INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN


**Bachelorarbeit**

# ARM Virtualization Using VMware ESXi Hypervisor For Embedded Devices

Caroline Frank

Draft vom March 17, 2020

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN


**Bachelorarbeit**

# ARM Virtualization Using VMware ESXi Hypervisor For Embedded Devices

Caroline Frank

| | |
|---|---|
| Aufgabensteller: | Prof. Dr. Dieter Kranzlmüller |
| Betreuer: | Jan Schmidt, MSc<br>Dr. Tobias Lindinger (VMware) |
| Abgabetermin: | 04. März 2020 |

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München,
March 17, 2020

.....
*(Unterschrift des Kandidaten)*

**Abstract**

One of the fastest-growing sectors in the digital technology market is the embedded device domain. Embedded devices are simple computing devices that form the basis of nearly all complex systems. However, embedded device development is limited by issues such as system complexity, security vulnerabilities, and resource constraints. Virtualization has been used since the 1970s to solve these same issues in mainframes and later servers and personal computers. This paper aims to answer whether hardware virtualization is a practical solution to the problems embedded devices currently face. First, the constraints that an embedded hypervisor would need to adhere to are elucidated and use cases examined for virtualization on embedded devices. A feasibility study was performed using a modified version of the server domain VMware ESXi hypervisor. The ESXi hypervisor, initially only capable of running on x86 processors, was modified to run on ARM processors, referred to as ESXi-ARM in this work to differentiate from ESXi for x86.

The ESXi-ARM hypervisor used for testing in this paper is a in-development, pre-release version with minimal performance optimization work done. This prototype ESXi-ARM hypervisor was found to be suitable for some embedded devices. Using the benchmarking suites LMbench3 and MiBench, it was found that the ESXi-ARM hypervisor on the MACCHIATObin Single Shot development board equipped with an ARMv8 processor has only modest virtualization overhead compared to native execution. Additionally, ESXi-ARM was able to host up to five virtual machines, with variable virtualization overhead depending on the application. Using Linux's cyclictest, it was determined that ESXi-ARM increased the scheduling latency, which lowered the deterministic behavior of the system and increased the worst-case execution time, both crucial characteristics of real-time devices. Future work on the prototype ESXi-ARM hypervisor should focus on improving the scheduling policy for devices that have real-time needs.

# Contents

# 1 Introduction

## 1.1 Motivation

Many embedded systems are striving to become more intelligent and offer greater functionality due to pressures such as market competition and resource constraints. As these systems become smarter and integrate better with their environment, they require more complex embedded devices to support this functionality. Market Research Future predicts the embedded software market to grow from $10 Billion in 2016 to $19 Billion by 2022, at an estimated compound annual growth rate of 9% [Fut19]. However, according to this same report, growth is constrained by processing power, battery usage, memory, safety, security, real-time constraints, and growing hardware and software complexity [Fut19]. Several studies on embedded systems have proposed virtualization to mitigate these constraints as this technology has been used since the 1970s to encapsulate functionality in the server and computer domains [Hei08, LX15, IDC14]. Nevertheless, embedded systems have fundamental differences to servers and computers in their purposes, design, and goals. A more in-depth analysis of requirements of embedded systems is warranted to define constraints that an embedded hypervisor must adhere to and evaluate use cases for virtualization [LX15].

## 1.2 Scope of this Work

The x86 processors are based on a complex instruction set computer architecture (CISC) and were first virtualized in 1998 by VMware. Virtualization had an enormous impact in the server space, such as lowering cost, increasing security, and simplifying management [VMw07]. Although the x86 is still the leading processor of today's servers and personal computers, reduced instruction set computer (RISC) based architectures lead the embedded and mobile devices market due to higher efficiency and lower energy consumption, leading to less heat generation [Sch19]. Recent changes in the instruction set architecture (ISA) and increases in memory and processing power for RISC-based processors have opened up the possibility of virtualizing embedded devices.

Of particular interest are processors that have built-in virtualization support, which increases the prospect of an efficient virtualization solution. ARM Holdings released its first core, the ARMv7, with virtualization extensions in 2010. Promising results have been published based on XEN/ARM and KVM/ARM hypervisors with modest performance and power costs compared to x86-based Linux virtualization on multicore hardware [BNT17]. The newer ARMv8 ISA processors have additional hardware virtualization extensions with relatively fewer performance studies compared to ARMv7.

Problem statement and contributions

This work aims to determine whether hardware virtualization is a potential solution to the problems faced by embedded devices. Contributions of this work are to better define the requirements, use cases, and limitations of virtualization for embedded devices. Further, to examine the viability of an embedded hypervisor, this work evaluates the performance of a prototype version of VMware's ESXi hypervisor modified to run on an ARMv8 ISA CPU. Finally, potential areas of work that are required to implement ESXi-ARM as an embedded hypervisor are identified.

## 1.3 Structure of this Work

Chapter 2 begins by providing an introduction to virtualization definitions and notations, defining the fundamental differences of hypervisors and the basics of hardware virtualization. Thereafter, the chapter discusses the principal functions of a hypervisor and potential areas of overhead. Chapter 3 provides an overview of the embedded devices landscape by defining the different embedded domains and the design demands of different subdomains. Further in this chapter, the requirements and challenges of an embedded hypervisor are surveyed, and use cases of virtualization in the embedded domain are discussed.

These introductory chapters are meant to give the background necessary to evaluate an embedded hypervisor. Chapter 4 provides the methodology and configurations of benchmarking the prototype ESXi-ARM hypervisor on the MACCHIATObin Single Shot development board. Results from the benchmarks are found in chapter 5. The final chapter 6 evaluates ESXi-ARM as an embedded hypervisor and identifies future research areas.

# 2 Background

The following chapter reviews virtualization fundamentals with focus given to hardware virtualization. First, section 2.1 introduces the basic virtualization techniques and type 1 and type 2 hypervisor architectures. The Popek and Goldberg requirements of a virtual machine are defined next in section 2.2. Section 2.3 then focuses on the functions of a type 1 hypervisor and ARMv8-A hardware extensions to clarify hypervisor design choices and how these design choices can affect system performance.

## 2.1 Virtualization Definitions and Notations

Virtualization is the separation of a service request from the underlying physical delivery of that service [VMw07]. Virtualization technology can be implemented at different abstraction levels, such as memory, operating system, or network virtualization, and within different domains, such as desktop or server virtualization. However, almost all virtualization solutions are based on three basic techniques: multiplexing, aggregation, and emulation [BNT17].

- **Multiplexing** is the process of exposing one resource multiple times. There are two variants of multiplexing. The first occurs in space, such as by partitioning a physical resource among virtual entities. The second variant occurs in time, by scheduling the physical resource temporarily to virtual entities [BNT17].
- **Aggregation** is the process of combining multiple physical resources to appear as a single resource or abstraction such as storage virtualization [BNT17].
- **Emulation** relies on software to expose a virtual resource that corresponds to a physical device that may or may not be present. The virtual resource exposed may also be different than the physical one [BNT17].

Hardware virtualization concerns itself with the creation of one or more virtual computing environments directly on a physical device, known as the host device. The virtualization software layer, known as the hypervisor, creates a simulated computer environment, known as a virtual machine (VM). The VM is capable of executing software such as applications and complete operating systems (OS) [BNT17]. Each VM is managed by a virtual machine monitor (VMM), which is a part of the hypervisor. The hypervisor works together with the VMMs to partition and assign the physical resources such as central processing unit (CPU) time, memory, and Input/Output (I/O) devices among the VMs [VMw07].

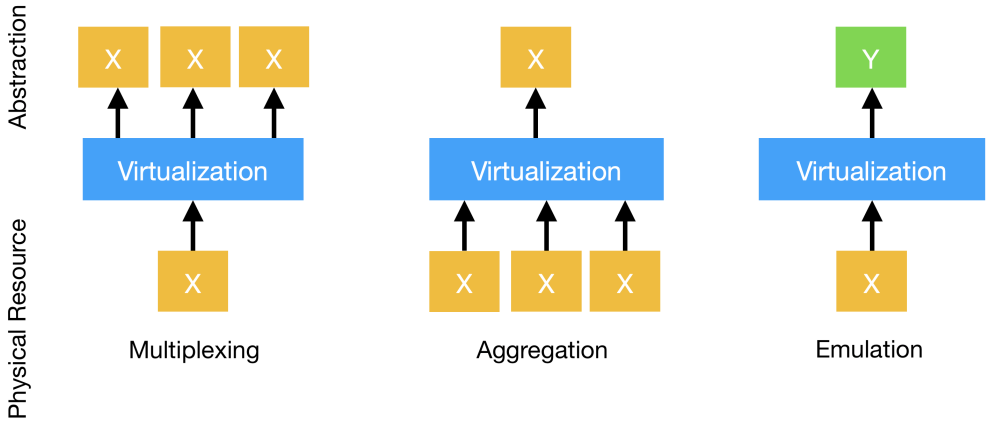


Figure 2.1: The three fundamental techniques to virtualization, adapted from [BNT17]
.

There is unavoidable overhead associated with hardware virtualization. The performance of a virtualized environment compared to a native (non-virtualized) environment depends strongly on the implementation and architecture of the hypervisor [VMw07]. There are two types of hypervisor architectures, classified on their location in the computing system.

A type 1 hypervisor sits directly above the host's hardware and is thus known as a Native or Bare Metal hypervisor. Type 1 hypervisors are in direct control of all hardware resources and integrate essential components of the OS, such as the kernel [BNT17]. Type 2 hypervisors, also known as Hosted hypervisors, run on extended hosts, meaning the host device has an OS on which the hypervisor software runs. For type 2 hypervisors, the host OS has sole responsibility for loading the hypervisor and communication with the hardware. However, the allocation of resources and the creation of virtual environments is typically performed by both the host OS and the type 2 hypervisor [BNT17]. The OS running in a VM on type 1 and type 2 hypervisors is known as the guest OS.

## 2.2 Popek and Goldberg Theorem

Popek and Goldberg defined the requirements of a VM as "an efficient, isolated, duplicate of the real machine" in their 1974 paper, *Formal Requirements for Virtualizable Third Generation Architectures* [PG74]. Popek and Goldberg further defined the three characteristics of a VMM as providing an environment essentially identical to the original machine, with only minimal decreases in speed and where the VMM is in full control of system resources [PG74]. The last characteristic necessitates that a program in a VM cannot access resources that are not explicitly allocated to it and that the VMM can regain control of resources already allocated to a VM [PG74].

For a virtualized system to fulfill the isolation requirement, instructions from a VM that could potentially interfere with another VM residing on the same host device need

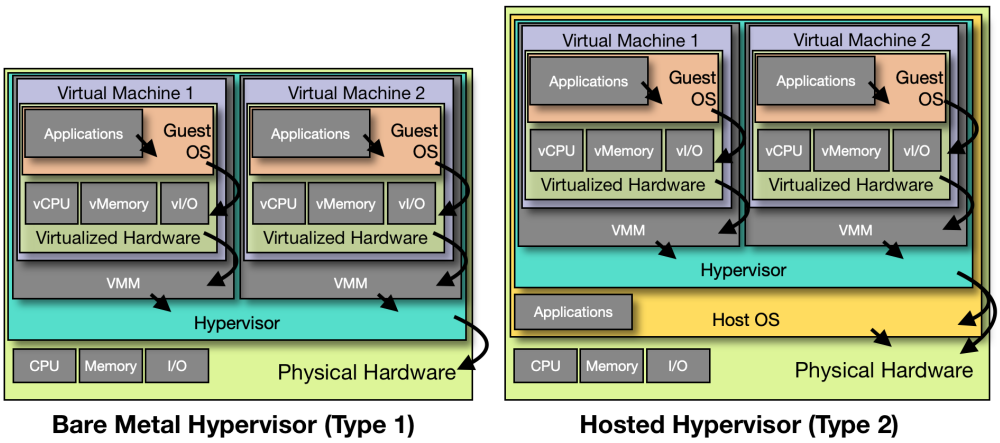


Figure 2.2: Comparison of a system virtualized using a type 1 hypervisor and a type 2. Arrows represent instruction flow.

to be modified to prevent unwanted interactions and unsafe operations. For a VM environment to be a duplicate of the real machine, instructions executed in a VM have to have the same effect as when they are executed on a native system. Finally, for a virtualized system to be efficient, a statistically dominate subset of the virtual processor's instructions must be executed directly by the real processor without hypervisor intervention [PG74].

To understand how a hypervisor achieves hardware virtualization, it is important to understand how instructions are executed. The CPU is the hardware that carries out instructions of a computer program by performing logic, controlling, and input/output operations. In a native device, the OS is responsible for communicating directly with the hardware, including scheduling instructions to execute on the CPU. A CPU executes instructions in different protection domains to protect data from faults and malicious behavior. Although most CPUs now have more than two execution modes, in essence, the modes can be divided into Kernel and User mode. Kernel mode is reserved for OS instructions and is the most privileged domain, allowing direct communication with the underlying hardware. Application instructions execute in User mode but can request OS services such as reading data from memory. In a virtualized device, the hypervisor executes in Kernel mode so that it can directly control hardware and manage resources for different VMs. All programs inside a VM, including the guest OS, is de-privileged to run in User mode. However, certain OS instructions need to be executed in Kernel mode to be executed properly, and often, a guest OS inside a VM does not know it is virtualized. A mechanism needs to be in place for these instructions to result in a trap, a type of synchronous interrupt which informs the hypervisor that it needs to intervene.

The set of instructions, $O$, in a *instruction set architecture* (ISA) can be divided into three sets of classes, $O = P \cup S \cup I$ [PG74, Xia16] where P, S, and I are defined as:

- **Privileged instructions** are those that can only be executed in Kernel mode,

thus always trap when executed in guest mode [Xia16].

- **Sensitive instructions** are those that may threaten the correctness of a virtual environment if they do not result in a trap. These are further divided into (1) control-sensitive instruction, which manipulate the processor or memory configurations and (2) behavior-sensitive instruction, whose results depend on the processor mode [Xia16].
- **Innocuous instructions** are those that can be executed in any mode without concern to correctness [Xia16].

According to Popek and Goldberg's Theorem, if all sensitive instructions are also privileged, then a system is classically virtualizable by trapping all privileged instructions, see Figure 2.3. Trapping all privileged instructions will also trap all sensitive instructions since $S \subseteq P$ [PG74]. However, this criterion is rarely met for most ISAs, necessitating additional handling of sensitive instructions that are not privileged. Three techniques have been developed to support efficient virtualization.

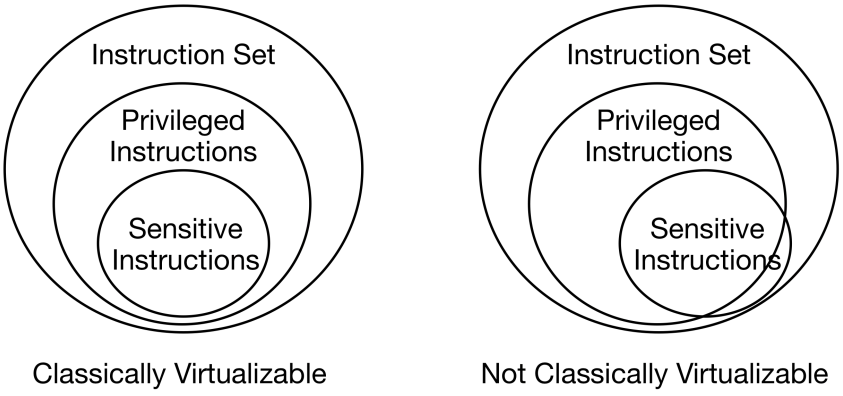


Figure 2.3: Popek and Goldberg's Definition of Classically Virtualizable and Non-Virtualizable ISAs

#### Technique 1 - Full Virtualization with Binary Translation

Full virtualization of a system requires that the complete underlying hardware is simulated, including the entire instruction set, input/output operations, interrupts, memory access, etc. Direct binary translation (DBT) uses emulation to translate from the source to the target instruction set through recompilation. In a system virtualized using DBT, the hypervisor examines all instructions from a guest OS instructions for sensitive instructions and translates them into safe, equivalent instructions before scheduling them to be executed by the CPU [Dan18].

The first successful virtualization of the x86 architecture used a combination of binary translation and direct execution, known as trap-and-emulate [VMw07]. In this approach, non-sensitive instructions are scheduled directly to be executed by the CPU while all

sensitive instructions must be examined by the hypervisor. This approach enforces encapsulation of the VM and enables any unmodified guest OS that runs on the bare metal hardware to execute in a VM. However, analyzing and translating VM instructions requires significant software complexity, processing time, and energy [SGB+16]. The slowdown, $S_v$, of a system using direct execution with binary translation is related to the fraction of privileged instructions executed by a VM, $f_p$, and the number of instructions needed to emulate a privileged instruction, $N_e$ [Men05].

$$S_v = f_p(N_e–1) + 1 \tag{2.1}$$

#### Technique 2 - Paravirtualization

Paravirtualization was developed to avoid the overhead from analyzing and translating instructions required by binary translation. Paravirtualization is based on the modification of the guest OS source code to replace the sensitive non-privileged instructions with special hooks to directly interface with the hypervisor, known as hypercalls (hypervisor calls) [VMw07]. The hypervisor provides an interface for communication with the guest OS that includes information on how to execute the instructions [BKL10]. Like binary translation, paravirtualization requires no specialized hardware to virtualize an ISA. It also has significant performance benefits over binary translation since instructions no longer need to be analyzed prior to execution. However, modifying the guest OS source code is often problematic since any changes to OS source code requires recertification that no errors or vulnerabilities occurred through the modifications [Xia16]. Also, most OS vendors do not publicize their source code, significantly limiting which OSs can be virtualized.

#### Technique 3 - Hardware-Assisted Full Virtualization

Hardware-assisted full virtualization evolved to promote faster VM execution without the need for guest OS modification. This approach uses a computer's physical components to support a hypervisor's virtualization tasks. Hardware extensions are usually built into the CPU and simplify virtualization by enabling nested paging, additional execution modes, and additional instructions [Dan18].

A pure hardware-assisted virtualization approach typically involves a high degree of VM traps, resulting in high CPU overhead and limited scalability [R19]. A hybrid approach of hardware-assisted full-virtualization with paravirtualized device drivers has achieved better performance [R19].

## 2.3 Functions of a Hypervisor

Type 2 hypervisors usually result in better hardware compatibility because the OS from the host device is responsible for the device drivers. However, type 2 hypervisors require a large codebase and suffer from high computational overhead, making them unsuitable for

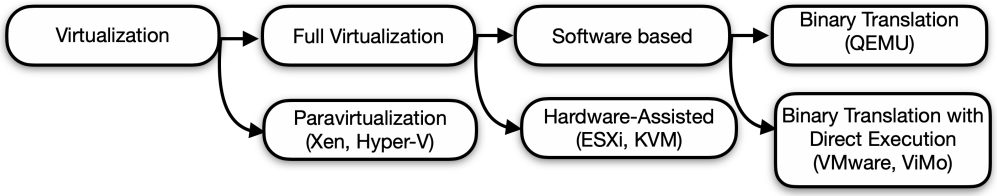


Figure 2.4: Taxonomy of Virtualization Techniques

Table 2.1: Advantages and disadvantages of the three basic virtualization techniques.

| Virtualization Technique | Advantage | Disadvantage |
|---|---|---|
| Binary Translation<br>Full Virtualization | - No special hardware needed<br>- Guest OS can remain unchanged | - CPU intensive to translate OS code on-the-fly<br>- Introduces substantial software complexity |
| Paravirtualization | - No special hardware needed<br>- Hypercall communication (usually) has better performance than DBT | - Guest OS must be modified<br>- OS (potentially) limited to open-source ones<br>- Modified OS must be re-certified |
| Hardware-Assisted<br>Full Virtualization | - Guest OS can remain unchanged<br>- Hypervisor can be implemented with less software complexity | - Relies on specialized hardware |

embedded devices. For this reason, the remainder of this paper concerns itself exclusively with type 1 hypervisors.

Hypervisors must simulate entire computing environments to fulfill Popek and Goldberg's requirements. Apart from instruction trapping, discussed in section 2.2, functions performed by type 1 hypervisors include, but are not limited to: context switching, exception handling, interrupt controller management, managing virtual exceptions, memory management, memory translation, and device emulation and assignment [ARM17a].

### 2.3.1 Context Switching

Context switching is the process of saving the currently executing task state so that it can be resumed from the same point at a later time. Context switches are computationally intensive, requiring a significant amount of administration work such as saving and loading registers and memory maps, updating various tables and lists, etc. Additional performance is lost due to running the task scheduler, TLB flushes, and indirectly sharing the CPU cache between multiple tasks.

In a native environment (non-virtualized), context switches occur due to processor mode switching, multitasking schemes, and interrupt handling. Virtualizing a system leads to additional context switches, such as when a hypervisor must intervene during normal VM execution or when switching the currently executing VM on a processor to a different VM. A critical difference in hypervisor performance is how often a context switch is required during normal VM execution and how long those context switches take.

On ARMv8 systems, an optimistic lower bound is determined by the amount of time required to perform 33 instructions that are responsible for saving and restoring 31x64-bit

general-purpose registers, 32x128-bit floating-point/SIMD registers, two stack pointers, and the pending and active states of private interrupts on the core [ARM19, ARM17a]. In reality, the time required for a context switch is much higher due to the time required for setting up a new context environment or finding a saved context.

### 2.3.2 Processor Mode Switching

The processor switches modes from User to Kernel when an illegal instruction is executed or when an exception occurs, synchronous or asynchronous. The mechanism to switch control to the kernel always follows the order: (1) change the processor to kernel mode; (2) save and reload the memory management unit (MMU) to switch to the kernel address space; and (3) save the program counter and reload it with the kernel entry point. Returning to User mode occurs in the following order: (1) change mode from kernel to user; (2) reload MMU state; and (3) load PC that was saved on entering. Not all processor mode switches require a full context switch, so depending on the task, guest OS, hypervisor, and hardware; processor mode switches can lead to varying amounts of overhead.

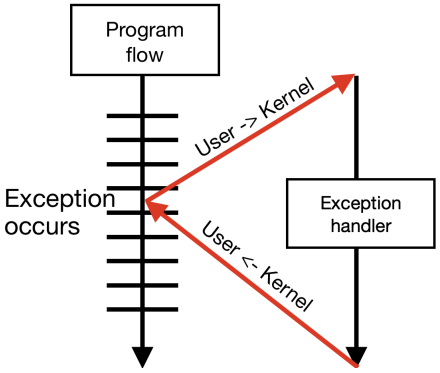


Figure 2.5: Visualization of the instruction flow when an exception occurs. The processor must switch from User to Kernel mode to invoke the exception handler routine (or interrupt handler). Once the exception has been handled, the processor switches back to User mode.

ARMv8 processors simplify processor mode switching by the introduction of additional execution modes, renamed exception levels (EL), see Figure 2.6. EL3 is the most secure mode and is reserved for the secure monitor. The secure monitor is an additional security feature available on certain ARM processors and is responsible for requesting processor mode switches. It allows the implementation of TrustZones that divides the world into secure and normal world. Each physical processor core provides a secure and non-secure processing element (PE), covering the processor, memory, and peripherals. In the normal world, EL2 is reserved for a hypervisor while EL1 is reserved for OSs. All other software and applications run in EL0. Virtualization extensions can only be accessed from EL2 or EL3 [ARM17a, ARM19].

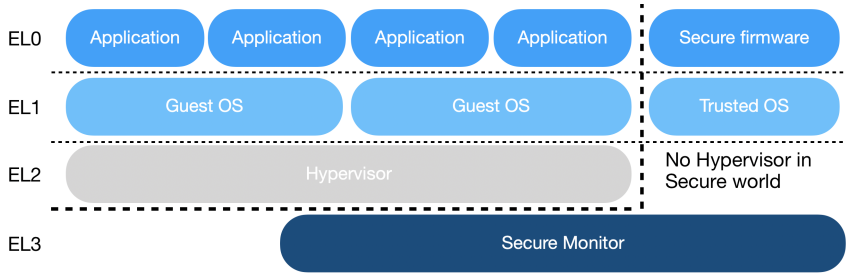


Figure 2.6: Exception levels in AArch64, adapted from [ARM16].

### 2.3.3 Exception Handling

Exception handling is the process of responding to anomalous conditions that require special processing, disrupting the normal flow of program execution. ARM defines exception handling as the process of handling synchronous exceptions, those that are raised internally by the processor during instruction execution. Synchronous exceptions always trigger a processor mode switch to handle the exception, see figure 2.7.

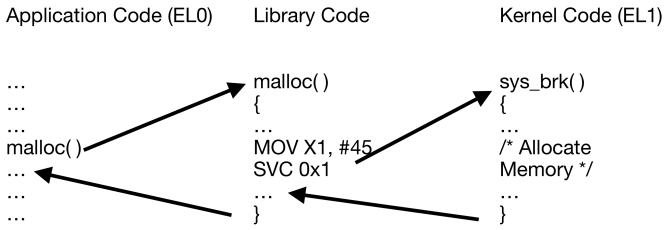


Figure 2.7: Synchronous exception handling in a non-virtualized system call for ARMv8 ISA. Application executing in User mode (EL0) initiates a system call requesting memory using `malloc()`, which is used to allocate blocks of memory on the heap. The processor switches to Kernel mode (EL1). Adapted from [ARM17b].

Synchronous exception handling is especially relevant when applications try to invoke system calls, which are a programmatic way that applications request services from the kernel of an OS with well-defined, safe operations. In a native system, the OS would request a processor mode switch directly from the processor and schedule these instructions to execute. In a virtualized system, system calls require hypervisor intervention. The hypervisor invokes an exception handler that decides how to process or emulate the instructions. System calls can be roughly grouped into six categories: process control, file management, device management, information maintenance, communication, and protection.

When an application executing in EL0 needs to request services from the OS in a native system, it performs a supervisor call (SVC). The OS then requests a processor mode switch from the secure monitor using a secure monitor call (SMC). In a virtualized

system, the guest OS cannot directly request a processor mode switch from the secure monitor. The guest OS must first address the hypervisor using a hypervisor call (HVC), which then invokes an SMC, see Figure 2.8. The required redirection to the hypervisor in a virtualized system results in execution time overhead compared to native execution.

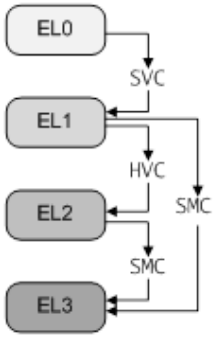


Figure 2.8: In a native system, applications executing in EL0 request services from the OS using a supervisor call (SVC). The OS can then directly request services from the secure monitor using a secure monitor call (SMC). In a virtualized system the OS must first request services from the hypervisor using a hypervisor call (HVC) which then addresses the secure monitor using an SMC. Adapted from [ARM17b].

### 2.3.4 Interrupt Controller Management

In contrast to the previously discussed synchronous exceptions, asynchronous exceptions come from external sources such as sensors or keyboards. These exceptions are known as interrupts and typically use a signal to the processor emitted by hardware or software, indicating an event that needs immediate attention. The processor responds by suspending the executing task, saving task-related data, and transferring control to the interrupt controller. The interrupt controller interprets the interrupt to determine what actions should be taken and calls the corresponding interrupt service routine (ISR). Once the ISR has finished executing, the processor clears the interrupt hardware for the next interrupt and determines which task should continue executing. Once a task is chosen, the processor retrieves the selected task information and transfers control to the task.

There are many different types of interrupts, and each has an associated priority level allowing the interrupt controller to rank them. In virtualized systems, there can be multiple external sources of interrupts that can have different priority levels for different VMs. Interrupt handling is further complicated when an interrupt needs to be handled by software within a VM. The targeted VM may not be running on a core when the interrupt is received and thus would require the hypervisor to perform a context switch before handling the interrupt. Other interrupts may need to be handled directly by the hypervisor [ARM13]. The hypervisor plays a crucial role in managing interrupts and can be the source of significant overhead if not appropriately handled.

To aid virtualization, ARMv8 processors have virtualization extensions such as virtual interrupts and the Generic Interrupt Controller (GIC). The GIC provides registers for managing interrupt sources, interrupt behavior, and interrupt routing to one or more processors [ARM16] and enables interrupts to be handled more effectively within a specific VM according to the source of the exception and execution state. See [ARM13] for more information regarding ARMv8 interrupt handling.

### 2.3.5 Scheduling and Core Management

The OS scheduler is responsible for deciding on which CPU a task will run and for how long. An OS inside a VM still schedules tasks but on the VM's virtual CPUs. The hypervisor schedules the ready tasks from all the VMs onto the physical CPUs. The two layers of scheduling policies, one from the guest OS and one from the hypervisor, is known as the double-scheduling property of virtualized systems.

Scheduling is a primary function of the hypervisor and can have significant performance consequences. The scheduling policy of a hypervisor has to take into account the processor architecture, such as if the CPU is a single-core or a multi-core processor. On a single-core processor, there is only one unit to perform work, so the hypervisor and VMs need to share the core. Multi-core systems can have several cores that are capable of independent instruction execution facilitating faster execution in comparison with a single-core processor. Multi-core processors can also permit different execution configurations for virtualized systems. One or more cores can be dedicated to a single VM or dynamically assigned according to the needs of each VM.

### 2.3.6 Device Emulation and Assignment

One of the most significant impacts on security and efficiency in virtualized systems is the approach for managing I/O across the VMs. The traditional method for I/O virtualization is emulation [KK13]. Device emulation requires that all guest OS accesses to a I/O device must be intercepted, validated, and translated into hypervisor operations. This method ensures that if a VM fails, other VMs are still able to use the physical I/O device [KK13].

However, device emulation is expensive since all guest accesses to the device have to be trapped and emulated in software [ARM17a]. Instead, a hypervisor can assign devices to individual guests so that the guest operates the device without or with minimal hypervisor intervention, known as the pass-through model [KK13]. The pass-through model often still requires the hypervisor to hide from the guest that the device is at a different physical address. This requires that the hypervisor handles interrupts as the interrupt ID is different from what the guest is expecting [ARM17a]. Hardware extensions such as transparent stage 2 mapping and interrupt virtualization can circumvent these challenges [ARM17a].

A hypervisor can also assign a device to a single VM, which then provides a virtual I/O interface to other VMs, enabling access to the device [KK13]. This method has the downside that improper access by a guest can bring down other guests, applications, or

even the entire system [KK13]. It also violates the primary security concept of isolation through virtualization.

### 2.3.7 Memory Virtualization and Management

There are different levels of memory that can be broken down into primary and secondary storage. Primary storage consists of memory directly available to the CPU, such as the random-access memory (RAM), processor registers, and processor caches. Secondary storage is not directly accessible to the CPU and must be accessed over I/O channels adding processing time overhead for each read/write. For this reason, primary storage is the preferred medium during computing.

Memory management is the process of controlling and coordinating physical computer memory to various running processes and abstracting the different levels of memory into a contiguous address space. In a native system, memory management is performed at three levels:

- The hardware-level involves components that physically access and store data, such as random-access memory (RAM) chips, memory caches, and flash-based solid-state drives.
- The OS-level involves the allocation and reallocation of continuous memory blocks, called pages, to individual programs as demand changes. Most OS's allow the utilization of more primary storage than is available by using virtual memory, called overallocation. The OS is responsible for the maintenance of page tables, the data structure that saves the mapping from physical to virtual addresses.
- At the application level, memory management ensures the availability of adequate memory for the objects and data structures of each program, which translates into two separate tasks: allocation and recycling.

In a virtualized system, the hypervisor is responsible for managing all physical memory, including dedicating a part of the physical memory for its own use and partitioning the rest to be assigned to the VMs. The hypervisor is also responsible for enforcing protection and redirecting access faults from VMs. Hardware virtualization introduces an extra layer of memory mapping so that each guest OS is responsible for mapping its virtual addresses to intermediate-physical addresses while each VMM maps the intermediate-physical addresses to physical addresses, see Figure 2.10.

The software-based approach to handle the two-stage address translations required by memory virtualization relies on shadow page tables. Shadow page tables are a composite set of page tables that rely on heuristics and memory tracing to keep track of changes to page tables in memory [BNT17]. This process can have significant performance overhead due to the hypervisor having to maintain synchronization of the shadow tables with the guest page tables [VMw19]. Workloads that involve a small amount of page table activity, such as process creation, memory mapping, and context switches do not incur significant overhead [VMw19].

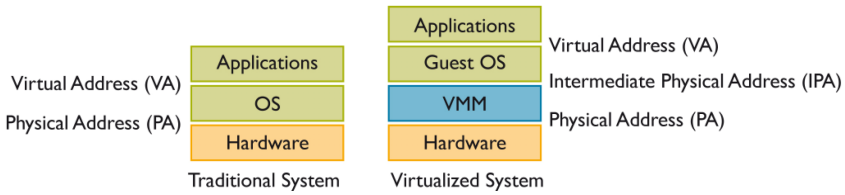


Figure 2.9: Address Translation in a traditional and virtualized system [MN10]

Hardware-based memory virtualization eliminates the overhead associated with software-managed shadow page tables through second-level address translation (SLAT), also known as nested paging. SLAT enables VMs to manage their own virtual memory structures without trapping to the hypervisor while still enabling the hypervisor to control the physical memory resources [Dal18]. The first layer of page tables stores guest virtual-to-physical translations, while the second layer of page tables stores guest physical-to-machine translation. The translation look-aside buffer (TLB) is a cache of translations maintained by the processor's memory management unit (MMU) [VMw19]. The first layer of page tables is maintained directly by the guest OS while the VMM maintains the second layer of page tables [VMw19].

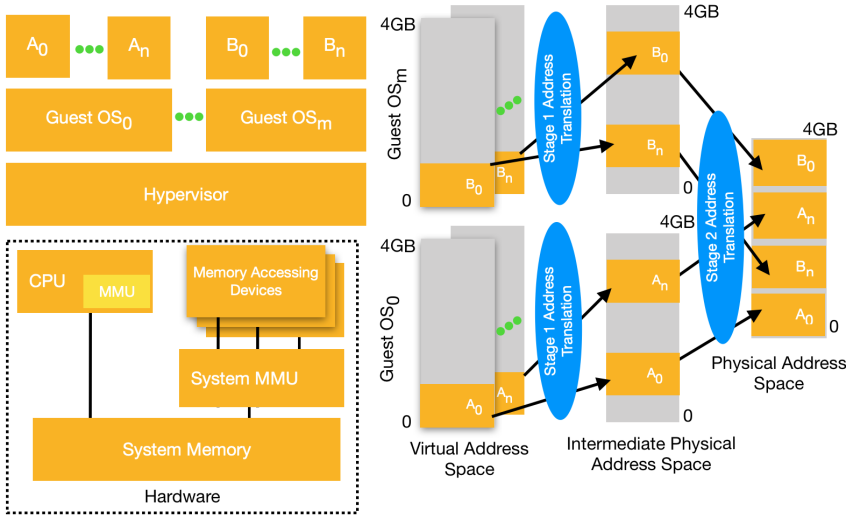


Figure 2.10: Two layer address translation in a virtualized system, adapted from [MN10].

When using hardware-assisted memory virtualization, a TLB miss can lead to significant overhead [VMw19]. Large pages in both the guest virtual to guest physical and guest physical to machine address translations can reduce the cost of TLB misses [VMw18b]. Workloads with a large amount of page table activity benefit the most from hardware-based memory virtualization [VMw18b].

There are many different techniques that a hypervisor can implement to improve memory usage, such as:

- Memory page sharing, also known as transparent page sharing, eliminates redundant copies of memory pages and is based on the observation that several VMs may be running instances of the same guest OS with common applications or components. The amount of memory saved by memory sharing depends on whether the workload consists of nearly identical machines [VMw18b].

- Memory ballooning is a reclamation technique that allows the hypervisor to retrieve unused memory from a VM and reallocate it to another VM [VMw18b].

- Memory compression occurs when a VM approaches the level at which host-level swapping will be required. The hypervisor will compress the memory to reduce the number of memory pages needed to swap out. Compression takes place in memory, whereas swapping takes place within the storage system, which is much slower than memory. Therefore memory compression results in significantly lower latency [VMw18b].

- Swap-to-host cache only occurs when the previous techniques are exhausted. The hypervisor cache is configured to be on SSD storage, which is much faster than regular swap files, which are typically on hard disk storage [VMw18b].

- Regular host-level swapping occurs when the host cache is full or not configured. The hypervisor will reclaim memory from the VM by swapping out pages to a regular swap file, some of which may still be active. This technique is typically associated with the highest performance degradation due to high access latency [VMw18b].

Refer to [Gan16] for a more detailed explanation of memory virtualization techniques and performance implications and to [VMw19] for ESXi [1] specific techniques.

### 2.3.8 Energy Management

Energy management is another crucial aspect of virtualization, particularly within the domains of battery-powered and heat-sensitive devices. Energy management entails the consumption of energy and the heat produced during execution. Execution speeds, energy consumption, and heat production are closely correlated, so optimizing one can negatively impact another.

A hypervisor needs to implement an effective energy management policy and must be able to consolidate different policies from guest OSs. Power management is typically a "private matter" of each OS, with different strategies to dynamically manage power consumption, such as varying the voltage, frequency, or the number of active cores. The hypervisor has to intercept hardware accesses from the guest OS power management driver and amalgamate different policies for the most favorable performance.

Power management policies are closely related to the processor version. For example, ARMv8 processors have several levels of power management, such as standby, retention,

[1] Virtualization techniques may differ in ESXi-ARM as it is a prototype based on ESXi for x86.

power down, dormant mode, and hot-plug [ARM13]. There are also different power state coordination architectures that enable the creation of power domains that are arranged in a logical hierarchy [ARM16]. There are two approaches to power management that hypervisors can use to control the power state of ARM cores. Physical OS power management, which comprises the software components that select the physical power states and virtual OS power management, which selects virtual rather than physical power states [ARM16].

## 2.4 Related Work

There are many more virtualization domains that have not been covered in this introduction. For information regarding operating system, network, application, server, storage, desktop and service virtualization, refer to [BKL10], which also discusses forensic techniques of virtualization. For an in-depth introduction to hardware and software virtualization, including hardware extensions of x86-64 and ARM, refer to [BNT17]. An evaluation of virtualizing ARMv7 and ARMv8 architecture can be found in [Dal18], including why ARM architecture does not meet the classical requirements for virtualization. Both [Dal18] and [DLLN15] compare performance of KVM and XEN on the x86 and ARM processors.

## 2.5 Summary

This chapter introduced the fundamental concepts of virtualization. In particular, the three basic virtualization techniques of multiplexing, aggregation, and emulation were presented and type 1 and type 2 hypervisor architectures discussed. Popek and Goldberg's requirements of virtual machines were explored and their theorem regarding classically virtualizable ISAs. This was followed by the three techniques to enable virtualization of not classically virtualizable ISAs through the use of Binary Translation, Paravirtualization, and Hardware-Assisted virtualization. Finally, functions of a hypervisor were discussed with consideration to overhead that each function adds into a virtualized system.

In chapter 3, the virtualization of embedded systems is explored. Particular consideration is given to the requirements, challenges, and use cases of an embedded hypervisor.

# 3 Virtualization for Embedded Devices

Virtualization basics were introduced in chapter 2, with a focus on virtualizing ARMv8-A architectures. In this chapter, virtualization of embedded devices is examined. In section 3.1, embedded devices are defined and an overview of the different embedded systems domains is given. In section 3.2, the design requirements of embedded systems are established, which closely relate to section 3.3, the requirements and challenges of an embedded hypervisors. Finally, specific use cases of virtualization for embedded devices are considered in 3.4.

## 3.1 What Are Embedded Devices?

Embedded devices are dedicated functionality devices that have historically relied on a single microprocessor or microcontroller to serve a single purpose in a larger system composed of hardware and software. These devices evolved out of a need for technology that could provide utility for a combination of cheap, lightweight, and low power consumption. Today, embedded devices build the foundation of a diverse spectrum of electronics, known as embedded systems, which vary widely on their functionality, purpose, and their level of interactions with humans and the environment. For example, cars rely on hundreds of simple embedded devices known as electronic control units (ECU) to support their expanding functionality, such as an anti-lock braking system, navigational system, and fuel injection pump. ECUs rely on sensors and actuators to accomplish their purpose. In contrast to ECUs, mobile devices, such as tablets and smartphones, have evolved to be similar to general-purpose devices, such as personal computers. Mobile phones rely on interacting with external networks (cellular, WiFi, etc.) for their functionality. An overview of the embedded systems market is seen in 3.1.

## 3.2 Embedded System Design Requirements

In order to be competitive in the market place, the entire embedded system (hardware and software) needs to be taking into account when making design decisions [Jr96]. Decisions to fulfill a system requirement can lead to a negative impact on other features and thus need to be weighed on a system-by-system basis. The utility of the end product, along with the size, weight, power, and cost (SWaP-C) are all major design driving factors. An overview of an embedded system architecture is shown in 3.3, adapted from [Jr96].

Table 3.1: IDC's Embedded systems market model device segmentation.

| Industry | Sub Industry Category | Device Category |
|---|---|---|
| Automotive | Automotive | Automotive control devices, infotainment devices, transportation telematics |
| Industrial | Aerospace and Defense | Avionics |
| Industrial | Industrial Automation | Manufacturing and process controls, motion controllers, operator interfaces, robotics, HVAC and other controls |
| Industrial | Services | Kiosks, POS, video surveillance, test and measurement |
| Industrial | Other | Industrial PC, handheld terminal, other |
| Health Care | Medical equipment | Proactive and consumers, diagnostic and monitoring, imaging, therapeutics |
| Energy | Consumption point | Smart metering, home/building automation |
| Energy | Renewable energy | Inverters controls, intelligent switches and controls, wind turbine control and communication systems |
| Energy | Electricity T&D | Distribution automation command and controls, fault detection and isolation |
| Communications | Broadband Access | Broadband access |
| Communications | Phones | Cellular phones, cordless phones |
| Communications | Network | Ethernet, infrastructure, network security, routers, optical, ATM multi-service switch |
| Communications | Services | Videoconference, VoIP |
| Communications | Wireless Infrastructure | Wireless infrastructure |
| Communications | Other | Communication other |
| Consumer | Consumer | Digital camera, digital TV, digital video devices, gaming, GPS, mobile internet devices, portable media devices, projector, consumer other |

Source: IDC and Final Study Report: Design of Future Embedded Systems (SMART 2009/0063) [IDC14]

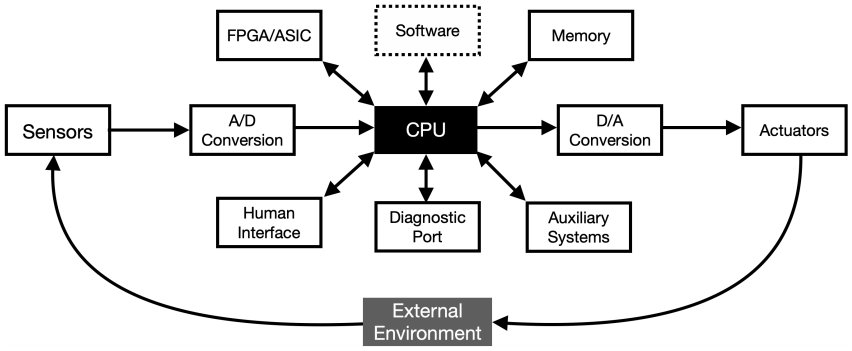


Figure 3.1: Example architecture of a possible embedded system, adapted from [Jr96]

### 3.2.1 Criticality

Embedded devices play integral roles in systems and are often categorized by the outcome of failure. This is often referred to as the *criticality of a system*, which can be distinguished by the function and associated consequence of failure.

- **Safety critical** systems may lead to a loss of life, serious personal injury, or damage to the environment in case of a failure. An example is the anti-lock braking sensors in a car.

- **Mission critical** systems result in an inability to complete the objectives for which the system was designed. An example is a navigational system used for landing a spacecraft.

- **Business critical** systems may result in significant economic costs. An example is any device that could lead to the failure of the automation system in a manufacturing plant.

- **Security critical** systems deal with the loss of sensitive data through theft or loss. An example is exposure of passwords on a mobile phone.

### 3.2.2 Real Time

Additionally, the failure of a system can have a different meaning in the context of real-time constraints. A real-time system is one in which the correctness depends not only on logical results but also on the timeliness, predictability, and dependability of those results [Ker11, HGC97]. These devices have tolerances of missed deadlines categorized into hard, firm, or soft real-time.

- **Hard real-time** any missed deadline is a system failure, often used with mission-critical systems. For example, the spacecraft Mars Pathfinder was nearly lost when a high priority task was not completed on time due to being blocked by a lower priority task, causing a system restart. The problem was corrected, and the spacecraft landed successfully.

- **Firm real-time** allows for infrequent, adequately spaced, missed deadlines, but the value of the task is zero after the missed deadline. For example, video conferencing applications send video and audio as packets over a network. Since the frames are time-order sensitive, a missed deadline causes jitter, diminishing Quality of Service (QoS). Late frames are disregarded as it would cause more jitter to display them. However, as long as jitter does not occur too often, users can still hear and see each other.

- **Soft real-time** allows for missed deadlines, and tasks continue to have value after the deadline as long as tasks are executed in a timely manner. For example, after taking a photo on a digital camera, it may take several seconds before the photo is processed and can be displayed on the screen. As long as all tasks involved with the image processing are completed relatively on time, the delay will have no critical effect.

## 3.3 Requirements and Challenges of an Embedded Hypervisor

Embedded hypervisors are type 1 hypervisors that are typically designed into the hardware device [BKL10]. As discussed in section 3.1 and 3.2, embedded systems encompass a large array of devices that all have different requirements making a one-size-fits-all hypervisor difficult. In this section, the requirements and challenges of embedded hypervisors are identified.

### 3.3.1 Efficiency

All hypervisors strive to be efficient, but embedded hypervisors must work under the additional resource constraints imposed on embedded systems due to the SWaP-c requirements. Fewer instructions to accomplish a task results in less power consumed. Power consumption is often a bottleneck for battery-powered devices, warranting hypervisors for battery-powered devices to add minimal overhead. This entails minimizing hypervisor intervention during VM execution and a need to consider power-consumption and performance trade-off [LX15]. Power consumption also has a direct correlation to heat-production, important for systems that are heat-sensitive, such as applications involving combustion in transportation [Jr96].

Most embedded devices are also memory constrained, requiring embedded hypervisors to be small and use memory efficiently. A small codebase has additional advantages, like simplifying Validation and Verification (V&V). V&V leads to a more reliable, stable, secure platform, essential for hypervisors that run in kernel mode and thus are part of the Trusted Computing Base (TCB). These properties are especially important in safety-critical and security-critical systems that go through rigorous certification processes [Xia16]. One way of minimizing the TCB of an embedded hypervisor is by using virtualization extensions provided by hardware. Virtualization extensions are hardware-specific, so for an embedded hypervisor to fully integrate, the hardware needs to be specified before development, decreasing OEM flexibility, or the hypervisor needs to be retrofitted to the hardware, increasing costs. Hardware extensions also play a significant role in the virtualization overhead of a hypervisor. Tasks that can be offloaded from the hypervisor to the CPU through hardware extensions decrease the CPU cycles spent outside of completing VM tasks.

### 3.3.2 Secure Communication and Isolation

Security in traditional server and desktop virtualization is achieved through isolation. However, most embedded devices are concerned with a different communication style from the server or personal computer domains. Whereas servers and personal computers are primarily concerned with inter-communication, between systems, embedded devices usually provide a specific function of a larger system and thus require intra-communication, within the system, to complete a function.

Additionally, embedded devices have real-time needs and are typically equipped with less powerful processors, creating a bottleneck and performance decline with bulk data transfers. An example of this is seen in Android's virtualization of the network interface baseband processor, responsible for processing all radio functions in smartphones. The mobile phone OS and the baseband processor code are run in separate VMs on a single processor. Data received by the baseband VM, such as video, needs to be shared with applications running in other VMs, such as the media player. Bulk data transfers, copying the memory between VMs, necessitates considerable processing time and energy, which results in a decline in QoS. Efficient inter-VM communication breaks the isolation principle since efficient energy-conserving bulk data transfers usually necessitate a shared

buffer [Hei07].

Additionally, most hypervisors are retrofitted with inter-VM communication facilities, leading to significant protocol overhead when communicating with other VMs or the outside world [KCSS11]. This is further complicated by security-critical devices, such as mobile phones that save passwords and bank account access codes. Security-critical data must be protected in case of loss or malicious behavior. This necessitates that an embedded hypervisor enables communication with the Principle of Least Authority (POLA) while minimizing processing time overhead [Hei07]. Embedded devices may also have to share data without a centralized host, such as car ECUs, which use a controller area network (CAN) bus.

### 3.3.3 Real-time Capabilities

Since most embedded systems have real-time constraints, a hypervisor must ensure a worst-case execution time (WCET) to function properly and safely [Hei07, EES+01]. The WCET does not necessarily entail the execution time to be fast, but rather, that the execution time has to be predictable, especially when the system is under peak load [Xia16, HGC97].

Additionally, a hypervisor must ensure that VMs that have different and sometimes conflicting timing requirements coexist safely and cooperatively. Temporal behavior of unrelated VMs sharing resources such as CPU, network, etc., must not interfere with one another [Ker11]. This is complicated since a virtualized system has a two-level hierarchy of schedulers, see 2.3.5. Each guest OS has a scheduler that selects a task to execute according to its policy with no insight into the importance of the task in relation to the other VMs [Hei07, KCSS11]. The hypervisor then schedules threads from individual VMs with no insight over what the task being scheduled is.

There are several methods for embedded hypervisors to try to adhere to VM timing constraints. One approach is through the association of a priority level with each VM; however, this can lead to a low-priority task from a high-priority VM blocking a high-priority task from a low-priority VM [Xia16, Hei07].

Another technique is through implementing bandwidth reservation schemes for CPU time, but this can lead to severe lag and jitter [Xia16]. Another approach is by using a multicore processor and assigning cores to individual VMs. However, this may result in underutilization of the CPU. Better utilization of multicore processors can be achieved by enabling hyperthreading and sharing cores between VMs that complement each other's real-time needs. Nevertheless, this complicates spatial isolation since there are multiple levels of cache with no guarantee where data may reside and which VMs have access to the data [BMB+18].

Apart from the disadvantages mentioned for the techniques available to a hypervisor to adhere to timing constraints, the biggest problem is that none of them can guarantee a WCET [EES+01]. Large systems often have too much hardware complexity and program state spaces to exhaustively explore all possible executions of a program [EES+01, Kel15]. The result is that measured execution times often underestimate the WCET and increases the chances of a time-related failure [EES+01].

#### Summary

There are further constraints that embedded devices face not discussed here, such as that embedded devices are usually non-configurable after manufacturing. However, most requirements for an embedded hypervisor can be summarized as [Hei07, LX15]:

- **Lightweight hypervisor** code base to minimize TCB and increase security and safety.
- **Strong encapsulation** between subcomponents for fault containment and security isolation.
- **Controlled communication** for low-latency, high-bandwidth information flow control between VMs.
- **Fine-grained encapsulation** of individual threads to impose a global scheduling policy that respects real-time constraints.
- **Minimal overhead** on system resources and performance.

## 3.4 Virtualization Use Cases for Embedded Devices

Virtualization offers viable solutions to some of the problems faced by embedded devices. However, as discussed in Section 3.3, a one-size-fits-all hypervisor is difficult due to the complexity and diversity of embedded devices. Individual use-cases for virtualization are identified in the following and the benefits of each.

### 3.4.1 Coexisting Operating Systems

Concurrently running multiple OSs on a device can lead to benefits such as hardware consolidation, real-time support, development simplification, and support for legacy systems. Supporting more than one independent user on a device can be achieved by virtualization and reduces costs through hardware consolidation. Even devices that only support one user can benefit from hosting more than one environment on a device.

#### Use Case 1: BYOD Devices

The concept of Bring-Your-Own-Device (BYOD) has recently gained traction in the enterprise domain, where employees typically need a personal and a separate business phone or tablet. BYOD offers separate environments on the same device, maintaining the separation of personal and enterprise data [SGB+16]. This reduces costs through hardware consolidation and enables the enterprise IT department to better control business data without accessing personal data.

#### Use Case 2: Decouple Application from Environment

High-level application programming interfaces (APIs) can be used to specify the interface between an application and the OS. APIs are typically OS dependent, so an application programmed for Linux would need to be changed to function in Apple's iOS. A hypervisor is able to support both environments in separate VMs, decoupling the application from the device. For example, a virtualized mobile phone can host both iOS and Android environments, enabling users to install OS-specific applications in the corresponding environment. This leads to application development simplification since applications are no longer bound to the device environment provided.

This use-case could also enable application developers to ship their applications with an OS image. Application developers would have a well-defined OS environment and would lead to reduced device failure due to configuration mismatch [Hei08]. Automatic elimination of replicated page content could be used to reduce memory usage [LUSG04].

#### Use Case 3: Vanilla OS Support

Virtualization can also be used to provide standard bloat-free OS environments. Vanilla Android is the most basic version of Android developed by Google. Android is open-source, allowing manufacturers to customize it with additional features and their own user interface. However, phones running the non-standard Android typically suffer from delayed updates since the manufacturer has to change updates to suit their version. This can shorten the lifespan of devices when the manufacturer no longer supports the device by providing updates. Running Vanilla Android in a VM and manufacturer specific code in a separate VM has the potential to provide better, faster updates and greener IT since devices can be supported longer.

#### Use Case 4: Legacy Application Support

Virtualization can also form the basis of legacy application support by enabling legacy software to operate in a suitable VM environment. Legacy support does not just reduce waste; it can also save money. Some applications have to go through costly and time-consuming recertification processes when upgraded. System hardware or software changes can result in the application no longer being supported without upgrading. Virtualization can form the basis of legacy application support by decoupling the hardware and software layers and dividing large systems into subsystems [Hei08]. A legacy application can be run on an older version of the OS in a VM while providing a user with all the functionalities of the new OS run a separate VM [Hei08]. This is especially applicable to industries with long certification processes such as automotive, avionics, or government work [Xia16]. Wrapping legacy software was also the basis of work done by [LUSG04], where legacy device drivers were isolated in VMs instead of integrated into the new OS.

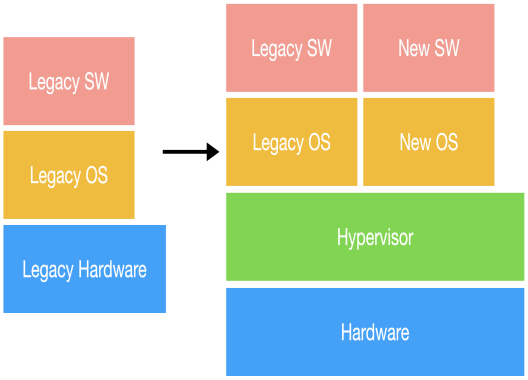


Figure 3.2: Legacy software support

#### Use Case 5: Real-Time Support

Many embedded devices require real-time responsiveness with low and predictable interrupt latency, which most mainstream OSs are not optimized for [Hei08]. A real-time operating system (RTOS) can be run alongside a general-purpose OS, proving an environment for applications that require real-time responses while maintaining the functionality provided by the general-purpose OS.

This use-case is implemented in some Android phones to reduce cost by consolidating the main processor and baseband processor [Hei11] [1]. The baseband processor is the network interface for radio functions for smartphones, GPS, or Bluetooth devices. Typically, the baseband processor is separate from the main processor because of real-time needs, certification requirements, and reliability concerns [KCSS11]. The baseband processor code is usually proprietary making it difficult to perform independent audits and raise security concerns. Virtualization enables the real-time needs to be satisfied, enables changes to the general OS without interfering with the radio functions, and enables separate certification of baseband code and OS. The use of a hypervisor also increases the security of the system through isolation of the subsystems. If vulnerabilities exist in the baseband processor code, the general OS data is protected by the hypervisor.

### 3.4.2 Dynamic Resource Management

Dynamic resource management entails the allocation and reallocation of physical resources to VMs. This can also include modules for monitoring and detecting the resource usage in VM for specified time periods in order to allocate the resource more suitably. This can lead more efficient use of hardware and less hardware required for a system. Dynamic resource allocation also increases the scalability, flexibility, and reliability of a system.

[1] Although implemented in some Android phones in early 2010; this use case is relevant for devices with very restrictive budgets and low network bandwidth.

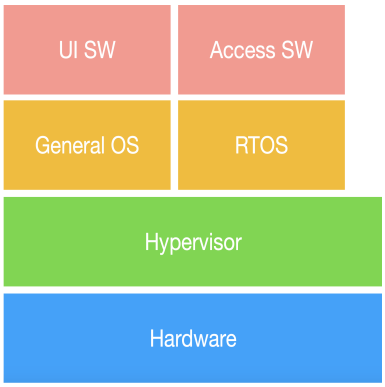


Figure 3.3: RTOS coexisting with general-purpose OS.

### Use Case 6: Multicore Management

As multicore processors are becoming cheaper to produce, they are becoming mainstream in the embedded systems market [Kel15]. Legacy software built for unicore processors is often not programmed to take advantage of multicore processors. A hypervisor can form the basis of a poorly scaling legacy software on multicore processors by performing analysis of the threads and scheduling them efficiently on the available cores.

A hypervisor can also dynamically add cores to an application domain, which requires extra processing power or can manage power consumption by removing processors from domains and shutting down idle cores [Hei08]. Fine-grained scheduling allows core cycles to be assigned to different VMs, enabling multiple VMs to share a core. This decoupling of the hardware and software also enables additional processing configurations, not just symmetric and asymmetric [Hei08]. This has the potential of increasing performance while lowering power consumption [Hei08].

However, there is an inherent security issue associated with certain multicore management techniques. Typical processors have several layers of cache. Each core has its own L1 data and instruction cache and shares L2 and L3 caches with other cores. Due to their hierarchical nature, complicated replacement policies, and quick turnover, it is difficult to know exactly where data is stored in the cache and to ensure isolation of data between VMs [CBS+18].

### Use Case 7: Hot Failover

Dynamic resource allocation can also increase the reliability of an embedded system [Jr96]. Downtime due to failure, can have devastating effects for many safety-critical and mission-critical applications. Flight control failure can lead to personal injury, while factory automation control failure can lead to downtown and costs. Some embedded devices cannot be shut down for maintenance due to safety reasons, such as reactor control systems, or due to being inaccessible, such as space systems. A common approach to reducing downtime is using redundant hardware, which directly increases costs [Jr96].

However, a substantial amount of system failures are due to a software error, often freezing the system and requiring a reboot. The traditional approach to reducing downtown in such a case is to use redundant processors where one is active, and at least one other is in standby. This approach adds overhead to maintain constant synchronization between the active and standby processors so that the standby processor can cleanly take control and keep the system alive if the active processor is unavailable [Jr96].

Most embedded devices are produced in competitive markets and have narrow profit margins. These devices cannot tolerate the costs of redundant hardware or additional processing capacity required for traditional fault tolerance techniques [Jr96]. Virtualization allows the active and standby systems to be set up as redundant VMs with redundant processing domains, enabling hot-failover.

#### Use Case 8: Migration

Another method of increasing reliability is through the use of migration. Hypervisors are able to automate the environment migration process while decreasing the downtime of a system by using a combination of migration techniques. Rather than having to install the software on the hardware, the entire VM can be copied from one domain to another regardless of the underlying hardware. This allows systems that cannot suspend activity, such as mission-critical systems, to be seamlessly migrated. This leads to the added benefit of decoupling the environment from the hardware leading to easy exchange from phones or other devices after hardware failure, device loss, etc.

#### Use Case 9: Dynamic I/O Management

Virtualization can also be used to manage I/O resources dynamically. VMs can be set to share a single physical I/O device without being aware that they do not have sole possession of the device. Virtualization can also enable dedicated I/O devices to be unplugged or hot-plugged to VMs without necessitating a restart. Especially embedded devices that need to communicate with the outside world, such as IoT devices, would benefit from dynamically assigning network cards [ASL+19].

### 3.4.3 Sandboxing

Security critical embedded devices are especially sensitive to malicious behavior. Sandboxing is the process of using software to isolate applications from critical system resources and other programs. This can be used for any applications that poses a security risk to a system to increase security.

#### Use Case 10: Operating System Isolation

Virtualization can be utilized to increase the security of an OS since a VM encapsulates the environment. If the standard OS is compromised, the rest of the system is still protected by the hypervisor. According to [Hei08], a hypervisor must adhere to two conditions to increase the security of an OS. First, the hypervisor must be much smaller

than the OS; otherwise, the hypervisor increases the size of the trust computing base (TCB) and is counter-productive for security. Secondly, the hypervisor must support the segregation of critical functionality into VMs different from exposed user- or network-facing ones.

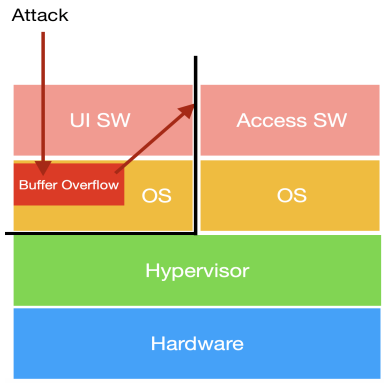


Figure 3.4: A user or network facing OS is compromised but the rest of the system is safe from exploit, adapted from [Hei08].

### Use Case 11: Monitor Software Isolation

Another potential source of attack are virus scanners themselves. To prevent this, a virus scanner can be installed in its own VM, which gets a read-only copy of the memory. The virus scanner is encapsulated so it cannot leak any data and is protected from network and user components which are at a high risk to be compromised.

### Use Case 12: Licensing Separation

Sandboxing can also be used for licensing separation. Linux is frequently used as a high-level OS due to its royalty-free status, independence from specific vendors, widespread deployment and strong developer community [Hei07]. However, linux is distributed under a GPL license, requiring derived code to be open source [Hei07]. There are discussions that this applies to device drivers that are loaded as binaries at run time into the kernel [Hei07]. Virtualization can be used to provide a proprietary execution environment for software that shares the processor with a Linux environment. A proxy driver is used to forward Linux driver requests to the real device driver using hypercalls [Hei07]. Sandboxing can also be used to protect licensed media content from copying. Embedded systems have a diversity of micro-controllers with different ISAs so when exchanging a micro-controller, the embedded system software must be ported to run on a different ISA [Ker11]. This is problematic if the source code is unavailable or inaccessible due to licensing restrictions. Through virtualization, software can run on a non-supported ISA through the usage of emulation [Ker11].

#### Use Case 13: Fault Containment

Even applications that were not created with malicious intent can cause devastating errors in a system. A hypervisor can encapsulate subsystems to keep faults contained and unable to interfere with other subcomponents.

## 3.5 Related Work

An overview of the embedded systems market was given in section 3.1. For a more detailed analysis of the market, refer to [IDC14], which evaluates the market of 2012 and predicts embedded system design trends until 2020. In [Fut19], the embedded systems market growth is evaluated with forecasts made until 2022.

Current design issues of embedded devices are evaluated in [Jr96]. Particularly important are the real-time needs of embedded systems. Fundamentals of real-time systems is given in [Ker11] and information to transforming real-time systems into virtualized real-time systems. Information regarding execution time analysis for embedded systems can be found in [EES+01]. Challenges associated with real-time virtualization are given by [ZG12, TSC14]. For information regarding the security aspects of virtualization, refer to the European Union Agency for Network and Information Security report [MJGB17].

Finally, virtualization motivation and use-cases for embedded device are discussed in section 3.4. For more information regarding use-cases for embedded device virtualization refer to [SGB+16, Hei11, Hei08]. A more in-depth analysis of use-cases is performed by [Hei07], with particular focus on the OKL4 microkernel hypervisor.

Performance comparison of microkernels L4 and OKL4 and hypervisors KVM and XEN on ARM was performed by [Xia16], which focuses on embedded real-time virtualization of reconfigurable platforms. Different virtualization platforms developed for ARM-based devices are summarized by [SGB+16]. Other ARM-based mobile virtualization platforms CodeZero, KVM-on-ARM and EmbeddedXEN are compared in [LX15]. MIPS and ARM-based Docker and LXC container virtualization solutions are compared in [NRLB18].

There are also several targeted commercial virtualization platforms available for a range of embedded systems. Cells [ADH+11] is a virtualization platform for mobile devices based on OS virtualization. Green Hills' Integrity hypervisor [Sof] is for high security systems certified by the EAL7 standard. Open-Source ARCN hypervisor is geared towards IoT development [LXRD19].

## 3.6 Summary

Embedded systems encompass a diverse array of devices that include both the hardware and software aspect. Not only are these devices constrained by design factors such as size, weight, cost, and power requirements but they also face varying challenges such as real time needs, safety, and security. This has led to design requirements of an embedded hypervisor that is efficient, lightweight, provides strong and fine-grained encapsulation,

and controlled communication. An embedded hypervisor offers solutions to problems faced by embedded devices by providing co-existing OSs, dynamic resource management, and isolation through sandboxing.

In the following chapters, a prototype version of VMware's ESXi hypervisor modified to run on ARM processors is evaluated as an embedded hypervisor. Chapter 4 details the research design and methodology and chapter 5 presents the results of this research.

# 4 Research Design and Methodology

In chapter 2, the architecture and role of a hypervisor is discussed. The performance of a virtualized system depends strongly on the design decisions of a hypervisor and the use of built-in virtualization support of processors. The performance of a virtualized system is measure as overhead, which is any combination of excess computation time, memory, bandwidth or other resources that are required to perform a specific task. In comparison to the personal computer or server space, embedded devices typically have decreased tolerance for overhead. In chapter 3, embedded system design criteria are defined with focus on what role an embedded hypervisor should ideally fulfill.

The objective in the rest of this paper is to determine the feasibility of an embedded hypervisor through the evaluation of a prototype version of the ESXi-ARM hypervisor. The methodology for testing is detailed in the section 4.1 while the rest of the chapter defines the hardware and software configurations of the testing environments.

## 4.1 Methodology

In a virtualized environment, overhead comes from the additional layers of processing required by the hypervisor and from sharing limited hardware resources. In complex virtualized systems, it is difficult to pinpoint the cause of performance degradation, which can result from specific tasks, such as context switching, or resource competition between VMs. Chapter 3 discusses many characteristics that an embedded hypervisor should adhere to. Not all these characteristics can be explored here so the memory footprint, scheduling latency, selected function latencies, and selected bandwidth metrics will be examined to give a starting point for future work. Additionally, domain specific applications will be examined to evaluate ESXi-ARM's performance in different embedded domains.

To measure the virtualization overhead of the ESXi-ARM hypervisor, different execution environments are used that compare native to virtualized execution and increasing instances of VMs, see Figure 4.1 for an overview. Environment 1 is used as a baseline of Debian executing natively on the bare metal MACCHIATObin development board. Environment 2 evaluates the performance of Debian executing in a VM on the ESXi-ARM hypervisor. Any deviations of the performance between environment 1 and environment 2 are due to virtualization, i.e. virtualization overhead. Environments 3 to 6 are evaluated to determine the increase in overhead from concurrently executing VMs on the ESXi-ARM hypervisor. The performance from concurrent execution primarily suffers due to resource contention such as when the hypervisor is busy performing tasks for one VM and unavailable for another VM. All benchmarks were run in the first VM while any

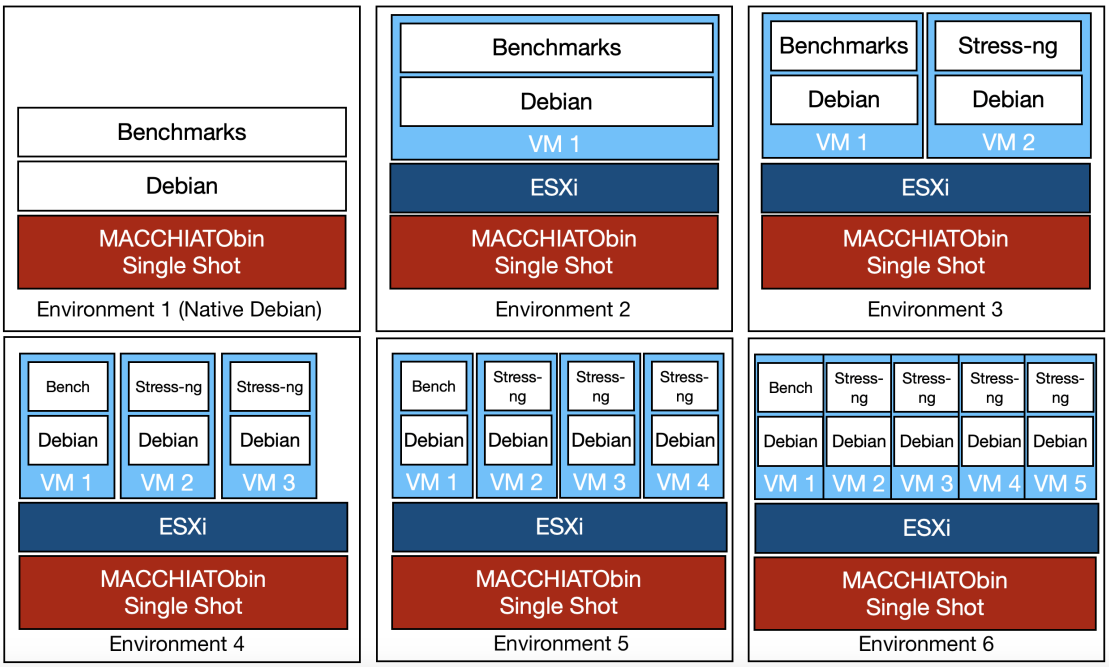


Figure 4.1: Testing environments

additional VMs hosted on ESXi-ARM were put under a simulated workload using the Stress-ng package, see [Fra18], with the command `stress-ng --cpu 4 --cpu-method matrixprod -timeout 60m`.

The memory footprint of ESXi-ARM and VMs was evaluated using the vSphere service console. The benchmarks were run directly after startup using a bash script found in 6.2. This script automates the testing process, first evaluating the boot time of the system and then running three different benchmark suites. First, the LMbench3 suite is run three times consecutively. The LMbench microbenchmarking suite [MS96, MS98] is used to evaluate basic operations of the OS such as system calls, memory access, and context switch time.

The script next runs 50 consecutive measurements of the synthetic benchmarking suite MiBench [GRE+01]. The MiBench suite is used to analyze the performance of domain specific algorithms to evaluate real-world performance. The categories and algorithms selected from each are as follows:

1. **Automotive and industrial control:** basicmath, bitcount, qsort, susan
2. **Consumer devices:** jpeg, lame, tiff [1], typeset
3. **Networking:** dijkstra, patricia

[1] Replaced with tiff_4.0.104.dsc for latest version check https://packages.debian.org/buster/libtiffdev.

4. **Office automation:** ghostscript [2], ispell [3], stringsearch

5. **Security:** blowfish, rijndael, sha

6. **Telecommunications:** ADPCM, CRC32, FFT/IFFT, GSM

The execution time of each MiBench algorithm is measured using the bash function `‘date + %s%N‘` and subtracting the execution start time from the end time.

Finally, the effects of virtualization on real-time performance are analyzed using cyclictest from the linux RT-tests benchmarking suite [ebu18]. Of interest here is the scheduling latency and WCET that the ESXi-ARM hypervisor adds to the system. Cyclictest is run at a priority of 90 with 10 million loops at an interval of 1000 or `./cyclictest -i1000 -l10000000 -h1500 -p90`.

Hardware was chosen to be comparable to embedded devices such as smart phones and tablets. An ARMv8-A processor was chosen since ARM is one of the largest market shareholders in the embedded market and this processor offers hardware extensions to support virtualization. Hardware and software configurations are unmodified as much as possible across benchmarking to negate effects from components not under testing. Excel was used to evaluate data and generate the graphs.

## 4.2 Hardware Configuration

Measurements were done on a Solid Run MACCHIATObin Single Shot development board [Mar19], equipped with a Marvell Armada 8040 SoC, see Figure 4.2. It is a cost effective and high performance single board computer aimed at the networking, storage, and connectivity space. The processor is an ARM quad-core 1.6 GHz Cortex-A72 processor, see section 4.2.1. It is important to note that the MACCHIATObin is a development board that was not tested for stability.

Onboard memory consists of a 4 GB RAM, 8 GB eMMC, and two 2.5” Intenso High Performance SSDs connected via SATA III. The native environment was installed on a 240 GB 2.5” Intenso High Performance SSD while the ESXi-ARM host and all VM environments shared a 960 GB 2.5” Intenso High Performance SSD.

### 4.2.1 Processor

The ARM Cortex-A72 is a low-power high-performance microarchitecture which implements the ARMv8-A64 instruction set, see Figure 4.3. ARMv8-A virtualization extensions include additional execution modes, a memory management unit (MMU), and a generic interrupt controller (GICv2m). Information related to ARMv8 ISA can be found in [ARM19].

[2] Replaced with ghostscript_9.27 dfsg2+deb10u3.dsc for latest version check https://packages.debian.org/buster/ghostscript.

[3] Replaced with ispell_3.4.006.dsc for latest version check https://packages.debian.org/buster/ispell

Table 4.1: Hardware configuration

| Component | MACCHIATObin Single Shot Rev is 1.3 |
|---|---|
| SoC | Marvell ARMADA 8040 |
| CPU | quad-core Cortex-A72 (up to 1.6 GHz) |
| Accelerators | Packet Processor |
| | Security Engine |
| | DMA Engine |
| | XOR engines for RAID 5/6 |
| Memory | 4 GB RAM |
| | 8 GB eMMC |
| | 240 GB 2.5” Intenso SSD via SATA (Native environment) |
| | 960 GB 2.5” Intenso SSD via SATA (Virtualized environments) |

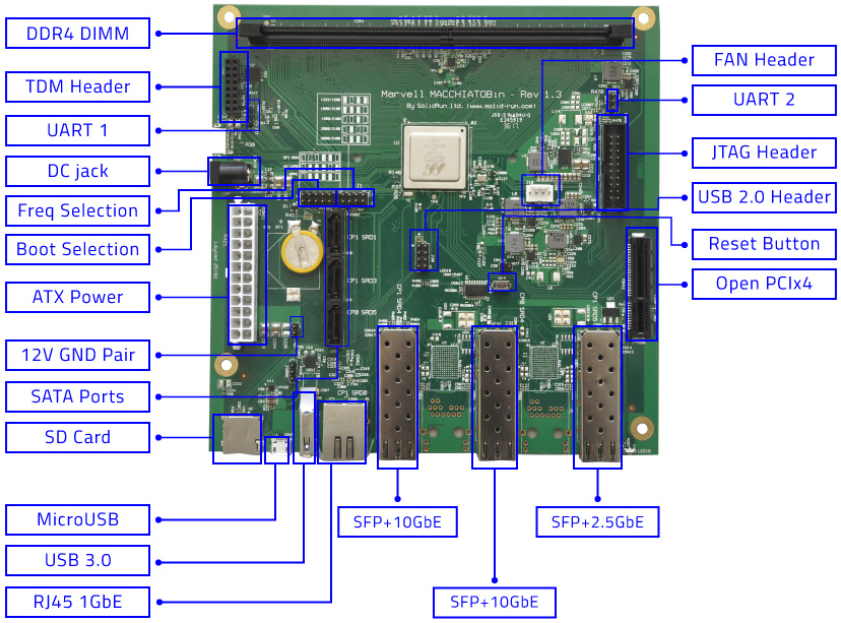


Figure 4.2: MACCHIATObin Single Shot development board, adapted from [Mar19]

### 4.2.2 Applicability

The MACCHIATObin Single Shot was chosen because of its comparable hardware to several embedded device domains. Most smart devices use six to eight core CPUs with a combination of high-performance cores, such as the Cortex-A72, and energy-efficient cores (prioritizes performance per watt), such as the Cortex-A53. The Apple A13 SoC appears in the iPhone 11 series and has two 2.65 GHz A72 cores and four A53 cores with 4 GB of RAM. Even low-end smartphones like the Huawei P smart released in 2017 has an Octa-core CPU with four 2.36 GHz Cortex-A53 cores and four 1.7 GHz Cortex-A53

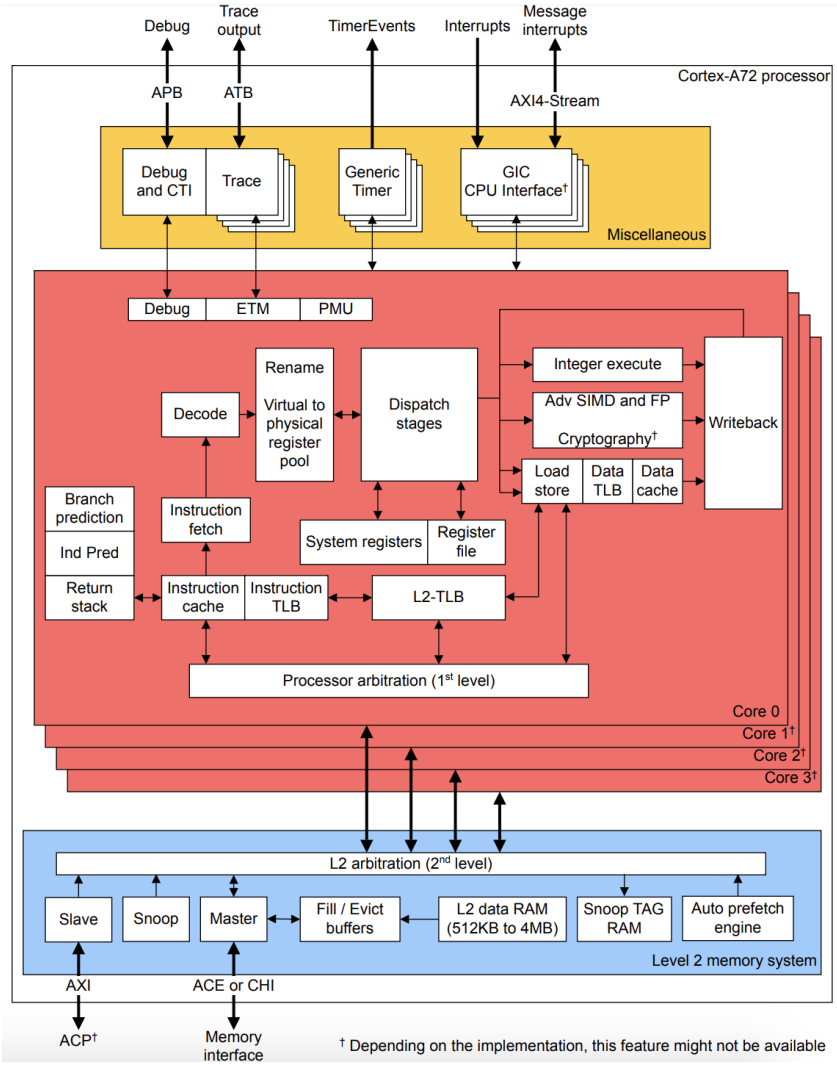


Figure 4.3: Block diagram of the Cortex A72 Processor from [ARM16]

cores with 3 GB of RAM.

Car infotainment ECUs typically use high-end ARM Cortex-A SoC processors and run applications such as Linux GENIVI. SoC from this category include Jacinto 6, developed by Texas Instrument with two ARM Cortex-A15 MPUs and two ARM Cortex-M4 processing subsystems and the Freescale i.MX6 based on ARM Cortex-A9.

Although the results of ESXi-ARM on the MACCHIATObin single shot will not translate exactly to all embedded domains, it offers a mid-scale performance environment that is a starting point to evaluate how an embedded hypervisor may perform.

## 4.3 ESXi Configuration

VMware's Elastic Sky X Integrated (ESXi) is a type 1 hypervisor [VMw18a]. VMware's ESXi only supports 64-bit x86 processors released after September 2006 [VMw18a]. In this paper a pre-release version of ESXi 6.9 modified to run on ARMv8 ISA with the vSphere service console was tested, referred to as ESXi-ARM to differentiate from the version of ESXi that runs on x86 processors. It is important to note that the version of ESXi-ARM tested for this work is a not publicly available pre-release prototype that is not complete and with minimal optimization work performed.

Table 4.2: Hypervisor configuration

| **Configuration** | **Description** |
|---|---|
| Hypervisor | ESXi-6.9.2 pre-release modified for ARM (ESXi-ARM) |
| BIOS Version | EDK II |
| Memory | 3.97 GB |
| Virtual Flash | 119.75 GB |
| Hyperthreading | No |
| Power Management Policy | Balanced |

## 4.4 Operating System Configuration

Debian 10 (Buster) was chosen as the OS since it is open-source, widely available and easily configurable. It is also compatible with ESXi-ARM and the bare-metal MACCHIATObin development board. The kernel used for both the native and hosted environment was 4.19.

Table 4.3: OS configuration

| **Configuration** | **Description** |
|---|---|
| OS Name | Debian GNU/Linux 10 (Buster) |
| Linux Kernel Version | Linux 4.19 |

## 4.5 Virtual Machine Configuration

The VM environment was setup in a single environment and then the VMDK was cloned to create additional VMs. This ensured that the environments were exact replicas.

Table 4.4: Virtual machine configuration

| **Configuration** | **Description** |
|---|---|
| CPU | 1 vCPU |
| Memory | 2 GB |
| Hard Disk | |
| SCSI Controller | VMware Paravirtual |
| USB Controller | USB 3.1 |
| Network adapter | VMXNET 3 |
| Video Card | 4 MB total memory |
| | 256 MB 3D Memory |
| Power management | Standby mode |

# 5 Results and Analysis

The performance of ESXi-ARM as an embedded hypervisor is evaluated in the following sections. First, the memory consumption of ESXi-ARM is analyzed in section 5.1 as the number of concurrently running VMs is scaled up. Next, sources of virtualization overhead are evaluated. As discussed in section 2, virtualization overhead can come from many different areas within a system, such as sensitive instruction virtualization, interrupt handling, sharing resources, etc. An important metric is the scheduling latency of the hypervisor, assessed in section 5.2 using Linux's Cyclictest. LMbench3 is used to evaluate low-level function latencies and bandwidth of the system in section 5.3. Finally, the execution time of different domain-specific applications are examined in section 5.4, to determine how the prototype of ESXi-ARM would perform in various real-world embedded domains. It is important to note that the version of ESXi-ARM tested in this paper is still under development and all benchmarks would need to be rerun on future versions of the software to give an accurate evaluation.

## 5.1 Memory Consumption

Embedded devices are often resource-constrained, especially in terms of memory. An embedded hypervisor should have a small code base and make efficient use of memory, see 3.3.1. Virtualization can severely impact the performance of memory access and the amount of memory needed to host a VM. The memory footprint of the ESXi-ARM hypervisor without hosting any VMs is 1.42 GB. From figure 5.1, it is seen that the memory consumption of the host remains relatively consistent when scaling the number of VM instances hosted on ESXi-ARM up. The maximum memory consumption of ESXi-ARM is 1.66 GB when hosting five VM instances. Overhead memory includes space reserved for the VM frame buffer and various virtualization data structures, such as shadow page tables [VMw18b].

From figure 5.1, it is seen that the memory consumption of individual VMs dramatically decreases as more VMs are hosted on ESXi-ARM. While only one VM is hosted on ESXi-ARM, it consumes 1.38 GB of memory, in contrast to when five VMs are hosted, this same VM only consumes 0.09 GB of host memory. In contrast to VM1, while five VMs are hosted on ESXi-ARM, VM5 consumes 1.22 GB.

As more VMs are hosted on ESXi-ARM, a decrease in memory consumption for individual VMs is expected as ESXi-ARM uses memory reclamation techniques to dynamically expand or contract the amount of memory allocated to a VM [VMw18b]. The most drastic memory reduction is likely from transparent page sharing (TPS), discussed further in section 2.3.7, which enables multiple VMs to share pages if the contents of

the page are the same. Since each VM is running the same OS, a high level of TPS is expected. Although this technique is effective when hosting identical OS environments in the VM, this effect would decrease when hosting different OSs, such as for Use Case 5: "Real-Time Support" where a RTOS is hosted alongside a general purpose OS, see 3.4.1. Research done by [LUSG04] already proposes this solution for use cases such as Use Case 4: Decouple Application from Environment; see 3.4.1.

Other memory reclamation techniques include the ESXi memory balloon drive `vmmemctl`, which reduces the VM's memory demands by forcing the VM to relinquish memory pages it considers least valuable, set in this environment to 65%. Another technique ESXi-ARM uses for memory reclamation is memory compression to reduce the number of memory pages it will need to swap out [VMw18b]. Compressing memory pages has significantly less impact on performance than swap-in [VMw18b]. If these techniques are not sufficient, ESXi-ARM will forcibly reclaim memory using host-level swapping to a host cache. These techniques should have little effect on the performance evaluated in the next sections [VMw18b].

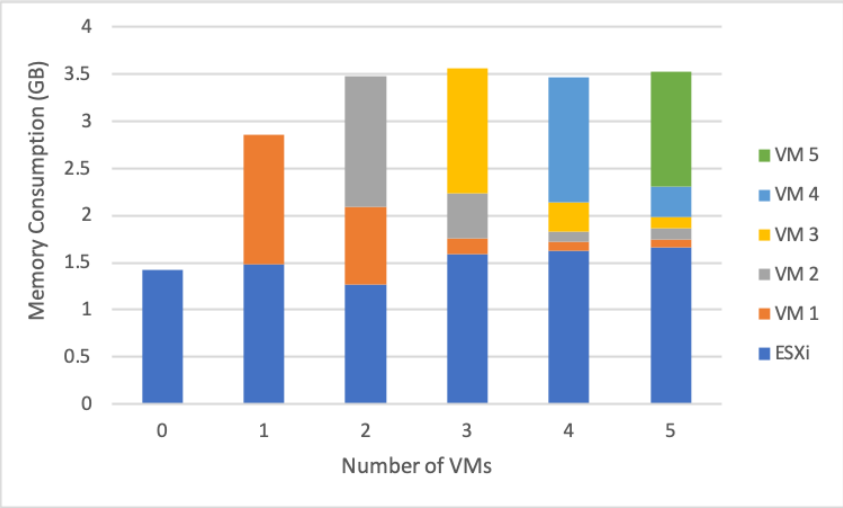


Figure 5.1: Host memory consumption while scaling the number of VMs hosted on ESXi-ARM up.

Table 5.1: Host memory consumption while scaling the number of VMs hosted on ESXi-ARM up.

| Number of Active VMs | ESXi-ARM Total Memory Consumption | VM 1 Memory Consumption | VM 2 Memory Consumption | VM 3 Memory Consumption | VM 4 Memory Consumption | VM 5 Memory Consumption |
|---|---|---|---|---|---|---|
| 0 | 1.42 GB | - | - | - | - | - |
| 1 | 2.86 GB | 1.38 GB | - | - | - | - |
| 2 | 3.48 GB | 0.82 GB | 1.39 GB | - | - | - |
| 3 | 3.56 GB | 0.17 GB | 0.48 GB | 1.32 GB | - | - |
| 4 | 3.47 GB | 0.09 GB | 0.11 GB | 0.31 GB | 1.33 GB | - |
| 5 | 3.53 GB | 0.09 GB | 0.11 GB | 0.13 GB | 0.32 GB | 1.22 GB |

## 5.2 Scheduling Latency

Latency and jitter are both critical characteristics of real-time systems, discussed in section 3.3.3. Latency is any delay in execution time, and jitter is the variance in execution time for repeated instructions. Latency can come from: interrupts being disabled when the event occurs; other interrupts arriving; nested interrupt handling; wake-up of the cyclictest executable; wait until preemption is re-enabled or higher priority task yields or same-priority task is pre-empted; scheduler overhead; processor sleep states, process migration, cache misses in the OS; lock priority inheritance etc.; and interactions between these [RV13]. It is important to determine the underlying cause for latency as summary statistics can hide outliers.

The scheduling latency, the delay between an event, and the start of real work is a crucial characteristic to determine the WCET. Cyclictest measures the amount of time that passes between when a timer expires and when the thread which set the timer executes. The measurement steps are as follows:

1. Switch the system to real-time priority and FIFO scheduling.
2. Acquire start timestamp using `clock_gettime`.
3. Call `clock_nanosleep` with specified timeout value.
4. Acquire end timestamp using `clock_gettime`.
5. Calculate latency as follows: $latency = end - timeout - start$.
6. Repeat steps 2-5 the specified number of loops.

Cyclictest starts several threads, each with different setup time so that they do not run in lockstep. In the ideal case, the requested timeout is identical to the measured time; the difference is the latency [RV13]. The resolution measured by cyclictest is 40 nsec for both the native and the virtualized environments.

Debian was patched Linux's PREEMPT-RT kernel for this test. This allows Debian to behave similar to a real-time OS meaning that a task in which a running thread can be stopped so that a higher priority task can run. This has the negative effect of making a system less deterministic since it is difficult to know exactly when the task will stop. Additionally, only tasks that have preemption enabled adhere to the policy and as such, a high priority task can be delayed indefinitely by lower priority tasks with preemption disabled. For the following benchmarks, Debian with and without the PREEMPT-RT patch was used to evaluate the system.

Debian virtualized with the ESXi-ARM hypervisor introduced additional scheduling latency into the system compared to native execution. In figure 5.2, the native Debian environment has a peak to the left of the virtualized environment. The peaks are the mode of the scheduling latency in the respective environments. The long tail of the Debian on ESXi-ARM environment suggests poor prioritization in the hypervisor's scheduler and needs to be addressed to make ESXi-ARM suitable for some real-time

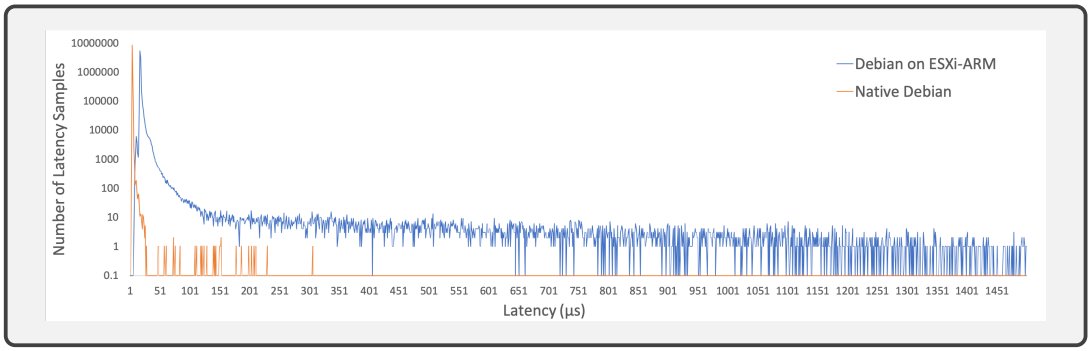


Figure 5.2: Cyclictest histogram comparing the latency of virtualizing Debian (without PREEMPT-RT patch) with ESXi-ARM compared to native Debian on a MACCHIATObin Single Shot development board. Cyclictest measured 10 million loops at an interval of 1,000 $\mu s$ at a priority level of 90. Please note, all threads exceeding a wakeup latency of 1,500 $\mu s$ are not displayed on the histogram.

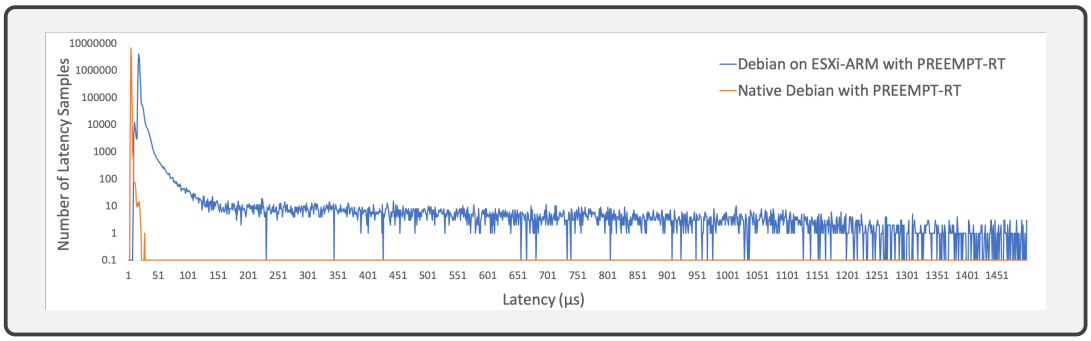


Figure 5.3: Cyclictest histogram comparing the latency of virtualizing Debian (with PREEMPT-RT patch) with ESXi-ARM compared to native Debian on a MACCHIATObin Single Shot development board. Cyclictest measured 10 million loops at an interval of 1,000 $\mu s$ at a priority level of 90. Please note, all threads exceeding a wakeup latency of 1,500 $\mu s$ are not displayed on the histogram.

systems. Another important metric is the maximum latency, see table 5.2. This reflects the systems WCET, discussed in section 3.3.3. Debian running natively on the MACCHIATObin had a WCET of 305 $\mu s$ compared to 50,835 $\mu s$ for the virtualized environment.

The PREEMPT-RT patch turns Debian into a real-time OS by giving all tasks scheduled from cyclictest a priority and ensuring that tasks with a higher priority will be scheduled on a CPU within a fixed time of the event waking the task. Repeating the measurements with the PREEMPT-RT kernel patch did not significantly improve the

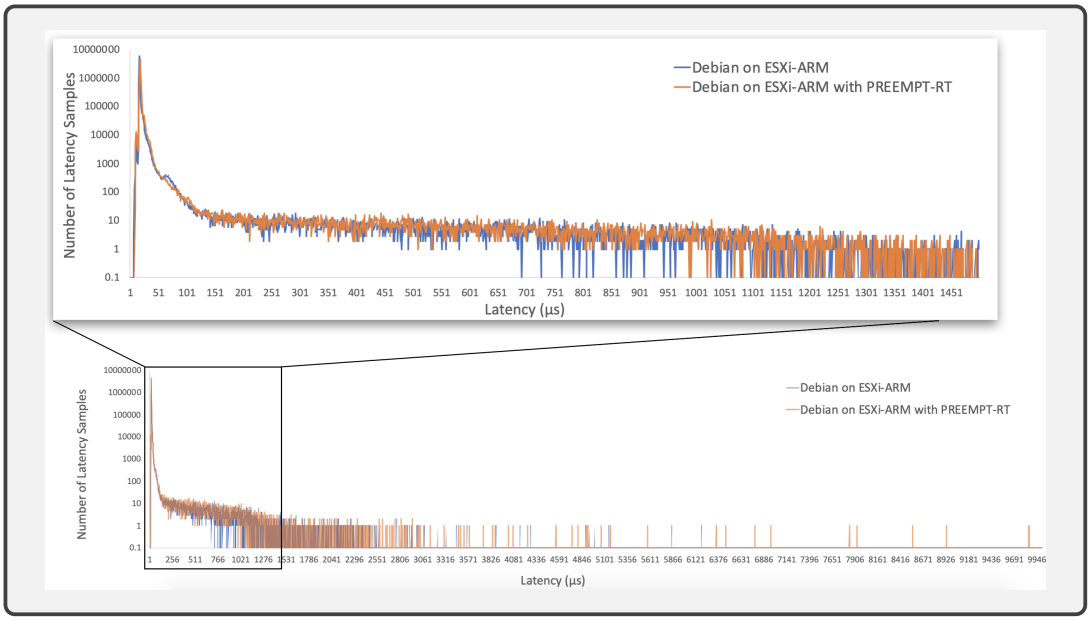


Figure 5.4: Cyclictest histogram comparing the latency of Debian with and without the PREEMPT-RT patch on ESXi-ARM on a MACCHIATObin Single Shot development board. Cyclictest measured 10 million loops at an interval of 1,000 $\mu s$ at a priority level of 90. Please note, all threads exceeding a wakeup latency of 10,000 $\mu s$ are not displayed on the histogram. The histogram is blown up to cut off at 1500 $\mu s$ to better display the effects at lower latency.

Table 5.2: Cyclictest latency measuring 10 million loops at an interval of 1000 $\mu s$ at a priority level of 90.

| Environment | Minimum Latency ($\mu s$) | Average Latency ($\mu s$) | Maximum Latency ($\mu s$) |
|---|---|---|---|
| Native Debian | 2 | 3 | 305 |
| Hosted Debian | 6 | 17 | 50,835 |
| Native Debian w PREEMPT-RT patch | 3 | 4 | 27 |
| Hosted Debian w PREEMPT-RT patch | 8 | 17 | 10,327 |

performance for Debian on ESXi-ARM, seen in figure 5.3. The graph shows that the peak of the native environment is still to the left of the virtualized environment, indicating that the scheduling latency of the native environment is better. Additionally, the virtualized environment still has a long tail compared to the native environment, which now has no peaks after 27 $\mu s$. The WCET of the virtualized environment improved from 50,835 $\mu s$ to 10,327 $\mu s$, but is still significantly below the performance of native Debian [1].

[1]It is important to note that this WCET is only an estimate and could dramatically increase with more

Although the WCET improved and the average latency remained the same, figure 5.4 shows that the addition of the PREEMPT-RT kernel patch collectively lowered the performance of Debian on ESXi-ARM. The environment with the PREEMPT-RT patch has a longer tail that does not start to decrease until around 1000 $\mu s$ opposed to Debian on ESXi-ARM without the PREEMPT-RT patch which shows a significant decrease in the number of latency samples at around 750 $\mu s$. Overall, the PREEMPT-RT patch decreased the scheduling latency performance for the virtualized environments.

It was expected that ESXi-ARM would introduce scheduling latency into the system compared to native execution since task scheduling requires unavoidable hypervisor intervention; discussed in section 2.3.2. Additionally, this version fo ESXi-ARM is a pre-release prototype which has not been performance optimized. The ESXi hypervisor for x86 is not a real-time hypervisor and instead optimized for the server domain. In the server domain, fair scheduling policies are preferred in contrast to embedded systems where real-time performance is often required. The scheduler of the ESXi-ARM hypervisor is a potential area of improvement for future releases to supported uses cases such as Use Case 5: Real-Time Support, discussed in 3.4.1.

## 5.3 Microbenchmarks

Microbenchmarks are small pieces of code intended to extract low-level performance metrics by focusing on specific operations such as communication between threads. The performance of simple functions is evaluated using LMbench3 benchmarking suite originally developed by Larry McVoy and Carl Staelin [MS96, MS98]. This suite is especially useful in evaluating low-level metrics related to latency and bandwidth and has been extensively used to evaluate the performance of hardware, OSs, and hypervisors [XBG+10].

LMbench3 uses the `gettimeofday` clock to measure execution time with a module `compute_enough` that automatically computes the time interval required to reduce clock resolution timing errors to less than 1% [2]. Some results from LMbench3 have also been excluded here, such as execution latency of primitive operations such as integer-, float-, double- due to no significant variation in execution time between testing environments.

### 5.3.1 Latencies

Apart from scheduling latency, virtualizing a system can also introduce low-level function latency due to sensitive instructions. The LMbench3 suit measures several different low-level function latencies. Measuring the latency of low-level functions is useful in identifying potential bottlenecks in system performance. The execution time of the native environment is the baseline of how quickly the system (hardware and software) can execute a set of instructions. Any additional execution time required by the virtualized

iterations or with additional workload

[2] `gettimeofday` has received criticism in regards to timing accuracy which the creators of LMbench have disputed. For more information, see the lmbench-3.0-a9 package textfile hbench-REBUTTAL.

environment is assumed to come from the addition of the ESXi-ARM hypervisor since hardware and software remain unchanged.

### Simple System Latencies

System calls (syscalls) are a programmatic way for applications to request services from the OS and requires the processor to switch modes, discussed in 2.3.2. Nontrivial entry into the system is measured by null call, null I/O, stat, and open/close. Null call measures the syscall overhead by repeatedly calling the function `getppid()`, stressing the syscall entry and exit path but doing no substantial work in between. In the virtualized environment, the physical process ID needs to be translated to the virtual one by the hypervisor. Null I/O function simulates writing and reading to a block device without actually doing any work; thus, the speed is primarily based on the processor and relevant syscalls. Zeros are read from `/dev/zero` and written to `/dev/null`, a pseudo device driver that does nothing but discard data [MS96]. The function `stat` repeatedly requests information about a file while the open/close function opens and then closes a file.

Table 5.3: Simple function latencies as measured by LMbench3 benchmarking suite. The values are all in microseconds averaged over three independent runs with standard deviation as the error estimate (smaller is better).

| **Test Case** | **Native Debian** ($\mu s$) | **Debian on ESXi-ARM** ($\mu s$) | $\Delta(\mu s)$ | **Percent Overhead** |
|---|---|---|---|---|
| null call | 0.35 ± 0.00 | 1.97 ± 0.01 | 1.62 | 464% |
| null I/O | 0.49 ± 0.01 | 2.16 ± 0.01 | 1.67 | 338% |
| stat | 1.33 ± 0.01 | 3.06 ± 0.05 | 1.73 | 130% |
| open/close | 3.76 ± 0.00 | 7.52 ± 0.26 | 3.76 | 104% |
| Select on 100 fd's | 2.57 ± 0.00 | 4.27 ± 0.04 | 1.69 | 66% |

Syscalls were expected to have high percent virtualization overhead compared to native execution due to their fast execution and the additional handling required from the hypervisor to execute these functions. The null call function is the simplest out of the measured syscalls and has the highest virtualization overhead compared to native execution at 464%. Comparatively, the open/close function requires $3.76\mu s$ to execute in the native environment compared to $7.52\mu s$ for Debian on ESXi-ARM, a 104% overhead. However, comparing the difference in execution time for syscalls between the native and virtualized environment, all the functions have a relatively consistent overhead with an average of 1.72 $\mu s$ [3]. A likely cause for the execution time overhead is that the hypervisor has to intervene to perform processor mode switch, discussed in section 2.3.2. Syscall execution time remained consistent when scaling the number of VMs hosted on ESXi-ARM up.

[3]The open/close function are two separate syscalls that are measured together.

### Process Creation Latencies

The fork, execve, and sh function are increasingly expensive forms of process creation. The function `fork`, creates a duplicate of the calling process, which then immediately exits the child process in the benchmark. The execve function, forks a process, and then has the child process execute a small hello world program. The parent process waits for the child to finish and exit. The sh function forks a process, and the child process calls execlp starting the shell, `/bin/sh`, with the new program as a command to the shell. This function includes the cost of the shell searching the file path for the program.

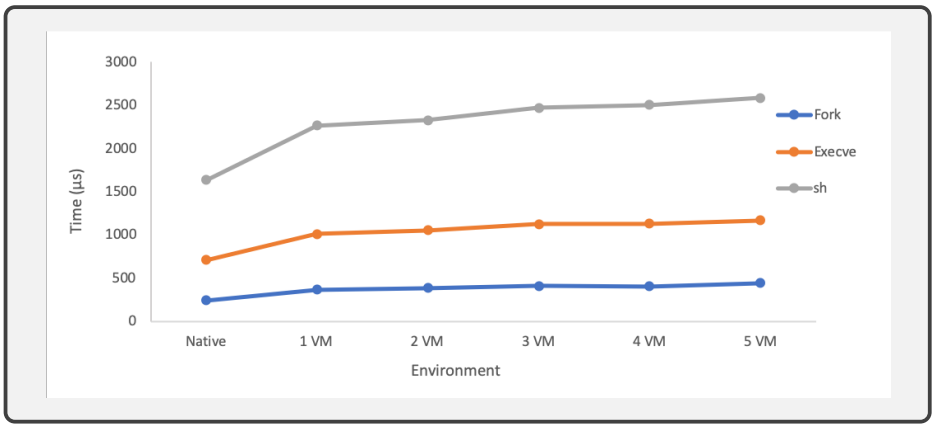


Figure 5.5: Process creation latencies. (Smaller is better)

Table 5.4: Process creation latency with native Debian compared to scaling the number of VMs hosted on an ESXi-ARM hypervisor. (Smaller is better)

| Environment | fork ($\mu s$) | fork + execve ($\mu s$) | fork + sh ($\mu s$) |
|---|---|---|---|
| Native Debian | $243 \pm 4$ | $709 \pm 4$ | $1635 \pm 3$ |
| ESXi-ARM + 1 VM | $366 \pm 27$ | $1010 \pm 55$ | $2266 \pm 135$ |
| ESXi-ARM + 2 VM | $385 \pm 7$ | $1053 \pm 8$ | $2326 \pm 33$ |
| ESXi-ARM + 3 VM | $411 \pm 14$ | $1123 \pm 53$ | $2472 \pm 76$ |
| ESXi-ARM + 4 VM | $404 \pm 14$ | $1129 \pm 39$ | $2502 \pm 71$ |
| ESXi-ARM + 5 VM | $442 \pm 26$ | $1169 \pm 26$ | $2586 \pm 16$ |

The percent virtualization overhead is between 39-50% for process creation, with fork having the highest percent overhead out of the three functions. The sh function had the highest $\Delta$ in execution time at 631 $\mu s$. This trend is also apparent as scaling the number of VMs hosted on ESXi-ARM up. The fork and execve functions only slightly increase in execution time, whereas the sh function has a faster increase in execution time, as seen in Figure 5.5.

### Signal Latencies

Signals tell another process to handle an event similar to interrupts in a CPU [MS96]. Signal handling is often critical to layered systems such as software development environments and threading libraries, which provide an OS-like layer on top of the OS [MS96]. LMbench3 measures signal handling by installing a signal handler and then repeatedly sending itself the signal [MS96]. This benchmark uses the POSIX sigaction interface because the signal stays installed as opposed to the more portable signal interface [MS96].

Table 5.5: Signal latencies as measured by LMbench3 benchmarking suite. The values are all in microseconds averaged over three independent runs with standard deviation as the error estimate (smaller is better).

| **Test Case** | **Native Debian** ($\mu s$) | **Debian on ESXi-ARM** ($\mu s$) | **Percent Overhead** |
|---|---|---|---|
| Signal handler install | 0.50 ± 0.01 | 2.11 ± 0.02 | 324% |
| Signal handler overhead | 2.23 ± 0.04 | 4.03 ± 0.08 | 81% |

The signal installer function had considerably more virtualization overhead at 324% as opposed to the signal handler, which had a virtualization overhead of 81%. This benchmark does not reflect real-world applications, as signals usually go to another process.Thus the true cost of sending that signal is the signal overhead plus the context switch overhead if they are running on the same core of a processor [MS96].

### File Latencies

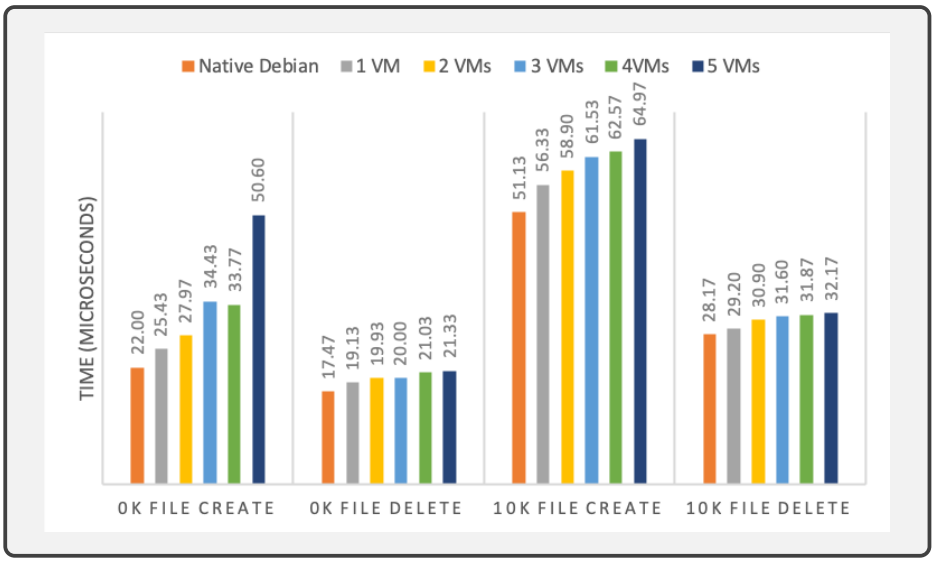


Figure 5.6: File System Latencies. (Smaller is better)

File system latency is defined by LMbench as the time required to create or delete a

zero length file [MS96]. Overhead for file creation consists of computing the label for the new file, performing permission checks for searching the path, modifying the directory, and creating the file. File deletion overhead for the 0K file delete consists of performing permission checks for searching the path, modifying the directory, and unlinking the file. The number of VMs executing has a stronger effect on file creation execution time than file deletion, seen in Figure 5.6.

### Context Switching Latency

Context switch time is the time needed to save the state of one process and restore the state of another process, see section 2.3.1. LMbench1 and LMbench2 measured context switch time using a "hot-potato" approach with Unix pipes connected in a ring [MS96]. LMbench3 uses N LMbench2-style process rings to measure the context switch times with all N rings running in parallel [MS96].

A process size of zero does nothing but pass the token on to the next process. A process size greater than zero simulates the summing up of an array of the specified size. The effect of this is that the data and instruction cache get polluted by some amount before the token is passed on. The overhead is measured using hot caches. As the number and size of the processes increases, the caches become increasingly polluted until the set of processes do not fit. The context switch times go up because a context switch is defined as the switch time plus the time it takes to restore all of the process state, including cache state. This means that the switch includes the time for the cache misses on larger processes [MS96].

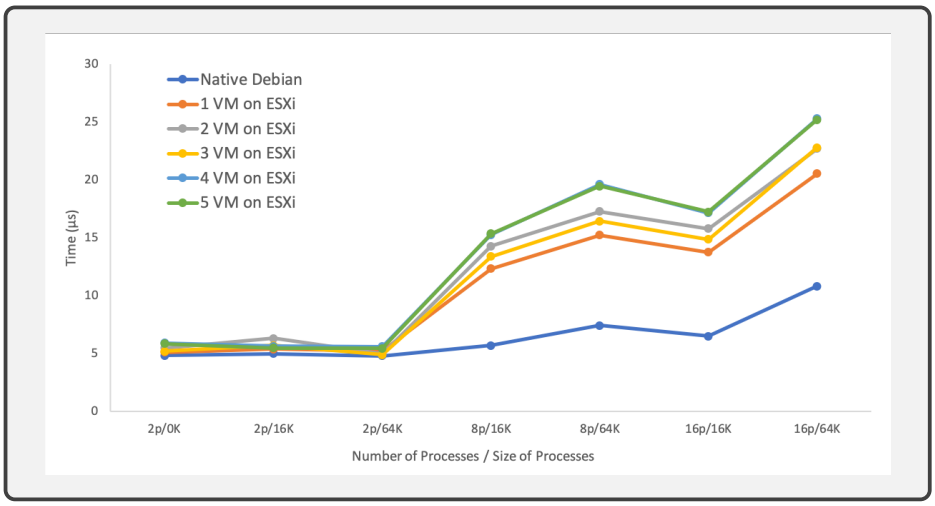


Figure 5.7: Context switching time of processes as a with varying process sizes. (Smaller is better)

The context switching time was consistent for two processes of varying size in all environments, see Figure 5.7. At 8 and 16 processes the native environment performed

significantly better than the virtualized environments. The size of the process had a more significant effect on the virtualization overhead than the number of processes. Both 8p/16k and 16p/16k had similar executions times within environments, whereas 8p/64k performed significantly better than 16p/64k.

### 5.3.2 Local Communication Bandwidth

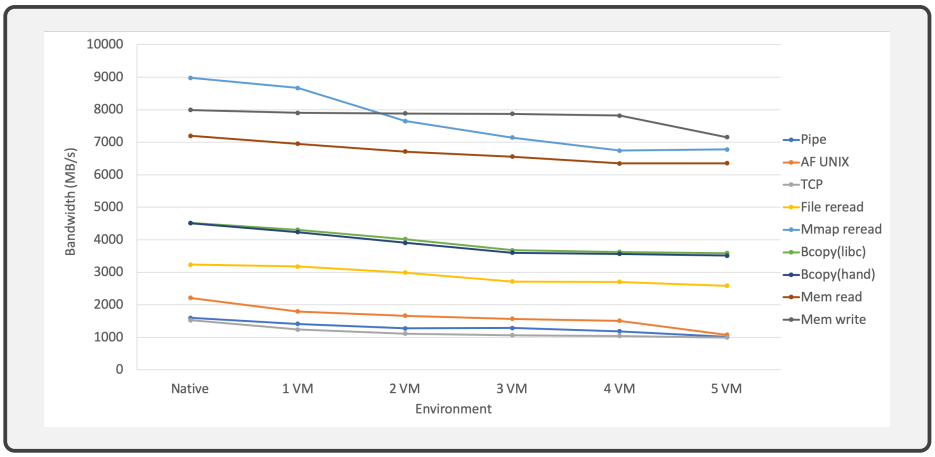


Figure 5.8: Local communication bandwidth as the number of VMs hosted on ESXi-ARM is increased. (Bigger is better)

Table 5.6: Local communication bandwidth in MB/s of native Debian compared to Debian in a VM Hosted on ESXi-ARM. (Bigger is better)

| Environment | Native Debian ($\mu s$) | Debian on ESXi-ARM ($\mu s$) | Percent Overhead |
|---|---|---|---|
| Pipe | $1599 \pm 1$ | $1409 \pm 48$ | 12% |
| AF UNIX | $2211 \pm 47$ | $1796 \pm 59$ | 19% |
| TCP | $1524 \pm 9$ | $1241 \pm 132$ | 19% |
| File reread | $3236 \pm 129$ | $3174 \pm 111$ | 2% |
| Mmap reread | $8979 \pm 3$ | $8670 \pm 450$ | 3% |
| Bcopy(libc) | $4517 \pm 1$ | $4309 \pm 68$ | 5% |
| Bcopy(hand) | $4510 \pm 1$ | $4238 \pm 81$ | 6% |
| Memory read | $7197 \pm 0$ | $6950 \pm 35$ | 3% |
| Memory write | $7990 \pm 1$ | $7904 \pm 6$ | 1% |

Bandwidth is the rate at which a particular facility can move data in the system [MS96]. There was not significant decrease to bandwidth due to virtualization overhead with most of the facilities experiencing less than 10% overhead, see Table 5.6, which can be attributed to normal deviance in execution time or latencies from hypervisor intervention.

Pipe latency is the zero-sized context switch overhead plus the pipe overhead. The 2p/0k context switch virtualization overhead was 6% or a $\Delta$ of 0.29 $\mu s$. This overhead did not significantly effect the more complex pipe function which has a virtualization overhead percent of 12%. The highest overhead is from AF UNIX, at 19%, which are sockets used for local interprocess communication. Although both AF UNIX sockets and pipes are mechanisms to enable interprocess communication, AF UNIX sockets are more flexible than pipes, such as enabling bidirectional communication. AF UNIX sockets also support additional features such as passing kernel-verified user and group identification. This flexibility requires intervention from the hypervisor to control the communication and permission processing but had the same amount of virtualization overhead at 19%.

The `read` and `mmap` functions are used to benchmark the overhead associated with data reuse [MS96]. The difference between `bcopy` [4] and `read` is the cost of the file and virtual memory system overhead [MS96]. As seen in Table 5.6, this difference did not significantly impact the system performance.

The bandwidth decreases as the number of VMs hosted on ESXi-ARM is scaled up. Interesting to note is that the bandwidth of the functions Pipe, AF Unix and TCP converge as the number of VMs is scaled up.

## 5.4 MiBench Performance Metrics

Microbenchmarks such as LMbench3 and Cyclictest measure basic system operations and behavior. Although microbenchmarks like these are useful in identifying potential bottlenecks in execution, they do not represent how real-world applications will behave. For this reason, a series of domain-specific benchmarks from MiBench benchmarking suite have been selected to evaluate ESXi-ARM performance.

### 5.4.1 Automotive and Industrial Control

The Automotive and Industrial Control category focuses on embedded control systems, such as air bag controllers or sensor systems. These systems rely primarily on basic math abilities, bit manipulation, and data organization [GRE+01]. Four algorithms were chosen:

**basicmath** Calculates simple mathematical functions needed for calculating road speed or vector values.

**bitcount** Counts the number of bits in an array of integers using five different algorithms.

**qsort** Sorts array of strings into ascending order using quick sort algorithm. Important for systems that require fast prioritization of data.

**susan** Image recognition package developed for recognizing corners and edges in Magnetic Resonance Images of the brain. Typical of a real-world application employed

[4] bcopy is deprecated but as we are testing the virtualization overhead and not the actual function the results are still valid.

for vision based quality assurance. Three separate susan algorithms were run: smoothing, edges and corners.

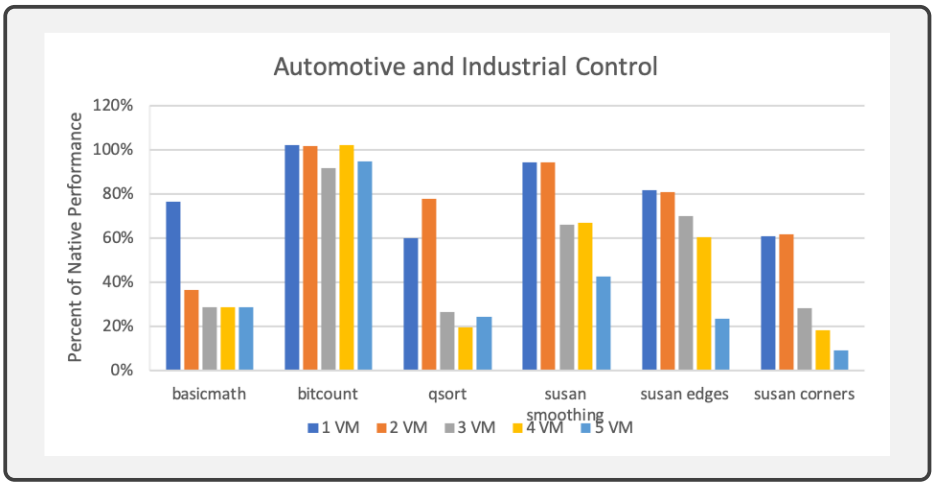


Figure 5.9: Virtualization overhead of Debian on ESXi-ARM measured with the MiBench Automotive Suite. Execution times are the average over 50 runs and have been normalized to native execution time.

Out of the Automotive and Industrial Control domain, Debian on ESXi-ARM performed best on the bitcount algorithm, with equal or close to equal performance to the native environment. This could be due to bitcount having the lowest total percentage of load and store instructions [GRE+01]. The only other algorithm that was able to achieve close to native performance was Susan Smoothing and only when less than three VMs were concurrently hosted on ESXi-ARM. Performance for less than three VMs was close to equal for qsort and susan but significantly declined as more VMs were hosted on ESXi-ARM.

### 5.4.2 Consumer

Consumer Devices includes devices such as digital cameras, phones, and tablets. This category of algorithms focuses primarily on multimedia applications [GRE+01].

**jpeg** Algorithm for image compression and decompression commonly used to view images in embedded documents.

**lame** MP3 encoder that supports constant, average and variable bit-encoding. Uses small and large wave files as data input

**tiff2bw** Image conversion. Converts a color TIFF image to black and white.

**tiff2rgba** Image conversion. Converts a color image in the TIFF format into a RGB color formatted TIFF image.

**tiffdither** Dithers a black and white TIFF bitmap to reduce the resolution and size of the image.

**tiffmedium** Converts an image to a reduced color palette.

**typeset** Captures the processing required to typeset an HTML document without any rendering overheads.

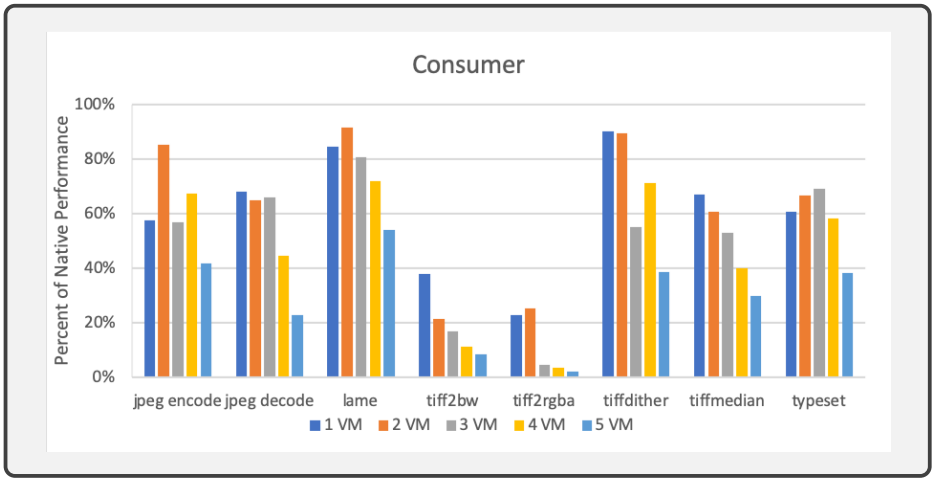


Figure 5.10: Virtualization overhead of Debian on ESXi-ARM measured with the MiBench Consumer Suite. Execution times are the average over 50 runs and have been normalized to native execution time.

The worst performing algorithm from the Consumer domain was tiff conversion. Debian on ESXi-ARM performed less than 40% as well as native Debian for tiff2bw and tiff2rgba, which significantly decreased as the number of VMs hosted on ESXi-ARM increased. When five VMs were running on ESXi-ARM, tiff2bw and tiff2rgba had a performance of only 9% and 2%, respectively, compared to native execution. For all the algorithms from the Consumer domain there was a significant performance decrease at five VMs.

### 5.4.3 Network

The network category represents embedded processors in network devices such as switches and routers. These devices focus on shortest path calculations, tree and table look-ups, and data I/O [GRE+01]. Two algorithms were chosen:

**dijkstra** Constructs large graph in adjacency matrix and then calculates the shortest path.

**patricia** Patricia tries are often used to represent routing tables in network applications.

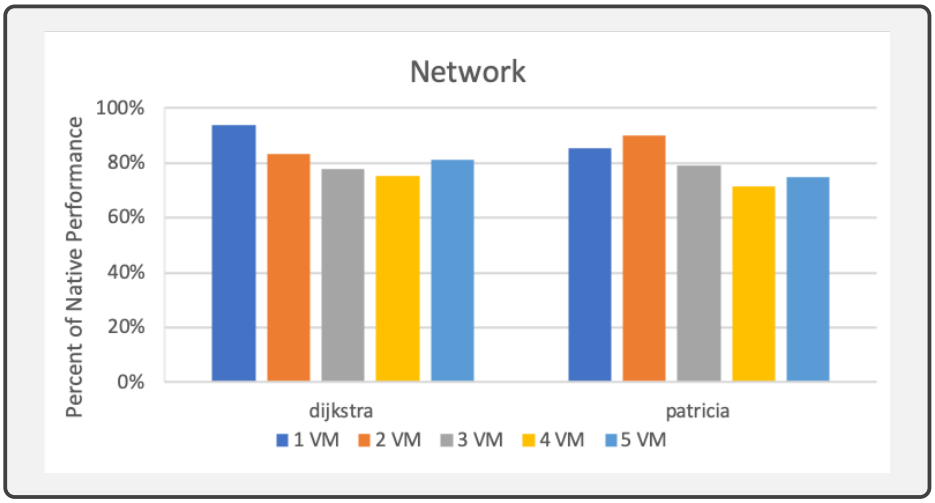


Figure 5.11: Virtualization overhead of Debian on ESXi-ARM measured with the MiBench Network Suite. Execution times are the average over 50 runs and have been normalized to native execution time.

Neither of the two Network domain algorithm's performance were significantly impacted by virtualization. Performance was consistent, around 80% of Native, even as the number of VMs concurrently hosted on ESXi-ARM was scaled up to five.

### 5.4.4 Office

Office category represents printers, fax machines, and word processors. The algorithms in this category focus primarily on text manipulation [GRE+01].

**ghostscript** postscript language interpreter without its graphical interface.

**ispell** Spell checker that is similar to the Unix spell, but faster.

**stringsearch** Searches for given words in phrases using a case insensitive comparison algorithm.

Performance was above 80% for both ghostscript and ispell when running less than three VMs on ESXi-ARM. Ghostscript performance significantly decreased at three VMs down to about 50%. Stringsearch performance decreased inconsistently while scaling up the number of VMs. Running two and four VMs performed significantly worse on the stringsearch algorithm compared to one and three VMs.

### 5.4.5 Security

Security is one of the most important aspects in regard to IoT embedded devices. The algorithms chosen focus on data encryption, decryption, and hashing [GRE+01].

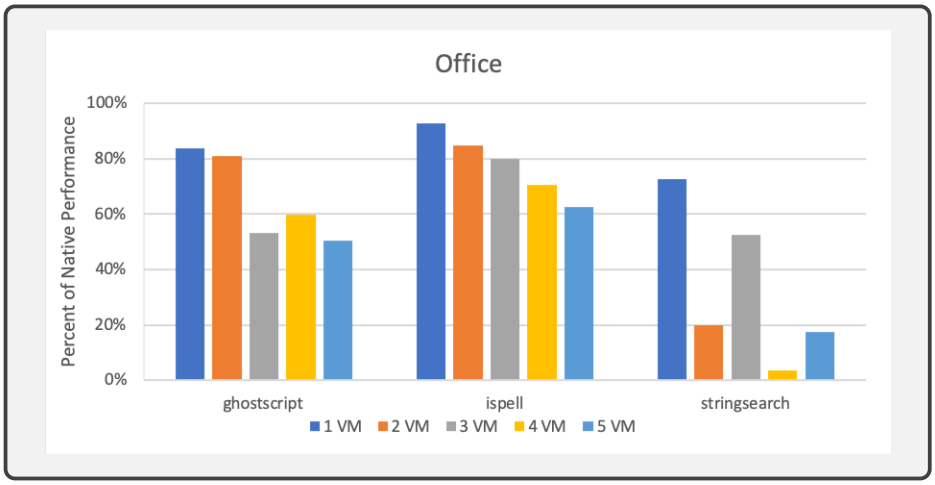


Figure 5.12: Virtualization overhead of Debian on ESXi-ARM measured with the MiBench Office Suite. Execution times are the average over 50 runs and have been normalized to native execution time.

**blowfish** Symmetric block cipher with variable length key (32-448 bits) input is a large text file.

**rijndael (AES)** Block cipher (128, 192, 256-bit key option) encrypts a large text file.

**sha** Secure hash algorithm used for generating digital signatures.

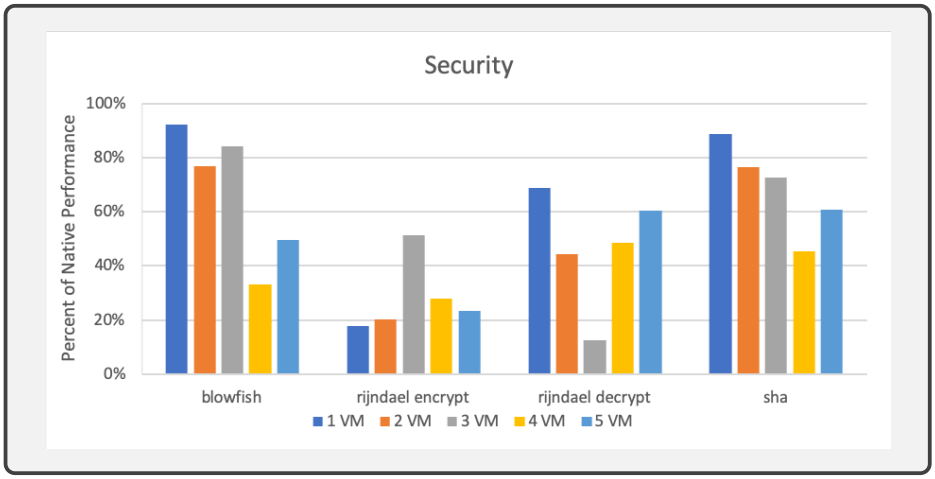


Figure 5.13: Virtualization overhead of Debian on ESXi-ARM measured with the MiBench Security Suite. Execution times are the average over 50 runs and have been normalized to native execution time.

Rjindael performed the worst out of the Security domain with all environments performing below 60%.

### 5.4.6 Telecommunications

Telecommunications focuses on voice encoding and decoding algorithms, frequency analysis and a checksum algorithm [GRE+01].

**ADPCM** Adaptive Differential Pulse Code Modulation is a variation of Pulse Code Modulation (PCM). The input data is large speech samples to encode and decode.

**CRC32** Performs Cyclic Redundancy Check (CRC) on a file. Used for error detection in data transmission.

**FFT/IFFT** Performs Fast Fourier Transform (FFT) and its inverse (IFFT) on an array of data. Input data is a polynomial function with pseudorandom amplitude and frequency sinusoidal components.

**GSM** Used for voice encoding/decoding input is small and large speech samples.

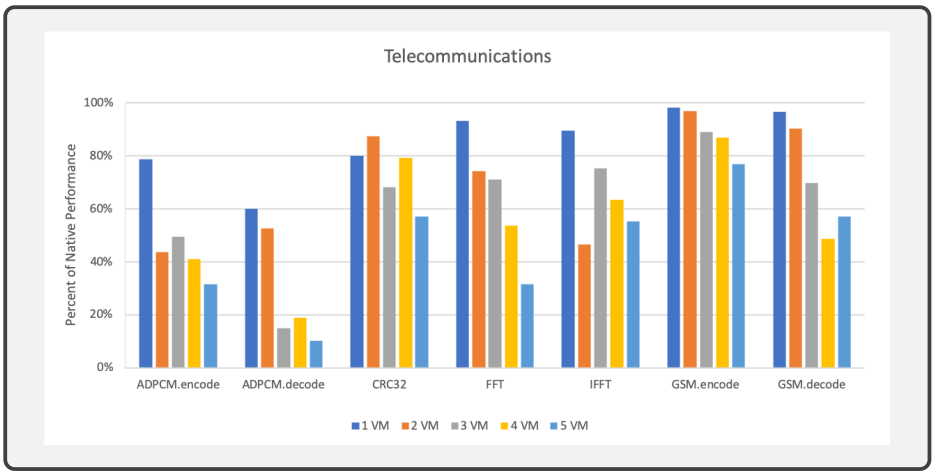


Figure 5.14: Virtualization overhead of Debian on ESXi-ARM measured with the MiBench Telecommunications Suite. Execution times are the average over 50 runs and have been normalized to native execution time.

When evaluating MiBench with only one VM hosted on ESXi-ARM, only ADPCM had a performance below 80% compared to native out of the telecommunications algorithms. Encoding from PCM to ADPCM performed closer to native than decoding except when running 2 VMs. GSM encode had the best performance even when scaling the number of concurrently running VMs up to five.

## 5.5 Summary

The performance of a prototype version of ESXi-ARM hypervisor is evaluated in this chapter. In section 5.1 the memory footprint of ESXi-ARM is evaluated using the vSphere service console. The ESXi-ARM hypervisor has memory footprint of 1.42 MB which is similar to the memory footprint displayed by XEN [nod19]. As discussed in section 3.3.1, embedded devices are often memory constrained so efficient use of memory is an especially important characteristic of an embedded hypervisor. As the number of concurrently hosted VMs was increased, the ESXi-ARM prototype made exceptional use of memory, most likely due to transparent page sharing. This effect would be diminished if different OS environments were used within the VMs, such as discussed in 3.4.1 for use cases 2 and 4.

In section 5.2, the scheduling latency of ESXi-ARM is evaluated using cyclictest. The ESXi-ARM hypervisor introduced additional scheduling latency compared to native execution, which was expected since this is a pre-release prototype that has not been yet been fully evaluated for anomalies and performance optimized. Native Debian with the PREEMPT-RT patch has a measured maximum latency of 27 $\mu s$ over 10 million loops compared to on ESXi-ARM with a maximum latency of 5348 $\mu s$. This could suggest poor prioritization in the hypervisor's scheduler but needs further evaluation to determine the source of latency.

The function latencies and bandwidth are evaluated using LMbench3 in 5.3. Simple syscalls have a high percentage of virtualization overhead but added only an average of 1.72 $\mu s$ to the execution time. The execution time of simple syscalls is not effected by the number of VMs hosted on ESXi-ARM. Process creation overhead using the fork, execve, and sh functions was at 39-50%. The virtualization overhead of process creation increased as the number of VMs executing on ESXi-ARM increased. In contrast, signal install and handle overhead was relatively consistent as the number of VMs hosted on ESXi-ARM was scaled up and added about 1.71 $\mu s$ of execution time compared to the native environment. File creation was impacted by the number of VMs hosted on ESXi-ARM to a greater degree compared to file deletion, which remained relatively consistent. Two process context switching has close to native performance but execution overhead increased substantially at 8 and 16 processes, however the size of the process has a greater effect on the virtualization overhead than the number of processes. ESXi-ARM has minimal decline in communication bandwidth compared to the native environment.

Since LMbench3 measures performance of basic system operations, it does not represent real world application scenarios. To evaluate how ESXi-ARM performs in different real-world embedded domains, MiBench benchmarking suite is used. Application specific benchmarks that reflect six categories: automotive and industrial control, consumer devices, office automation, networking, security, and telecommunications were chosen. ESXi-ARM performance varied between and within domains. Not all applications had a linear decline in performance as the number of VMs scaled up, so it can be concluded that the overhead is not only due to resource contention. Some of the variance in execution time can be attributed to the wake up latency but further analysis is warranted

to make conclusive statements in regards to the source of overhead.

# 6 Conclusion

Further development of embedded systems is currently constrained by issues such as growing hardware and software complexity, SWaP-C constraints, etc. This work aimed to answer whether virtualization is a viable solution to the problems currently facing embedded devices. To answer this question, first requirements of an embedded hypervisor were evaluated and use cases for embedded virtualization established. Thereafter, a feasibility study was performed using VMware's well-established server hypervisor, ESXi modified to run on ARM processors.

The diversity of the embedded domain makes generalizations of an embedded hypervisor difficult, however prevailing attributes required of an embedded hypervisor include a minimal code base, strong and fine-grained encapsulation, controlled communication, and minimal overhead. The pre-release prototype version of ESXi-ARM tested for this work is evaluated for potential virtualization use cases in embedded devices, listed in table 6.1.

## 6.1 Contributions

Before evaluating the benchmark results of ESXi-ARM, it is important to note that this version of the hypervisor software is an in-development prototype that has not undergone the extensive optimization that is performed for market released software. For this reason, gaps in functionality and performance are expected and any results only apply to this early pre-release version of ESXi-ARM. All benchmarks would need to be rerun on the final version of ESXi-ARM to give an accurate evaluation of performance.

Based on the results of the different benchmarks performed, it is concluded that the ESXi-ARM hypervisor could be a feasible solution for many embedded device domains. Although it is still in-development, the prototype ESXi-ARM hypervisor exhibited high stability throughout benchmarking, hosting up to five VMs concurrently with little performance degradation. The ESXi-ARM hypervisor also exhibited efficient use of memory due to its memory management policies, such as transparent page sharing. For low-level system calls, ESXi-ARM added only minor latency compared to native execution with an average of 1.72 $\mu s$. However, depending on the amount of syscalls called by an application, this minor latency could add significant overhead to a system. Scaling the number of concurrent VMs hosted on ESXi-ARM did not affect the execution time of the systems calls such as null call, null I/O, stat, open, close, signal install, and signal handle.

For other low-level functions such as file create, file delete, process creation, and context switching, the latency increased linearly as the number of VMs hosted on ESXi-ARM

Table 6.1: Constraints and use cases identified in chapter 3 with the evaluated relation to benchmarking tests performed in this work. A checkmark indicates application areas where the prototype ESXi-ARM could potentially be implemented as-is, and an X-mark indicates areas for future work. Blank spaces indicate areas that were not covered by the benchmarks performed in this work and necessitate further testing.

| Background | Constraints and Application Areas | Results | Section |
|---|---|---|---|
| 3.3.1 | Power Constrained | | |
| 3.3.1 | Heat Sensitive | | |
| 3.2.2 3.3.3 | Real-Time Systems | ✗ | 5.2 |
| 3.3.1 | RAM Constrained | ✓ | 5.1 |
| 3.3.2 | Intracommunication | ✓ | 5.3.1 5.3.2 |
| 3.3.2 | Intercommunication | | |
| 3.4.1 | Use Case 1: BYOD Devices | ✓ | 5.1 |
| 3.4.1 | Use Case 2: Decouple Application | ✓ | 5.1 |
| 3.4.1 | Use Case 3: Vanilla OS Support | ✓ | 5.1 |
| 3.4.1 | Use Case 4: Legacy Application Support | ✓ | 5.1 |
| 3.4.1 | Use Case 5: Real-Time Support | ✗ | 5.2 |
| 3.4.2 | Use Case 6: Multicore Management | | |
| 3.4.2 | Use Case 7: Hot Failover | | |
| 3.4.2 | Use Case 8: Migration | ✓ | 4.5 |
| 3.4.2 | Use Case 9: Dynamic I/O Management | | |
| 3.4.3 | Use Case 10: Operating System Isolation | ✓ | 5.3 |
| 3.4.3 | Use Case 11: Monitor Software Isolation | ✓ | 5.3 |
| 3.4.3 | Use Case 12: Licensing Separation | ✓ | 5.3 |
| 3.4.3 | Use Case 13: Fault Containment | ✓ | 5.3 |

increased. In comparison to the native environment and as the number of concurrently executing VMs increased, local communication bandwidth decreased slightly. From the LMbench3 latency and bandwidth results, it is concluded that virtualization solutions need to be evaluated on a case-by-case basis. The frequency of system calls required by an application and the tolerance for overhead are all device specific characteristics.

Additionally, according to the results from Linux's cyclictest, a weakness of the prototype ESXi-ARM is the scheduler. Real-time devices require deterministic behavior from a system such as WCET constraints. Results from cyclictest showed that in comparison to native execution, the prototype ESXi-ARM introduced additional scheduling latency and reduced the deterministic behavior of the system.

## 6.2 Future Work

Further evaluation should be done for characteristics identified in chapter 3. Table 6.1 summarizes these characteristics and the results of benchmarks run in this work. A

checkmark identifies application areas in which the prototype version of ESXi-ARM may already be able to adequately perform. Future work could focus on implementing the listed use cases. A blank space indicates areas for which no tests were run and need further evaluation, such as power constrained and heat sensitive devices.

An X-mark indicates areas for which the ESXi-ARM prototype needs further performance optimization to be applicable. One such area is improving the scheduling policy to adhere to priority constraints of different VMs. A more in-depth analysis of different real-time systems should be performed to determine their tolerance for missed deadlines. Determining the tolerance for missed deadlines would enable better scheduling policy decisions to be made between adhering to real-time and fairness constraints of the VMs executing. Further, additional tests from Linux's rt-test suite should be evaluated to determine other latency characteristics, such as scheduler or interrupt latency, to establish where ESXi-ARM adds latency into the system and identify optimization areas for future versions of ESXi-ARM.

A detailed analysis of the LMbench3 and MiBench suite should be performed to determine the source of the latencies in the virtualized environments. It would be beneficial to determine why certain functions take more time as the number of VMs hosted on ESXi-ARM increases. Additionally, attempts should be made to determine the source of jitter between MiBench runs.

Finally, to get an accurate representation of the performance of ESXi-ARM, other embedded hypervisors should be benchmarked on the MACCHIATObin development board as a comparison. Comparing the performance of different hypervisors would allow better insight into the embedded hypervisor market.

# List of Figures

# Bibliography


[ADH+11] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: A virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, number DOI: 10.1145/2043556.2043574 xiv, xvi in SOSP '11, pages 173–187, New York, NY, USA, 2011. Association for Computing Machinery.

[ARM13] ARM. Arm generic interrupt controller architecture version 3.0 and 4.0. Architecture Specification ARM IHI 0069D ID072617, ARM, www.arm.com, 2013.

[ARM16] ARM. Arm cortex-a72 mpcore processor. Reference Manual r0p3, ARM, December 2016.

[ARM17a] ARM. Aarch64 virtualization. Technical Report 0100, ARM, March 2017.

[ARM17b] ARM. Version 0.1 aarch64 exception and interrupt handling. Technical Report 1, ARM, February 2017.

[ARM19] ARM. Arm architecture reference manual armv8, for armv8-a architecture profile. ARM Architecture Reference Manual ARM DDI 0487D.b ID042519, ARM, www.arm.com, April 2019.

[ASL+19] Iqbal Alam, Kashif Sharif, Fan Li, Zohaib Latif, Md Monjurul Karim, Boubakr Nour, Sujit Biswas, and Yu Wang. Iot virtualization: A survey of software definition and function virtualization techniques for internet of things. *Cornell arXiv*, February 2019.

[BKL10] Diane Barrett, Gregory Kipper, and Samuel Liles. *Virtualization and Forensics A Digital Forensic Investigator's Guide to Virtual Environments*, volume 1. Elsevier, 2010.

[BMB+18] Alessandro Biondi, Mauro Marinoni, Giorgio Buttazzo, Claudio Scordino, and Paolo Gai. Challenges in virtualizing safety-critical cyber-physical systems. In *Embedded World Conference*, 2018.

[BNT17] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. *Hardware and Software Support for Virtualization*. Synthesis Lectures on Computer Architecture, 2017.

[CBS+18] Alfons Crespo, Patricia Balbastre, José Simó, Javier Coronel, Daniel Pérez, and Philippe Bonnot. Hypervisor-based multicore feedback control of mixed-criticality systems. In *IEEE Access*, volume 6, pages 50627–50640, 2018.

[Dal18] Christoffer Dall. *The Design, Implementation, and Evaluation of Software and Architectural Support for ARM Virtualization.* PhD thesis, Columbia University, 2018.

[Dan18] PD Dr. V. Danciu. Virtualisierte systeme. Lecture Notes, 2018.

[DLLN15] Christoffer Dall, Shih-WEi Li, Jintack Lim, and Jason Nieh. A measurement study of arm virtualization performance. Technical Report CUCS-XXX-XX, Columbia University, November 2015.

[ebu18] ebugden. Cyclictest. "`https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start?s[]=cyclictest`", August 08 2018. "[Last Accessed: December 1, 2019]".

[EES+01] Jakob Engblom, Andreas Ermedahl, M Sjdin, Jan Gustafsson, and Hans Hansson. Execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer - STTT*, January 2001.

[Fra18] Dann Frazier. stress-ng. "`https://wiki.ubuntu.com/Kernel/Reference/stress-ng`", December 18 2018. "[Last Accessed: December 5, 2019]".

[Fut19] Market Research Future. Global embedded software market research report- forecast 2022. Market Research Report MRFR/SEM/1571-HCRR, Market Research Future, https://www.marketresearchfuture.com/reports/embedded-software-market-2103, May 2019.

[Gan16] Jayneel Gandhi. *Efficient Memory Virtualization.* PhD thesis, University of Wisconsin-Madison, 2016.

[GRE+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, number 10.1109/WWC.2001.15 in WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[Hei07] Gernot Heiser. Virtualization for embedded systems. White Paper OK 40036:2007, Open Kernal Labs, Inc., November 2007.

[Hei08] Gernot Heiser. The role of virtualization in embedded systems. *Open Kernel White Paper*, pages 11–16, April 2008.

[Hei11] Gernot Heiser. Virtualizing embedded systems - why bother? In *DAC*. NICA and University of New South Wales, 2011.

[HGC97] W. A. Halang, R. Gumzej, and M. Colnaric. Measuring the performance of real time systems. In *Real Time Programming*, pages 93–98. International Federation of Automatic Control, September 1997.

[IDC14] IDC. Design of future embedded systems. Industry Report SMART 2009/0063, IDC, 2014.

[Jr96] Philip J. Koopman Jr. Embedded system design issues (the rest of the story). In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, pages 310–317, October 1996.

[KCSS11] Robert Kaiser, Massimo Conti, Orcioni Simone, and Ralf Seepold. *Solutions on Embedded Systems*, chapter Applicability of Virtualization to Embedded Systems, pages 215–226. Springer Netherlands, 2011.

[Kel15] Timon Kelter. *WCET Analysis and Optimization for Multi-Core Real-Time Systems*. PhD thesis, Dortmund, March 2015.

[Ker11] Timo Kerstan. *Towards Full Virtualization of Embedded Real-Time Systems*. PhD thesis, University of Paderborn, March 2011.

[KK13] David Kleidermacher and Mike Kleidermacher. *Embedded Systems Security*. Elsevier, 2013.

[LUSG04] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *6th Symposium on Operating Systems Design and Implementation*. USENIX association, 2004.

[LX15] Wenzhi Chen Lei Xu, Zonghui Wang. The study and evaluation of arm-based mobile virtualization. *International Journal of Distributed Sensor Networks*, 11(7):310308, 2015.

[LXRD19] Hao Li, Xuefei Xu, Jinkui Ren, and Yaozu Dong. Acrn: A big little hypervisor for iot development. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 31–44, New York, NY, USA, December 2019. Association for Computing Machinery.

[Mar19] Marvell. Macchiatobin. Website, October 2019.

[Men05] Daniel A. Menascé. Virtualization: Concepts, applications, and performance modeling. In *31th International Computer Measurement Group Conference*, pages 407–414, January 2005.

[MJGB17] Antonio Maña, Eduardo Jacob, Lorenzo Di Gregorio, and Michele Bezzi. Security aspects of virtualization. Technical Report ISBN 978-92-9204-211-0, DOI 10.2824/955316, Enisa, February 2017.

[MN10] Robert Mijat and Andy Nightingale. Virtualization is coming to a platform near you. White paper, ARM, 2010.

[MS96] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference*, 1996.

[MS98] Larry McVoy and Carl Staelin. Mhz: Anatomy of a micro-benchmark. In *USENIX 1998 Annual Technical Conference*, 1998.

[nod19] nodiscc. Xen wiki. "`https://wiki.debian.org/Xen`", September 09 2019. "[Last Accessed: December 16, 2019]".

[NRLB18] Vivian Noronha, Maximilian Riegel, Ekkehard Lang, and Thomas Bauschert. Performance evaluation of container based virtualization on embedded microprocessors. In *30th International Teletraffic Congress*, pages 79–84, 2018.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualization third generation architectures. In *Communications of the ACM*, volume 17, pages 412–421, New York, July 1974. ACM.

[R19] Lingeswaren R. Para virtualization vs full virtualization vs hardware assisted virtualization. Website, October 2019.

[RV13] Frank Rowand and Arnout Vandecappelle. Using and understanding the real-time cyclictest benchmark. "`https://mindlinux.wordpress.com/2013/10/25/using-and-understanding-the-real-time-cyclictest-benchmark-frank-rowand-son` October 25 2013. "[Last Accessed: December 10]".

[Sch19] Matthias Schorer. Hypervisor esxi: Virtualisierung für arm-systeme. Internet, May 2019.

[SGB+16] Junaid Shuja, Abdullah Gani, Kashif Bilal, Atta Ur Rehman Khan, Sajjad A. Madani, Samee U. Khan, and Albert Y. Zomaya. A survey of mobile device virtualization: Taxonomy and state-of-the-art. *ACM Computing Surveys*, 0(0):36, April 2016.

[Sof] Green Hills Software. Integrity real-time operating system. "`https://www.ghs.com/products/rtos/integrity.html`. "[Last Accessed: December 10, 2019]".

[TSC14] Gilberto Taccari, Luca Spalazzi, and Andrea Claudi. Embedded real-time virtualization: State of the art and research challenges. In *16th Real-Time Linux Workshop*. Università Politecnica delle Marche, 2014.

[VMw07] VMware. Understanding full virtualization, paravirtualization, and hardware assist. White Paper WP-028-PRD-01-01, VMware, 2007.

[VMw18a] VMware. Esxi. Internet, October 2018.

[VMw18b] VMware. Performance best practices for vmware vsphere 6.7. Manual 20180727, VMware, July 2018.

[VMw19] VMware. vsphere resource management update 2. Technical documentation, VMware Inc., 2019.

[XBG+10] Yang Xu, Felix Bruns, Elizabeth Gonzalez, Shadi Traboulsi, Klaus Mott, and Attila Bilgic. Performance evaluation of para-virtualization on modern mobile phone platform. In *International Journal of Computer and Systems Engineering*, volume 4, pages 229–236. World Academy of Science, Engineering and Technology, 2010.

[Xia16] Tian Xia. *Embedded Real-Time Virtualization Technology for Reconfigurable Platforms*. PhD thesis, Université Bretagne Loire, https://hal.archives-ouvertes.fr/tel-01905886, October 2016.

[ZG12] Qingling Zhao Zonghua Gu. A state-of-the-art survery on real-time issues in embedded systems virtualization. In *Journal of Software Engineering and Applications*, volume 5, pages 277–290. College of Computer Science, Zhejiang Univesity, January 2012.

# Bash Script

```
#!/bin/bash
#######################################################################
# Written by Caroline Frank for LMU bachelor thesis
# Last updated 9 December 2019
#######################################################################

# creates new folder to save results of run with date and time
home_dir=$(pwd)
folder=$(date +"%Y-%m-%d_%T")
mkdir "${home_dir}/results/$folder" && RESULTSPATH="$_"
TESTNUMBER=1 # Organizes tests in the order they are run

########################## Boot time ###################################

touch $RESULTSPATH/${TESTNUMBER}_boottime.txt
systemd-analyze >> ${RESULTSPATH}/"${TESTNUMBER}_boottime.txt"

######################### LMBench3 Suite ###############################

RUNNUMBER=1
cd ${home_dir}/lmbench-3.0-a9/src
make rerun
mv ${home_dir}/results/debian.0 ${home_dir}/results/debian.${RUNNUMBER}

((RUNNUMBER++))
make rerun
mv ${home_dir}/results/debian.0 ${home_dir}/results/debian.${RUNNUMBER}

((RUNNUMBER++))
make rerun
mv ${home_dir}/results/debian.0 ${home_dir}/results/debian.${RUNNUMBER}

######################### MiBench Suite ###############################

MIBENCH_PATH="${home_dir}/mibench-embedded"
```

```
36 SRCDIRS="automotive/basicmath automotive/bitcount automotive/qsort
   ↪ automotive/susan consumer/jpeg/jpeg-6a consumer/lame/lame3.70
   ↪ office/ghostscript/src"
37
38 SUMMARY=zSummaryTimes.txt
39 touch ${RESULTSPATH}/$SUMMARY
40
41 # $1=<filename> $2=<test> $3=<test name> $4=<options1 ...>
42 RUN_MIBENCH () {
43     local FILE="${1}_${2}.txt"
44     cd ${MIBENCH_PATH}/$CURRENT_DOMAIN/$2
45
46     echo Starting ${2} Test.......
47     touch ${RESULTSPATH}/$FILE$x
48     echo Ran $(date +"%Y-%m-%d_%T") Run Options: $@ >>
       ↪ ${RESULTSPATH}/$FILE
49     start=`date +%s%N`
50     ${@:3}
51     end=`date +%s%N`
52     echo Execution time was `expr $end - $start` ns. >>
       ↪ ${RESULTSPATH}/$FILE
53     echo ${2} `expr $end - $start` >> ${RESULTSPATH}/$SUMMARY
54 }
55
56 repeat=1
57 while [ $repeat -le 50 ]
58 do
59     #function_name <filename> <test> <executable> <options>
60     ######################  Automotive    ######################
61     CURRENT_DOMAIN=automotive
62
63     # Basic Math
64     ((TESTNUMBER++))
65     RUN_MIBENCH ${TESTNUMBER}_large basicmath ./basicmath_large >
       ↪ ${RESULTSPATH}/${TESTNUMBER}_output_bm_large.txt
66
67     # Bit Count
68     ((TESTNUMBER++))
69     RUN_MIBENCH ${TESTNUMBER}_large bitcount ./bitcnts 11250000 >
       ↪ ${RESULTSPATH}/${TESTNUMBER}_output_bc_large.txt
70
71     # QSort
72     ((TESTNUMBER++))
```

```
73      RUN_MIBENCH ${TESTNUMBER}_large qsort ./qsort_large input_large.dat
        ↪  > ${RESULTSPATH}/${TESTNUMBER}_output_qs_large.txt
74
75      # Susan
76      ((TESTNUMBER++))
77      RUN_MIBENCH ${TESTNUMBER}_large_s susan ./susan input_large.pgm
        ↪  ${RESULTSPATH}/${TESTNUMBER}_output_large.smoothing.pgm -s
78      RUN_MIBENCH ${TESTNUMBER}_large_e susan ./susan input_large.pgm
        ↪  ${RESULTSPATH}/${TESTNUMBER}_output_large.edges.pgm -e
79      RUN_MIBENCH ${TESTNUMBER}_large_c susan ./susan input_large.pgm
        ↪  ${RESULTSPATH}/${TESTNUMBER}_output_large.corners.pgm -c
80
81      ########################   Consumer
        ↪  ############################
82      CURRENT_DOMAIN=consumer
83
84      ## jpeg
85      ((TESTNUMBER++))
86      RUN_MIBENCH ${TESTNUMBER}_large_encode jpeg ./jpeg-6a/cjpeg -dct int
        ↪  -progressive -opt -outfile
        ↪  ${RESULTSPATH}/${TESTNUMBER}_output_large_encode.jpeg
        ↪  input_large.ppm
87      RUN_MIBENCH ${TESTNUMBER}_large_decode jpeg ./jpeg-6a/djpeg -dct int
        ↪  -ppm -outfile
        ↪  ${RESULTSPATH}/${TESTNUMBER}_output_large_decode.ppm
        ↪  input_large.jpg
88
89      ## lame
90      ((TESTNUMBER++))
91      RUN_MIBENCH ${TESTNUMBER}_large lame ./lame3.70/lame large.wav -S
        ↪  ${RESULTSPATH}/${TESTNUMBER}_lame_output_large.mp3
92
93      ### tiff2bw
94      ((TESTNUMBER++))
95      RUN_MIBENCH ${TESTNUMBER}_large_bw tiff-4.0.10 ./tools/tiff2bw
        ↪  ${MIBENCH_PATH}/${CURRENT_DOMAIN}/tiff-data/large.tif
        ↪  ${RESULTSPATH}/${TESTNUMBER}_tiffbw_output_large.tif
96
97      ### tiff2rgba
98      ((TESTNUMBER++))
99      RUN_MIBENCH ${TESTNUMBER}_large_rgba tiff-4.0.10 ./tools/tiff2rgba
        ↪  -c none ${MIBENCH_PATH}/${CURRENT_DOMAIN}/tiff-data/large.tif
        ↪  ${RESULTSPATH}/${TESTNUMBER}_tiffrgba_output_large.tif
```

```
100
101     ### tiffdither
102     ((TESTNUMBER++))
103     RUN_MIBENCH ${TESTNUMBER}_large_dither tiff-4.0.10
        ↪  ./tools/tiffdither -c g4
        ↪  ${MIBENCH_PATH}/${CURRENT_DOMAIN}/tiff-data/largebw.tif
        ↪  ${RESULTSPATH}/${TESTNUMBER}_tiffdither_output_large.tif
104
105     ### tiffmedian
106     ((TESTNUMBER++))
107     RUN_MIBENCH ${TESTNUMBER}_large_median tiff-4.0.10
        ↪  ./tools/tiffmedian
        ↪  ${MIBENCH_PATH}/${CURRENT_DOMAIN}/tiff-data/large.tif
        ↪  ${RESULTSPATH}/${TESTNUMBER}_tiffmedian_output_large.tif
108
109     ### typeset
110     ((TESTNUMBER++))
111     rm ${MIBENCH_PATH}/${CURRENT_DOMAIN}/typeset/*.ps
112     RUN_MIBENCH ${TESTNUMBER}_small typeset ./runme_large.sh
113     cp ${MIBENCH_PATH}/${CURRENT_DOMAIN}/typeset/*.ps ${RESULTSPATH}
114
115     ######################  Network ######################
116     CURRENT_DOMAIN=network
117
118     # Dijkstra
119     ((TESTNUMBER++))
120     RUN_MIBENCH ${TESTNUMBER}_large dijkstra ./dijkstra_large input.dat
        ↪  > ${RESULTSPATH}/${TESTNUMBER}_output_dijkstra_large.dat
121
122     # Patricia
123     ((TESTNUMBER++))
124     RUN_MIBENCH ${TESTNUMBER}_large patricia ./patricia large.udp >
        ↪  ${RESULTSPATH}/${TESTNUMBER}_output_patricia_large.txt
125
126     ###################### Office ######################
127     CURRENT_DOMAIN=office
128
129     ## ghostscript
130     ((TESTNUMBER++))
131     RUN_MIBENCH ${TESTNUMBER}_large ghostscript gs -sDEVICE=ppm
        ↪  -dNOPAUSE -q
        ↪  -sOutputFile=${RESULTSPATH}/${TESTNUMBER}_gs_output_large.ppm --
        ↪  ${MIBENCH_PATH}/$CURRENT_DOMAIN/ghostscript/data/large.ps
```

```
132
133     ## ispell
134     ((TESTNUMBER++))
135     RUN_MIBENCH ${TESTNUMBER}_large ispell ./ispell -a -d
        ↪ tests/americanmed <
        ↪ ${MIBENCH_PATH}/$CURRENT_DOMAIN/ispell/tests/large.txt >
        ↪ ${RESULTSPATH}/${TESTNUMBER}_output_ispell_large.dat
136
137     ### stringsearch
138     ((TESTNUMBER++))
139     RUN_MIBENCH ${TESTNUMBER}_large stringsearch ./search_large >
        ↪ ${RESULTSPATH}/${TESTNUMBER}_output_ss_large.txt
140
141     ######################## security ######################
142     CURRENT_DOMAIN=security
143
144     ## blowfish
145     ((TESTNUMBER++))
146     RUN_MIBENCH ${TESTNUMBER}_large_encrypt blowfish ./bf e
        ↪ input_large.asc ${RESULTSPATH}/${TESTNUMBER}_output_bf_large.enc
        ↪ 1234567890abcdeffedcba0987654321
147     RUN_MIBENCH ${TESTNUMBER}_large_decrypt blowfish ./bf d
        ↪ output_large.enc
        ↪ ${RESULTSPATH}/${TESTNUMBER}_output_bf_large.asc
        ↪ 1234567890abcdeffedcba0987654321
148
149     # rijndael
150     ((TESTNUMBER++))
151     RUN_MIBENCH ${TESTNUMBER}_large_encrypt rijndael ./rijndael
        ↪ input_large.asc output_large.enc e
        ↪ 1234567890abcdeffedcba09876543211234567890abcdeffedcba0987654321
152     RUN_MIBENCH ${TESTNUMBER}_large_decrypt rijndael ./rijndael
        ↪ output_large.enc
        ↪ ${RESULTSPATH}/${TESTNUMBER}_output_rij_large.dec d
        ↪ 1234567890abcdeffedcba09876543211234567890abcdeffedcba0987654321
153
154     ## sha
155     ((TESTNUMBER++))
156     RUN_MIBENCH ${TESTNUMBER}_large sha ./sha input_large.asc >
        ↪ ${RESULTSPATH}/${TESTNUMBER}_output_sha.txt
157
158     ########################  telecomm ######################
159     CURRENT_DOMAIN=telecomm
```

```
160
161     ## adpcm
162     ((TESTNUMBER++))
163     RUN_MIBENCH ${TESTNUMBER}_large adpcm bin/rawcaudio <
        ↪  ${MIBENCH_PATH}/${CURRENT_DOMAIN}/adpcm/data/large.pcm >
        ↪  ${RESULTSPATH}/${TESTNUMBER}_output_large.adpcm
164     RUN_MIBENCH ${TESTNUMBER}_large adpcm bin/rawdaudio <
        ↪  ${MIBENCH_PATH}/${CURRENT_DOMAIN}/adpcm/data/large.adpcm >
        ↪  ${RESULTSPATH}/${TESTNUMBER}_output_large.pcm
165
166     ## CRC32
167     ((TESTNUMBER++))
168     RUN_MIBENCH ${TESTNUMBER}_large CRC32 ./crc ../adpcm/data/large.pcm
        ↪  > ${RESULTSPATH}/${TESTNUMBER}_output_crc_large.txt
169
170     ## fft
171     ((TESTNUMBER++))
172     RUN_MIBENCH ${TESTNUMBER}_large_normal fft ./fft 8 32768 >
        ↪  ${RESULTSPATH}/${TESTNUMBER}_output_fft_large.txt
173     RUN_MIBENCH ${TESTNUMBER}_large_inverse fft ./fft 8 32768 -i >
        ↪  ${RESULTSPATH}/${TESTNUMBER}_output_fft_inv_large.txt
174
175     ## gsm
176     ((TESTNUMBER++))
177     RUN_MIBENCH ${TESTNUMBER}_large_encode gsm ./bin/toast -fps -c
        ↪  ${MIBENCH_PATH}/${CURRENT_DOMAIN}/gsm/data/large.au >
        ↪  ${RESULTSPATH}/${TESTNUMBER}_output_large.encode.gsm
178
179     RUN_MIBENCH ${TESTNUMBER}_large_decode gsm ./bin/untoast -fps -c
        ↪  ${MIBENCH_PATH}/${CURRENT_DOMAIN}/gsm/data/large.au.run.gsm >
        ↪  ${RESULTSPATH}/${TESTNUMBER}_output_large.decode.run
180
181     repeat=$(( $repeat + 1 ))
182 done
183
184 ######################## RT-tests Suite
    ↪  ##################################
185
186 cd ${home_dir}/rt-tests-1.5
187 ./cyclictest -i1000 -l10000000 -h1500 -p90 > output
188 mv ${home_dir}/rt-tests-1.5/output ${RESULTSPATH}/output
```