KERNKONZEPT

White Paper

MCU VIRTUALIZATION THE L4Re MICRO HYPERVISOR

Employing Virtualization on MPU-based Processors

Jan Klötzke, Dr.-Ing. Adam Lackorzynski | Kernkonzept GmbH

HARDWARE CONSOLIDATION IN AUTOMOTIVE BY VIRTUALIZATION

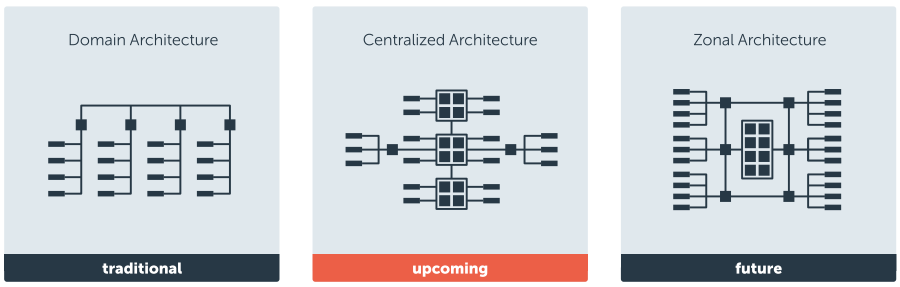

Figure 1: Automotive ECU Architecture Roadmap

- **Automotive software challenge: growth of ECUs and safety critical functions**
- **Centralizing compute approach lowers average cost per function**
- **Virtualization is broadly used in HPC systems**
- **Freedom of interference between safety-related functions required**
- **Hypervisors are state-of-the-art solution for partitioning shared systems through virtualization**

The automotive industry is faced with the challenge that an ever-increasing share of the product value is defined by software. This has led to the growth of electronic control units (ECUs) that are deployed in a vehicle. It also implies that more and more software defined functions are safety critical and must be designed according to functional safety standards such as ISO26262.

Nowadays it is not uncommon that a car sports well over 100 individual ECUs. This number has been steadily increasing for many years. To counter the associated costs, weight, and physical space requirements, the industry is moving towards a more centralized compute approach: Several adjacent ECUs are consolidated into more capable “zonal controllers”, which brings down the average cost per function.

As these previously physically separated applications now share the same computing resources, there is an increased need for virtualization solutions. This is partially driven by the desire to reuse existing ECU software stacks and deploy them almost unchanged in virtual machines on controllers along with other virtual ECUs. In the realm of automotive HPC systems, the utilization of virtualization is considered best practice and is used already on a broad scale.

Another requirement is the freedom of interference between different safety related functions. Hypervisors are the state-of-the-art solution to partition a shared system through virtualization, so that designers can ensure individual functions are separated in time and space while still meeting real-time and safety requirements.

This also applies to other safety and real-time critical application fields like medical devices and avionics. By using a hypervisor, a system can be partitioned into isolated parts, ensuring safety and non-interference while also enabling convenience functions.

1. VIRTUALIZATION-ENABLED REAL-TIME MCUs

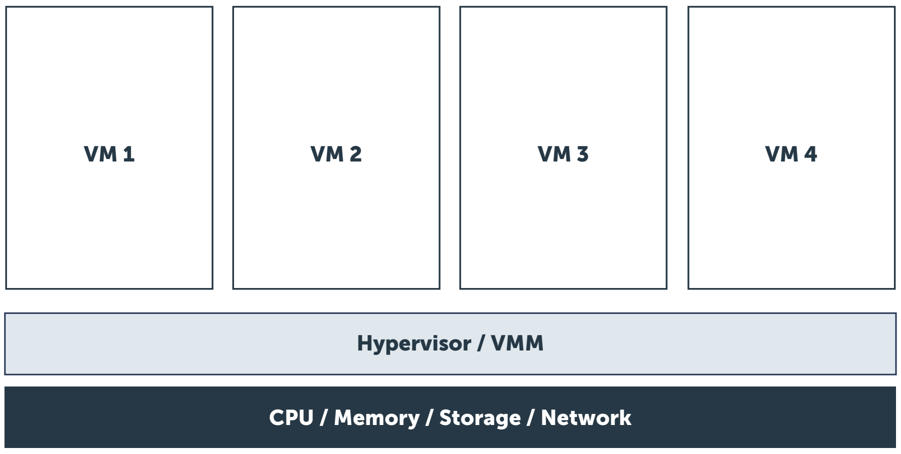

Figure 2: Generic Hypervisor System Architecture

- **Armv8-R architecture enables virtualization for MPU-based processors**
- **Hypervisor manages access to hardware for multiple guests, emulates virtual machine model by VMM**
- **Performance and resource spectrum addressed by 2 sub-architectures**

To meet the demands of automotive and storage designers, Arm has defined the Armv8-R architecture to enable virtualization for MPU-based processors. To understand the unique design challenges of these systems, let us first examine the coarse architecture of a hypervisor. Figure 2 shows the common system architecture, as it is currently used in general-purpose computing.

The hypervisor must multiplex the access of multiple guests (depicted as "VM 1 to 4" in the above figure) to the underlying hardware resources. Additionally, the hypervisor has to emulate a virtual machine model towards the guest. The component which provided this virtual machine model is usually called „virtual machine monitor" (VMM). The guest operating system interacts with the virtual machine model and is unaware of any other virtual machines executing in parallel. Access to hardware resources is always mediated by the hypervisor, either by giving selective access to hardware resources or by emulating devices in software.

From a system architecture perspective, the Armv8-R architecture has to cover a wide range of the memory and computational resource spectrum. Effectively, there are two ends of the performance and resource spectrum that are addressed by two sub-architectures:

- Arm Cortex-R52 first implementation of embedded MCUs in automotive ECUs
- Arm Cortex-R82 announced as first implementation of 64-bit CPU
- Armv8-A architecture uses same virtualization support in both subarchitectures
- Newly introduced: second MPU, controlled by hypervisor

+ **Armv8-R AArch32:** 32-bit CPUs for smaller systems with typically just a few megabytes of memory. This is the domain of embedded MCUs typically found in automotive ECUs. The Cortex-R52 is the first implementation of this architecture. It is beginning to be widely employed in the automotive industry.

+ **Armv8-R AArch64:** While still meeting real-time demands, these systems have much more memory available, typically in the range of gigabytes. These 64-bit CPUs also have much more computing power. Arm has announced the Cortex-R82 processor as the first implementation.

Both sub-architectures use the same underlying building blocks to enable virtualization in real-time systems. By and large, the virtualization support of the Armv8-A architecture is carried over almost unchanged. Meanwhile, a second – nested – MPU controlled by the hypervisor has been introduced into the architecture system.

This architecture is shown in Figure 3:

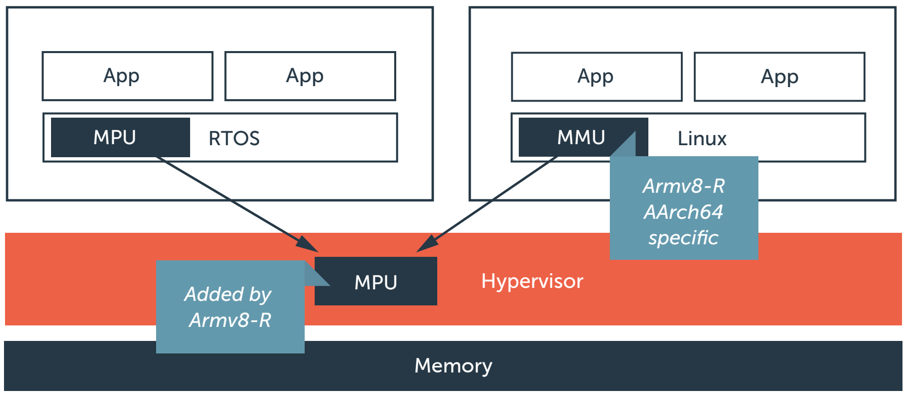

Figure 3: Armv8-R Virtualization Architecture

2. CHALLENGES FOR HYPERVISOR ARCHITECTURE

- Off-the-shelf hypervisors impossible to use with Armv8-R AArch32 architecture
- Common hypervisor architecture should combine Cortex-A and Cortex-R
- A scalable hypervisor delivers flexibility and reduced system complexity

Due to the resource constraints of MCUs that use the Armv8-R AArch32 architecture, off-the-shelf hypervisors such as KVM or Xen are not suitable, even if they would be adapted to this Armv8-R profile. Their reliance on the availability of an MMU and their memory footprint make them impossible to use on this kind of systems.

On the other hand, it is common for high-end automotive systems to combine Cortex-A compute clusters and Cortex-R real-time cores. A common hypervisor architecture is desirable that includes support for both MPU and MMU based systems. As hardware and software systems evolve, the deployment of software components will change, too. A scalable hypervisor should therefore provide common APIs on all systems to enable easy migration of workloads.

This leaves three options for more specialized hypervisors to cover the full range of the Armv8-A/R architecture:

+ **A small, bare-metal architecture:** Socalled “separation kernels” can also be counted under this architecture. This implies a ***limited feature*** set that might not be able to support all use cases on the upper end of the resource spectrum.

+ **Two implementations:** Each one would target the different ends of the resource spectrum. This approach induces ***additional complexity and costs.***

+ **A scalable architecture** that supports both AArch32 and AArch64, as well as MPU and MMU based systems. This is the desired approach from a software-architectural and maintenance view because it delivers ***flexibility and reduced system complexity.*** On the other hand this poses significant challenges to meet the low-end resource requirements.

3. L4Re HYPERVISOR & MICRO HYPERVISOR FOR MAXIMUM FLEXIBILITY

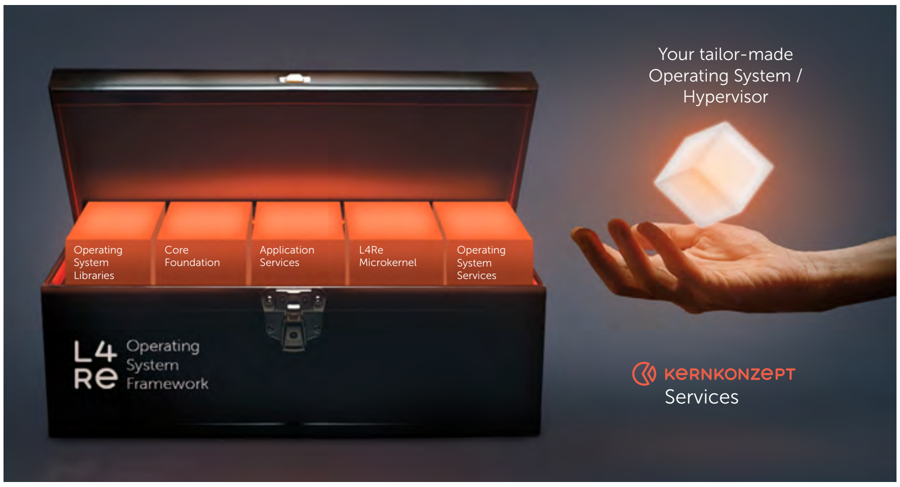
Your tailor-made Operating System / Hypervisor
Operating System Libraries
Core Foundation
Application Services
L4Re Microkernel
Operating System Services
L4 Re Operating System Framework
KERNKONZEPT Services

Figure 4: The L4Re Operating System Framework: a flexible Toolkit for tailor-made Software Solutions

- **L4Re Operating System for security, safety, and real-time focusing use cases**
- **2 specialised components provided for conflicting requirements: employable as L4Re Hypervisor or L4Re Micro Hypervisor**

A scalable solution that covers the full range of the Armv8-A and -R architectures can be achieved with a small, microkernel-based system, like the L4Re Operating System Framework.

The L4Re Hypervisor and L4Re Micro Hypervisor are part of the L4Re OS Framework, which builds upon the L4Re Microkernel, the L4Re OS Libraries, the L4Re Core Foundation, the L4Re Application services, and the L4Re OS services – several user-level components and libraries, allowing for use cases that focus on security, safety, real-time, or any combination thereof.

The L4Re Operating System Framework provides specialized components for the various tasks in a system. Because of conflicting requirements (e.g. size vs. features), two specialized implementations of selected components exist, e.g. the VMM. These can be deployed in a very flexible way to create tailored systems for the specific use case.

Depending on the requirements of the customer, the components can be employed as L4Re Hypervisor or L4Re Micro Hypervisor.

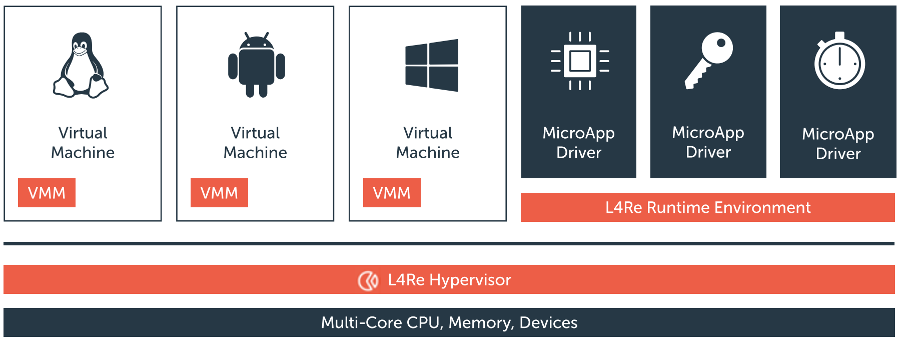

Figure 5: L4Re Operating System and Hypervisor Framework

- **State-of-the-art L4Re Microkernel with object capabilities & dedicated VMM**
- **User-level components allow tailored designs for different use cases**
- **Very small TCB makes obtaining safety and security certifications for L4Re systems easier**

The state-of-the-art L4Re Microkernel is the foundation of both the L4Re Hypervisor and the L4Re Micro Hypervisor. It is a crucial piece of the architecture, which provides access-control by object capabilities. It is the only component running in kernel-privilege mode of the processor.

While the microkernel only implements mechanisms, functionality is built by user-level components running on the system. This design allows to build applications and use cases with tailored minimal trusted computing bases (TCBs), providing secure, safe, and realtime applications.

Virtualization is a crucial functionality of the system, and its implementation is split in two parts. The microkernel implements functionality concerned with so-called world switching and thus isolation, while a virtual machine monitor (VMM) implements the virtual platform presented to guest OSs. A typical design is to run one VMM per VM to ensure isolation between different VMs running parallel on the system. However, a single VMM can also host multiple VMs.

Compared to other hypervisors, L4Re provides a vastly reduced TCB (see figure 6). This makes it easier to obtain safety and security certifications for systems built with L4Re. Functionality that does not have to be co-located is separated into different components. That way, the microkernel can enforce their separation.

In a standard setup of the L4Re system, there is a root task that owns all the resources (memory, CPUs) of the system. Another task is then responsible for starting applications and virtual machines, acquiring the required resources from the root task, and assigning them to the different tasks in the system as needed.

This hierarchical delegation of resources is key to partitioning the system in a reliable and traceable way.

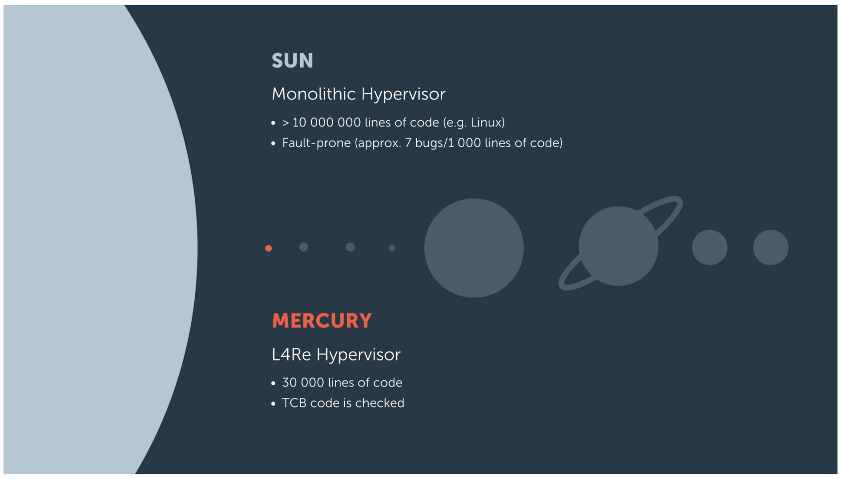
SUN
Monolithic Hypervisor
• > 10 000 000 lines of code (e.g. Linux)
• Fault-prone (approx. 7 bugs/1 000 lines of code)
MERCURY
L4Re Hypervisor
• 30 000 lines of code
• TCB code is checked

Figure 6: Trusted Computing Bases compared: L4Re Hypervisor versus Monolithic Hypervisors

- ***Tinit*** **process handles system setup, uniting several components**
- ***Tvmm*** **is hosting VMs, requiring minimal code**
- **Optimal balance of isolation and resource usage**

3.1 The L4Re Micro Hypervisor for small-scale System Architectures

Small-scale MPU systems (Armv8-R AArch32 – Cortex R52) typically only have a few megabytes of memory to host the hypervisor and the VMs with their actual applications. Additionally, platforms provide multiple processor cores that need to be managed by the hypervisor. Such systems usually have an almost static setup. The only typical exception are updates based on A/B schemes, where a different image is loaded on the next boot after a successful update.

Given the static setup and the tight resource constraints, the system initialization is consolidated into a single task called *tinit*. It unites the jobs of several components to save precious resources by trading it with the (in this use case unnecessary) isolation of these components. Likewise, a specialized virtual machine monitor variant named *tvmm* is used.

Tinit processes a small text-based configuration file to launch an application with its shared memory regions, and it also launches *tvmm*. The individual VMs are also defined within this file, through descriptions of the RAM range the VM uses, the device interrupts and priorities, the MMIO regions, and the payload to load into each VM. In this way, the system setup is exclusively handled in the *tinit* process and *tvmm* is solely responsible for running the VMs.

Tvmm is capable of hosting one or more VMs. Because its setup is done solely by *tinit*, it requires minimal code to support the execution of the guests. The listing below shows an example of a *tinit* configuration file where one *tvmm* instance is started that hosts two VMs.

Based on the non-functional requirements of the system, the designer can choose the optimal balance of isolation and resource usage.

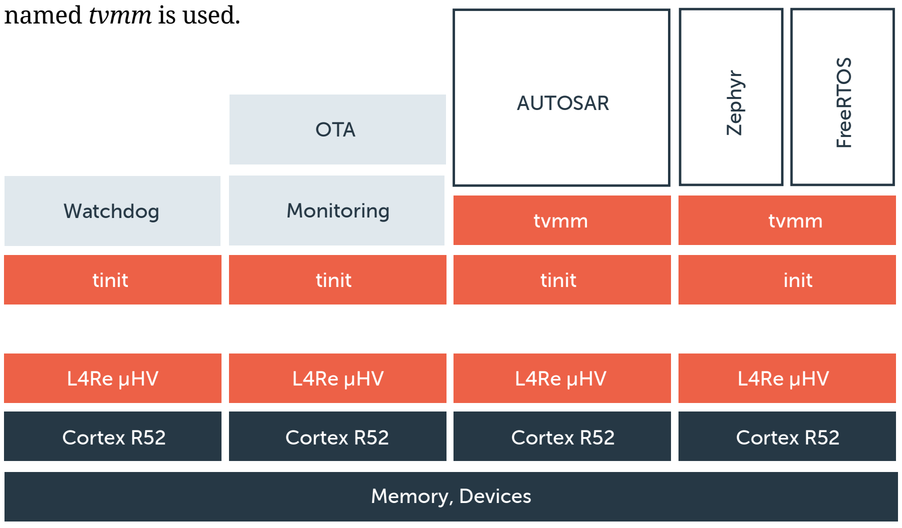

Figure 7: L4Re Micro Hypervisor Example on Cortex-R52

```
start tvmm
  # define high-priority Zephyr VM
  defvm zephyr 0x80
        ram 0x34800000 0x100000
        shm 0x34a00000 0x800 # shared with
  FreeRTOS VM
        mmio 0x25a00000 0x10000
        mmio 0x25610000 0x10000
        irq 40
        irq 41
        irq 48
        irq-priorities 0 0x7f
        load zephyr_image
  end
  # define low priority FreeRTOS VM
  defvm freertos 0x10
        ram 0x34900000 0x100000
        shm 0x34a00000 0x800 # shared with
  Zephyr VM
        load freertos_image
        irq-priorities 0 0x0f
  end
end
```

- **L4Re Hypervisor on Armv8-A and Armv8-R architecture supports realtime OS and fully featured Linux VMs**
- ***uvmm* in L4Re Hypervisor also supports MPU-based guests (FreeRTOS, Zephyr)**

3.2 The L4Re Hypervisor for high-end System Architectures

On the high end of the resource spectrum (Armv8-R AArch64 – Cortex R82), memory is not a scarce resource. These systems are used for computation-intensive tasks. They can host multiple, fully featured Linux VMs and real-time operating systems. Some of these VMs may even be provisioned dynamically. These kinds of setup are well supported by the L4Re Hypervisor when running on the Armv8-A architecture. Thanks to the common L4Re APIs, all L4Re Hypervisor components also work on Armv8-Rbased devices.

The standard VMM in the L4Re Hypervisor, *uvmm*, has been extended to also support MPU-based guests. It is possible to run guests like FreeRTOS or Zephyr in virtual machines as well. *Uvmm* targets general-purpose guest operating systems and is designed to cover common use cases, from high-performance embedded to cloud scenarios. It is highly configurable and offers multiple features:

· modular design
· pass-through and emulated devices
· support for VirtIO and VirtIO devices
· virtual PCIe
· suspend/resume support
· device tree support, augmenting the device tree for the guest
· support for UEFI
· multi-core guest support
· a monitor interface to control and inspect the VM state

```
vmm.start_vm({
  id = 1,
  mem = 128,
  rd = „rom/ramdisk-armv8-64.cpio.gz“,
  fdt = „rom/uvmm_arm_virt-64.dtb“,
  bootargs = „console=hvc0 earlyprintk=1“,
  kernel = „rom/Image.gz“,
});
vmm.start_vm({
  id = 2,
  mem = 32,
  kernel = „rom/FreeRTOS.elf“,
  fdt = „rom/uvmm_freertos.dtb“,
});
```

- **Modular L4Re OS Framework with common APIs gives system designers choice of components, as required by use case**

The standard application to bring up the system is called *ned* in the L4Re Framework. It parses a Lua script to spawn the desired components in the system. As an interpreted language, Lua allows for almost unlimited freedom of configuration. Despite merely creating VMs, it is possible to act upon termination of a VM, regardless of a VM crashing, rebooting, or shutting down.

The listing below shows a simplified excerpt of an example system configuration. It starts two VMs that are then executed concurrently in the same core(s). The virtual machine itself presented to the guest is defined by the device tree passed by the “fdt” argument.

This approach allows for a very flexible definition of the virtual machine. It allows execution of an unmodified guest, which is especially crucial for operating systems that were running directly on a compatible MPU platform.

3.3 Use Case specific Tailoring

Thanks to the modular nature of the L4Re Operating System Framework, the choice of components is up to the system designer. Because the APIs are common, the above components can be mixed and matched between the different profiles, as required by the system architecture.

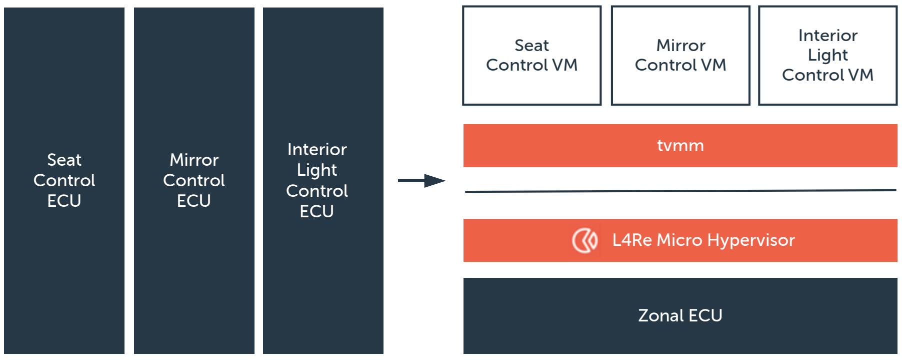

Figure 8: ECU consolidation: Multiple separate ECUs are co-located into a single zonal ECU

3.3.1 Use Case: Automotive ECU with multiple VMs

With a virtualization-capable platform you can integrate multiple applications into a single ECU (Electronic Control Unit). The applications are safely and securely separated from each other to avoid interference. This enables the design of zonal ECUs, running multiple logical ECUs on one physical ECU (see Figure 8).

By employing virtualization techniques, you maximize legacy reuse with minimal development effort. Reducing the number of physical ECUs drives down the overall energy use, the wiring, and the complexity of the vehicle.

The L4Re Micro Hypervisor provides all required functions to isolate the various workloads and to guarantee freedom from interference. By moving the previously physical ECUs into VMs, the investment into the existing software stack can be retained and made independent from the underlying hardware platform. Additionally, it enables smooth migration of the VMs into future ECUs that are even more capable.

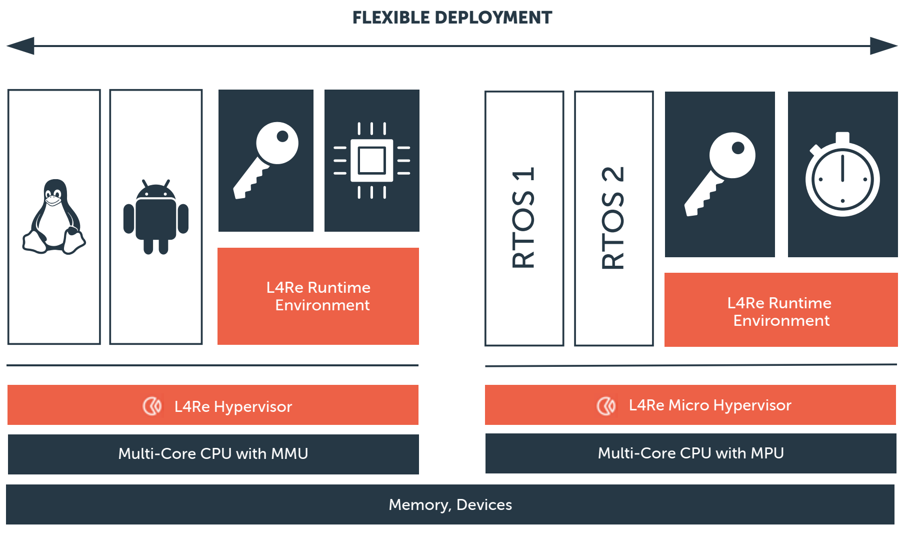

Figure 9: Heterogeneous SoC with L4Re Hypervisor and Micro Hypervisor

- **L4Re Micro Hypervisor guarantees freedom from interference in ECUs**
- **L4Re Framework with common APIs enables flexible deployment of components in SoC**

3.3.2 Use Case: Heterogeneous Automotive SoC

With ever growing integration, automotive SoCs are a composition of different, specialized subsystems. This typically includes throughput-oriented, high-performance CPUs with MMU on one side and real-time capable CPUs for latency and safety critical workloads on the other side. Both subsystems are interconnected and realize different parts of the ECU functionality. Where individual software components are deployed depends on the specific requirements, but also on resource availability.

The L4Re Framework provides common APIs across all systems. This architecture enables the flexible deployment of components in the SoC. You could also migrate workloads between different kind of processing units, MPU-based or MMU-based, if requirements allow for such variance.

- L4Re Hypervisor separates storage controller functions from data processing for computational storage
- L4Re Hypervisor runs on Armv8-R AArch64, including Cortex-R82
- L4Re Framework ensures isolation of storage RTOS on Armv8-R

3.3.3 Use Case: Storage Controller

Storage controllers are gaining more computational resources with every new generation. This has led to the requirement to move to 64-bit architectures to keep up with the required memory capacity. Likewise, the increasingly complex functions that need to be deployed on the storage controllers require more sophisticated operating systems, such as Linux.

Computational storage is a trend that takes this development even further, by deploying data processing functions directly into the storage controller (see Figure 10). The unique property of such a system is that real-time functionality is co-located next to general purpose, throughput optimized workloads on the same CPU.

Deploying such value-added functions directly to the storage controller maximizes performance while driving down power consumption and overall hardware costs. However, it requires proper separation of the core real-time storage controller functions from the use case specific data processing. This separation is provided by the L4Re Hypervisor.

The L4Re Hypervisor has been adapted to run on Armv8-R AArch64, including the upcoming Cortex-R82 processor that enables such a use case. Building on the foundation of the field-tested Armv8-A L4Re Hypervisor, the L4Re Framework runs multiple Linux VMs and ensures the proper isolation of the storage RTOS on Armv8-R as well. Depending on the requirements it might even include dynamic provisioning of the user VMs.

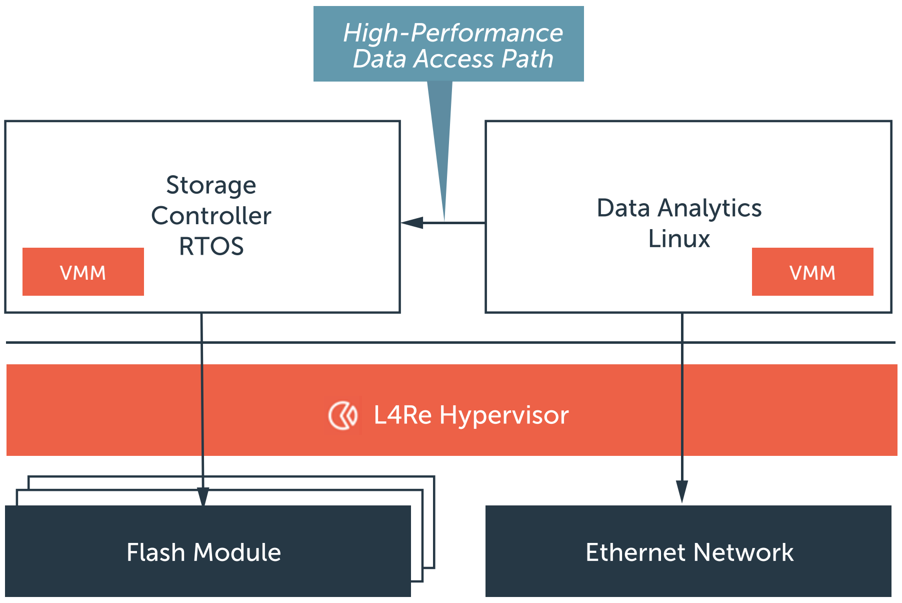

Figure 10: Storage Controller Use Case employing real-time and Linux Workloads

ADVANCED SAFETY & SCALABILITY WITH THE L4Re HYPERVISOR FAMILY

L4Re benefits for automotive software design:

- Safe upgrade path in the software stack
- Reduced complexity for developers
- Certification-ready systems, adhering to functional safety as well as security standards like Common Criteria EAL
- Consistent frame-work from small MMU-based to cloud server and HPC systems

The L4Re Operating System and Hypervisor Framework provides the technology to cover the whole spectrum of ARMv8-R virtualization. By leveraging the same APIs on both ends of the resource spectrum, applications can be deployed with no or only minimal changes on the whole spectrum. As systems are evolving and gaining more resources and capabilities, the L4Re architecture provides a safe upgrade path, economizing investments in the software stack.

By deploying both the L4Re Hypervisor and the L4Re Micro Hypervisor alongside each other, you further reduce complexity for designers and developers.

With the L4Re Operating System Frame-work, especially the combined use of L4Re Hypervisor and L4Re Micro Hypervisor for MPU, the objectives of consolidation and safety can be pursued simultaneously with a solution with uniform APIs and allowing full scalability.

The modular microkernel approach is key to enabling certification of systems, according to functional safety as well as security standards such as Common Criteria EAL. The L4Re Micro Hypervisor provides the foundation to build certifiable, virtualized safety systems on the ARMv8-R technology. The small code base minimizes effort and time for certification and enables using MPUs for safety and security purposes in the first place.

Along with the L4Re Hypervisor for MMU based systems, L4Re provides a consistent framework from small, safety-critical systems up to cloud server virtualization scenarios.