularity	Groups of apps + guest-OS	One OS with groups of apps
MMU usage	Level 2 hypervisor, Level 1 guest OS	Level 1 host-OS
Performance	More expensive context switches	Just host-OS
Integration process	Incremental integration	All applications must be verified together
Software update	Affects one guest-OS	Affects the entire platform
Security breach	Affects one guest-OS	Affects the entire platform
Safety concern	Can decompose into subsystems	Must consider all applications

Among the techniques available to achieve virtualization, containers can offer a relatively lightweight solution in terms of software overhead, helping to keep performance. On the other hand, the platform can be vulnerable to failure if one of the hosted applications causes a malfunction in the kernel, since the very principle of containers is to have a single kernel host multiple containers. Also, security vulnerabilities are applicable to the entire platform.

In contrast, a hypervisor can offer a solution to avoid some of these potential hazards. One strong advantage is that the hypervisor relieves interdependencies between the various applications and guest OSes hosted on the platform. This can simplify the development of each guest system drastically. Then, various approaches are available and careful consideration of each is necessary to ensure an appropriate blend of performance, system robustness, and security.

In general, a hypervisor is a stronger isolation technique. Each guest OS, including its kernel, runs in a full virtual machine, and each guest kernel is responsible for coordinating its own applications inside its local world. This requires heavier context switches when two applications in two different guests communicate but provides a verifiably stronger protection. In contrast, a container runs multiple OSes over a single kernel. This means that switches between applications are much faster, but a failure in the kernel causes a full system malfunction.

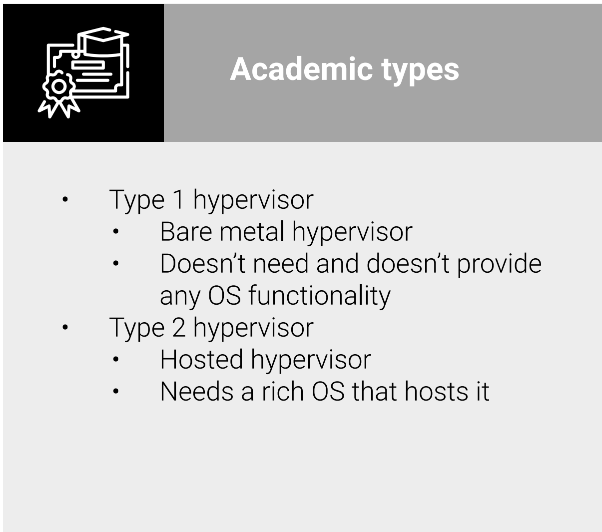
Another aspect is the possible gradual integration and upgrade path that the hypervisor proposes: because it allows running multiple full execution environments, upgrading a guest kernel will only impact that environment without modifying whatsoever the other OSes. On a container, upgrading a container requires the review of all the containers to ensure that the kernel behavior is unchanged.

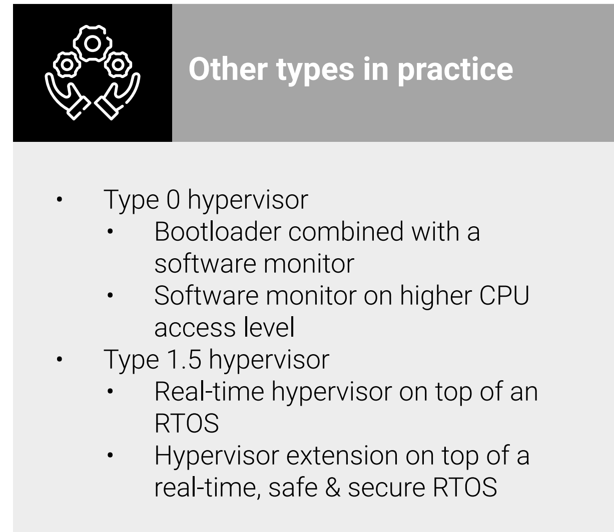
Handling of security and safety is also potentially more robust in the hypervisor context. With the hypervisor an application may be able to corrupt its guest OS. However, the fact that each guest is isolated helps limit the impact on security and safety, because it is also isolated from other guests. Another application running in another guest can continue to run unaffected on the same platform. A security breach in the guest kernel, for example, affects one guest OS running on the hypervisor, whereas the containerized approach leaves all users affected because of the single kernel design. As far as safety is concerned, the hypervisor can be decomposed into multiple subsystems whereas the safety implications of one container failure must be considered for all applications.

Finally, a hypervisor-based system is potentially easier to analyze because – unlike the containerized approach - it is not necessary to assess the impact of any error on all other aspects of the system. The “divide-and-conquer” approach makes the isolation of concerns more clear, at least from a CPU application scope perspective.

Hypervisor Choices

Conventionally, there are three categories of hypervisor. They are termed type 0, type 1, and type 2. All have their own strengths and drawbacks.





Type 0 hypervisor is relatively lightweight, similar to a bootloader with additional monitoring capability. Indeed, it contains a minimalistic bootloader that loads guests on different CPU cores. In addition, there is only a small monitor to detect any issues like incorrect accesses by guests. When the platform is running smoothly, the monitor does nothing. Hence the main role of a type 0 hypervisor is to start different operating systems with some level of isolation on different cores. This can be suitable when the system is relatively simple, there are enough cores to dedicate one or more to each function, and there is no device sharing to manage. Different guests are simply assigned to different cores in a static way.

A type 1 hypervisor is more sophisticated and performs the role of a real scheduler and protection environment. It can work with dynamic workloads but typically a fixed number of guests. At bootup, the hypervisor starts the first guest as the master guest. The master then starts the other guests. The hypervisor coordinates the execution of these environments and ensures they run correctly and in a real-time, deterministic manner, switching between processes according to a time-based trigger or depending on the relative priorities of the different guest operating systems. The type 1 hypervisor simplifies retaining real-time and deterministic properties. One drawback is that the master OS – being responsible for loading the guest OSes – is also often responsible for device sharing. Thus, a safe OS running as a guest on the platform is effectively dependent on a non-safe operating system. This may not ensure sufficient robustness, placing the entre platform at risk. If the master guest malfunctions, the entire hypervisor platform may malfunction.

The type 2 hypervisor is significantly different in operation. After starting the general-purpose operating system, POSIX applications start normally. If and when required, virtual machine applications can be started as an environment to run a guest OS. This ensures full flexibility to start one or many guests and run each independently through a virtual machine manager (VMM). Another advantage is that the VMM also manages some level of security from the host OS application context, which is less privileged than in a Type 1 environment. On the other hand, the definition of a Type 2 hypervisor defines that a general purpose is used as the host OS, like Linux or Windows, but not a real-time OS and thus it is not able to maintain real-time behavior or organize and schedule different processes in a fully deterministic way. This means that the general consolidated system remains unpredictable as a whole.

Type 1.5: eMCOS Hypervisor®

In contrast with each of these approaches, eMCOS Hypervisor combines a privileged RTOS and minimal hypervisor code. Based on eMCOS® POSIX, which is a multi-process RTOS, it is capable of starting multiple POSIX applications then running multiple guest OSes while maintaining the real-time properties of both the VMMs and POSIX applications as guests. In this respect, this approach is similar to that of a type 2 hypervisor. However, since POSIX applications run directly on the POSIX OS, there is less overhead than when running inside a guest, and the real-time scheduling keeps a proper deterministic execution environment. Consequently, eMCOS Hypervisor can be seen as a type 1.5: a real-time platform capable of running direct real-time applications, while at the same time giving the possibility to run additional OSes and their applications as guests.

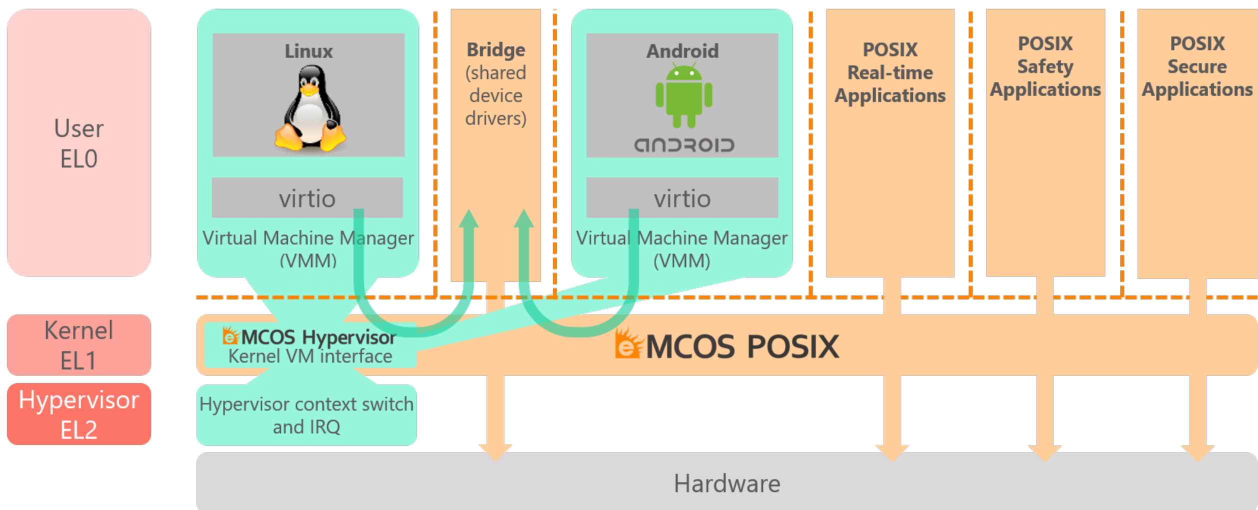


Figure 2. The eMCOS Hypervisor permits running multiple guest OSes on common hardware

It is possible to develop various different applications, including POSIX and open-source applications, to run on the eMCOS POSIX RTOS (figure 2). This can include safety applications and security applications, which may have real-time aspects. VMMs are responsible for running the guests and handling any faults. At the same time, virtual device drivers with access to physical hardware can run as processes, allowing guest OSes to share access through a suitable bridge application handling each a single device, for example for networking or console sharing. There is a direct API for native applications and standard hypervisor APIs for communications: guest OSes communicate through virtio interfaces. This enhances the flexibility of the platform, because Virtio is widely recognized throughout the industry.

Multikernel Architecture

An aspect that is particularly important is that every attempt has been made to minimize the code running in the hypervisor domain and in the kernel domain to ensure that most of the code is running with the minimum amount of privilege in the platform.

However, to take consolidation and aggregation properly into account it is also necessary to consider multicore behavior.

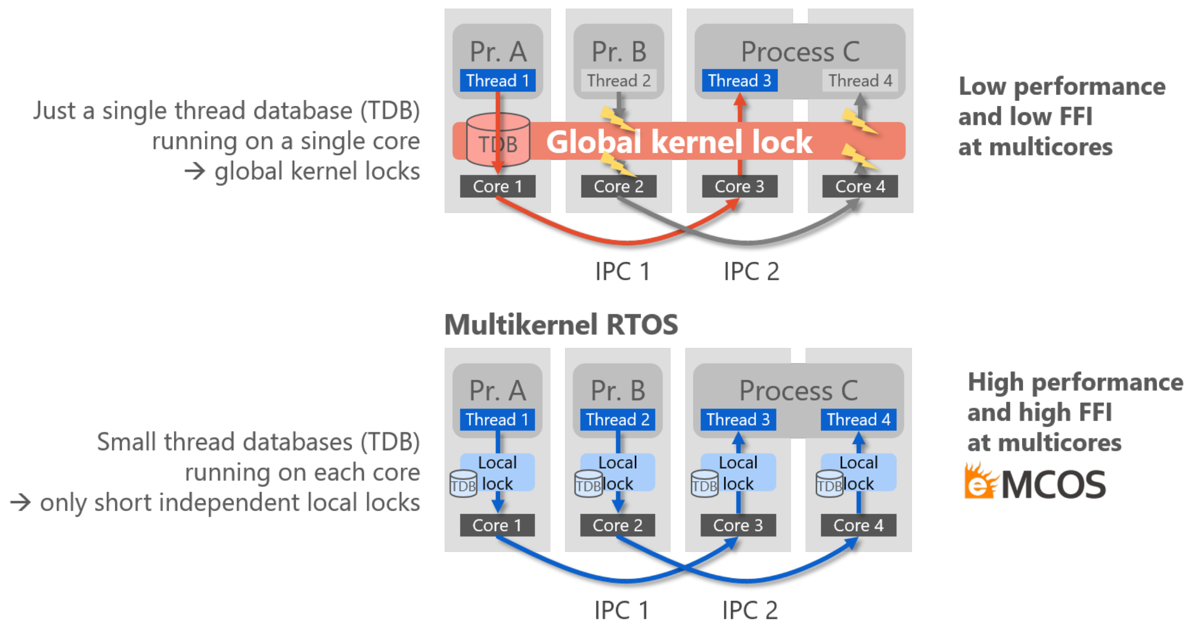


Figure 3. Multikernel versus SMP locking

In a type 1.5 (or type 2) hypervisor, a virtual CPU (vCPU) is managed by a thread of the host OS. The host OS schedules all the threads in the platform with their own priority, time quota, and memory allowance, regardless of whether said threads are managing a POSIX application running directly, or a guest vCPU. This can be done using a microkernel, which behaves in a way to keep the kernel side as simple as possible. Microkernels typically employ symmetrical multiprocessing (SMP) that uses a global locking system for managing access to threads (figure 3). This global locking allows only one set of interactions between cores to take place at a time, which effectively prevents real CPU parallelism. eMCOS hypervisor is based on the eMCOS POSIX RTOS that uses an innovative microkernel AND multikernel design, in which each core runs a scheduler that is largely autonomous and allows all cores to communicate together using asynchronous messages. This allows cores to schedule local threads freely using only minimum local locking thereby enabling multiple application and guest execution to proceed simultaneously and truly in parallel. This also enhances freedom from interference (FFI) in multicore systems, as cores will never depend on each other at the microkernel level for rescheduling.

In other words, the use of a multikernel as an operating system scheduler allows general better system parallelism than a SMP scheduler. As an extension, a multikernel-based hypervisor also does not introduce a global lock in the system and keeps the full parallelism of guest OSes.

Hypervisor Device Management

Embedded applications are critically dependent on correct device management to ensure proper use of the common hardware resources. Since sharing requires a bridge to coordinate accesses, with the related additional latency and occasional copies, it is useful to limit the amount of shared devices. There is also an impact on both safety and security when considering failures cases in devices, as a device on a shared physical bus may affect other device given to other guests.

From the beginning of the project, it is therefore important to consider how the various guests will use the hardware, including the on-chip and off-chip peripherals.

eMCOS Hypervisor device management

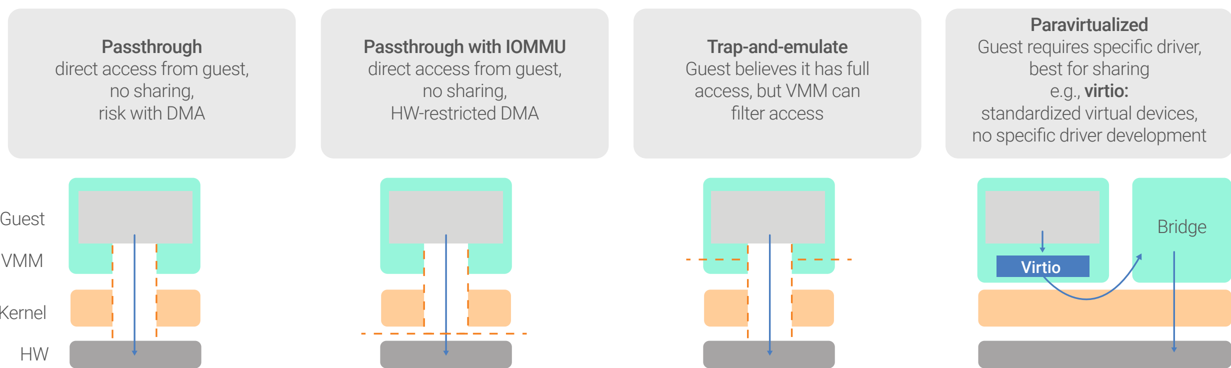


Figure 4. Device management techniques supported in eMCOS Hypervisor

A number of approaches are effective, as illustrated in figure 4:

Passthrough

In this case, the guest OS has direct access to the hardware and both the VMM and kernel agree that the guest can access the hardware. This is suitable when there is no need for device sharing. However, when hardware is capable of direct memory transfers e.g., a DMA HW, there is a risk that the hardware can corrupt the guest, but also the VMM or even the host OS.

Passthrough with IOMMU

When direct memory transfers are required e.g., by a DMA HW, passthrough with IOMMU (input-output memory-management unit) allows direct access to resources. The IOMMU can be configured to ensure only access to guest memory is allowed. The device can then only corrupt the guest at worse.

Trap and Emulate

In this approach, the guest behaves in the same way as with direct access but in practice the VMM traps the request to access the resource and decides whether it can be permitted. This brings the benefit of increased security, because the VMM can filter hardware accesses. For stateless devices like clock controllers or GPIO (general purpose input/output), this can allow basic sharing.

Paravirtualized

The categories described previously do not allow sharing. Paravirtualization is suitable for sharing and uses a bridge to manage hardware directly. The guest OS no longer has direct access to the hardware and typically communicates with the hypervisor using a standardized interface; the guest is aware that it runs on a hypervisor. In eMCOS Hypervisor, this is usually a virtio interface. The virtual device will talk to the bridge to get access to the real world.

Developers of embedded systems need the flexibility to use any and all of these techniques together as appropriate to ensure the optimal performance. All are supported by eMCOS Hypervisor.

General Advantages of eMCOS Hypervisor

In summary, eMCOS Hypervisor offers several advantages to embedded systems developers seeking a robust and flexible platform for virtualization. These include faster development, without requiring any modification, because Linux drivers can be reused for sharing physical or virtual resources due to eMCOS' POSIX API.

Further advantages include the opportunity to ensure a faster boot sequence by taking advantage of the opportunity to completely control the startup and execution sequence. This gives developers the freedom to choose sequential and parallel multicore startup, including parallel loading of Linux and Android guests and real-time applications.

Also, true parallelism is possible, building on eMCOS POSIX's real-time multikernel design. Truly parallel guest scheduling limits cross-core interference.

eMCOS Hypervisor in Action

As a practical example, it is possible to show how eMCOS Hypervisor supports virtualization in a real-time control system, such as an industrial robot or automated teller machine (ATM). Both use cases involve mixed-criticality applications including the user interface as well as robot safety controls and ATM security features that require deterministic real-time response.

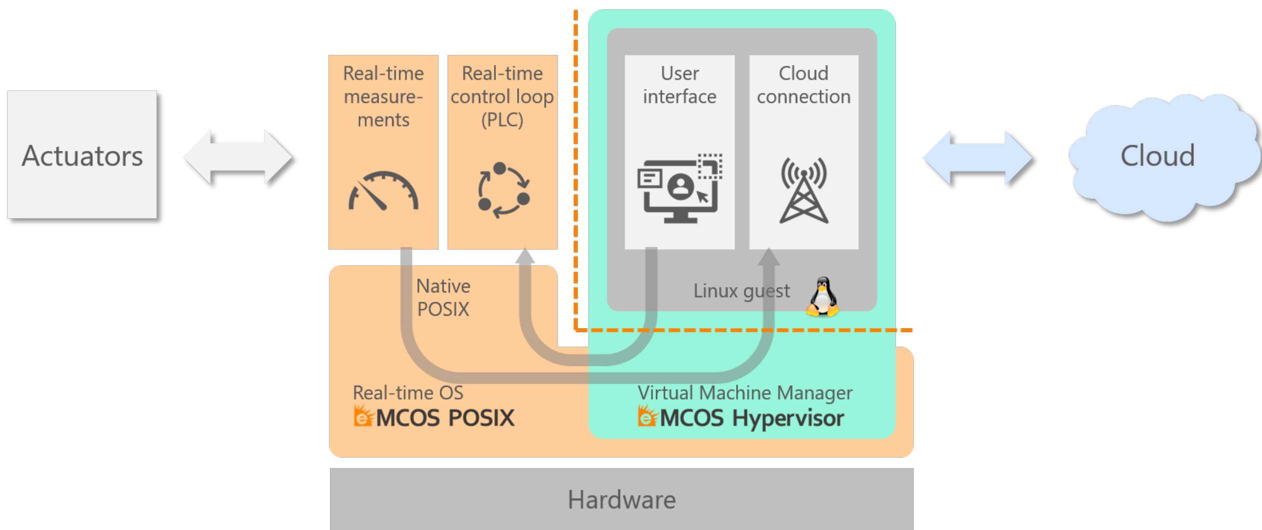


Figure 5. eMCOS Hypervisor managing applications running on native RTOS and Linux guest OS

In the example shown (figure 5), eMCOS Hypervisor runs real-time applications on the native eMCOS POSIX RTOS, alongside user-interface and connectivity applications on a Linux guest OS. eMCOS also provides valuable extra help to enhance data security and thwart known cyberattacks, including safeguards to handle authentication and ensure applications start and run correctly.

Conclusion

The relentless demand for more sophisticated functionality in embedded systems in turn calls for greater system performance and features. At the embedded edge, systems are aggregated into reduced hardware to save the bill of materials. The next step is to consolidate the systems through software evolution. Consolidating applications with mixed criticality that require high standards of security and functional safety is best handled using a hypervisor that combines the best aspects of type 1 and type 2 platforms. The eMCOS Hypervisor, which is built on a native POSIX RTOS, brings these advantages forward, has along with its unique multikernel architecture to combine high computing performance with a keep safe and secure architecture.

© eSOL 2002-2022. All rights reserved

*eSOL, eSOL Co. Ltd, eMCOS, eMCOS Hypervisor are registered trademarks or trademarks of eSOL Co., Ltd. in Japan and other countries.
*Other company or product names are trademarks or registered trademarks of their respective companies.

eSOL **E-mail: sw-inq-en@esol.com** **https://www.esol.com/**