Secure Virtualization of Latency-Constrained Systems

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Adam Lackorzynski

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig Technische Universität Dresden
Zweitgutachter:	Dr.-Ing. Attila Bilgic KROHNE Messtechnik GmbH
Fachreferent:	Prof. Dr. Christof Fetzer Technische Universität Dresden
Verteidigung:	6. Februar 2015

Acknowledgments

There are many people to thank. First, I want to thank my advisor Prof. Hermann Härtig, especially for his support and encouragement over the course of the years. Moreover, the friendly, motivating and inspiring environment in the operating systems group has been a pleasure to be with. I want to thank Björn Döbel, Marcus Völp, Frank Mehnert and Prof. Hermann Härtig for proof-reading different drafts of this work and giving valuable comments. Thanks also go to Alexander Warg for the continuous (team-)work on our operating system that makes it the basis of so many interesting applications. Last but not least, I want to thank my parents, my grandparents and my brother for their support and believing in me throughout this journey.

Contents

List of Figures

Listings

List of Tables

1 Introduction

A modern world cannot be thought of without computers, they became an integral part of our lives. Computers are used in nearly all facets of our daily routine, aiming to boost our productivity and provide a more enjoyable being. Mobile phones, desktop systems, and laptops are the most visible computing platforms. However, the majority of employed computing systems are embedded systems that are part of a device, spanning from small ones such as pacemakers up to airplanes.

The tasks conducted by embedded systems are manifold, covering control, monitoring as well as entertainment applications. For example, a modern high-end car can have up to 100 Electronic Control Units (ECUs) [Cha+12] to operate all the actuators, sensors and other functionality within the vehicle. This includes convenience functionality such as the entertainment system and the motor-controlled outside rear view mirrors but also critical components such as engine control and brakes. The ECUs themselves are mostly not developed by the car manufacturers but are components bought from different suppliers which have specialized in their specific area. The software running on the ECUs is equally diverse and ranges from self-made systems through widely-used real-time operating systems up to generic operating systems including applications.

All ECUs in such a car must be powered and connected within the car. As each ECU is a computing system on its own, including memory, persistent storage, CPUs, input and output facilities, it requires appropriate power, space and cabling. The number of systems also attributes to the overall weight of the car and thus to the fuel consumption that shall be minimized for environmental and economic reasons. If the many systems could be merged into fewer and possibly more powerful systems, overall power requirements and weight could be reduced.

A second area that uses multiple systems is factory automation, where computing systems are used for controlling machines but also for monitoring and maintaining them. Due to the timing requirements of the control part, this functionality is laid out in an individual system while monitoring and maintenance runs on a different system. Here again, multiple computing systems are used to run the machine and a consolidation into a single system can save energy and cost.

In this thesis I will evaluate how virtualization technology can be used to consolidate multiple systems with latency constraints into fewer systems for the benefit of lowering overall resource usage.

Currently, virtualization is used predominantly in server environments, where it allows multiple systems to be run on a single host machine, in particular when the individual systems are controlled by different users. Virtualization allows to provide full control over each virtual machine to a specific user while isolating the guest operating systems inside the

virtual machine from other guests running on the same physical system. Virtualization is also one of the key enablers for cloud computing, especially for *Infrastructure-as-a-Service* (IaaS) type of systems where customers rent computing capacity in the form of virtual machines. In those scenarios the focus is on efficient usage of available host processing capacity. Isolation is also an important characteristics as customers should not be able to access the data of other customers running on the same physical machine.

Due to the huge diversity of embedded software systems, virtualization is the primary technology to use for consolidation as it avoids re-implementation and, more importantly, re-evaluation of existing software.

However, the latency and predictability required by guests when running in their original environment must be preserved when running in a virtual machine. Only then guests can work from within VMs, for example, to control the brakes and the actuators of an industrial robot without risking the safety of people operating these devices. To the best of my knowledge no current implementation supports the combination of latency and secure isolation in a VM setting.

In the following, I will propose and evaluate an operating system that achieves security and legacy support by running components in virtual machines while providing real-time guarantees to both native and virtualized applications. In particular, my design addresses the following challenges:

- **Generality**: The system shall be applicable to any computing system that offers hardware-based isolation mechanisms. It shall be portable and work equally well on different hardware platforms.
- It shall support **virtualization** to run multiple legacy operating systems without or with only limited need for modification.
- It shall support **latency-constrained** applications, running either natively on the host or as part of the guest system in a virtual machine.
- The **virtualization and isolation overhead** as well as **resource** requirements of the system shall be minimal.
- The system must offer and enforce **security** to prevent unauthorized access to data of other subsystems and VMs.
- The proposed extensions for guests shall have a good **applicability**, allowing for easy integration into existing systems.

As a base platform I have chosen a microkernel-based architecture because it comprises the best characteristics for the outlines requirements. Microkernels run only the absolutely necessary functionality in the most privileged kernel mode of the CPU, all other components are run as user-level components. This is the reason why microkernel-based systems allow to build application-specific small Trusted Computing Bases (TCBs) and thus provide a good fit for security-conscious use cases [Sin+06]. Microkernels also provide an abstraction layer for the hardware and thus provide generic, yet thin, interfaces for portable applications across hardware architectures.

Virtualization is also a good fit for microkernel-based systems as virtualization integrates well into the architecture. Simplified, a virtual machine (VM) is much like a traditional

process that, however, has more state attached that must be handled during runtime: a complete machine state instead of just the registers of threads running in a process. The microkernel only implements the virtualization features required for isolation, while virtual machine monitors (VMMs) are implemented as user-level components to provide the virtual platform. This design follows the principles required for small TCBs because guests only depend on their respective VMMs and in particular not on the VMMs of other guests.

Due to the small kernel and most functionality running in user-level, the preemptibility of the microkernel cannot be negatively influenced by operating system components and applications. Microkernels are therefore a good foundation to support applications with real-time constraints [Här+98].

In my thesis I have integrated and extended an existing microkernel-based operating system with the possibility to execute latency-constrained workloads from within virtual machines. The specific peculiarity of virtual machines is that they not only execute one task but many and that most of those tasks are best-effort jobs. However, on the virtualization layer, the distinction between best-effort and latency-constrained tasks within a virtual machine is invisible. A virtual machine is only represented by one or multiple virtual CPUs. Consequently, for adequately supporting real-time tasks, both in virtual machines and as tasks running directly on the host, mechanisms need to be added to the host system.

The specific implementations of virtualization vary between architectures. Hardware-assisted virtualization has only been added in recent years to mainstream hardware such as x86 and even more recently to the ARM architecture. However, there is also a generic approach that allows to execute legacy operating systems and works on any architecture featuring privileged execution and isolation. I will describe how such a generic virtualization approach can be built for running latency-constrained guests.

When running multiple virtual machines with real-time applications inside it turned out that simple approaches such as high-frequency context switching are not sufficient to achieve the required event dispatching characteristics, see Section 4. However, mixed-criticality scheduling theory is a good fit and can be applied to the question of running multiple latency-constrained virtual machines.

It turns out that additions of up to 30 lines of source code are sufficient to enhance guests systems to make use of the presented mechanisms. The applicability and generality of the presented approach is shown by an evaluation of the two mainstream architectures x86 and ARM.

Outline of the Thesis

This thesis continues with Chapter 2 discussing related work and giving an introduction to microkernel-based systems in general and L4Re in particular. Chapter 3 continues with the changes and improvements requires to build a hardware-independent virtualization solution that is able to run latency-constrained applications. Chapter 4 will introduce the background and mechanisms required for running real-time applications from within virtual machines. Chapter 5 follows with an evaluation and the thesis concludes with Chapter 6 summarizing and giving an outlook for further topics.

2 Fundamentals and Related Work

In this work I want to combine real-time, security and virtualization functionality to build a system that allows to securely consolidate existing real-time systems onto a single system. I am proceeding to introduce into the relevant areas:

- I introduce hardware for running systems on and the resulting challenges due to hardware characteristics for predictability and real-time. Bringing existing real-time systems from their original platform to a platform suited for consolidation also requires to reconsider platform characteristics.
- I review existing real-time systems and show their drawbacks concerning security and isolation.
- I look at virtualization as means to run complete software systems by providing a virtual environment on the host system.
- I present systems that focus on security.
- Finally, I discuss how microkernels appear to be a suitable platform for my work as they have already showed to provide real-time and security guarantees. My thesis then combines these features.

2.1 Platforms, Architectures and Devices

Computing systems span a wide range of diverse platforms, ranging from simple and cheap processing units for special purpose tasks up to systems with many computing cores and magnitudes more of addressable memory. For building systems with virtualization, real-time and security in mind I will restrict to platforms that offer hardware means to support and run software as close to the hardware as possible. Moving existing real-time systems from their original platform to a platform suitable for consolidation also requires to evaluate those platform for their real-time execution behavior. Use of pure software mechanisms to achieve isolation of components, such as use of specific languages and language runtimes [HL07; Coo14] are excluded.

An exception to hardware requirements is the support for virtualizing guests that may need to be modified and adapted to run in a virtualized environment whenever the hardware does not provide virtualization support. For that reason I will address systems with hardware-assisted virtualization support and systems without.

2.1.1 Hardware Architecture

The considered hardware architectures all have the same basic structure. They consist of one or multiple processing units (CPUs) that are logically connected to the same memory (RAM), thus all the cores share the same memory. The system also use multiple different types of caches to speed up the overall system performance. I will go into detail of caches and their influence on the execution behavior in the next section. Another characteristic of the considered multiprocessing systems is that their memory is implemented in a cache coherent fashion. That is, the memory and caching subsystems ensure that all cores always read the same data written by the core that has last modified a memory location. However, the same is not generally true for modifications by hardware devices that are connected to the memory bus but not to the cache subsystem.

For the scope of this work I will use two common hardware architectures: x86 and ARM. Using more than one architecture allows to show the applicability and generality of the presented approaches. The x86 architecture is the standard in the portable computer, desktop and server markets but increasingly also targets smaller systems such as embedded systems and mobile phones. It has the most modern set of features, including support for virtualization, and is easily available. Systems using the ARM architecture are dominant in the growing market of mobile devices, such as smart phones and tablet computers but are also used in all kinds of embedded devices. ARM systems are available in a variety of configurations. In the following I will only refer to those offering virtual memory features, such as the Cortex-A series. Hardware-assisted virtualization [MN10] has been added to the ARM architecture and is available in an increasing number of deployments.

2.1.2 Influence of Hardware on Execution Behavior

The execution behavior of applications largely depends on the hardware platform they are executed on. For timing-critical programs it is thus important to understand the characteristics of the hardware so that timing constraints can be met. When virtualizing existing systems, those are run on different hardware platforms that have different characteristics and thus require evaluation. In the following I will describe hardware characteristics and their influence on the execution behavior of software, in particular with regards to their timing characteristics. I will cover both platform neutral characteristics as well as architecture-specific ones.

Memory Caches

Memory caches are transparent memories[1] that buffer accesses to main memory. They are available in several stages as Level-1 to Level-X, short L1 - LX. In typical existing platforms X can range up to 3, where the L1 cache is the closest to the CPU and L3 the most far away. Small embedded system usually have an L1 cache, more advanced embedded system also have an L2 cache, especially multi-core systems. High-end desktop and server systems also use an L3 cache. The closer the cache is to the CPU the faster it is, at the cost of limiting the size to a few kilobytes. Cache sizes and access times increase with the levels farer away from the CPU.

Caches may be used for different content. If a cache is used for any data it is called a *unified*

[1] Originating from the french word *cacher* ⇔ *to hide*.

cache. In a Harvard architecture the cache is split into a cache for data and a cache for instructions, abbreviated *I-cache* and *D-cache*. This split allows for a better performance by arranging the I-cache closer to the part of the CPU that is fetching instructions and by simplifying the I-cache design as the I-cache can be read-only. In common existing platforms, L1 caches are usually separated in an I-Cache and D-Cache, whereas L2 and further caches are unified caches.

In a multi-level cache architecture with at least two levels of cache several modes of operation are possible within the cache hierarchy. An *inclusive* cache means that data is also contained in all lower caches at the same time whereas *exclusive* operation means that data is in exactly one cache. In practice a combination of both modes is used. The difference of the configurations is about replacement strategies, interaction with other cores, coherency and capacity, as for example, exclusive cache arrangements can store more data than inclusive caches, which in contrast require less effort on eviction.

Caches can be configured with different options regarding their behaviour on access hit and miss events. A cache *hit* is a memory access where the data is available in the cache. A cache *miss* is a memory access where the data is not available in the cache and thus needs to access the memory or cache behind that cache level. A cache hit on a memory write can be handled in two ways: *write-through* and *write-back*. Write-through caches will store the contents in the cache and also directly write the data back to lower cache levels and main memory. Write-back will just write the contents to the nearest cache. The contents will be written to main memory in case a modified cache line is evicted. When a write to a memory location does not have a corresponding entry in the cache, the cache can keep the content in the cache, called *write-allocate*, or it can leave the cache unaffected, called *write-no-allocate*.

A cache is organized in *cache lines*. A cache line is the smallest chunk of memory that is read or written by the cache. The size of a cache line is typically 32 or 64 bytes. Generally the size needs to be at least the size of the biggest register available in the CPU. Along with the cache line itself the cache needs to store information to which location in main memory that cache line refers to. That information is called the *tag*. The *index* is the position of the cache line in the cache. Principally two approaches are possible to map memory addresses to cache locations: using the virtual or the physical address of the memory access. In combination with the index and the tag, four combinations are possible:

Physically-indexed, physically-tagged (PIPT) Both the index and the tag use the physical address. PIPT caches have the benefit of avoiding any aliasing problems where multiple entries in the cache point to the same memory location. Their downside is that accessing the cache requires a virtual to physical address translation, possibly memory accesses due to a potential miss in the translation look-aside buffer (TLB), a cache that caches virtual to physical address translations. Using physical addresses to addressing the cache thus requires more effort including more energy per access than using virtual addresses.

Virtually-indexed, virtually-tagged (VIVT) In a VIVT cache virtual addresses are used to avoid translations. However, VIVT caches suffer from aliasing problems: multiple virtual addresses can point to the same physical address, leading to the situation that multiple cache entries exist for the same memory location. Consistency between those entries must be maintained in software. Another problem is that when virtual to physical address translation changes, for example, by modifying a page table entry, virtually cached data can point to the old physical address. To avoid

this problem the cache needs to be partially flushed on page table modifications. In particular, this implies to flush the entire cache on address spaces switches.

Virtually-indexed, physically-tagged (VIPT) VIPT caches combine the benefits of using physical addresses for the tag and virtual addresses to index into the cache. While the cache is queried using the virtual address, the virtual to physical translation for the tag can be done in parallel. When both operations are successful a cache entry has been found. As the virtual address is used for the index, the cache can contain multiple entries for the same physical memory location which must be handled by software.

Physically-index, virtually-tagged (PIVT) Only a theoretical construct that is not useful in practice.

A prominent representative of an architecture implementing VIVT caches is the ARMv5 architecture. Due to the requirement of flushing the cache upon each page-table switch those architectures are not suited for systems that perform frequent process switches, such as microkernel-based multi-server systems. Especially when running legacy real-time systems in a virtualization context, where address space switching is required among different guests due to isolation requirements, the switching overhead might be prohibitive. Modern architectures, including ARMv6 onwards and x86, implement physically tagged caches avoiding those flushing problems and thus being friendlier to systems that change processes frequently.

Multiple configurations are possible when mapping an address of a memory access to a cache location. The simplest one is the *direct-mapped* cache where each memory address has a fixed location in the cache. On the other end of the scale is the *fully-associative* cache which is able to place data in any cache-line size location. Generally, fully-associative caches are not used for large caches as their hardware implementation is costly. The cache must be able to lookup all cache lines in parallel and reach agreement on which cache line hits or whether all cache lines miss upon a request. Typical implementations usually use *n-way-set-associative* caches that allow to place a datum in n possible cache lines, with n being up to 8, as this is the best compromise between implementation cost and runtime performance.

Associative caches need a replacement strategy to decide which cache line to evict out of the n possible cache lines. Ideally, the cache line should be evicted which will be be the least useful in the future. As caches cannot look into the future they use algorithms such as Least-Recently-Used (LRU) or variants thereof to determine which cache line to evict.

The execution performance of code running on the system depends not only the processing unit itself but also on the transfer capacity and latency of the memory bus and the caches. In this regard the cache plays an important role as it significantly reduces the access time to data that is required by the processor. Typically, access to the main memory is about two magnitudes slower than the L1 cache [HP11, p. 72, Chapter 2.1].

This large difference in memory accesses has significant influence on the execution behaviour. Especially in multi-tasking environments the contents of the cache may vary and lead to substantially different execution timing. When the system is on load with different programs being multiplexed concurrently, caches will be filled with the working set of each program running. When switching to the next one, most of the cached data from the previous run will have been evicted (*cold cache*) and are required to be repopulated. For programs with latency requirements this has the consequence that in the worst case all caches are cold and all data, including instructions, need to be fetched from main memory first [Meh+01].

Additionally, depending on the cache configuration, the cache may contain data from other programs that still need to be written back to main memory, adding additional delay. For a worst case estimation, cold caches need to be considered when estimating the execution behaviour of a program.

To avoid or at least reduce the effect of the cache the following options are available:

- Switching off the cache enhances predictability but also tremendously reduces the execution speed of programs, especially of best-effort programs that do not have any tight latency requirements. Additionally, in Symmetric Multiprocessor Systems (SMP) caches cannot be disabled as they are required for maintaining memory consistency across cores.
- With multi-processor systems, one or more cores can be dedicated to run tasks with latency requirements exclusively. This allows exclusive use of core-local caches. However, the tasks need to make sure to only use core-local caches and further cache levels that span multiple cores can also have influence, for example, with inclusive cache setups. In an overall view, dedicating cores in a multi-processor system to specific tasks can also lead to a low utilization of the system.
- *Tightly Coupled Memory* (TCM), also called scratchpad memory, is a memory that can be accessed as fast as cache memory and can be available as an optional feature, especially in embedded processors. The size of the TCM is usually in the range of 256KiB. The operating system can maintain this memory and allow latency-constrained tasks to place code and data in the TCM to benefit from cache-like access performance without eviction. However, placing the task's data and code there is not enough as all the operating system functionality that is required for the latency-constrained task would also need to be placed into the TCM. Due to the very limited size of the TCM this requires very careful layout of the code and data structures. We have evaluated the use of TCM in a microkernel-based environment [Hes+08].
- Systems may offer the possibility to lock cache lines and thus prevent their eviction. This avoids that the program for which the cache lines have been locked, is stalled due to cache misses. However, the approach also takes away cache capacity from other running programs.
- Systems with latency-constrained programs might want to use write-through caches instead of write-back caches to avoid stalling their execution due to write-back of data loaded by other programs.
- Cache coloring [LHH97] is a technique that assigns memory to applications in such a way that memory of different applications map to different parts of the cache. This way no other program can evict cache lines. However, cache coloring requires knowledge on how the cache maps addresses to entries in the cache.
- As already mentioned the ARMv5 architecture suffers from the problem that its caches must be flushed upon every address space switch. To counter that the ARMv5 architecture offers the Fast-Context-Switch-Extension (FCSE). FCSE uses 128 process IDs (PIDs) to map the lower 32MB of the address space to unique regions of the whole 4GB address space and thus avoids cache flushes when switching between these processes. As the number of domains is limited to 128, an operating system needs to provide means to handle more tasks in an appropriate manner, such as dynamically

swap tasks between an FCSE and normal mode. The limitation to a 32MB address space also requires specifically built applications and thus the benefits of FCSE cannot be used transparently. FCSE has been implemented in a microkernel-based system and in Linux [Wig+03; CC09].

Translation Look-Aside Buffer

The Translation Look-Aside Buffer (TLB) caches translations from virtual to physical addresses. Operating systems maintain a translation table, called page table, for each task in the system which maps virtual addresses to physical page frames. A page is a fixed size and size-aligned region of virtual or physical memory. A page table is a multi-level data structure which is stored in memory. Current implementations of page tables contain two to four levels, requiring two to four memory accesses for each virtual to physical translation. As programs run with virtual addressing, a translation is needed for each memory access as well as for each instruction fetch. Caching those translations is beneficial as they are needed frequently.

With any change of the page table, the corresponding TLB entry needs to be invalidated. When setting a new page table base pointer, the TLB will be flushed by the processor to invalidate any old entry. As TLBs are crucial for performance, some architectures implement *Address Space IDs* (ASIDs) which store an operating system defined tag with each TLB entry. This allows the MMU to differentiate between different address spaces and thus avoids flushing the TLB upon address space switches. A prominent representative that is implementing ASIDs is the ARM architecture starting with ARMv6. The x86 architecture does not offer IDs for address spaces but Intel offers IDs for virtual machines, called Virtual Processor Identifiers (VPIDs) [Cor14, Volume 3, Chapter 28.1].

MMUs usually offer support for multiple page sizes. To reduce pressure on the TLB it is beneficial to use big page sizes as far as available and usable. This allows to cover bigger areas of memory with fewer TLB entries. However, this also requires that the memory of the used page size is available. For example, on the x86-64 architecture, the choice can be made between 4KiB, 2MiB and 1GiB pages, where requiring 2MiB of memory for a single program section often already wastes a considerable amount of memory. The ARM architecture is more flexible in this regard and offers more page sizes, such as 1KiB, 4KiB, 64KiB and 1MiB.

Latency-constrained programs should use big pages as far as possible to reduce their usage of the TLB and thus also to reduce TLB misses, especially right after address space switches. However, with using bigger pages programs are also likely to occupy a larger amount of physical memory.

Platform Induced Latency

Hardware platforms as a whole increasingly use software to implement features. For example, recent systems provide legacy devices such as PS/2 keyboards with software, using keyboards connected via the USB bus. This allows to transition from older hardware technology to new technologies without keeping legacy devices while remaining compatible with older software. In such cases performance does not play a key role so that emulation in software is sufficient.

In the following I will describe sources of latency that can be built into hardware platforms:

System Management Mode (SMM) is a mode that is built into x86-based processors [Cor14, Volume 3, Chapter 33.1]. Software running in SMM is provided by the Basic Input/Output System (BIOS) and aims to be completely transparent to the operating system. It is for example used to provide legacy devices such as PS/2 keyboard and floppy drives with devices connected via USB. Experiments have shown that SMM may cause tremendous stalls in execution of the processor. For example, when inserting an USB storage device, experiments showed that a randomly selected system might run in SMM and not be interruptible for as long as 500 ms.

Most common x86-based systems make use of SMM, not only for legacy device emulation but also for other required system maintenance tasks, with scarce details available on their behavior. Consequently it is not generally possible to disable SMM. Latency demanding software running on such systems should therefore disable "USB device legacy emulation" in the BIOS, if available. If disabled, no keyboard is available in old operating systems or simple software, such as DOS or boot loaders. Modern operating systems will re-initialize the USB sub-system and have drivers for all common USB devices, including keyboards. Any USB connection event will be handled by the operating system and not by SMM. However, the operating system is required to initialize the USB host controller using a USB host driver, which is considerably more complex and bigger than a driver for PS/2 keyboard and mouse.

If disabling legacy device emulation is not offered, the operating system must initialize the USB subsystem to bring it under its own control. The other option is to avoid any plug-in of USB devices to the platform while latency-constrained programs are running.

BIOS/UEFI The BIOS on x86-based platforms is responsible for setting up the system and launching the boot loader which then finally launches the operating system. The BIOS also offers services which can be used by programs, however, those are only available when running in 16 bit mode which has long been obsoleted. Modern operating system run in 32 or 64 bit mode, independently of any BIOS functionality.

UEFI (Unified Extendible Firmware Interface) is the successor of the BIOS [Coo]. Besides platform initialization and boot services UEFI also offers runtime services which are available while the operating system is already running. The UEFI implementation is delivered by the platform provider. Any software must rely on a proper behaviour of UEFI functionality if used, especially under timing constraints. Open-source implementations, such as Coreboot [Coma], can provide viable solutions when the runtime behavior of the supplied platform firmware is not sufficient.

Microcode is part of the processor itself and used to implement complex functionality within the CPU. Microcode can be updated [Cor14, Volume 3, Chapter 9.11], for example, to correct errors in the processor. As such microcode can be a source of latency for executing code. However, as an integral part of the processor unduly long execution behavior from the microkernel is unlikely.

Trustzone is a technology available in some ARM processors to offer two worlds, a *normal world* and a *secure world* [Lim14]. A typical usage scenario is that the normal operating system, for example Linux, is running in the normal world while the secure world offers security services. For that the secure world is running a small operating system which can be called from the normal world. For security reasons certain hardware functions, such as cache maintenance operations, are only accessible from the secure

side, requiring the secure world operating system to offer an interface for those to the normal world.

Implementations of the secure world may come pre-installed with the device, not allowing them to be disabled. Examples are the OMAP line of ARM systems by Texas Instruments [Ins].

Implementations of the secure world may also allow to integrate custom software modules. Examples are the 'Trusted Foundation™ Software' by Trusted Logic®. Latency-constrained software running in the normal world must make sure that the secure world does not perform any regular task that may interrupt the execution of the normal world. Furthermore, the normal world software must not call services on the secure side when execution behavior of that functionality is not known.

System designers should either choose a platform without pre-installed Trustzone software, or, given they require such a functionality, choose a system with a known execution behaviour.

Built-in Virtualization comes pre-installed with the system, for example to offer remote administration features, and cannot be disabled or circumvented. At the time of writing, no such systems are known, however, technology exists and is emerging to build such systems. As systems with a built-in and closed Trustzone secure world component exist today, desktop and mobile system might contain similar technology in the future.

System designers of real-time systems should avoid such systems, especially if they want to use hardware virtualization features of the platform requiring the built-in virtualization solution to support nested virtualization. This will lead to increased and likely unknown latency for any virtualization relevant operation in the system.

Device Firmwares are software programs or even complete operating systems that are running on devices and implement their functionality. The program code is either stored on the device or loaded to the device by the driver running on the host operating system. The firmware is running on the device, independent of the main processing cores. However, the device and the main processing cores might share common resources, such as the memory subsystem. This can lead to congestion on the memory bus and thus lead to increased latency for the main processor cores. As the firmware can be supplied by the device driver, an update of the driver might also change the behavior of the device regarding its interaction with the system or system software. Latency-constrained systems must therefore pay attention to driver and firmware updates in the system.

Devices that have direct memory access (DMA) capabilities can access to whole physical memory, giving devices the possibility to manipulate the system. Malicious drivers can gain access to unauthorized system areas by using a modified firmware. Modern systems posses IO-MMUs to limit the possibilities of access to main memory to predefined regions, avoiding erroneous memory corruption or attacks [Cor11].

Power Management techniques are used to reduce the power consumption of a system during runtime. Two common techniques are used to achieve a reduction in power consumption of a system. The first one is to switch off components when they are not required. The second is to control energy consumption with frequency and voltage scaling where available, foremost with CPU cores. Depending on system load the performance and thus also the power consumption can be controlled.

Using power management in latency-constrained systems must be considered carefully. For example, x86-based processors offer a set of sleep states that put the processor into power-saving modes. The different modes are a trade-off between power-savings and latency of transitioning to and from the mode. The greater the power-savings, the more time it requires to enter the mode and, more importantly for latency considerations, the more time it requires to restore a working state. Latency-constrained systems must therefore be aware that power-saving modes contradict quick event response times. Often, operating systems offer means to avoid power-saving modes for the benefit of shorter event latencies.

Multi-Core Systems

In a shared-memory multi-core system the operating system typically runs on each core and uses hardware-provided cross-core communication mechanisms to implement operating system features as well as providing (transparent) cross-core communication functionality for applications. Multi-socket systems are likely to have memory connected to each socket while accessing that memory from either socket is transparent, yet the performance characteristics are different. Thus, applications with latency requirements must be aware of the specific characteristics of such a system and further, the operating system must offer functionality to the applications to adjust its behavior and configuration to such a system.

The performance characteristics for message passing operations differ significantly depending on whether the two threads are running on the same core or on different ones. Further, the distance of the cores, for example whether they are on the same socket or on different ones, also has an influence on the communication latency. Multi-threaded applications, that require an upper bound on their communication latency must therefore ensure that their threads are appropriately placed on the physical cores and that this placement can only be changed explicitly by the application itself. In this case the operating system must not transparently migrate threads between cores.

Concerning accessing memory, similar reasons exist for the requirement to pin threads to cores. In a multi-socket system memory accesses depend on the distance of the core to the used memory and thus a migration of a thread can change memory access characteristics.

Multi-core systems can also be beneficial in setups with real-time applications. Given the system provides enough cores, real-time applications can be executed on a dedicated core so that they do not need to share core-local resources such as TLB and L1 caches with other applications. However, further caches (L2, L3) are typically shared among cores within a socket and thus are still a shared resource. This can be brought further to separating applications to different sockets, where no caches are shared. However, the downside is a largely underused processor.

Latency Induced Through Virtualization

Hardware-assisted virtualization allows to run unmodified guests in a virtual machine. Whenever an interrupt triggers, the virtual machine is exited and the host system can handle the interrupt. However, the guest operating system inside the virtual machine can execute any valid system-level instruction including any long-running ones. Examples of such instructions are the `WBINVD` instruction on the x86 architecture which writes back and invalidates modified cache lines of the whole cache. We have studied the behavior of those

instructions in a setting with AMD's SVM-based virtualization [SLW09]. Measurements showed that the `WBINVD` instruction inside a virtual machine on a particular system can run up to over 600 µs. However, x86-based virtual machines can be configured to exit to the host system before any such instructions are executed. In a setup with timing constraints the host system shall make use of those intercept possibilities and implement the functionality with an interruptible algorithm.

On the ARM platform, operations that run on the whole cache have been removed from the ARMv7 architecture [Lim14]. System software must accomplish cache maintenance covering the whole cache by using *set/way* cache operations within a self-implemented loop. This avoids long-running instructions and allows the cache maintenance to be interruptible.

2.2 Real-Time Systems

Real-time systems are systems that are able to guarantee timely execution of *tasks*. According to a schedule a real-time system selects tasks so that their timing constraints are fulfilled. The schedule is created by an admission where all constraints of all real-time tasks in the system are used to compute whether a given task set can be scheduled at all, and if, how the schedule looks like. Non-real-time tasks in the system are executed in the slack of real-time tasks, the time that is left after real-time programs have been executed.

The most prominent resource to be scheduled is the CPU, that is the point in time and the duration a task is scheduled to run on a core. Other resources that can also be scheduled, especially in setups with combined real-time and best-effort clients, include disks [Reu05], network [Lös06], and buses [Sch02].

Real-Time System Variants Operating systems implementing real-time functionality are usually called *real-time operating system* (RTOS). Three main groups of implementations exist:

1. Non-isolating systems do not use nor require virtual memory and thus provide no protection against malfunction in any task. With no hardware protection mechanisms used, those systems can run on a wide range of platforms. Platforms may offer a simple form of memory protection via *Memory Protection Units* (MPUs) to partition memory. A benefit of those hardware platforms is the runtime predictability as the whole system is run within the same operating mode that is never changed and caching mechanisms, such as TLBs, are not required nor available.

2. Co-located approaches hook into an existing multi-tasking operating system. The low-level interrupt path is modified to branch to the real-time handler which will run real-time tasks. The multi-tasking kernel will be executed in the idle loop of the real-time dispatcher. With this approach, real-time programs are executed within the privileged mode of the CPU (2a). Extended versions allow user-level applications with the benefit of leveraging user-level software environment (2b).

3. Multi-tasking, multi-user operating systems can run multiple applications isolated from each other using hardware mechanisms. The operating system kernel provides functionality that allows its running programs to obey timing requirements.

One main difference of the three groups is their use of specific hardware features. Group 3 is the most demanding, requiring a CPU that offers multiple privilege levels and, as real-time programs are run as user-level applications, means to isolate programs from each other while allowing efficient and fast context switching. Systems of group 2a require virtual memory capabilities. However, as the real-time programs run within the kernel-privileged mode and in any currently running context, the isolation features are only used for non-real-time programs. Assuming the operating system kernel offers adequate preemtibility, real-time programs are not affected by the behavior of non-real-time programs. Approaches of group 2b use user processes and thus virtual memory to run their real-time tasks and thus benefit from efficient context switching in the same was as group 3 systems. The least demanding group is the first one that does not require any privilege levels nor isolation features.

Hardware Platforms The selection of a hardware platform for use in the context of virtualization, real-time and security, makes the availability of hardware functionality for isolation indispensable. At least two privilege levels and virtual memory that can be efficiently handled, are an absolute requirement to isolate components from each other. Additional hardware-provided functionality to support virtualization is beneficial to reduce the virtualization overhead in both runtime performance and engineering effort. However, they are not an absolute requirement.

Considering the x86 platform, the whole line offers the same feature set of functionality: User-level and kernel-level privilege modes and virtual memory. The ARM processor line is more diverse and offers a wide range of CPUs with different capabilities but with a homogeneous instruction set architecture (ISA). For example, considering the Cortex line of CPUs, they range from the Cortex-M0, an implementation of the ARMv7 ISA with just 12,000 gates [ARM] and no virtual memory nor privilege levels, over the Cortex-R line offering an MPU with between 150,000 and 290,000 gates depending on the configuration [Tur10], up to Cortex-A line with virtual memory including address space IDs and multiple privilege levels. Later Cortex-A implementations also add support for virtualization and the 64-bit ARMv8 instruction set.

In the context of this work, the x86 and ARM-based Cortex-A processors can be used as they are featuring the required functionality.

2.2.1 Mixed Criticality Systems

Mixed-criticality (MC) systems play an important role for this work. Mixed-criticality systems are a special form of real-time systems which combine tasks of different criticality. The criticality of a task determines the worst-case execution time (WCET) to be used in admission: the more critical a task is, the higher the assurance level required, the more pessimistic the assumptions to be made and hence the longer the WCET. A comprehensive overview on MC systems is given by Baruah et al. [Bar+11].

In the following I will explain the MC scheduling problem as far as required for this work. Each task T within an MC system is analyzed at its own criticality level and all the criticality levels of all other less critical tasks in a system. Concerning scheduling, the resulting problem then is whether the *actual* execution times of all tasks of a given criticality level λ and of higher criticality stay below the WCET of all these tasks as determined for λ. If so, all these tasks must meet their deadline. Otherwise the completion of lower critical tasks than λ is no longer guaranteed.

The MC scheduling problem can be explained using an example for tasks with three levels: low, medium and high. For brevity and readability I will further refer to tasks T^H, T^M, T^L, as high, medium, and low criticality tasks respectively and to $C(H)$, $C(M)$, $C(L)$ as high, medium, and low WCET estimates. With those three levels, the scheduling criterion is:

- If the execution times of all high-level tasks remain within their high-level execution times, they will meet their deadline.
- If the execution times of all high-level and all medium-level tasks remain within their medium-level execution times, they will meet their deadlines.
- If the execution times of all high-level, all medium-level and all low-level tasks remain within their low-level execution times, they will meet their deadlines.

In other words, if a low-level task exceeds its low-level execution time, it must not lead to high- or medium-level tasks to miss their deadline. Generally, for the WCETs of any task, the following holds: $C(H) \geq C(M) \geq C(L)$.

2.2.2 Admission

Admission is the process of planning the jobs in a system so that they can be run by the system with the available resources. In a real-time system, an admission service will only allow further jobs if the resources for that job are available, preventing an overload of the system.

Admission can be *online* and *offline*. Online admission is done at runtime, the system can dynamically permit newly created jobs as well as handle the termination of jobs. Offline admission is done with a fixed set of jobs at setup time of the system and yields a fixed schedule that is then executed by the system.

In both types the execution behavior of the jobs must be known to the admission service. For real-time systems this comprises the worst case execution time (WCET) and the highest allowed latency.

For Virtualizing Systems the admission is two-layered. The *Local Admission* is the admission that is done by the guest in the virtual machine, with the local knowledge of that guest only. The *Global Admission* then combines the schedules of the guests and the host to an admission for the whole system.

2.2.3 Real-Time and Devices

Up to now I described real-time in the context of the processor alone, that is handling of processing time. However, the processor is not the only resource in the system that is shared between multiple clients. Devices such as disk controllers with their attached disks and network interface cards are commonly shared among multiple client programs on the system. The operating system has to offer means to multiplex and abstract the services offered by those devices in a safe way.

Considering applications with timing requirements, the device drivers need to provide guarantees concerning the service they offer to the application. Additionally, the devices

themselves should behave in a deterministic way to allow execution time estimations. Providing device services in combined real-time and best-effort systems has been researched in the past [Reu05; Lös06]. As devices and their drivers also need to be considered in real-time systems that use virtualization I will briefly introduce their role in the virtualization stack.

Generally, applications with timeliness requirements can be located in either a component running as a native program on the host kernel, or, as part of different programs, in a VM, as depicted in Figure 2.1. Here, the interesting setup is the VM. Any real-time program inside the VM makes the VMM a real-time program in the host context, that is real-time requirements for device accesses must also be considered in the VMM. If the VMM has further knowledge about the device requests it receives from the VM it can use two channels to a driver and issue real-time and non-real-time request to the device driver, allowing for better resource usage by the device driver.

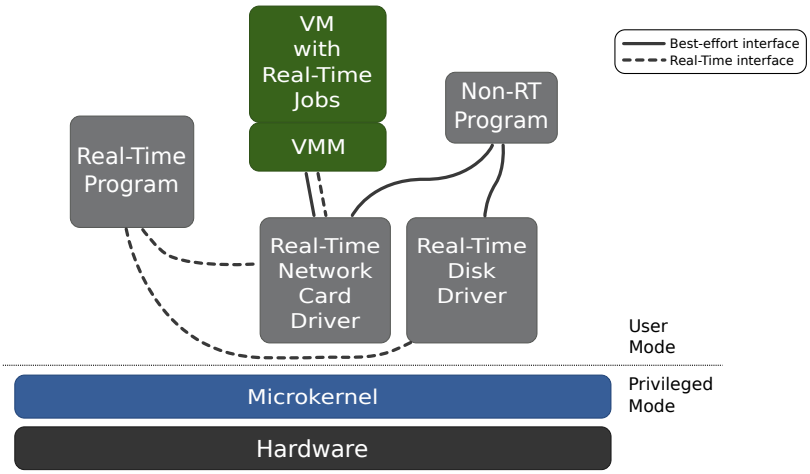

Figure 2.1: Setup of microkernel-based system with real-time applications, virtual machines with real-time parts and device drivers. The dashed lines indicate a real-time interface and communication while the straight lines indicate the best-effort interface.

2.3 Security Focused Systems

Systems that involve multiple parties, such as different virtual machines or multiple users, require that the host provides means to protect the system and each party thereon from potentially malicious behavior of one or multiple parties on the system. The system can be a single computer or, in a broader sense, multiple systems that are connected over a network.

In today's world, where computing systems are connected to other systems over the Internet, security plays an important role in every system design. Thus any system is required to provide a security concept. Research has, free of practical constraints of existing systems, brought forward systems that are designed with stringent security requirements from the ground up.

The basic principle of information security is to maintain the following three properties: confidentiality, integrity, and availability. *Confidentiality* requires that data is not available to individuals or systems that are not authorized to access this data. *Integrity* means that any unauthorized modification of data is detected. *Availability* means that data can be accessed when it is needed. Any system, which is concerned with an all-embracing security model,

bases its design on those three principles. The classical work by Saltzer and Schröder [SS75] describes how to handle the protection of information on a system with multiple users.

Operating systems need to offer proper functionality for applications to provide the three security properties. Additionally, the operating system also needs to be tamper-resistant against modification and against attacks to gain unauthorized access. Eventually this boils down to the likelihood of programming errors in the operating system code, as this code is running in the most privileged mode of the system. Any successful penetration of that code allows the attacker to take over the system. This observation leads to the conclusion that small kernel designs, such as microkernels, are more appropriate for security concerned systems than monolithic kernel approaches. Studies of monolithic systems have shown that programming errors are significant and require consideration [Cho+01; Pal+11a].

One of the main reasons for programming errors in operating systems are the used programming languages. All widely used operating systems are written using the C or C++ language, with additional assembly code for hardware-specific operations not available in the language itself. C and C++ are well suited for programming as they closely model the type of execution of the hardware. However, they also allow direct manipulation of memory (pointers) and modification of the executed code. This gives programmers the flexibility to write efficient code, however, subtle programming mistakes, for example with regards to handling buffers in memory, can lead to erroneous code and thus incorrect or even insecure execution behavior. This may make the code prone to attacks from non-authorized components on the system. Singularity [HL07] is an operating system that is based on language security. Software components are written in the type-safe language Sing#, derived from C#. As no hardware protection is required in this system, the software isolated processes (SIPs) run in the most privileged mode of the processor, making their handling light-weight. Of course the compiler and the runtime have to ensure the isolation between the components. The downside of such an approach is that every component needs to be written in a particular type-safe language and thus already existing software cannot be used.

Another approach to rule out any programming mistakes in the operating system itself is to formally validate the code of the operating system against a well defined model. The small size of a microkernel makes its verification practically possible. The VFiasco project [HTS02] worked with the L4/Fiasco kernel. seL4 is the first formally proven L4 microkernel [Kle+09].

The design of the security model that is used by the operating system is of relevance as well. Two major paradigms are commonly used to implemented access control: *Access Control Lists* (ACLs) and *Capabilities* [Tan08, Chapter 9.3]. Both types utilize subjects and resources to describe access permissions. With ACLs resources posses lists of subjects which describe which subject is allowed to access the resource, possibly accompanied with an access type. A typical representative of this approach are UNIX-like systems, where the right to access a file is stored with the file itself. On the contrary, in capability systems subjects posses the right, a capability, to access a resource. As with ACLs a capability might include specific access rights for a resource. A comprehensive overview on ACLs and capability system is given by Miller et al. [MYS03]. Generally, capability systems are regarded as being superior over ACLs, especially as they allow to implement the *principle of least authority* [SS75], where each subject should only have those privileges which are necessary to complete its job. Systems implementing a capability-based security model are, for example, EROS [Sha99], seL4 [Kle+09] and Fiasco.OC.

Summarized, security-focused systems shall prefer a system employing a small kernel and using a capability-based access control mechanism. Those kernels are written in a C/C++-

like language for efficiency reasons and provide an execution interface that allows to run legacy software. Small-kernel systems can be closely evaluated concerning their security properties.

2.4 Virtualization

According to Popek and Goldberg [PG74] a *virtual machine* (VM) is "an *efficient, isolated duplicate* of the real machine". The virtualization is provided by a *virtual machine monitor* (VMM), which shall have the following three characteristics:

- the environment provided by the VMM for the VM is as identical as possible to the original system (→ *duplicate*),
- the performance shall not suffer (→ *efficient*), and
- the VMM shall be in full control of the resources of the system (→ *isolated*).

Although they published their work in 1974 and systems making use of virtualization have been used since then [Gum83; MS70], virtualization only gained momentum about a quarter of a century later when it entered the mainstream market. The Disco project [BDR97] started the ongoing interest in virtualization and spawned several widely-used virtualization solutions such as VMware [Bug+12], Xen [Bar+03], KVM [Kiv+07] and Hyper-V [Mic].

By that time the x86-based hardware got powerful enough to run guest operating systems in virtual machines. However, the initial x86 architecture is not virtualization friendly because it has flaws in the instruction set, so-called virtualization holes, that break the *trap-and-emulate* approach as described the Popek and Goldberg criterion [AA06]. Trap-and-emulate requires that any *sensitive instruction* traps when executing the instruction in a less privileged processor mode so that it can be handled by the VMM. Virtualization holes make it impossible to implement a VMM solely based on the technique of trap-and-emulate. However, with additional techniques such as guest code inspection, virtualization can also be implemented on architectures with virtualization holes.

Over the course of the evolution of virtualization several techniques have been developed and combined. I will introduce them in the following sections.

2.4.1 Parts of Virtualization

Generally, virtualization covers a whole system. Going into more detail, several areas need to be covered: Virtualization of the CPU, the memory and devices. CPU and memory are usually handled together whereas providing virtualized devices and the multiplexing to real ones can be handled separately.

CPU Principally there are two possibilities to provide a virtual environment for a given instruction set architecture (ISA). The first is the emulation of each instruction, building the whole semantic of each instruction in software. This has the benefit that any ISA can be handled on a host, including detailed execution behavior, however, it induces a significant performance overhead. An example for an emulator is Bochs [Law96]. The other possibility is to execute the code of the guest on the host, called native execution. This requires an ISA which fulfills the trap-and-emulate criterion. If that criterion is not fulfilled, several

slight modifications are available to circumvent the virtualization holes.

If the guest operating system kernel can be modified, any instruction which does not trap on sensitive operations can be avoided and replaced with trapping instructions and hypervisor calls. This approach is called *paravirtualization* and a popular representative of that approach is the Xen hypervisor, see 2.4.4. When the guest cannot be modified, the virtualization layer has to detect virtualization holes, for example by analyzing the guest code for instructions that do not trap. Popular for that kind of technique are VMware [Bug+12] and VirtualBox [Ora].

Memory When an operating system uses virtual memory, its kernel will manage page tables for each process and describe therein which part of the physical memory is accessible to this process. However, in a virtualized environment the host system has control over the physical memory and a guest operating system is not allowed to modify its own page tables in the host. Consequently the virtualization layer needs to virtualize the use of virtual memory in the guest operating system. Several options are available. With the paravirtualization approach, where the guest can be modified, the operating system kernel will use host functionality to construct the virtual address spaces of its processes. In the case where the guest cannot be modified, two principle approaches are available. The first approach uses the possibility that setting a page table base register traps out of the guest kernel and thus the virtualization layer can inspect the page table of the guest and construct validated page table entries. As this approach behaves like a TLB it is called *virtual TLB* (vTLB). A second common name is *shadow paging* because the host keeps a second page table of the guest's one. Shadow paging incurs a significant performance overhead because of using traps and inspecting the guest's page table in software [SK10]. To overcome this penalty, processor vendors have added a mechanism generally referred to as *nested paging* [Adv08]. With this approach the MMU of the processor is capable of using the page table of the guest and one for the VM provided by the VMM to translate guest virtual addresses to host physical ones. Nested page tables have been the key feature to get the performance of VMs very close to the performance of non-virtualized systems.

Devices In the same way a computer system communicates with the outside world using devices, such as a network card or a keyboard, a VM requires a similar infrastructure to communicate outside of its VM, either with other VMs or the outside world. To run unmodified guests, the VMM needs to supply emulations of hardware devices, for example an interrupt controller, a timer, a network card and a storage device. One device emulation per device class is sufficient, the particular type is mainly selected by ease of emulation and availability of device drivers in popular guest systems.

As the communication between the operating system and the device has been designed for IO memory or IO ports, this interface is not ideal in a virtualized environment. Therefore, the performance of the guest can be greatly improved by using special purpose drivers inside the guest that are specifically designed for optimal communication with VMM components. Another possibility is the so called *device pass-through* where a VM is permitted access to a physical device and can thus use it directly, without any virtualization layer in between. This results in a better performance, however, also needs prerequisites considering security aspects and the device cannot be shared with other VMs or system components. Devices are accessed using IO memory and access to this memory is granted through page tables. Page tables have a minimal granularity, called a page, which requires two devices to not

have their IO memory within one single page. If that is the case a VM would have access to both devices and if that is not desired then device pass-through cannot be used.

Another point to consider is the way a device and the processor exchange data with the device. *Direct Memory Access* (DMA) is a commonly used mechanism that allows the device to transfer data between itself and the main memory without interaction of the main processor. This increases the performance of the system as the processor does not need to copy data to and from the devices itself. For the copy operations the device drivers need host physical memory addresses which they pass to the device. The devices can then access this memory to read and write data. However, the access of the memory is not restricted. A device can access any memory in the host. As the device driver programs the device it has the possibility to access any physical memory. In the case of device pass-through to a VM, the VM is running without host privileges and shall not have access to memory not assigned to the VM. To prevent such attacks, and to improve failure resistance, several options are available. In a software-based approach a trusted component must be developed to validate any DMA-related request for invalid addresses [Meh05]. Another possibility are *IO-MMUs*, memory management units for IO devices. IO-MMUs are implemented in the device buses and are programmed by the host system. They are similar to the MMU in the processor, adding virtual memory for devices. Only memory pages granted access to by the host system are accessible for the device, making it impossible to access non-granted memory. Ideally an IO-MMU is available for every DMA-capable device in the system, however, this is not usually the case and multiple devices may need to share one IO-MMU.

Due to the increasing use of virtualization, hardware device vendors are adding virtualization support to their devices. The most prominent example are network cards. Device pass-through is a very efficient way to make a network card available to a VM, however, one card per VM is required. With an increasing number of VMs per host it is not possible to plug as many network cards into the system. To overcome this limitation, devices may expose their functionality through multiple virtual-function devices, allowing the host to pass one of those virtual functions to a VM. The devices themselves implement functionality to aggregate the accesses from the VMs to a single physical device. For the PCI bus this approach has been standardized under the name *Single Root I/O Virtualization* [SIG]. In combination with IO-MMUs this allows efficient use of devices from within VMs without sacrificing security of the system.

Techniques to protect the system from misbehaving or malicious devices and drivers are not only useful for virtualization. Any system that runs device drivers in user-level needs means to protect the host system from faulty or malicious components. Generally, all software architectures that run driver in non-privileged modes, such as microkernel-based systems, benefit from the presented techniques.

2.4.2 Implementation Options

In this section I will introduce how virtualization can be implemented, focusing on the x86 and ARM platforms.

Both platforms initially suffered from virtualization holes, making a pure trap-and-emulate approach impossible. Choices to overcome this limitation with software approaches are the adaption of the guest system or using additional means to address the virtualization holes. For unmodified VMs several virtualization solutions exist that apply techniques to overcome the architectural limitations [Bug+97]. The adaption of the guest is limited to the operating

system kernel. Examples are the Xen system (see Section 2.4.4) and L^4Linux. I will introduce L^4Linux in Section 2.7.2.

With the increasing popularity of using virtualization, hardware vendors have integrated specific support into their hardware to aid and simplify the implementation of virtualization software as well as improve virtualization performance. Generally those additions plug the virtualization holes in the ISAs but also add features specifically targeted to the virtualization use case.

In the following I will briefly introduce and describe the virtualization related functionality of the x86 and ARM architectures. For the x86 architecture, Intel and AMD have come up with technically similar approaches but differing implementations, requiring that virtualization software must implement both variants to support both processor vendors. On the Intel platform the virtualization support is called VT-x, while the name VMX (virtual-machine extensions) is also popular. On the AMD platform the virtualization extension is officially called *AMD Virtualization™* (AMD-V), however, the term *secure virtual machine* (SVM) is also common. Virtualization is supported by all current processors by Intel and AMD, except a few models by Intel which do not have VMX enabled. On the ARM platform, virtualization support has been introduced with the Cortex-A15 processors. A technique called TrustZone has been available starting with ARM11 type processors, allowing to run two operating systems on one processor, one controlled by the other. We have explored how to multiplex multiple guest operating systems with TrustZone [Fre+10]. A complete documentation can be found in the processor manuals [Cor14; Dev11; Lim14].

As already outlined in Section 2.4.1, virtualization support is provided in the categories of CPU, memory and devices.

CPU Both the x86 and ARM architectures introduce new execution modes specifically targeted at running guest virtual machines. Those modes do not suffer from virtualization holes and allow to run guests at native near processor speed without any requirement to detect virtualization holes.

The x86 architectures uses a hardware-defined data structure that describes the configuration of a *virtual CPU* (vCPU). With VMX it is called *virtual-machine control data structure* (VMCS) and with SVM *virtual machine control block* (VMCB). I will use the term *VMCx* as a generic abbreviation whenever no specific variant is required. A VMCx is a page of memory and contains the complete CPU state of a virtual machine. A VMM needs to set up one VMCx per vCPU prior to launching a VM. Main choices are to configure exit conditions, that is conditions when the CPU shall stop executing the VM and return to host execution. Such an action is called *VM-exit*. Launching a VM is done using the explicit VM start instruction which takes the address of a VMCx. Multiplexing multiple VMs on a host CPU is possible using different VMCx's, however, special care might be necessary to flush cached state, depending on the hardware implementation. Any hardware interrupt hitting a host CPU will lead to a VM-exit and thus allow the host system to take over control.

The ARM architecture introduces a new execution mode called *Hyp* mode for executing VMs. The host has to configure the virtual machine through control registers. Loading and storing VM state is up to the virtualization software as no hardware data structure has been defined. The host system is interrupted when device interrupts occur so that it can regain control over the CPU.

Memory The key issue with memory is the virtual memory used by the guest. As already described in 2.4.1 the concept of nested paging is used to avoid expensive handling of guest virtual memory in software using a vTLB.

The main difference between VMX and SVM on the x86 architecture is the choice of the page table format. VMX introduces a new page table format whereas SVM uses the standard page table format also used for normal paging. A pointer to the host page table is stored in the VMCx to be used by the VM.

The ARM architecture allows to set a page table for the VM. To be able to address more than 4GB of physical memory in 32 bit systems, ARM introduced a new page table format called *Large Physical Address Extension* (LPAE). This is especially interesting for the virtualization use case where the host may want to run multiple VMs and a 32 bit address space does not suffice.

Devices Device virtualization is termed as *Intel® Virtualization Technology for Directed I/O* (VT-d) for Intel systems [Cor11] and AMD-Vi for AMD systems [Dev09]. Both subsume the ability of the platform to provide an IO-MMU and the possibility for interrupt remapping. Generally, any platform using the *Peripheral Component Interconnect* (PCI) together with devices capable of initiating DMA requests is required to have IO-MMUs to guard against malicious or faulty devices or device drivers.

Contrary to the x86-based platforms using PCI for devices which VMs shall have direct access to, ARM systems primarily use the concept of dedicated DMA engines. Since they are separate from the device itself, access by a driver can be virtualized and direct programming by a device driver can be prohibited. However, providing a virtual model of a DMA engine requires engineering effort and is slower than direct access and thus ARM systems also offer IO-MMUs.

2.4.3 Software Components for Virtualization

Virtualization software must provide the environment that enables a guest system to run. All of the three previously described areas, CPU, memory and devices, must be covered by the software. Commonly, the software implementing virtualization is named *hypervisor* or *virtual machine monitor* (VMM) and both terms are used interchangeably. Two major design concepts exist:

Type-1 hypervisors implement a dedicated kernel that exclusively handles virtual machines. An example for a type-1 hypervisor is Xen[Bar+03].

Type-2 hypervisors are part of a general purpose operating system which allows to run both applications and virtual machines. An example for a type-2 hypervisor solution is KVM[Kiv+07].

For the purpose of this work I want to use the terms hypervisor and VMM for specific and distinct functionality. Reflecting a virtualization architecture in the context of a microkernel-based system, the hypervisor is implemented in the microkernel. Still the microkernel remains an operating system kernel and thus the hypervisor is a role, among others, that the microkernel provides. Virtualization functionality that must be implemented in the host kernel are those that must run privileged or are required to be in the host kernel for

isolation and security reasons. All other functionality shall be implemented in a user-level component, the VMM. However, the distinction is not strict, as for performance and interface transparency reasons it might be reasonable to implement more virtualization functionality in the host kernel.

The VMM is the user-level component that primarily configures and controls VMs and implements virtual devices. A system can run multiple VMMs independently. Figure 2.2 depicts a possible setup of such a system. The hypervisor is a functionality of the host kernel and the VMMs are user-level components which are running one or multiple VMs. The VMMs can be different implementations that target different use cases.

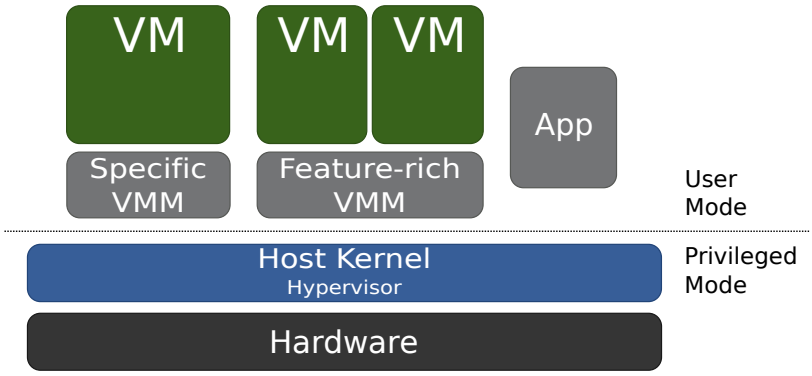

Figure 2.2: A microkernel-based virtualization architecture with different kind of virtual machines monitors (VMM). The host kernel only provides necessary virtualization functionality upon which VMMs of different feature sets are based on.

2.4.4 Existing Work

The following presents a selection of existing work and solutions in the area of virtualization.

Xen Xen [Bar+03] is one of the early adopters of virtualization technology on the x86 platform. It started with a paravirtualization approach and added support for hardware-assisted virtualization later when this technology was becoming available. Xen is a widespread solution and as it is available as open source, it is also the base for many other virtualization solutions, both in academia and industry.

The Xen architecture consists of the Type-1 Xen hypervisor and a management VM, called *Dom0*. Dom0 is always a paravirtualized VM running Linux [Comb] which runs the management tools and is in charge to control the other VMs, called *DomU*. DomUs can run paravirtualized VMs and, if supported by the hardware, unmodified guest VMs.

From the security perspective, the trusted computing base of any VM running on a Xen system consists of the Xen hypervisor itself and the complete Dom0 management VM, being a Linux kernel together with the user-land infrastructure running on it. This increases the trusted computing base compared to a monolithic Linux system and thus Xen does not provide the necessary design for a secure system foundation I envisage.

KVM Kernel Virtual Machine (KVM) [Kiv+07] is a Type-2 virtualization system, adding a virtualization functionality to the Linux kernel and making it a hypervisor. KVM builds

upon hardware support for virtualization and started with the x86 architecture. For device virtualization KVM uses an adapted version of the QEMU system emulator [Bel05]. Any VM running on a Linux with KVM needs to rely on the Linux kernel and the adapted QEMU driving the VM. Analogous to the reasoning for Xen, the trusted computing base includes a monolithic kernel design and is thus not suited for a secure system.

VMware Mobile Virtualization Platform The VMware Mobile Virtualization Platform (MVP) [Bar+10] targets embedded and mobiles devices, most prominently smartphones. Their primary use case is to run a second Android operating system on the mobile phone to cover contradicting goals of a private and business usage of the phone. The OS for private use shall be as open as possible and allow to install any third-party application. On the contrary, the business OS shall be as secure as possible and controlled by company IT.

The architecture of MVP uses a standard Android configuration and places a hypervisor module into the host Android Linux kernel, making the host system a type-2 hypervisor. Several programs are executed in the host to provide functionality to the guest system, such as virtual private networking and storage. MVP emphasises the BYOD (Bring Your Own Device) approach, which requires easy adaptability of the system to any mobile device and as such favours the approach of adding hypervisor functionality to an existing system. The host system runs the private Android and the guest system runs the enterprise version. Consequently the enterprise Android has to rely on the private Android system, that is the trusted computing base of the enterprise Android system includes the open private Android system.

MVP relies on the security properties of the host Android system. Besides relying on the Linux kernel, the user interface and application starting subsystem must be also trusted as for example the switching between the domains is done with a special application.

NoHype NoHype [Sze+11] is an architecture that statically allocates CPUs and memory to virtual machines and lets a single virtual machine run exclusively on a number of cores. The system uses state-of-the-art Intel processors to leverage virtualization features such as nested page-tables. During startup Xen is used to bring a virtual machine into a fully running state. When the startup is completed, the hypervisor is exchanged with a very minimal one that will kill the virtual machine on any virtual machine exit event. Consequently the VM must be fully working without causing any exit events. As this architecture removes the traditional hypervisor during runtime of a VM there is also no attack vector a VM could use to break out of the virtual machine and attack other VMs or the whole system. NoHype uses a slightly modified kernel to mitigate problems where the hardware provides no means to guard against misbehaving other virtual machines, such as with inter-processor interrupts, which according to the authors, posses no problem to the applicability of their solution. Their target use case is secure virtualization in the cloud-computing area where VMs can be statically allocated to the hardware and overprovisioning is not required to better utilize the available hardware.

Concerning running real-time workload in a virtual machine, the NoHype architecture is interesting, as no intercepting hypervisor also means that it cannot induce any additional latency. Only the latency through the hardware virtualization functionality is added (see [SLW09]). NoHype requires to have a CPU local timer which for the Intel platform is the APIC timer that would need to be sufficient for the VM workload. NoHype is exclusively targeted at virtualization and the architecture does not allow to run any additional services

besides virtual machines where each virtual machines occupies at least one physical CPU at all times, even if the VM is idle.

CloudVisor CloudVisor [Zha+11] is a virtualization system that targets the cloud environment, aiming to reduce the trusted computing base of each VM running in the system. The threat model describes VMs that exploit the VMM to gain access to other VMs as well as the cloud provider that can use management tools to inspect customer VMs. To prevent that, CloudVisor uses secure booting techniques to plug a tiny security monitor below the VMM and run it in guest mode, using nested virtualization. Nested paging is used to hide the guest memory from the VMM and each VM exit and entry is trapped by CloudVisor to adapt the nested page table. VM requests on storage devices are detected and transparently encrypted by CloudVisor, presenting only encrypted data to the VMM.

The CloudVisor architecture targets common virtualization systems and introduces an isolated layer between the VMM and guest so that a, potentially penetrated VMM, cannot access data from within a VM. It runs systems with a single VMM and, due to the encryption of storage data, induces an overhead of up to 54.5%. Secure booting techniques shall ensure that a guest must only trust CloudVisor and not the VMM. CloudVisor might be a solution for existing VMM systems, however, a system should include security considerations from the ground up and integrate those in the system.

SPUMONE SPUMONE [Nak+11] is a virtualization system for multi-cores that is developed on the SH platform. It runs guest systems in privileged mode. This offers benefits with regards to the event latency achieved by the system up in to the guest. The low engineering effort required to bring a guest OS to the platform is also put as an advantage of SPUMONE. Regarding devices the platform requires that devices can be exclusively given to a guest, any possible sharing cannot be prohibited and can be done cooperatively among the guests. As the guest kernels are running in the privileged mode of the processor, they cannot be protected from other subsystems using an MMU. Only the user processes on the guest system use virtual memory protection. SPUMONE proposes to use core-local memory to run code on a core. Core-local memory is not accessible from other cores and thus provides protection from code running on other cores. However, this requires that all code running on a core fits into the core-local memory, which size is only a few hundred kilobytes and thus does not suffice to run for example Linux. To support guests that do not fit entirely into the core-local memory, SPUMONE uses a swapping mechanism with hashes to exchange memory pages between the core-local memory and the main memory. Code is only executed in the core-local memory and the hashes are used to check the integrity of the memory when it is copied to from the main memory to the core-local memory. SPUMONE is not able to protect the virtualization layer itself on the core nor other guest running on the same core. A monitor service is proposed to check integrity. This monitor must run on a separate core so that it cannot be tampered with. To protect guests from each other the system can only support one guest per core.

Cells The Cells architecture [And+11] targets mobile phones by providing multiple virtual phones on a single device. It uses virtualization based on operating system mechanisms, such as namespaces and device proxies, to run multiple instances of mobile phone software on the same kernel. This approach has the benefit that all functionality related to device drivers, noteworthy 3D acceleration, does not need to be virtualized and can be used in any

of the virtual phones. The downside is that all the virtual phones depend on the proper working of the Linux kernel and any attack on Linux puts all virtual phones on risk.

NOVA The NOVA microhypervisor [SK10] is a tiny hypervisor. It uses the mechanisms provided by the x86 architecture to implement virtualization and splits the functionality of a traditional VMM into the hypervisor, which runs in privileged mode of the CPU and handles CPU based virtualization, and the actual VMM which handles device virtualization. Using one VMM per VM, the VMM is just exposed to the VM it handles but not to other VMs. Proper isolation is ensured by the NOVA microhypervisor. NOVA outperforms other virtualization solutions, showing that splitting hypervisor-functionality from the VMM does not induce any performance penalties.

2.5 Real-Time, Scheduling and Nested Systems

In this section I will give an overview on related work that covers previously introduced topics, especially those combining multiple of them such as real-time and virtualization.

2.5.1 Scheduling Theory with Stacked Systems

In scheduling theory the handling of multiple subsystems, which use a scheduler itself, has been of interest. Because in those systems two schedulers are stacked on each other the scheduling theory is referred to as *hierarchical scheduling*.

Fundamental results in scheduling theory have laid the grounds for later research of stacked subsystems. Among this ground work are the rate monotonic scheduling algorithm [LSD89], work on schedulability of real-time systems using static priority preemptive schedulers [Aud+93], combining hard real-time, periodic tasks using EDF and soft aperiodic requests [SB96], and the constant bandwidth server (CBS) [AB98].

More recently, scheduling theory has also focused on hierarchical setups, where a base systems runs multiple other systems on top of it. In their terminology, the scheduler in the host resides on a *global* level, whereas the subsystems, also called *applications*, schedule on a *local* level. Scheduling with both the global and the local schedulers being fixed priority and preemptive has been described by Davis and Burns [DB05]. Using EDF at the local level and EDF or a fixed priority scheme at the global level has been researched by Zhang and Burns [ZB07], applications can be both periodic or sporadic. All analysis have been restricted to single CPU systems.

2.5.2 Real-Time and Virtualization

Virtualization systems have been specifically evaluated for their use in real-time setups. Gandalf [IO08b] is a virtualization system that has been evaluated for its real-time use [IO08a]. Proteus is a system for the PowerPC architecture which can virtualize several guest systems [BK09] and provides partitions with temporal isolation [KBG10]. No spatial isolation of virtual machines is provided. Kinebuchi et al. [Kin+08] use an L4 variant as a hypervisor in an embedded environment with a paravirtualized real-time operating system and evaluate the system by measuring achieved latencies. With regard to scheduling, tasks and their

threads in guest systems are exposed to the host. Follow-up work of Kinebuchi developed a new system [Kin+], called SPUMONE, running on a SH-4A platform. The system is able to run virtualized systems and targets real-time use cases. The authors explicitly state that, for their target use cases, security can be ignored for better performance of the system. The system is described in more detail in Section 2.4.4. Xtratum [CRM10] is a bare-metal hypervisor that targets avionics environments and allows to run multiple paravirtualized guest operating systems, among them Linux and RTEMS, in a statically partitioned system. It supports scheduling schemes such as ARINC 653 as required for avionics use cases.

RT-Xen [Xi+11] extends the Xen Hypervisor with four different and known real-time scheduler algorithms, allowing to run guest operating systems under one of those real-time schedulers. The scheduler implementations are compared among each other and the authors conclude that RT-Xen only gives a "moderate overhead" using 1kHz scheduling frequency.

2.6 Terminology

Basic concepts of computing are used throughout computing industry and academia, however, different communities may also use different wording for the same concept. Thus, in the following, I want to define terminology and wording and reflect it against terminology used by different communities within academia and industry and finally I want to specify the terminology used in this thesis.

Real-Time and Operating Systems The real-time community as well as the microkernel community are using the same words with different meanings:

Task In the real-time context a task is an entity that executes work, which is executed in a thread provided by an operating system. In the microkernel context a task describes an container that is implemented with address spaces and may contain threads and other task-local resources, such as capabilities. To avoid confusion I will not use the term task without specifying the meaning in the context of real-time and operating systems.

Server In the operating system terminology and also within networking environments the term server refers to a provider of a service, which a client can use. A server can also be a client of other services. Contrary, in real-time scheduling a server subsumes a group of tasks that are scheduled under a scheduler.

Addressing in Virtual Environments Any type of virtualization, starting with virtual memory up to nested paging, involves different types of addresses. *Physical addresses* are those addresses that are used on the memory and devices buses. To provide flexible memory handling and isolation of software components, *memory management units* (MMU) are used to provide an abstraction from physical memory. *Virtual addresses* are translated to physical ones by the MMU, the translation table is maintained by the operating system. Considering virtualization, a VM has its own notion of physical and virtual memory, however, a VM gets only access to a subset of the actual available memory in the host. In virtualization context the terms *guest virtual address* (GVA) and *host virtual address* (HVA) are used. To denote the physical address of the host, the term *host physical address* (HPA) is commonly used in this context.

A special role is taken by paravirtualized guests, such as L^4Linux. In Linux kernel differentiates between virtual and physical addressing, however, in the context of paravirtualization three different types must be used. The physical addresses need to be divided into ones used for devices, thus being the real host physical addresses, and those used internally in the L^4Linux kernel for addressing, being virtual in the context of the L^4Linux kernel running in a host address space, and being different from those virtual addresses used for Linux processes. Operating system kernels typically do not offer support for differentiating between the two types of addresses and special care must be taken when adding paravirtualization support to those kernels.

2.7 Microkernel-based Systems

Microkernel-based systems promise to be a good fit for security and real-time conscious environments. Their design principle of only running absolutely necessary code in the most privileged mode of the CPU and building any functionality on top as non-privileged components allows to build system with small *Trusted Computing Bases* (TCBs) [Hoh+04]. The composition of a system through multiple modules allows to design a system in such a way that a component only needs to depend on those other components which functionality it requires. Any other functionality available in the system cannot influence this component, especially regarding the existence of faults in those components that can lead to the infringement of confidentiality, integrity and availability demands. This is in contrast to monolithic designs where a broad set of functionality is implemented in the host kernel and part of any feature implemented on top. Any programming fault in a monolithic kernel has direct influence on all components running on the kernel, whether or not this component requires the affected kernel component. As the number of programming errors correlates with the amount of code [Pal+11b], it can be assumed that systems which allow a small TCB are better suited for security conscious systems. Due to their small size, microkernel-based systems also offer the possibility to be evaluated with formal methods to prove that their code adheres to a formal specification, ruling out any programming errors that would violate the specification. An initial step is to provide a proof of the kernel which has been accomplished in the seL4 project [Kle+09].

Considering real-time systems the predictability and preemptiveness of the system are of interest. Similar to the security argument for microkernel-based systems, less dependencies reduce the complexity of the system and thus make the predictability of the system better understandable. At the lowest level the kernel controls the preemptiveness of the system, requiring that the kernel provides a good interruptability to be suitable for real-time use. However, user components still need to be designed with latency requirements in mind to be able to react to events appropriately. DROPS [Bau+98] is a system that has been specifically designed to build real-time systems using a microkernel-based design.

Microkernel-based systems have gone through several generations, starting with the Mach microkernel [Acc+86]. The approach to Mach was to strip down an existing monolithic kernel and allowed to implement microkernel-based design concepts as well as running guest operating systems, such as MkLinux [PSR96]. However, Mach suffers from slow IPC performance, attributed to the asynchronous IPC mechanism used in Mach. This observation lead to the design of second generation microkernels [Lie95], namely L4, that use a “from scratch” implementation and put their main focus on IPC performance. Comparisons between Mach and L4 show that a design from scratch can tremendously improve performance of the

overall system [Här+97]. Security considerations led to the design and development of the third and current generation of microkernels. With the introduction of using a capability-based design, any resources in the system, including communication, can only be used when being in the possession of an access right to that resource. Contrary to second generation systems, third generation kernels provide the possibility to isolate subsystems in such a way that those subsystems do not and cannot know from each other, except when explicitly configured to do so [LW09]. At the time of this writing third generation microkernel systems provide the best choice for designing security and real-time conscious systems.

Due to the familiarity with L4-based systems I chose the TUD:OS system for the remainder of this work. I will introduce this system in the following.

2.7.1 The TUD:OS System

The *TU Dresden Operating System* (TUD:OS) is a microkernel-based system, consisting of the Fiasco.OC microkernel and the *L4 Runtime Environment* (L4Re). L4Re is a flexible, portable and component-based software layer that runs in user-space on top of the Fiasco.OC kernel, providing runtime services such as memory management, to be able to run applications. Existing applications can be run in a port of the Linux kernel to the system, called *L^{4}Linux*.

TUD:OS evolved out of DROPS [Bau+98], that used the Fiasco microkernel and *L4 Environment* (L4Env) as its user-level services. L^{4}Linux was one of the first applications developed within the DROPS project.

Figure 2.3 shows a general architecture overview of a TUD:OS system. The Fiasco.OC kernel is the only component running in the privileged mode of the processor. All other components are implemented in isolated, non-privileged tasks, depicted as separate boxes.

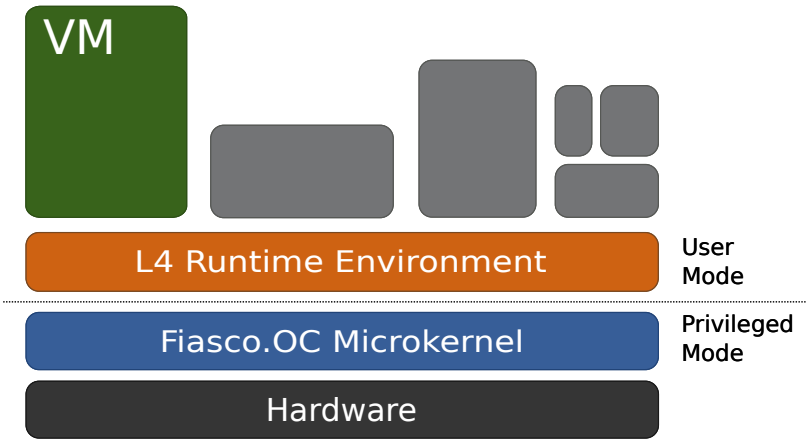

Figure 2.3: General architecture of a TUD:OS system. The kernel is the only component running in the most privileged processor mode. All other functionality is implemented in services, running in non-privileged and isolated tasks.

Fiasco.OC

The kernel is the only software component running in the most privileged processor mode of the system. Being a microkernel, it only covers basic and essential functionality, foremost ensuring isolation of components running on the kernel. Consequently the kernel must run

any platform functionality that requires execution in privileged mode. For example, this covers setting up address spaces and dispatching hardware interrupts. Other functionality is located in the kernel for usability reasons. For example, scheduling of threads is implemented in the kernel to ensure proper performance of the system.

Access to kernel functionality is guarded using a capability system [LW09]. Any functionality in the kernel is modeled as an object, and any access to such an object using a system-call is directed through a capability table. Each protection domain in the system has its own capability table, which is maintained by the kernel. This table contains pointers to kernel objects which a protection domain can invoke. If a capability table does not contain a pointer to a particular object, the protection domain of that table cannot access this object. This scheme implements the principle of least authority (POLA) [SS75; SSF99].

Factories are the concept through which objects in the system are created. The kernel implements factories to create kernel objects, such as protection domains or threads, or other factories that are equipped with less resources. The factory concept is also used to create user-level objects.

Overall, the kernel covers the following key functions:

Protection domains By providing different protection domains the kernel ensures that code running in different domains cannot influence each other. This protection is achieved by using the memory management unit provided by the platform and creating address spaces for each domain, called tasks. An address space is an abstraction from physical memory, building virtual memory. The kernel implements such an address space by constructing a page table, which is a data structure used by the MMU to map virtual addresses to physical ones, and thus to memory areas. Each entry in the page table also carries a set of flags describing memory access characteristics, such as read-only permissions or caching behavior. Using one page table for each task gives the kernel the ability to grant each task access to specific memory areas. Access to other memory is not possible as no entry in the page table exists that points to that memory. Only the kernel is able to set a current page table in the system as well as to modify the page table.

For user-level programs the kernel is offering a *map* operation that allows to give memory a program possesses to other programs, building up a hierarchy. The memory can be given out with the same rights or with less rights. The hierarchy denotes parent/child connections where a parent is responsible for maintaining the memory of its children. Shared memory between tasks is established by giving multiple tasks access to the same physical memory. The system has a root memory component that possesses all memory and which is the start of the memory mapping hierarchy.

Units of execution One unit of execution is a thread. A thread executes code and is subject to scheduling by the kernel. It is bound to a task and therefore an address space wherein it executes, multiple threads can be within the same task. The kernel offers a mechanism to migrate threads between CPUs.

Communication plays an important role within a microkernel based system. When all services are implemented in different tasks, good performing communication mechanisms become essential to the performance of the whole system. The system supports multiple ways of communication.

Message passing is an operation where two threads explicitly invoke a message passing operation and possibly exchange a limited amount of payload data. In L4 systems, message passing is called IPC, short for *Inter Process Communication*. The name is misleading but traditionally used, as threads are the communication partners rather than processes. For reasons of simplicity in the kernel, IPC is synchronous, meaning that when one partner has not invoked an IPC system call the other partner will block until both have invoked the function. Synchronous IPC avoids to queue up any messages in the kernel, which would require buffer handling therein. Both partners may choose to abort the operation specifying send or receive timeouts when one partner is not getting ready to communicate in time. Several combinations of sending and receiving are also available.

Another way of exchanging data, and thus communication, is to establish shared memory between two protection domains. The kernel supports this by allowing a task to share memory it possesses with other tasks.

Asynchronous notifications are supported by using interrupts that are used both to implement real physical device interrupts as well as soft interrupts implemented by programs. An interrupt can be triggered either by a hardware device or by a software component. Unlike message passing, the receiver of an interrupt needs not be ready when the interrupt occurs, it can fetch it at any later time. Interrupts do not transfer any payload except the interrupt itself. Shared memory is often accompanied with interrupts to signal changes of shared memory contents.

The relationship of two communication partners is established using a capability system that can create communication channels between domains and give them access to the channels. These channels must be explicitly created and maintained, a creator can only build those channels between domains to which it has access. It is the duty of the creator to enforce a given communication policy for its children. This way only allowed communication channels are created, all other cross domain communication is prevented automatically as those domains cannot address other communication partners.

Scheduling Scheduling is the functionality that decides which thread to run next. Typically, the scheduler is invoked periodically by a timer event, or when a thread is being blocked and another ones needs to be selected. Generally, the decision, which thread to select, can be placed anywhere in the system, especially a user-level component. However, user-level scheduling for the whole system induces a performance penalty and thus scheduling is implemented in the kernel [Sto07].

The kernel implements a fixed priority, round robin scheduler that always selects the thread with the highest priority that is ready to execute. Within one priority level a round robin scheme is applied. The length of a timeslice can be configured for each thread separately. The scheduler functionality is also used to put threads on different cores that might be available in a platform. Multi-processing systems run one scheduler per core and perform core-local scheduling. User-level policy can instruct the kernel to migrate threads between cores.

Besides the described key functions the kernel also provides optional debugging facilities such as serial output and an interactive debugger as well as other functionality that must be implemented in there, including interrupt controller and hardware virtualization functionality.

Runtime Environment

As described in the previous section the kernel only implements basic core functionality itself and offers possibilities to enforce policies that have to be implemented in user land. The runtime environment implements those policies and also offers services and libraries that abstract from the pure kernel interface and add functionality that is commonly used by programs and developers. The environment consists of services running in tasks, called servers, and task local functionality provided by libraries.

The system server, which is the root of the system, provides several services, including memory provision for clients, naming services and a simple read-only file system. Application loading is done by an application loader that interprets scripts that launch applications and set up communication channels between those.

For productive and convenient application development the runtime environment provides several sets of libraries. Message passing calls for communication with the servers are wrapped in libraries that are used by the programs [WL11]. Standard functionality, for example a subset of POSIX, pthreads and a C++ runtime environment are provided by corresponding libraries such as a C library.

Device Access

The classical RTOS allows every task to access all devices and thus trusts all applications to handle the platform devices properly, or not to access them at all. In the contrary, in a system with protection domains the operating system must prevent that one untrusted subsystem can control or negatively influence the platform, for example because it has access to platform-critical devices.

Considering that the operating system must control the platform leads to the conclusion that a central platform management component is needed in the system. Given client-specific configurations it will hand out access rights for the available devices. Clients of the platform manager are components that need to run devices for themselves, for example a protocol stack, or a system component that multiplexes a device for multiple clients. A good example for this type of usage is a network device driver that also implements a virtual switch to handle network connections of multiple clients in the system.

Clients may use device functionality either by calling a specific interface offered by the device driver component, or they are expecting a hardware device, in which case the device can be emulated by a device model, typically implemented in a virtual machine monitor. Modern devices designed with virtualization in mind offer several virtual devices that can be passed through to multiple clients independently, avoiding the need for a slower software implementation. Devices or controllers that use direct memory access (DMA) must be specially treated when they are driven by untrusted components as the DMA functionality gives access to the physical memory and therefore to all protection domains and the kernel. Hardware provided IO-MMUs allow to restrict access to physical memory for these devices. The platform management component is responsible to setup the IO-MMU hardware for use.

2.7.2 L^{4}Linux

L^{4}Linux is a port of the Linux kernel to the L4 microkernel system family. The first publication dates back to 1997 [Här+97] where Linux 2.0 was ported to the initial L4 kernel by Jochen Liedtke. With the hardware architecture available in that time, L^{4}Linux showed an application performance that was close to native performance and showed that the unfavorable performance characteristics of MkLinux [PSR96] were not due to the microkernel idea itself but due to design considerations with and within the Mach kernel [Acc+86].

In the following I will introduce L^{4}Linux and provide sufficient knowledge for the subsequent work with L^{4}Linux. L^{4}Linux was among the first solutions to provide virtualization on common of the shelf hardware. To run Linux on L4, the Linux kernel is adapted and wrapped into an L4 task and executed in user-level. For each Linux process L^{4}Linux creates an L4 task so that the Linux processes are isolated among each other and the L^{4}Linux kernel itself is also isolated from its processes. All Linux processes are fully controlled by the L^{4}Linux kernel by using available L4 mechanisms.

The first port of Linux to L4 had the goal to evaluate the generality of the L4 interface [Lie96]. L^{4}Linux showed to be working and performing well. However, several peculiar solutions had to be taken to provide the required Linux semantics and binary compatibility. For example, each Linux process required a *signal thread* to allow the L^{4}Linux kernel to handle signals for the process [Här+97].

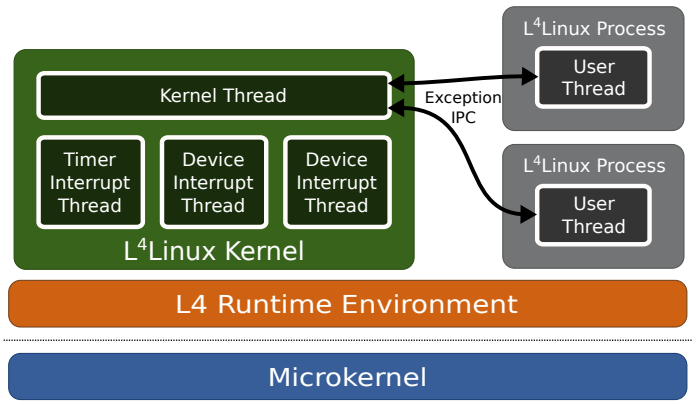

Figure 2.4: Architectural overview of L^{4}Linux using threads. Besides the kernel thread, separate threads for each interrupt are required as well as for each user process.

With the availability of the Fiasco kernel [Här+98], interface adaptions and enhancement of the kernel interface were possible. This allows for an interface co-design between the kernel and the application. Subsequent work enhanced the Fiasco interfaces to allow for a better exception handling and control of L4 threads which in turn simplified L^{4}Linux [Lac04].

At this point, L^{4}Linux is using one L4 thread per process. A schematic view is shown in Figure 2.4. Within this setup, only one thread is running at a time as this matches the expectations of an operating system kernel. The newly introduced exception IPC mechanism allows to retrieve and set the full CPU state of a thread. User processes are executed by replying to their previous exception IPC with a new state and waiting until they trigger the next exception. For handling preemption, the L^{4}Linux has to interrupt a currently running Linux process. This is done in the interrupt context of Linux by injecting a special preemption exception into the Linux process. The Linux process will then return to the

L^{4}Linux kernel via an exception IPC, allowing the L^{4}Linux kernel to run internal work, including scheduling.

Compared to the initial version of L^{4}Linux, no changes in the memory handling were done or were deemed necessary. Internally, L^{4}Linux uses Linux' page tables to track the page allocation to the Linux processes. L4 mappings are established upon faults of user-level programs by looking up the corresponding memory page in the Linux-internal page table. Access right removals, including page removals, are done immediately.

Device interrupts are handled with interrupts threads that are running in the L^{4}Linux server. Initially, an L4 thread could only receive from a single interrupt, requiring one interrupt thread per device interrupt in L^{4}Linux. Fiasco added the capability of being able to wait for multiple interrupts, requiring only a single interrupt thread. For generating the Linux-internal timer interrupt, an L4 thread, similar to the interrupt threads, is used.

Initially, the base synchronization primitive was disabling processor interrupts, which was allowed by the microkernel. Later work addressed the security and latency implications of allowing to disable processor interrupts by user-level and added a user-level-based synchronization mechanism to L^{4}Linux. A lock is used to implement the critical section and a helper thread is installed to allow a thread to block whenever the lock is already taken [HHW98]. The helper thread has to respect the priorities of the blocking thread when waking them up to preserve expected behavior [Meh05].

L^{4}Linux has been integrated into the TUD:OS system by drivers that connect to L4 services, such as input devices, frame-buffer, network[LH04] and block devices [Reu05]. L^{4}Linux is being continuously upgraded with Linux releases and available for the x86 and ARM architectures.

2.8 Summary

In this chapter I have presented fundamentals and related work required for securely consolidating real-time systems using virtualization techniques. TUD:OS will be the platform used for the integration.

3 Generic Virtualization

Running virtual machines with latency-constrained work requires that the base system offers adequate support for such a configuration. The software platform must be designed so that events can be dispatched with low overhead to and within virtual machines. Virtual machines shall be supported with a uniform interface for different virtualization technologies as well as different architectures so that virtualization software can support a wide range of different configurations. Still, latency is not the only consideration. The design must also include proper isolation so that multiple virtual machines and applications can run concurrently on the same system without being able to negatively influence each other both in the spatial and temporal domains. And finally, as we are extending an existing microkernel-based system new mechanisms shall be integrated into the system such that it works well with existing applications. New virtualization functionality shall also only extend the trusted computing base for unrelated programs where absolute necessary which requires that virtualization functionality is split into a security relevant part and user-level components.

As the base for this work I chose TUD:OS, a universal L4 microkernel-based operating system. As already outlined in 2.7.1 it provides the necessary features of a small trusted computing base together with required isolation properties. It supports multiple architectures and platforms as well as the possibility to run legacy systems.

With the initial development of L4 by Jochen Liedtke there was the first user-level application on L4: L^4Linux. One of the goals of the L4 interface design was, and still is, to be universal in a regard that any type software can be built using this interface. With porting Linux to L4 it had been shown that the L4 interface is sufficiently generic so that a complete operating system including its applications can be hosted. While developing L^4Linux, microkernel interface changes where explicitly not considered [Här+97]. While Linux could be run on L4 and performance was close the native Linux using application-level benchmarks, some rough edges remained.

In the course of on-going development those limitations were addressed and consequently improved L^4Linux. The initial design has been discussed in Section 2.7.2. In this Chapter I will proceed with a new virtualization interface that addresses an yet unresolved issue with preemptibility and thus low latency.

3.1 The Base System

The base system providing the core operating system functionality includes the host microkernel, Fiasco.OC, and the L4 Runtime Environment, L4Re. In the following I will briefly introduce and analyze its characteristics as a versatile platform, supporting latency and

security requirements.

The host kernel is the only component running in the privileged mode of the processor and thus plays an essential role to the overall behavior and security properties of the whole system. Besides security properties, the duration and latency of kernel operations is important for the real-time behavior of the system.

Security Properties The host kernel provides isolated compartments for software components. Both spatial and temporal isolation are required. Spatial isolation is provided by means of address spaces. Each compartment in the system has multiple address spaces, targeting different resources. The virtual memory address space is enforced by hardware and the capability address space is protected by the kernel. Further address spaces may be provided, for example, on the x86 architecture I/O ports form an own address space.

To provide temporal isolation the host kernel implements preemptive multitasking and ensures that the scheduler can preempt running threads to select other ones which are ready to run. The scheduler implements a configurable policy and allows to prioritize threads over others.

The enforcement of resource sharing must be implemented by the host kernel. The kernel itself manages the processor by means of the scheduler and memory by controlling the page-tables, as outlined. As described in Section 2.7.1 the kernel provides *map* and *unmap* operations to grant and revoke access rights to memory. Access rights to memory pages can only be granted with the same or less privileges. The same mechanism is used analogously for capabilities. Access to other resources is managed by user-level components, for example, for hardware devices, however, the host kernel needs to supply a communication channel for components to interact securely with each other. Fiasco.OC supplies IPC-gates for that purpose which provide a communication channel between two tasks, whereas other tasks in the system cannot interfere with that channel.

Security-related considerations need to continue in the user-level environment. Implementing resource access policies for resources for multiple compartments requires appropriate care. Negligence during implementation may lead to resource access for unauthorized compartments.

Latency Properties For latency-constrained programs it is important that the blocking time of host kernel invocations and calls to other user-level components is known. Two groups exist in this regard. The first group are operations that always take a fixed amount of time, independent of previous operations executed by other components. The second group are operations that depend on some system state and thus their runtime is not constant and potentially unbound. For the calling context that shall be no problem as it is known to the caller that the execution time of the call is unknown. However, execution progress of other unrelated threads must not be delayed. Regarding the kernel that requires that long-running operations are preemptible so that potentially runnable threads can be executed. For user-level services principally the same requirements exist, however, the situation is more relaxed. Given there is a service that has multiple clients and one client requires a bounded reply during its latency-constrained operation, the server must specifically consider this client in its request processing. Concluding, a latency-constrained program must be aware which operations are potentially unbound and not call those in code paths with timing constraints. This is true for both native applications as well as legacy code running in a virtualization

environment. Typical programs are divided into a start-up phase where they call potentially long-running operations, such as memory allocations, and a control phase which does not use such calls. Further, any intermediate layer, especially in a virtualization environment, must not synchronously call any unbound operation while executing latency-constrained code.

The Fiasco.OC microkernel and the L4 runtime environment were designed with both latency and security requirements as two of their primary goals. It is therefore well suited as a base system for this work.

3.2 L4Linux

L4Linux is a major application in TUD:OS and plays an integral role in the overall system. It is a virtualization solution allowing to run standard Linux applications and drivers, required in many use cases. Linux is becoming more and more real-time friendly, allowing to run real-time tasks within Linux, side by side with normal, non-real-time applications [FPT14]. The same can be done using L4Linux, allowing to run legacy real-time applications side by side with other legacy applications, in the same or other instances of L4Linux. Of course, programs can also be native L4 applications or application in other virtualized environments, and they may also have real-time requirements.

Considering the Linux kernel, the whole kernel must provide good preemption characteristics itself, that is, handling of incoming interrupts shall not be stalled by long-running non-preemptive operations. As Linux is a monolithic system this poses big challenges to the internal design and code quality of Linux, especially with regard to drivers. However, Linux is being used as a real-time operating system which lets us assume those characteristics are given.

Considering L4Linux, the added L4 adaption must also have these characteristics. However, the initial design did not focus on those but rather had to adapt to a given and fixed interface, as described previously. Regarding interrupt responsiveness L4Linux shows limitations. In the following I will outline those limitations and present possible solutions.

From the security perspective L4Linux offers the same characteristics as native Linux, protecting the Linux kernel and Linux user processes from each other by the means of address spaces. Therefore no improvement is required in that regard. However, any modification made to the system must consider that security properties are being hold.

The following list describes the problems that I identified within L4Linux regarding interrupt responsiveness:

Semantic gap between Linux and host scheduling

In the initial L4Linux design, the Linux scheduler has been modified such that any runnable Linux process was exposed to the L4 scheduler. That was done because it was assumed that the L4 host scheduler can schedule more efficiently among the threads that are ready, avoiding any additional decision from the Linux kernel code which would need to be switched to.

However, there is a semantic gap between Linux and the L4 scheduling, as the L4 host scheduler does not know about any additional attributes the Linux kernel is giving each process, such as Linux-internal priorities and thus treats every process equally. For

example, a low priority Linux process is treated equally to a high priority, interactive process. Adapting the L4 priority of the corresponding L4 thread was discussed for the initial design [Här+97], but has never been implemented.

Additionally, keeping the modifications of the Linux scheduler has been causing considerable manual effort as the Linux kernel scheduler was going through multiple revisions in the early 2.6 series. Thus, due to the mentioned disadvantages with the handling of scheduling decisions, I decided to drop any modification of the Linux scheduler. Consequently, with the standard Linux scheduler, L^4Linux only exposes one Linux thread (per CPU) to the host system to run. This change also has a positive effect in the context of real-time processes as only the Linux thread selected by the Linux scheduler is eligible to run. As a side effect, the change that only one Linux context (per CPU) can be active at any time also simplifies the code within L^4Linux that handles release and reception of user processes executing in L4 threads.

Unbounded host interaction

The L4 host system offers potentially long-running operations. One such operation is rights delegation, for example on memory, using the calls *map* for granting access rights and *unmap* for revoking access rights. Mapping operations build up a hierarchy among tasks as threads in each task can map memory or capabilities to other tasks, given the right to do so. The counter operation, unmap, is more expensive because it requires the kernel to keep a data structure which stores the mapping hierarchy to be able to remove mappings again. This data structure is called *mapping database*. Revoking a right works recursively, that is if a task A has mapped a page to B and B has mapped it further to a task C, then when A revokes the page, it will be removed from B and C. In this case A does not know that the page was mapped further by B. Consequently, the unmap operation depends on the length of the mapping tree behind a page and it thus not fixed in time. In a general case, real-time programs must be aware that unmapping a resource is not fixed in time.

Revoking and possibly deleting a resource can also lead to the revocation of other resources. For example, unmapping a task also causes unmap operations of the memory pages in there as well as of all capabilities. Thus, the time needed for unmapping a task is highly dependent on the amount of virtual memory and number of capabilities mapped in this task. If the task to be deleted has further mapped out pages or capabilities, those are also deleted recursively.

This behavior places a special challenge to virtualization systems: Since kernel code runs within a single thread, a call to unmap causes the guest kernel to stall for the time unmap needs, as it is a synchronous operation. This contradicts to the requirement of good preemption of a system as needed for real-time workloads. I will describe a mechanism to overcome this problem in this chapter.

Interrupt handling

Interrupt handling plays an important role in a latency-conscious system where overhead and thus delay shall be minimized. However, in the initial L4 design each interrupt had to be handled by a separate L4 thread as one L4 thread could just bind to a single interrupt. The required interactions between the interrupt threads and the Linux server cause additional overhead and thus delay. The type of interaction between the interrupts threads and the Linux server also depends on the state of the Linux server:

The interrupt threads are configured with a higher host priority than the Linux server so that the Linux kernel is interrupted immediately when an interrupt is triggered. In older Linux versions, the interrupt handling is divided into two parts, the *top-half* and the *bottom-half*. The top-half part is responsible to run any immediate work regarding handling of the device interrupt and designed in a way that it must not depend on any specific Linux context. Consequently it can run in the L4 interrupt threads, outside of a proper Linux kernel context. This ensures that the top-half of a device interrupt is handled immediately when it is triggered. An interrupt can trigger further work in the kernel itself, which is flagged by the top-half code. In native Linux this is handled on the return path from the top-half handler. However, with L4Linux running the top-half in a different host thread, this interrupt thread needs to notify the Linux server. Three cases must be considered here:

- The Linux kernel is idling and thus the idler context needs to be woken up so that Linux's scheduler is called.
- A user process is running. The user process must be interrupted so that the Linux kernel gets active again to eventually handle work triggered by the interrupt.
- The Linux kernel is active, for example, handling a system call. Nothing is done by the top-half handler because the kernel code will check for any interrupt triggered work before resuming a user process or going to idle.

Delay is not only added by the interaction between the Linux server and its interrupt threads but also because the Linux server is not preempted when it is executing Linux kernel code and an interrupt is received. Although the interrupt threads are prioritized over the Linux server and call their top-half functionality, any further pending work does not immediately preempt the Linux server so that it can pick up pending work. The reason is that the initial L4 design did not provide necessary means to preempt an L4 thread with the required state information. Instead, checks for pending work are handled in the L4 adaption layer once the Linux server has finished its current operation, such as handling a system call.

I will address those problems with a new execution model which I will introduce later in this chapter.

Missing of events

In the original design of L4Linux there is a race condition to miss a notification from an interrupt thread. The problem lies in the state variable that is required to decide within the interrupt handlers which way to notify the Linux kernel, or not to notify at all, as described previously. This state variable contains the L4 thread identifier of the user process that shall be running. However, setting this variable and doing the IPC call atomically is not possible. Thus it can happen that the variable is set with a valid L4 thread ID and the IPC call has not yet happened, an IRQ is delivered in the meantime, the IRQ handler wants to interrupt the user process, which is not yet running and thus the wake-up event is lost. Eventually the work triggered by the interrupt will be done, for example with the next interrupt, for example from the scheduling interrupt, or a system call invoked by the user process. However, such a miss causes a latency in the event delivery delaying any required processing of work.

The very first version of L4Linux used to disable interrupt delivery at the CPU level for synchronization, being allowed to do so. This avoided the race because invoking an IPC call enabled the interrupts again. However, the possibility to disable CPU interrupts

from user-level programs was removed for security and robustness reasons in later kernels such that this way of handling atomic sections is not possible anymore [HHW98].

Overall, the paravirtualization approach taken by L^4Linux does work, however, also uses rather complicated constructs to make Linux run on L4. The root of this design is owed to the fact that the initial design goal was to stay with the originally proposed L4 kernel interface. Secondly it has been shown that the interface is capable of supporting such a workload. The problems described are not unique to Linux but affect all guest operating systems such as shown in our L4/Symbian port [Bra+08].

However, with the freedom to modify the Fiasco kernel it is possible to develop an interface that is specifically designed for running guest operating systems (virtualization) and integrates seamlessly into the existing microkernel-based system.

The basic idea is simple: provide an interface that matches a real CPU as close a possible, because an operating system is expecting this type of execution environment. I will describe this interface in the following.

3.3 A Generic Virtualization Interface

To address the problems mentioned above, we added new mechanisms to the L4 kernel meant to genuinely support virtualization of CPUs, referring to the resulting mechanism as a *virtual CPU* (vCPU).

The goal is to provide a virtualization interface that supports both faithful virtualization as well as paravirtualization with the same functional approach, covering CPU and memory virtualization. Further goals include the integration into microkernel-based systems so that VMs and native microkernel applications can coexist and work together, creating a symbiosis. The last goal is to provide a platform for latency-conscious workloads.

The basic idea for the virtualization interface is straightforward. Operating systems have been developed to run on hardware and hardware follows a simple model of execution. A processor executes instructions until the executed instructions cause an exception or an external interruption occurs. In this case the processor continues execution at handler code that has been provided by the operating system. The operating system will then handle the exception and remove the reason for the exception so that the original execution can continue where it has been interrupted.

Virtual machine monitors follow a similar approach: They execute the guest operating system and handle any exception that is caused by executing the guest operating system.

The virtualization interface provided by the microkernel shall have similar characteristics when executing guest operating systems: let the guest execute code and branch any exception to a predefined exception handler function within the guest kernel. For integration with the host system the virtualized guest shall also be able to communicate with other system modules by means of L4 operations such as IPC. As such a virtualization interface provides *virtual CPUs* to the guest operating systems we call this interface briefly the *vCPU* interface.

Overall, the following features need to be provided by a vCPU:

- One or more entry points must be defined where execution continues when an event shall be delivered to the vCPU.

- It must be possible to disable injection of events to the vCPU to be able to implement atomic sections.
- A memory area must be available which can hold the state of one interrupted execution.
- Communication with other microkernel threads must be possible through the same mechanism, allowing to use any existing L4 component directly and without restrictions.
- Multi-processor guests must be supported.

Entry Point(s) The entry point is the start of a function similar to an interrupt vector on physical processors. Execution is branched to this function whenever an interrupt is injected to the vCPU. Besides external events there are also synchronous events that are triggered during code execution, for example, an exception when executing an undefined instruction or faulting on memory accesses. Those types of events can be handled in the same way as external events, given the possibility to differentiate them.

Event Delivery Flag The event delivery flag indicates whether the vCPU is ready to be interrupted and to be branched to an entry point. This flag is used very frequently by prospective guest operating systems and must therefore be modifiable with very low overhead.

State Save Area The state save area is an area of memory that is used to store sufficient state to resume execution after handling interrupts and exceptions in the guest operating system. Both the host system and the vCPU access this region. The event delivery flag prevents reentry and thus locks this region and thereby prevents inconsistencies from concurrent modifications. The kernel disables event delivery whenever the vCPU is branched to the entry point. When the context information has been saved from the state save area, the guest can re-enable event delivery to signal that the state save is clear for new content.

Receiving and Sending Messages The primary communication method on L4 systems is IPC which is synchronous. This is in contrast to the asynchronous nature of vCPUs that receive messages whenever event delivery is enabled without invoking a specific operation. However, vCPUs shall be able to communicate with other servers and to use the respective client libraries of the servers without the need to redesign those services and libraries. It is therefore crucial that L4 IPC works the same way in vCPU mode as in non-vCPU mode.

For transferring message data, L4 IPC uses a user-level thread control block (UTCB). Each L4 thread has a UTCB and UTCBs are located on kernel-provided memory. Senders store their message into the UTCB prior to invoking the corresponding IPC operation and get messages out of the UTCB when receiving messages. The UTCBs are also used for memory and capability mappings.

When in vCPU mode, receiving an IPC message can be twofold, depending on the state of event delivery:

Event delivery is enabled: Whenever a message is received, execution continues at the entry point. Event delivery is disabled and message contents are stored in the UTCB and the state save area.

Event delivery is disabled: The vCPU behaves as in non-VCPU mode and messages can be received by explicitly invoking a message receive operation.

With vCPUs a thread is able to receive an IPC message while not blocking in a corresponding IPC receive operation. Receiving interrupt notifications works in the same way as those are also delivery via the IPC mechanism. Asynchronous event reception is not only required in virtualization environments but shall also prove useful for any kind of asynchronous usage models.

Sending an IPC message, or more generally, invoking any IPC operation, can be done anytime while in vCPU mode. However, for the duration of the IPC event delivery through the entry point will be disabled so that the semantics of IPC are retained. Further, as sending and receiving IPC involves copying data to and from the UTCB, event delivery must be disabled throughout handling the UTCB. Otherwise an asynchronously incoming IPC message will overwrite UTCB contents. Consequently, event delivery to the vCPU must be disabled before putting data into the UTCB and only enabled again after retrieving the reply from the UTCB. Event delivery must also be disabled for library calls that may invoke IPC operations.

Paravirtualization and Multiple Processes Operating systems commonly use multiple virtual address spaces to provide isolation among different processes on their system. They also guard the guest kernel from the user processes. Protection of the kernel could be accomplished by using a separate address space, however, that would require address space switches for all kernel invocations such as system calls. To avoid the switching, the virtual address space is divided into a part accessible to the user process and a part that is only usable by the kernel. Furthermore the kernel is part of every user address space. With such a configuration kernel requests require only two privilege changes and address spaces are not switched.

To apply the same approach to paravirtualized setups, the system would need three privilege levels: for the host kernel (hypervisor), for the guest kernel and the user processes to guard the host kernel from the guest kernel and to guard the host and guest kernels from the user processes. However, commonly hardware only offers two privilege levels. For this reason guest setups can only use the non-privileged mode and therefore have to use separate address spaces for the guest kernel and user processes. A side effect of using separate address spaces is that both the guest kernel's address space as well as the user's address spaces can use the full size of the virtual address space available to user-level programs on the host system. In a potential system with three privilege levels the virtual address space must be divided in three parts for each level, yielding in less available address space for each part. This is in particular interest on 32-bit architectures and today's systems with multi-gigabyte memory setups.

With the separate address spaces it is required that those address spaces are switched when changing from vCPU guest kernel mode execution to a vCPU user process and again when switching back. This ensures that the guest kernel code and data is protected from accesses by the user processes. To support this common use case of address space switching, a vCPU can switch between host tasks and thus different guest address spaces. When continuing execution in a user process, the guest kernel will supply the host kernel a corresponding address space where the vCPU will be migrated to. Whenever an event shall be delivered or the user code triggers an exception, the vCPU is switched back to its guest kernel address

space to continue execution at the entry point.

Besides the address space argument, the corresponding vCPU operation can also be supplied with pages to be mapped to the target address space. This allows to handle page faults of user tasks within the same call instead of using separate map operations, speeding up page fault handling.

Figure 3.1 depicts a schematic view on a paravirtualized operating system, consisting of an address space for the guest operating system kernel and two user processes.

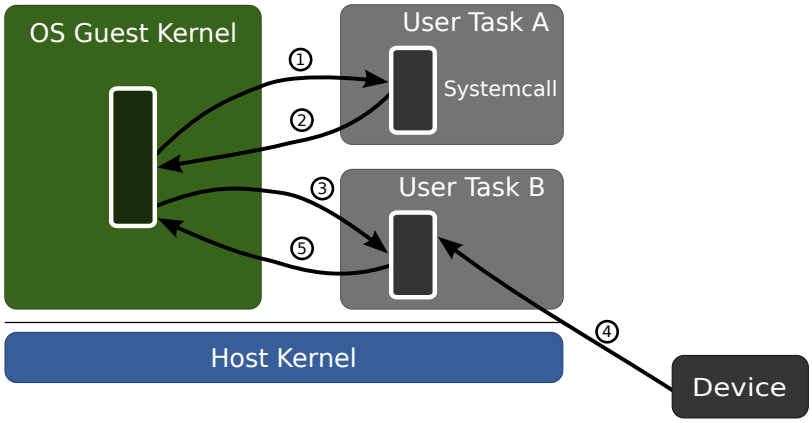

Figure 3.1: Using multiple address spaces for a vCPU to implement user-processes in a paravirtualized operating system. The execution starts in the kernel task and continues with switching the vCPU to task A (1). Executing user code, the vCPU will execute a system call and thus transition back to the kernel task (2). Similarly, executing in task B (3), an external event (4) switches the vCPU back to the kernel task (5).

3.3.1 Multi-Processor Guest Operating Systems

Symmetric multi-processor systems have multiple processors within one system that can be used in parallel. Operating systems make all the processors available to applications. In virtualized setups multiple processors can be made available to VMs, allowing guest systems to use multiple host processors.

With vCPUs, the guest operating system is supplied with multiple vCPUs that are placed on the available host processors. No specific features or enhancements are required by the vCPU functionality to run multi-processor capable guest operating systems. Inter Processor Interrupts (IPIs) are mapped to L4 interrupts that can be triggered by software.

3.3.2 Full Virtualization and vCPUs

Modern processors such as x86-based systems from AMD and Intel provide hardware support for operating system virtualization [Cor14; Dev11]. The virtualization extensions provide an execution mode that is free of virtualization holes and further provide several performance targeted features for virtualization, such as nested paging (see 2.4.1).

The execution of a virtual machine is very similar to the execution model of vCPUs. As already introduced earlier, each virtual CPU of a virtual machine is configured by a VMCx. The CPUs provide additional privileged instructions for managing the VMCx and handling

virtual machines. Whenever an event inside the virtual machine cannot be handled within the virtual machine, the CPU will exit the VM and return to the VMM, delivering sufficient information to let the VMM handle the event.

Mapping this type of execution to vCPUs is straightforward. Virtual machines always require a task for execution comprising the resources, such as memory, for the VM. The VMCx is configured by the VMM, which is running a vCPU for each CPU of the virtual machine. To hold the VMCx data, a vCPU requires a bigger vCPU state save area than for a normal vCPU. Resuming into the virtual machine is done through the kernel, which copies and sanity checks the VMCx and calls into the virtual machine. The exit from the virtual machine goes through the kernel back to the VMM. Depending on the state of the event delivery within the vCPU, the VMM is either entered through vCPU's entry or when the vCPU-resume call returns.

On Intel processors, when using nested paging, the page table for VMs uses the Extended Page Table format (EPT) [Cor14] which is different from the standard process page table format. For that reason a VM needs a different type of address space and cannot use a standard *L4::Task* for holding a VM's page table. Instead it has to use an *L4::Vm* which is derived from *L4::Task* and behaves in the same way but uses the EPT format.

Recent versions of the ARM architecture also include support for virtualization [Lim14]. On ARM, the CPU-based virtualization functionality has been integrated differently than on x86, for example, the guest VM configuration must be saved and restored register by registers instead of being handled as a whole by a single instruction as on x86. For memory virtualization ARM uses the same principle as x86, providing nested page-tables for VMs. Overall, the ARM virtualization extensions fit well into the vCPU model. As the format of the nested page-table is the same for VMs and host applications, no specific *L4::VM* object is required as on x86 and the standard *L4::Task* object works for both VMs and host applications. For VMs an extended state save area has to be used to store the additional VM state. The data layout of the area is defiined by the host kernel.

Although ARM and x86 follow different approaches for CPU virtualization, both virtualization architectures can be made available through the vCPU interface. I thus believe that hardware-assisted virtualization functionality on other architectures can be integrated in the same way.

3.3.3 Conclusion

Concluding, the vCPU interface provides an execution model with asynchronous event delivery for user-level programs on L4. It is generic to support any guest operating system, including multi-processor configurations, and multiple hardware architectures. Compared to other virtualization solutions, such as DISCO [BDR97] and Xen [Bar+03], the vCPU model integrates and naturally extends an existing system, allowing to use system services in virtualization components as well as using virtualization functionality in microkernel-based applications. As an example, the vCPU mechanism is used to implement redundant multi-threading for replication-based fault tolerance [Döb14].

3.4 Implementing Virtual CPUs

Previously I described the requirements for a generic virtualization interface that can both handle paravirtualization and hardware-assisted full virtualization. In the following, I will provide details on the implementation of this interface and the virtualization functionality in the Fiasco.OC microkernel. The following has also been published in [LWP10]. I will start with the para-virtualization interface and add the interface for hardware-assisted virtualization in a second step.

The base for the para-virtualization interface is the virtual CPU (vCPU) that executes the code of a guest. A virtual machine can consist of multiple vCPUs, building a virtual multi-processing system.

As described in the previous section, the following five distinct features are required to implement vCPUs:

- An event delivery enable/disable flag.
- A state save area.
- One or more entry points.
- Support for messages.
- Address space switching.

Threads are being used to execute code and are scheduled by the host kernel. The same is required for vCPUs and thus threads are the basis for implementing the vCPU functionality in the host kernel. A thread is promoted to a vCPU by adding configuration settings, such as an entry point and the location of the state save area. Implementation-wise the first choice to make is whether threads shall be replaced by vCPUs altogether. Technically this is possible as vCPUs offer thread semantics with disabled event delivery. However, vCPUs also require more memory resources for the state save area, which can be up to a whole page in the current implementation for hardware-assisted virtualization. However, most threads do not need any state save area. Thus normal threads are preserved and remain available.

The next choice to make is when a vCPU is created. Principally, two options exist. One is that the system offers a way to create a thread and similar but separate function to create a vCPU. The other choice is to add vCPU-functionality to a thread at any later point. The first option looks simpler to implement. However, implementation-wise it is not required to know whether a thread will eventually become a vCPU. Creating a thread is independent of promoting an existing thread to a vCPU and can thus be done in two separate steps. Further, knowing beforehand whether a thread shall also be a vCPU possibly requires this knowledge in the loader of the application because this loader is creating the first thread of the application. For reasons of usage flexibility and additional memory requirements for vCPUs, any thread can be promoted to a vCPU at any later point, saving system resources.

State Save Area One of the features required for vCPUs is the state save area. In this memory area the kernel stores the state of the vCPU prior to branching to an entry point. The handler that is called by the entry point in the guest then uses this state to handle the entry. The state consists of a reasonable set of CPU registers and the state can be modified

by the handler to influence the execution of the vCPU. When finishing the handler with a resume operation, the contents of the state save area are loaded into the physical processor and the vCPU continues execution.

Referring to a physical CPU, the state save area is either a kernel stack or banked registers and only a minimum of state is saved. Most of the state is kept in the CPU registers and the handler is responsible for saving this live state to an appropriate place, if required at all for the functionality of the handler. A similar approach of only saving a minimal set could be taken by the vCPU functionality. However, the host kernel is required to save the whole register state of a vCPU when entering the host kernel as only a minimal set of CPU registers is not sufficient to run the host kernel. Therefore the saved state is stored into the state save area right away, containing all general purpose CPU registers and flag values, instead of re-loading them into the CPU and leaving them for the user handler to save them again.

The state save area itself must be a memory area, which is shared between the kernel and the user. Pushing the state to save onto the stack of the user's handler would be a natural approach, however, the host kernel shall not access user memory as it can cause page faults when accessing it. Any effort in handling this situation will be avoided when using memory with the same characteristics as the UTCBs. Alternative ways of accessing the state, for example, using a system call, would add additional latency because that system call is likely to be required on every entry and resume operation. Also, such a system call would exchange data with the kernel using the UTCB, which in turn leads to the conclusion that the kernel can store the saved state directly into it before calling the entry point. However, using the vCPU's UTCB for this state is not possible as the vCPU may also receive IPC messages. The IPC messages will place their message contents into the vCPU's UTCB which is therefore not available as a state save area.

As the state save area needs to be accessed by both the kernel and the user, in the same way as the UTCBs, the memory for the state save area is created in the same way as UTCBs. With the introduction of vCPUs, the memory used for UTCBs and the state save areas has been renamed to the more generic name *kernel-user memory*. *Kernel-user memory* is allocated to a task through the *L4::Task* interface.

Entry Point The entry point designates a function within the guest code which is used by the host kernel to branch the execution of the vCPU to whenever an event shall be delivered to the vCPU. Native CPUs typically have multiple entry points, for example, called *interrupt vectors*. However, in a software approach, a single entry point turned out to be sufficient. The address of the entry point is stored in a configuration area of the state save area. The single handler can then determine the cause for the exception by inspecting architecture-specific state.

Event Delivery Flag The event delivery flag indicates whether an entry into the vCPU through an entry point is allowed. The so-called *interrupt flag* is the corresponding flag in physical CPUs. The flag is required in order to implement vCPU-local critical sections. For example, upon entry through an entry point, the state save area contains the previous state of the vCPU. Any consecutive entry to the vCPU has to be prevented until the state has been saved elsewhere and can be discarded from the state save area. For that reason the kernel is disabling the event delivery flag upon entry. The guest operating system can enable event delivery again at any point in time. The event delivery state can also be set in

a vCPU resume operation.

Operating systems use interrupt disable and enable operations to implement CPU-local atomic operations. On physical CPUs this is a fast operation and is thus used frequently in operating systems. That means for the vCPU that the manipulation of the entry flag must perform equally well. For that reason the flag is located in the state save area to be manipulated with memory writes only and without interacting with the host kernel. An alternative approach would be to use a system call to set the entry flag state of the vCPU in the host kernel, but that comes with an unacceptable performance penalty. However, a benefit of the system-call approach is the involvement of the host kernel when enabling interrupts. In the case an event was received while event delivery was disabled, and the event is still pending when event delivery is enabled again, the host kernel could immediately branch to the entry point. When the event delivery flag is just in memory, without involvement of the host kernel, the host kernel does not know when event delivery is enabled again and thus cannot branch the vCPU to the entry point. For that reason, the state save area also hosts a *pending* flag where the host kernel flags any pending event. Thus, when vCPU enables event delivery again, it has to check the pending flag, and if it is set, poll any pending event by issuing a message receive operation.

vCPU Attributes The state save area does not only contain the entry and pending flags but also additional flags that control whether a vCPU can be branched to the entry point. The event disable/enable flag is responsible for IPC and IRQ messages. Entry through the vCPU entry point is additionally available and separately controllable for page-faults and exceptions. In the same way as L4 threads, any vCPU has a pager and exception handler to which page faults and exceptions are forwarded. For vCPUs, the page-fault and exception flags allow to configure whether the vCPU itself will handle page faults and exceptions or those will be forwarded to the appropriate handler thread. This allows to setup a vCPU as required by the use case.

The state of the Floating-Point Unit (FPU) can also be controlled through a state flag, allowing to share the FPU between the vCPU kernel mode and the user-mode, and providing the possibility to implement FPU switching in the same way as native operating systems do. With an enabled FPU flag, the host kernel will not switch the FPU contents when switching to a user process but leave the FPU as setup by the guest kernel. This way the guest kernel can handle the FPU state of its user processes.

The paravirtualization of operating systems with user processes, such as Linux, requires to use address spaces provided by the host to isolate guest user processes among each other and the virtualized operating system kernel. The vCPU mechanism supports that by allowing to specify a task in which execution shall be resumed. The *user-mode* flag in the state save area is used to indicate that the execution shall switch to the currently used user task and allows to reuse the last provided task capability, providing improved performance as no capability look-up needs to be performed by the host kernel. Upon an entry the user-mode flag indicates whether the saved state is from a user-mode execution or from within the task running the kernel code. The stack pointer stored in the state save area is only used when transitioning from user to kernel vCPU mode. When entering from vCPU kernel mode, the stack pointer is unchanged. This behavior is in line with the behavior of native CPUs.

3.4.1 Common vCPU Functionality

Common functionality of a vCPU that is required in any or most of the guest operating systems is gathered in a library *libvcpu*. Performance critical functionality is thereby implemented in header files so that it can be fully inlined by the guest code.

The following listing shows the most used functions of the library.

```
1  void
2  l4vcpu_irq_disable(l4_vcpu_state_t *vcpu);
3
4  l4vcpu_irq_state_t
5  l4vcpu_irq_disable_save(l4_vcpu_state_t *vcpu);
6
7  void
8  l4vcpu_irq_enable(l4_vcpu_state_t *vcpu, l4_utcb_t *utcb,
9                    l4vcpu_event_hndl_t do_event_work_cb,
10                   l4vcpu_setup_ipc_t setup_ipc);
11
12 void
13 l4vcpu_irq_restore(l4_vcpu_state_t *vcpu, l4vcpu_irq_state_t s,
14                    l4_utcb_t *utcb,
15                    l4vcpu_event_hndl_t do_event_work_cb,
16                    l4vcpu_setup_ipc_t setup_ipc);
```

Most interesting in the shown listing are the *enable* and *restore* functions because they provide the functionality for the aforementioned requirement of checking the *pending* flag after enabling event delivery again. The caller of the function must provide a function callback that will be invoked whenever an event is pending. The interface of the enable/restore functions is designed in a way that the callback is an integral part of the function and cannot be forgotten. The `setup_ipc` function is provided to do any preparatory work before calling the event reception function or before opening event delivery.

Guest operating systems must also have the chance to idle, that is to stop execution and wait for incoming interrupts. CPUs offer specific instructions to stop execution, such as *hlt* on the x86 architecture. Those instructions can only be called in privileged processor modes. In virtualized environments, the guest should inform the host that it is idling so that the host can schedule other work in the system, or idle itself.

The libvcpu offers a corresponding `l4vcpu_wait_for_event()` function that blocks the vCPU and waits for any incoming IPC message. It shall be used as an replacement for the idle function in the guest operating system.

```
1  void
2  l4vcpu_wait_for_event(l4_vcpu_state_t *vcpu, l4_utcb_t *utcb,
3                        l4vcpu_event_hndl_t do_event_work_cb,
4                        l4vcpu_setup_ipc_t setup_ipc);
```

As with the `l4vcpu_irq_enable()` and `l4vcpu_irq_restore()` functions, `l4vcpu_wait_for_event()` takes the necessary call-backs to setup and handle incoming events.

3.4.2 User-Mode Virtualization Specifics

L^{4}Linux virtualizes the Linux kernel and runs it on an L4-based host system as a user-level application. Due to the design of hardware only supporting two privilege levels at most in general, L^{4}Linux has to use a configuration where the Linux kernel and its user-level application run both unprivileged on the host. To maintain binary compatibility for applications to the native Linux version, L^{4}Linux requires host operating system features that are normally not needed or required for normal applications. In the following I will explain what is required for the x86 and ARM architectures.

x86 Architecture

On the x86-32 and x86-64 architectures, Thread Local Storage (TLS) is typically implemented by using segment registers. Those registers point to a table defining segments of memory together with access rights. Setting a table index in the segment register is possible from user-level, however, defining the contents of the table, that is the memory ranges, can only be done by privileged mode and thus in the host kernel. To be able to provide the necessary segments, for example for TLS or Linux's `modify_ldt()` system call, the host kernel needs to offer the option to set entries in those tables. Fiasco.OC offers a system call to define user-accessible ranges of memory on those tables. There are two different types of tables: the Local Descriptor Table (LDT) and the Global Descriptor Table (GDT). The LDT is local to the task and thus needs to be switched on task changes. The GDT is per CPU and three entries of the GDT are available for each thread to use. The entries are set to the thread-specific value on each thread switch. TLS is implemented using the GDT.

Fiasco.OC uses TLS to make the UTCB address available to threads through a segment register. However, in virtualization setups this segment register may be used by the guest code so that it either cannot be used by L4 code or must be multiplexed. In any case Fiasco.OC can only initialize the segment register with the UTCB address upon thread creation and has to save and restore it for further thread switches instead of just resetting it. L4Re handles this by providing a wrapper implementation of the `l4_utcb()` function for getting the UTCB address that can be overwritten by programs.

L^{4}Linux makes use of that wrapper as Linux is using the segment registers itself. The alternative way of getting ones UTCB address in L^{4}Linux is to put in the TCB of Linux's threads. The TCB is embedded in the stack memory which follows size and alignment rules so that the TCB can be found using appropriate masking on the stack pointer.

ARM Architecture

The evolution of the ARM architecture brought changes to the way low-level functionality must be implemented in user-space programs. For example, with the introduction of multi-processor systems, going from architecture version v5 to v6, new facilities to implement atomic operations and thread local storage were introduced. To keep applications independent and binary compatible across different architecture variants, the Linux kernel provides a set of functions that shall be called by user-programs. Those functions live in the kernel space area of the virtual memory and implement functions such as compare-and-exchange with the hardware functionality provided and needed by the hardware. This way a compiled user program works on any sub-architecture as the Linux kernel provides the right implementation

of the function. These functions are not implemented by system calls, which would be an alternative, as they are used frequently in programs and must have low overhead.

The following list shows the functions that are provided by the Linux kernel.

- `cmpxchg` and `cmpxchg64`: Atomically compare and exchange function on 4 and 8 bytes of memory. Provided by the kernel as the v5 and early versions of the v6 sub-architectures do not provide means to implement this functionality without the help of the kernel.
- `memory_barrier`: Required for SMP systems, an empty function on uni-processor systems.
- `get_tls`: Get the thread local storage (TLS) register. For v5, the kernel uses a single and global memory location to store the TLS value of the current thread. This is possible as v5 only supports single processor systems. With v6 and upwards the CPU offers dedicated TLS registers per CPU which allows a thread to get the TLS register value without kernel interaction.

For L^{4}Linux this poses a problem as the designated address range for those functions (`0xffff0f60 - 0xffff0fff`) is defined by the ABI but is not available for L^{4}Linux as this range of the address space is occupied by the host kernel. Linux programs have those addresses hard-coded in their binaries. Binary compatibility can be retained by emulating access to those functions in L^{4}Linux, however, this significantly slows down applications as handling the page-fault is much more expensive than a function call.

The following possibilities exist to improve the situation:

- The host kernel implements the same set of functions with the same semantics as Linux. This solution provides the same execution speed as the native implementation, however, it also has drawbacks. The interface is provided for a particular user program or VM such as L^{4}Linux and may collide with other implementations. Further, it would be necessary that the value the `get_tls` function returns is known to the host kernel and it needs to be updated for every user thread switch that L^{4}Linux does. The update can be included in the corresponding user thread activation function to avoid any extra kernel system call.

  Another possibility could be to allow programs to install an arbitrary page in the kernel's address space to let the programs define the functions. However, this is not possible as the ARM processor uses the same page to host its entry vectors which are located at `0xffff0000`. Giving the user the ability to define code for the Linux functions would require copying code to that location on thread or task switches in the host kernel to protect the kernels own entry code.
- If the source of the user program is available, especially the C library, it can be modified to use an address range provided by the L^{4}Linux kernel. L^{4}Linux offers the same kernel functions and maps those to the address space of each user program. No other change is required besides changing the base address of the function area.
- If recompilation is not possible, code can also be patched at runtime. Whenever a page-fault to the original function area is detected, the corresponding caller is modified to use the proper address provided by L^{4}Linux. Successive calls of a patched function

will not cause a page-fault anymore but use the L4Linux address right away. The downside of this approach is that finding the right place to modify is not always (easily) possible as the user code might load the address into a register and jump to a different code part to actually jump to the kernel-provided function. At the time of the page-fault, the L4Linux kernel might not be able to find the place where the address is loaded into the register without scanning the surrounding user code and a proper knowledge of the ARM instruction flow. Furthermore, user code that performs integrity checks on its code, for example, for security reasons, will find differences to the original.

In practice, runtime patching proved to be simple for programs using *glibc* as this C library is using a sequence of inline assembly instructions in its source code to call the kernel-provided functions. Consequently the calling code is always the same and can be easily detected and patched. Linux distributions, such as Debian GNU/Linux, will thus be patched at runtime and run, except for patching the first invocation, without slow trap-and-emulate mechanisms.

When patching user code, the new code needs to jump to the address supplied by L4Linux instead to the original location at 0xffff0f60. The choice of that address is important. The new instructions including its target jump address must be encodable with the same amount or less instructions as the original code. The ARM instruction set allows to encode values within an instruction, optionally with an additional shift value, allowing to also encode big values within the instructions itself, without requiring an extra memory load instruction. As the original Linux kernel code is located in the upper part of the virtual address space, a big 32-bit value is required. With this technique a call to addresses in the range 0xffff0000 to 0xffff00ff, the range of the kernel provided functionality, can be constructed with a few ARM instructions without any memory reference. The following code is used to get the TLS value and represents a typical example:

```
0: mvn    r0, #0xf000   ; r0 <= 0xffff0fff
4: mov    lr, pc        ; return address
8: sub    pc, r0, #0x1f ; jump to 0xffff0fe0
```

The first instruction loads the value 0xffff0fff into the register r0 using a negative move instruction. The second instruction sets the link register for the function to be called to return to after the jump and the third instruction finally jumps to 0xffff0fe0 by substracting 31 from r0 and setting the result as the new program counter.

The code which replaces the original code needs thus to fit into these three instructions. As the address space available to L4Linux is limited to 3GB (0xc0000000), the provided code has to be placed below that address limit and needs to be encodable within the available space of the original instructions. A straight-forward approach is to limit the available address space of Linux programs below the 3GB limit required by the underlying kernel and use that area to provide the Linux code. This allows to map the code to a page at virtual address 0xbffff000. The code to be patched in the applications looks as follows:

```
0: mov    r0, #0xc0000000 ; r0 <= 0xc0000000
4: mov    lr, pc          ; return address
8: sub    pc, r0, #0x20   ; jump to 0xbffffe0
```

This code jumps to the same offset (0xfe0) within the provided page at 0xbffff000, allowing to use the same code as in the original implementation. An analogous implementation is used for the other kernel-provided functions.

Other C libraries, such as the one used in the Android operating system, may not allow to do runtime patching. The Android C library gives the compiler better means to integrate the calling of the kernel-provided functions. Additionally, Android prefers to circumvent the ABI provided by the Linux kernel and use sub-architecture-specific code directly to avoid the additional indirection. Generally, such setups can always use the trap-and-emulate approach, or recompile to adapt to the changed settings.

A third option is to provide the environment which the user programs expect. An example is again providing TLS. Starting with version v6 the ARM architecture provides dedicated TLS registers and Linux uses the TPIDRURO register for TLS. This register can only be written by the host kernel and thus requires an interface for setting the value of the registers for user programs such as L^{4}Linux. The host kernel also has to load the thread's value on each thread switch. With support for the TPIDRURO register the aforementioned Android system, which avoids the `get_tls` function provided by Linux, can run unmodified with regard to TLS.

Coming back to the patching approach, we can also avoid to call the `get_tls` function by directly patching the original code with the appropriate load instruction. As this is only a single instruction, there are not space constraints:

```
0: mrc 15, 0, r0, cr13, cr0, {2} ; r0 <= TPIDRURO
4: nop
8: nop
```

Overall, all programs can be at least handled via trap-and-emulate and with multiplexing of registers as those accesses do not trap. In specific cases, such as when programs use the *glibc* library, runtime patching avoids the requirement to use the costly trap-and-emulate mechanism.

To suggest improvements, Linux on the ARM architecture should announce the base address for the kernel functionality, for example via the `aux` vectors as done on the x86 architecture. This way L^{4}Linux can use another memory area than native Linux. User programs should also keep to the ABI defined by Linux to be able run unmodified binaries without substantial effort. However, both suggestions come with a minimal performance penalty. Non-ABI functionality should thus only be used when the source of the applications is also available for a possible recompilation by the end-user.

3.4.3 Integrating vCPUs into the L4 System

A virtual machine also needs to communicate with other host system components, for example, with memory managers and external device drivers. Using those services is accomplished by invoking corresponding capabilities and thus by doing L4 IPC. IPC within a vCPU works in the same way as in non-vCPU mode and allows to reuse existing code. This is especially important for using libraries that use IPC, for example, client libraries that implement communication with a server.

However, attention must be paid to the handling of the UTCB that is required for storing and retrieving IPC payload data. While event delivery is enabled in the vCPU, asynchronous

notifications can be received anytime, including IPC messages that deliver payload data in the UTCB. Thus, for preparing an IPC message, a vCPU must disable interrupt delivery before filling the UTCB and only open them again after retrieving all data from the UTCB after receiving an answer.

As an example, in L^{4}Linux, the wrapping code looks like this:

```
unsigned long flags;
local_irq_save(flags);
 ... // l4_ipc-calls-with-utcb-handling, L4 library calls
local_irq_restore(flags);
```

For easier use, a set of function wrappers is available that wrap the above code sequence and avoids writing repetitive code sequences. Still developers need to remember that disabling virtual interrupts also means that the vCPU cannot receive any event during this time which in turn attributes to the latency of L^{4}Linux. The code sequences under lock should therefore be minimized to one call a time.

3.5 Implementing Full Virtualization

Modern CPUs provide built-in support for virtualization to run unmodified guest operating systems, reducing the complexity for the VMM for running unmodified guests and also providing a more accurate behavior of the virtual platform. However, running unmodified guests requires that the VMM emulates a real hardware platform sufficiently well. Considering the x86 platform, this includes device models of an interrupt controller, keyboard, mouse, UART, graphics, to name the most basic ones only. Additionally, x86-based systems require a BIOS to boot and launch with running 16-bit code that needs to be supported by the VMM. Later versions of the x86-based virtualization hardware add more features to handle virtualization in hardware, however, the need for a VMM remains.

The hardware virtualization functionality has to be used from the privileged host processor mode, and thus requires that the host kernel is extended to support the hardware virtualization features. However, not all virtualization functionality, such as device emulation, is required to be in the kernel. The virtualization functionality is therefore split into two parts:

1. The host kernel is required to run virtualization functionality that can only be executed in privileged host mode and all functionality required for isolation with regards to virtualization in the system.

2. The remaining parts that can be implemented in a user-level component. Implementing as much functionality as possible outside of the kernel reduces the size of the trusted computing base.

We distinguish between the two parts and call a host-kernel that supports virtualization a *hypervisor* and the user-level component *virtual machine monitor* (VMM).

The VMM has to provide a virtual platform the guest operating system is running in. The VMM needs provide resources to the VM, such as memory and CPU, as well as provide virtual or pass-through devices. Virtual devices can be emulations of existing physical devices so that existing device drivers in the guest can be used or virtual devices that are specifically

designed for virtualization environments. The latter provide a better fit for virtualization environments as they avoid emulation and can use the best possible and best performing interface between the VM and VMM. In the following, I will provide an overview on three different approaches to virtualization that have been implemented on Fiasco.OC and can be used to run guest operating systems.

3.5.1 Fiasco.OC's Full Virtualization Interface

Using the hardware virtualization features of modern CPUs in a host operating system requires the execution of privileged processor instructions. Thus the host kernel needs to explicitly support those virtualization features. Fiasco.OC has been extended [Pet+09] to support both AMD's [Dev11] and Intel's [Cor14] variants of x86-based virtualization.

Both virtualization variants implement the same model for running guest operating systems, however, differ substantially in their implementation so that the VMM component must be aware of the differences. Both virtualization extensions duplicate x86's four privilege levels and split them into *root mode* and *non-root mode*. A hypervisor is running in root mode and the CPU provides the functionality to switch into non-root mode. Any event that cannot be handled by the guest operating system, or is configured to trap out of the VM, transfers the CPU back to root mode for the hypervisor to take control. During the transition between the two modes the state of the CPU is saved to and loaded from the corresponding VMCx.

As the size of the VMCx is bigger than the standard vCPU state, a whole page of kernel-user memory must be provided when a thread is promoted to *extended* vCPU mode. 'Extended' indicates that the hardware virtualization feature shall be used for this vCPU. From this point on, the VMM can use the *L4::Thread::vm_resume* operation to switch to the VM. Depending on the state of the event delivery flag of the vCPU, VM exits to the VMM will either continue in the vCPU's entry point (event delivery allowed) or the *vm_resume* operation call will return (event delivery disabled). The availability of those two options proved useful to support different execution models of VMMs.

The VMM is also responsible for handling the VMCx for each vCPU. Fiasco.OC will perform sanity and security checks so that a VMM cannot gain more system privileges or harm the host kernel or other user-level components. The format of the VMCB is defined as a memory layout and thus the vCPU interface uses this format as described in AMD's reference manual. In contrast to AMD, Intel does explicitly not define the memory layout of the VMCS but defines field numbers to be used with the *vmwrite* and *vmread* instructions. Both instructions are only available in privileged mode and thus cannot be used by the VMM. For that reason Fiasco.OC uses a memory layout for the VMCx that derives the offsets into the memory area from the field numbers used for *vmread* and *vmwrite*.

A VMM is also required to provide memory to a VM. The virtualization features of the CPUs provide the feature of nested paging for that purpose (see 2.4.1). With nested paging, the CPU first translates virtual addresses within the VM to so-called guest physical addresses, followed by a translation using the VM's page-table to host physical addresses. This outer page-table is managed by the host. The concept of nested page tables maps very well to L4-based systems. Principally, an *L4::Task* could be used for the outer page table. As already outlined previously, on the x86 architecture, the page table formats are different so that the derived *L4::Vm* is necessary. Using the usual L4 operations *map* and *unmap* any available memory can be given to the VM and withdrawn from it.

Differences between Paravirtualization and Hardware-Assisted Virtualization No major performance differences are to be expected when comparing systems with and without hardware-assisted virtualization. Both virtualization methods use the vCPU mechanism so that they operate similarly. However, larger effort is required for context switching as for hardware-assisted virtualization the host kernel is required to save and restore more state (VMCx) than for standard vCPUs, which induces additional overhead. For guests that use an MMU nested paging is a significant performance improvement that cannot be accomplished with pure software-based solutions such as used with paravirtualization [SK10].

Conclusion With the described extensions implemented in Fiasco.OC it is now possible to implement VMMs that are required to let VMs run. The design allows to run different VMMs on the system as already depicted previously in Figure 2.2. Different VMM implementations can cover different use cases and run simultaneously on the same system. For example, a feature-rich VMM can support a wide variety of guests but also comes with a large code size and complexity. On the contrary, a specialized VMM for a specific guest setup can be small and of low complexity and thus be a better fit for security and latency-conscious environments.

The following three VMMs have been developed or been adapted to run on Fiasco.OC: L4/KVM, Karma and the Palacios VMM.

3.5.2 L4/KVM

L4/KVM [Pet+09] uses L^{4}Linux with KVM [Kiv+07] to run unmodified guests in virtual machines. KVM is a Linux module that implements hypervisor functionality using hardware-assisted virtualization features of the system. For using KVM in L^{4}Linux, KVM has been adapted to use vCPU functionality instead of privileged instructions to provide memory to VMs as well as to run VMs.

Using L^{4}Linux with KVM provides a feature-rich VMM that supports many guests, including Microsoft Windows, however, also resorts to a large and complex code base that is not suited for security-conscious nor latency-critical use cases.

3.5.3 The Karma VMM

The Karma VMM takes a different approach to virtualization on the x86 architecture. Instead of emulating hardware devices and providing the functionality solely required for booting an x86-based guest system, Karma slightly modifies the guest operating system kernel and omits the boot path of an x86-based system altogether. Connection to devices in the host is accomplished by virtualization-aware drivers in the guest. The approach of the Karma VMM significantly reduces the VMM's software complexity required for virtualizing operating systems.

The Karma VMM has been implemented by Steffen Liebergeld during his Diploma thesis [Lie10; LPL10]. By design, the Karma VMM requires adapted guest systems but can use this as an advantage for a low-complexity VMM that is well suited for applications that demand security as well as latency properties.

3.5.4 Palacios VMM

Palacios [LD] is a VMM library that is built for being linked to kernels and provide the system with VMM capabilities. In its original use case it is linked to the Kitten kernel [Lan+il], an operating system kernel used in high-performance computing environments.

The Palacios library has open ends that need to be filled with functionality by the user of the library. For example, Palacios needs to allocate memory and requires locking functions that must be provided. Because Palacios is a library it looked promising to be used for a VMM on top of L4Re and Fiasco.OC. As Palacios is targeted to run in kernel mode, it does not offer hooks for functionality required when run in user mode. Such hooks include functionality to modify memory mappings of the guest or use the virtualization instructions of the processor as those can be directly manipulated and called when run in kernel mode. Hooks for halting a vCPU, resuming a vCPU and handling guest memory have been added to Palacios and been implemented by the VMM using appropriate functionality of Fiasco.OC and L4Re.

Linking Palacios to a user-level VMM program provided no major difficulties and was of low engineering effort. For the execution model of Palacios it proved useful that the vCPU-resume operation can also return from its call instead of branching to the vCPU's entry point. As the resuming the VM is implemented by a function that is used by the library, the function needs to return to the library. Continuing execution at the entry point would require additional effort to return to calling function that can be avoided here.

Summary Overall, with the implementation of three different types of VMMs, Fiasco.OC's virtualization interface shows to be versatile enough to cover different requirements and demands.

3.6 Guest Operating Systems

Using vCPUs, I have virtualized two commonly used operating systems. The first is FreeRTOS, a classical real-time operating systems which is not using privilege separation. The second is Linux, a popular general-purpose operating system that uses hardware provided isolation mechanisms for process isolation. Linux is also increasingly used for real-time workloads.

The paravirtualization technique is generic and thus applicable on any hardware architecture with minor architecture-specific adaptions. Its abstraction level is also higher than that of native hardware which gives advantages regarding support of target architectures. I will evaluate FreeRTOS and Linux on both the x86 and ARM architectures. For FreeRTOS the higher abstraction level yields to a particular advantage which I will describe in the following section on FreeRTOS.

3.6.1 vCPU-based Paravirtualization of FreeRTOS

FreeRTOS is a classical RTOS which does not make use of multiple privileges and thus can run on CPUs without an MMU. On a system with an MMU, FreeRTOS will not make use of it. However, FreeRTOS can make use of ARM's memory protection unit (MPU) that offers to guard areas of memory from access and allows to guard the FreeRTOS kernel from access by user code.

FreeRTOS is highly configurable and supports multiple tasks with different priorities. The scheduler can be called preemptively via a timer interrupt or cooperatively and selects among runnable tasks with the highest priority. Interrupt handlers are available to service interrupts.

Typically, the FreeRTOS kernel and the application tasks are linked together, building a single binary that is loaded onto the target. For paravirtualization that means that this single binary is executed in vCPU's kernel-mode. The adaptions required for paravirtualization are the following:

- In native FreeRTOS critical sections are guarded by disabling and enabling interrupts to avoid preemption. For running in a vCPU this is replaced with disabling and enabling event delivery to the vCPU.
- A timer is required for periodically invoking FreeRTOS's scheduler. For the paravirtualized FreeRTOS the timer is emulated using a separate host thread that periodically triggers a software interrupt that is delivered to the vCPU executing the FreeRTOS code. Alternatively, a hardware timer can be used to trigger the scheduler of FreeRTOS.

The vCPU and the timer thread must be scheduled in the host system. Assuming a fixed-priority scheduling in the host, both the vCPU and the timer thread have a host priority. To allow the timer thread to preempt the vCPU, the timer thread is given a higher host priority than the vCPU thread. The priorities of the FreeRTOS system must also be considered for the whole system to define its responsiveness relative to other subsystems in the host system.

The required vCPU-adaptions are, with very few exceptions, hardware independent and thus the paravirtualized version of FreeRTOS is running on ARM and x86 platforms whereas the original FreeRTOS is not available for x86. As the specific architecture adaption is provided by the host system, guest operating systems do not need to provide architecture-specific code and can exploit the host system to provide a wider range of architectures and systems.

As FreeRTOS supports task protection using MPUs, it is desirable to support the same features in the paravirtualized variant. Separate host tasks can be used to run tasks with a specific view on the address space. Those address spaces will only contain those parts that the FreeRTOS task shall be able to access and the task switching functionality of vCPUs will be used.

A difference between the MPU hardware and page tables is the size of the protection granularity those mechanisms support. The minimal size of ARM's page table is one kilobyte, while the smallest page size on Fiasco.OC system is four kilobytes. The smallest region supported by MPUs is 32 bytes. Thus when running the paravirtualized FreeRTOS system the alignment of the FreeRTOS regions must be adapted to the supported page sizes of the host system.

3.6.2 vCPU-based L^{4}Linux

Linux, as a full-fledged operating system kernel, uses all features of the vCPU model to be paravirtualized. In the following I will describe the differences between the thread-based variant presented in Sections 2.7.2 and 3.2 and the vCPU-based version.

The thread-based L^{4}Linux uses multiple threads internally to implement a logical CPU. Synchronization of interrupt state among the threads is done by using a central synchronization service. Threads are blocked entering a critical section when another thread already has virtual interrupts disabled by sending an IPC messages to the synchronization service. The service will unblock the thread when the other thread leaves the critical section. With the vCPU-based variant, this handling becomes much simpler. First, multiple threads to build a logical CPU are not required anymore due to the asynchronous entry possibility to handle interrupts and other events. Secondly, synchronization is achieved by manipulating the vCPU event disable flag that is directly mapped to the corresponding low-level Linux operations. Consequently, a central synchronization service is not required anymore as disabling event delivery to the vCPU will automatically prevent any interruption of the vCPU execution.

The handling of the user-level processes is also significantly changed between the two variants. In the thread-based version, each user-level thread is mapped to its own L4 thread and exceptions triggered by a user-level thread are sent to the Linux kernel via L4 exception IPC, generated by the host kernel. In the vCPU-model, the vCPU transitions between the kernel task and the user tasks. Thus only a single vCPU is required for any number of processes used in L^{4}Linux, saving host resources compared to the thread-based model. The handling of address spaces itself is not changed. Both variants require one L4 task per user-level process to provide process isolation.

Adding support for virtual multi-processor support is straight-forward. Multiple vCPUs are created, one for each virtual CPU, running the Linux kernel code and handling user-level processes in the same way as on native hardware. The L4-specific work in L^{4}Linux must be multi-processing aware and handle concurrent access to shared resources. One example of such a code path is the handling of page mappings to user processes. As L^{4}Linux manages its own shadow page table, upon answering a page fault, L^{4}Linux will need to look up the page fault address and translate it to its internal memory address. That address is then mapped to the user process when resuming its execution. In this case the look-up and the resume operation must not be interrupted so that the mapping does not become invalid after the look-up and before the mapping has been finished. Otherwise, the contents of the source memory page could have been changed already, for example, due to swapping. Further, unmap operations on other CPUs, to perform rights downgrades on memory pages, must make sure that map operations on any CPU are done before doing the unmap. This can be achieved by performing a remote procedure call on all other relevant CPUs, so that when the call returns, all map code paths have been left. Any new map operation needs to look up the source memory address again, which could have since been updated.

Summarized, the vCPU approach significantly simplifies L^{4}Linux compared to the previous thread-model. The vCPU-model also made it possible to support a virtual multi-processor L^{4}Linux with only very few adaptions.

3.7 Responsiveness of VM Systems

In the following I will qualitatively evaluate the VM architectures regarding their ability to provide a continuous and timely execution of their guest systems. This is an important property for two reasons:

- VMs must be able to run periodically. Guest operating systems commonly schedule regular work, interact with devices or other external systems. To ensure proper

operation the VM must be scheduled timely by the host whenever the VM requests execution so that no communication with the external world times out because a guest operating system did not react within a certain time frame. Delays until the VM is actually required to be scheduled can be typically in the hundreds of milliseconds or more without negative effects. Longer delays might trigger error or warning conditions within the guest operating system due to abnormal timing and lead to misbehavior.

- When guest systems are handling timing critical events, the VM must be scheduled according to the requirement of the event handling. Periodic execution of the VM is not sufficient in this case as the process within the VM shall immediately. Colloquially speaking, the VM, which is handling the event, must be scheduled as fast as possible.

Both reasons have in common that a VM must not be blocked from receiving events unduly long, in either case for seconds or microseconds. Reasons that a VM cannot receive events are:

- The VM is not scheduled.
- The VM is scheduled but the guest operating system has interrupts disabled.
- The guest operating system or the VMM are engaged in an L4 IPC operation or system call.

The first reason is due to scheduling decisions in the host and we will cover a possible solution in Chapter 4. The second reason is due to the internal handling of the guest operating system and cannot be influenced by the VMM or host system. The third reason can be influenced or avoided by obeying design rules which we will study in more detail.

3.7.1 Invoking Host Services and Communication

Virtual machines and VMMs need to invoke host services, for example, to allocate memory or for input/output operations. Still the VM shall not block while invoking these services. Ideally, this requires that all communication between the VM and VMM and other components in the host system is asynchronous. The client, the VM or VMM, would put a request into a shared memory region, notify the server and would in turn be notified when the request has been handled so that the result can be retrieved from the shared memory. While the request is in progress, the VM can continue to run and, for example, receive interrupt messages. However, L4 systems dominantly use synchronous communication, predominantly for simplicity in both the server and the client.

Besides communication between components on the system, VMs and VMMs also need to issue system calls that are handled by the host kernel. System calls may require a fixed time to execute the request while for others no upper bound can be specified. Still, those system calls are implemented such that they allow the host kernel to be preemptive. Further, calls to host kernel functionality might be handled by user-level proxies so that generally system calls must also be treated as communication with other components. Calling other user-level components must also consider the following:

- The communication partner, typically a server, might not be ready to receive any message, for example, because it is handling another request. New requests will

have to wait until the server has finished handling previous requests. Statements on communication duration might be difficult to make depending on the service and its potential clients.

- Priorities in the host system must be considered. Communicating with services that have a lower host priority might be subject to priority inversion. Although timeslice donation, temporarily elevating the receiver's priority to handle the request, has been addressed in the past, open issues, however, remain, especially with multi-processor setups [SBK10].

To avoid blocking of the VM and VMM, the VMM must make sure that, when the guest system runs, L4 IPC and system calls, called from its vCPUs, do not block for an unbound amount of time. The following approaches can be taken to avoid blocking:

1. Sending IPC with zero timeouts can be used to avoid blocking if the receiver is currently not ready to receive IPC. This way the sender can poll until the receiver is ready to communicate and run the VM during polling tries. This method only prevents blocking until the communication partner is ready to receive the message and does not affect the duration of the IPC once the connection has been established.

2. Servers are implemented such that they can handle requests from multiple clients. With the current interface of Fiasco.OC this communication pattern requires one IPC-gate per client. Furthermore, each IPC-gate must be bound to a separate thread as there can be only one (implicit) reply capability. In any case multiple concurrent client requests require that the server prioritizes one client over other clients according to a policy to provide better response times. Due to multi-threading in the server, the server will also get more complex as data structure need to be locked and concurrent execution needs to be considered.

3. The clients, such as the VMM, can use a worker thread to carry out the IPC and thus implement asynchronous handling internally itself.

The option with the best applicability is option 3, as it works with any server interface and regardless of the response behavior of the server. In any case asynchronous handling of communication adds additional code and thus increases code complexity.

3.7.2 Device Emulation

For running guest systems, the VMM needs to emulate devices so that the guest can use its standard hardware drivers. The interface between a device driver and its hardware device is always built for asynchronous communication. The driver puts a request into the device memory (shared memory between host and device) and receives notification via device interrupts.

When designing interfaces for virtual hardware that is used between a guest operating system and its VMM, the same principle must be followed. This allows the guest operating system to stay responsive to events. Furthermore, the VMM also needs to follow this approach, as already addressed in the previous section.

3.7.3 Responsiveness in Paravirtualized VMs

Paravirtualized VMs need to follow the same rules as hardware virtualized VMs regarding interfacing between the guest system and the VMM that implements the communication with other host components. Additionally paravirtualized VMs also use functionality provided by the host kernel, exposed via system calls, which do not follow the typical device pattern.

A typical example in paravirtualized guests is the required creation and destruction of host tasks in the guest environment to work as an isolating container for guest processes. In their native operation the creation and destruction of an address space are mainly operations of memory allocation to get or free memory of process-related data structures such as the page-table. Activating the page-table and thus making it the current one on the CPU is a simple operation of writing a privileged system register. A paravirtualized guest cannot be allowed to control the memory used for the page-table as this would circumvent the isolation provided by the host system. Instead, the host must create, destroy, and activate page-tables. To request that service from the host, the guest must call appropriate host functionality. Remembering the latency requirement of the guest, calling this host functionality must not exceed the guest's latency demands.

Considering an L4 system, operations handling tasks might expose a call duration that is not within the range of required latency and thus not acceptable for the guest. Consequently the guest must call this functionality asynchronously to be able to receive events while the call is being processed. However, it turned out that implementing asynchronous task handling in L^{4}Linux is more complex than initially envisioned. In the following I will walk through a solution for implementing asynchronous host task handling in L^{4}Linux [Lac12].

The critical operation in L^{4}Linux is the destruction of a task as the host kernel needs to not only free the memory for the page-tables used for the task but also any other resource associated with it. Generally, this involves capabilities associated with the task as well as memory and port I/O mappings that need to be removed not only in this task but also in all tasks that have received mappings from the task to be destroyed. Furthermore, in multi-processor setups the Fiasco.OC kernel uses Read-Copy Update (RCU) [Mck04] to synchronize task destruction on multiple cores. RCU involves waiting periods until other cores have gone through a quiescent state. In the case of L^{4}Linux no memory and capabilities have been granted to other tasks from user processes, which reduces the potential work to be carried out by the host kernel. However, this does not alleviate the need for an asynchronous task handling solution as the RCU handling remains.

The first approach to that problem is to create an additional host worker thread, as described previously, called *deletion-thread*. This deletion-thread is performing the actual deletion of the task in the host system. The vCPUs are queuing deletion requests into a work-queue, a data structure available in Linux. The deletion-thread takes out requests from the queue and handles them independently and asynchronously from the vCPUs. The host priority of the deletion-thread is lower than the priorities of Linux's vCPUs, so that those can keep to their event latency requirements. However, establishing the deletion-thread and the work-queue is just the first step. Due to the lower host priority of the deletion-thread, the vCPU must pause to give the deletion-thread a chance to run and perform any task deletions. As the vCPU is executing many Linux processes, the vCPU is never idling as long as any Linux process is ready to run. New processes can be created, which in turn also requires the creation of corresponding host tasks. As the deletion-thread cannot run, no host tasks are deleted, leading to an out-of-resource situation for L^{4}Linux, where no host task cannot be

created anymore.

A solution to this problem might be to wait in the process deletion in the L^{4}Linux kernel until the deletion-thread has deleted the corresponding host task. While waiting, L^{4}Linux is able to receive incoming events and switch to a different context to handle them. However, in the code path for process destruction in Linux, a process cannot sleep anymore as its context is being destroyed. Thus it is not possible to suspend a context at that point.

Contrary to the process exit path, the creation path is able to sleep and thus to wait. This is the point to give the deletion-thread a chance to run when the work-queue size indicates that there are tasks to be deleted. With this approach new processes free the host resources they need for creating new host tasks. An upper boundary for the number of tasks to be deleted can be used to not let new processes stall unduly long.

However, one problem remains in this setup. The deletion-thread needs to be able to run, however, the vCPU will not necessarily release the CPU on its own so that the deletion-thread can run. Thus a process creation will wait on the deletion-thread as long as the vCPU has any process running. For the purpose of suspending the vCPU I have added a kernel-internal Linux process, called *waiter* that is woken up when host tasks shall be deleted by the deletion-thread, and suspended again when the deletion-thread has done its run. This waiter thread calls a host-specific `halt` instruction to suspend the execution of the vCPU and thus give the deletion-thread a chance to run. However, although the waiter suspends the vCPU, the vCPU must still be able to run any real-time process that might be able to run. For that reason the waiter thread is assigned a Linux priority of -20 which translates into the highest priority for any non-real-time Linux process. Consequently, a real-time process must use a priority out of the real-time priority range provided by Linux. Those can, for example, be set with the `chrt` program.

Another design choice for a short latency process deletion is to place the deletion-thread on a host CPU that is not used by L^{4}Linux, if such a host CPU is available and has enough free capacity to not block the progress of the L^{4}Linux. However, it is unlikely to leave a free host CPU unused instead of using its capacity, for example, for the L^{4}Linux. The waiting in process creation also means that this operation can take considerable more time than in native Linux. Consequently a program should not assume any timely execution of process creation functions. As real-time programs are typically divided into a non-real-time initialization phase and the actual real-time operation, and process creation is done in the initialization phase, an enlarged process creation time shall not be an issue for real-time programs.

Data Handling The handling of the shared data structure between the vCPU running operating system code and the deletion-thread must also follow defined rules. Any resource allocation must be handled solely in either the vCPU or the host thread. The reason is that, for example, the allocation of memory for an item in the work-queue uses the Linux memory allocator and that allocator requires a valid Linux context, that is the calling code needs to run in the Linux vCPU and on the appropriate Linux stack. However, the deletion-thread is a host thread and unknown to the Linux kernel itself. Thus, it must not use any Linux functionality that requires the existence of a Linux context. For the allocation of the work-queue items that means, that the deletion-thread is not allowed to delete the items in the queue when the host task has been deleted. The freeing of the memory must be performed by a Linux context. Nevertheless, the deletion-thread must inspect the work-queue to find out about the host tasks to delete. The work-queue is implemented using a Linux

`list` that works independently of a Linux context and can thus be traversed in any context. The deletion-thread only flags items as done so that they can be removed from the list and deleted by a Linux context, which is also hooked into the process creation function.

Besides the shared work-queue, the vCPUs and the deletion-thread are using an interrupt by which the deletion-thread is notified when new tasks to be deleted have been queued. The deletion-thread uses the generic L^4Linux-internal server infrastructure to receive those interrupt events. A prototypical implementation in L^4Linux add about 230 lines of source code.

3.8 Summary

Summarized, the vCPU-based virtualization interface provides a better fit than the previous thread-based virtualization approach for running legacy operating systems as it more closely resembles the execution model of physical CPUs. The model is also better suited for running latency-constrained workloads in paravirtualized setups compared to the thread-based approach as it provides better preemptibility. The concept of vCPUs is also portable across hardware architectures, and has been implemented on x86 and ARM. Conveniently, the vCPU model also fits for providing support for hardware-assisted virtualization, allowing a better combination of virtualization technology.

4 | Real-Time and Virtualization

In the previous chapter we looked at how different flavors of virtualization technology can be integrated into an existing microkernel-based system. Building on the presented solutions we want to proceed with integrating real-time systems.

Real-time systems are systems where timeliness is important, for example, systems where externally imposed deadlines must be met by the system. Schedulers in the real-time systems ensure timeliness of task execution.

Real-time operating systems come in different flavors: A thread library is a software package that provides basic operating system support and a scheduler. The library is bundled with the application. Thread libraries run on wide range of systems as they have low hardware requirements, for example, they do not require hardware isolation mechanisms. On the contrary, general purpose operating systems are increasingly used for real-time use cases as they provide access to a rich set of applications and device drivers.

The question we address in this chapter is this: can we use virtualization to consolidate real-time systems such that all timeliness guarantees of the native systems can also be honored when running these systems in a virtual machine?

These techniques are not limited to classical control-based real-time systems but are also interesting for systems where timeliness is important. Both desktop and server environments can be targets. For desktop systems adequate handling of user interaction is crucial for a decent user experience. On the server side, processing of real-time protocols, such as voice-over-IP, or streaming of video data, requires timely processing for a pleasant user experience.

In this chapter we first enumerate design alternatives, then discuss "Mixed Criticality" and then follow-up with our proposed mechanisms of scheduling contexts and their implementation.

4.1 Design Alternatives

At the core of running real-time systems in virtual machines lies a hierarchical scheduling problem. The scheduler in the host schedules virtual machines, which in turn have their own scheduler as depicted in Figure 4.1.

The generic isolation properties of a virtualization system also include the schedulers. All schedulers in the system run independently. The scheduler in the hypervisor schedules the VM and does not have any knowledge which guest process within the VM will be scheduled when the VM runs. Similarly each individual scheduler in the guest has a local view on its

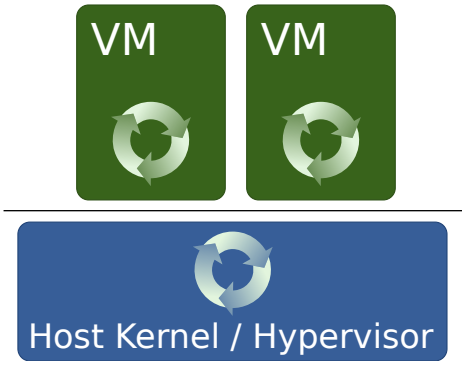

Figure 4.1: A scheduler hierarchy in a virtualization system. Both the hypervisor and the VM guests have a scheduler on their own.

VM-local processes only and neither will nor can consider any global system state.

When a guest is running latency sensitive processes, for example reacting on an event triggered by a sensor, the VM must be scheduled to receive the event so that the guest scheduler is able to run the proper process for handling the specific event.

Common virtualization systems are optimized for throughput rather than event latency. For improving the latency until a VM is scheduled, the following approaches describe two possible options:

Timeslicing VMs Assuming the hypervisor uses a proportional share scheduler with fixed time-slices, a first approach to improve the event reaction latency of code running in a virtual machine is to increase the frequency with which the virtual machines are activated. A typical scheme used by hypervisors is a round-robin based selection of virtual machines. Commonly VMs are run for a long time, for example, Xen runs VMs for 30ms in the default setting [CGV07], allowing the VM to build up a cache working set and benefiting from it.

For VMs with latency requirements, the commonly used large time-slices of VMs is considered too long. For example, assuming 3 VMs with an equal share on the CPU and a 30ms time-slice, a VM would need to wait up to 60ms until it is scheduled again to be able to process a pending event. With more VMs the event would be pending accordingly longer.

Enhancing the hypervisor to switch immediately to a VM, instead of waiting for the next time-slice, would allow to reduce the latency of activating the VM. However, to guarantee the assured CPU share for all VMs, the hypervisor has to implement appropriate budgeting mechanisms to avoid that a VM can exceed its CPU share due to external events. This may in turn lead to the situation that despite a pending event a VM has to wait for CPU time.

To counter the possibly long waiting times, Sisu Xi et al. [Xi+] propose to increase the switching frequency of VMs. The authors report that splitting up budgets into chunks of 1ms does not lead to significant performance problems. They furthermore state that with much smaller chunks the additional scheduling overhead becomes prohibitive. This overhead comes from multiple sources. The available time for a VM to build a cache working set is greatly reduced with an increased scheduling frequency. Further, due to the increased scheduling frequency the host system is also used more often and thus the scheduling and switching overheads in the hypervisor itself sum up. Section 5.4 contains an evaluation of the relation between scheduling frequency, scheduling overhead and application performance.

Priority-based Systems In a host system that supports priorities, a VM requiring timely event reaction latency can be prioritized over other running components, such as other VMs or applications. This way the hypervisor schedules the real-time task whenever an event is pending for it because it has a higher priority than other activities in the system. However, this has a downside, the VM occupies the CPU for as long as it is runnable, blocking all other activities in the system that might also be ready to execute. As previously described for VMs, VMs are likely to also run best-effort work, which will in turn be executed on the high host priority of the VM. In such a scenario, as shown in Figure 4.2, *low*-priority, best-effort work is preventing the *medium*-priority work from being executed because the VM itself must be configured to a high host priority to be able to timely react on events.

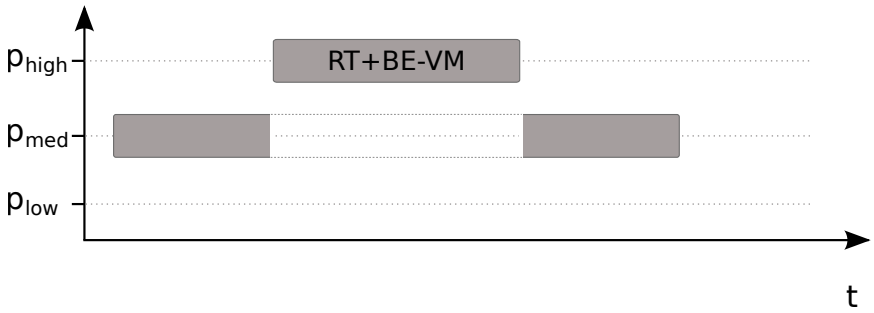

Figure 4.2: A host system running a virtual machine with real-time (RT) and best-effort (BE) tasks as well as a medium priority application. Assuming the best-effort tasks in the VM are low-priority in a global scope, they are preventing the medium prioritized application from running as their container, the virtual machine, is running at the high host priority.

Budgets can be used to limit the execution of the *RT+BE-VM*. However, since such a VM can run both real-time and best-effort work, the VM might not be scheduled upon reception of a real-time-relevant event because the budget has already been depleted by best-effort work. In this case the VM has to wait until a new time-slice begins and the budget is refilled, as depicted in Figure 4.3. This is the same problem as with long time-slices as described previously.

Setting the time-slice length at reach of the latency demands requires to decrease the time-slice and thus increase the host scheduling frequency. This contradicts the goal to keep the host scheduling frequency small, as outlined previously.

Global View Taking a step back, we notice that VMs may run, colloquially speaking, "important" and "less important" work. Taking a global view on a system with multiple VMs we have a system that consists of subsystems of differing importance running in multiple VMs, as shown in Figure 4.4. Such a configuration can also be handled as a mixed-criticality system which we are going to explore in the following. The use the mixed-criticality paradigm will allow us to understand that the simple time-slicing approach is not sufficient to handle the systems as described previously.

4.2 Definition of Terms

Before proceeding with mixed-criticality systems, we want to introduce basic terms used for real-time systems and scheduling(see also 2.6). In real-time systems, a *task* denotes an

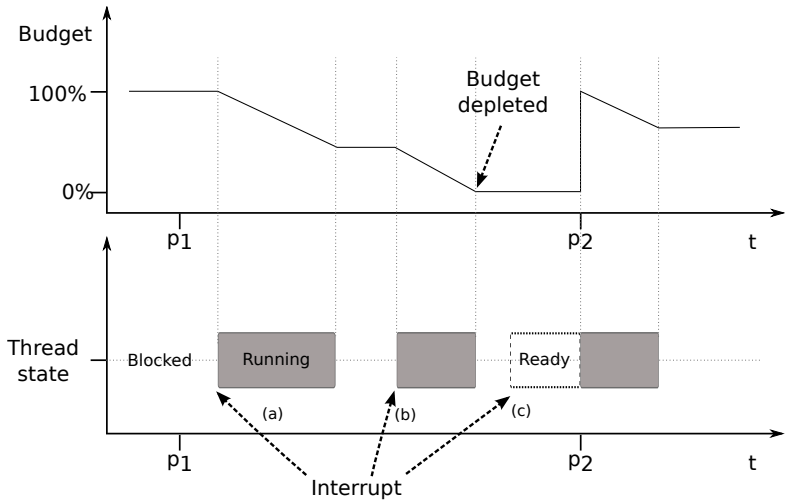

Figure 4.3: A thread with an assigned budget handling interrupt events. With the occurrence of interrupts (a) and (b), budget is available and the thread can run. However, when interrupt (c) occurs, the budget is exhausted and execution of the thread has to wait, depicted by the dotted box, until the budget is refilled at point p_2.

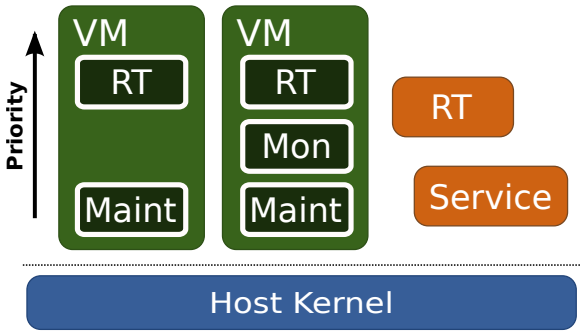

Figure 4.4: View of a system consisting of multiple VMs and native applications with tasks of differing "importance" running both as native applications and inside the VMs.

entity that executes *jobs*.

- A *job* is a piece of work that is characterized through attributes such as deadlines and periods.
- The *WCET*, the worst-case execution time, is the maximum time a job needs for completing its work. The WCET can be, for example, determined by theoretical examination of the whole system, considering worst behavior, or by practical experimentation, depending on requirements. WCETs are also specific to the computing system the software is running on as the execution behavior is largely defined by the hardware platform.
- A *deadline* is a point in time when a job must have finished its work.
- The *period*, also called *time-slice*, defines a time range when a job can be executed. Deadlines are often at the end of a period. Periods repeat, forming *periodic tasks*.

- A *scheduler* is a component that selects the next task to run.
- *Priorities* are a prevalent mechanism in real-time systems to drive scheduling. Tasks have differing priorities which allow the scheduler to select the task with the highest priority.
- A *budget* is used to limit the available CPU share for a task within its period, allowing the scheduler to enforce a maximum CPU usage of a task.

4.3 Mixed Criticality Systems

Mixed-criticality systems [Bar+11] are systems that can handle systems with multiple software components with differing criticality. As analyzing every component in the system to the same degree is time-consuming and costly, mixed-criticality systems allow to restrict analysis of components to the criticality requirement of the particular component, and combine multiple of the components into a system. This scheme can also be applied for real-time systems where the level of evaluation for any component is defined by their timing criticality. In a setup with multiple real-time virtual machines we find such a configuration. The following sections have also been published in [Lac+12].

Mixed-criticality (MC) systems must make sure that tasks with lower criticality will only be allowed to run if tasks with higher criticality are guaranteed to complete. MC systems work under the assumption that a whole system, consisting of multiple components, cannot reasonably be certified under the requirements for the most critical component. Thus only critical components are examined under that regime, using the appropriate tools and effort. Less critical system components are certified with less strict demands, requiring less effort and time. The main objective of a mixed-criticality system is to make sure that a less critical component can never cause a situation where a higher critical component misses its deadline. Microkernel-based systems are especially suited for running mixed-criticality systems as they provide secure and strong isolation for a component-based system architecture.

Defining mixed-criticality systems formally, each task τ is assigned an assurance/criticality level L up to which τ is certified, that is τ is certified for each level up to L. For real-time systems, the certification yields the worst-case execution time C, and as the certification is done for each criticality level up to L, each task has a vector $\mathbb{C}$ of WCETs. The WCETs are only increasing with every criticality level, so that $\mathbb{C}(L) \geq \mathbb{C}(L')$ for $L \geq L'$. For the following we will assume a sporadic task model. That is, a task is characterized by its period, a relative deadline, its criticality level and WCET vector: $\tau = (T, D, L, \mathbb{C})$, and we assume implicitly constrained tasks ($D = T$).

The mixed-criticality scheduling problem can now be phrased as follows: for each criticality level L, if no job of a task τ_j with criticality level $L_j \geq L$ executes longer than $\mathbb{C}_j(L)$, find a schedule such that all jobs of all tasks with a criticality level greater or equal than L complete by their deadline. A schedule is *feasible* if it is a solution to the scheduling problem and it is *optimal* if it finds a feasible schedule whenever there exists one. The MC scheduling criterion requires schedulers to deny the kth job $\tau_{i,k}$ of a lower-criticality task τ_i its requested service if a job $\tau_{j,l}$ of a higher-criticality task τ_j executes longer than $\mathbb{C}_j(L_i)$. In this case $\tau_{j,l}$ denies $\tau_{i,k}$ and we call the earliest point in time by which $\tau_{j,l}$ has executed longer than $\mathbb{C}_j(L_i)$ without completing the *criticality decision point* of L_i.

Looking at virtual machines again, as depicted in Figure 4.5, we see that scheduling in such

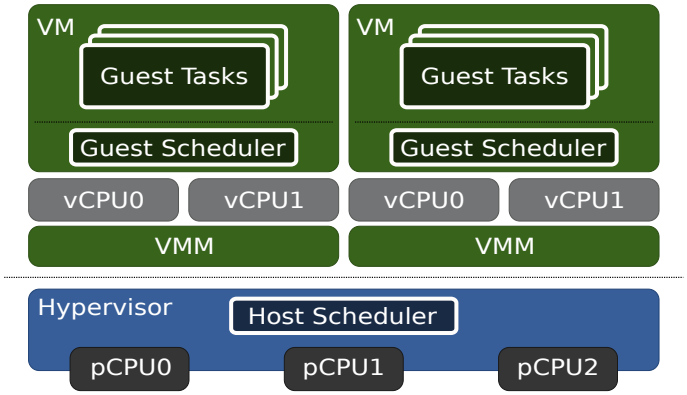

Figure 4.5: A system using virtualization to host two virtual machines.

systems is typically strictly hierarchical. Each VM has its own scheduler, which is responsible to meet all deadlines of the tasks within the VM. The host scheduler then combines these guest schedules by assigning each VM a share of the CPU time. This share is typically characterized by a budget.

The host scheduler does not only have to guarantee timeliness for the guests but also needs to enforce a certain degree of isolation between all guests. For such systems, the host must at least ensure that:

(R1) Scheduling in VM A must not depend on another VM B providing information about the tasks it schedules; and

(R2) The scheduling and, in particular, the feasibility of a schedule in a VM A must not depend on the correctness of any component (including the scheduler) in any other VM B.

Because a guest scheduler in a virtual machine must be certified at least up to the criticality level of the tasks it schedules, we can relax the latter requirement for mixed-criticality systems:

(R2') The feasibility of a schedule produced by VM A for a certain criticality level L_i must not depend on the correctness of lower than L_i certified schedulers and components in other VMs.

Mapping those requirements to our example as seen in Figure 4.5, the hypervisor including the host scheduler must necessarily be trusted by all guests and hence certified at the highest criticality level of any guest task. The VMMs must be trusted only up to the extent that the guest operating system and its scheduler have to be trusted. That is, they must be certified at the highest criticality level of the tasks in the task set of the particular VM.

4.3.1 Mixed-Criticality Examples

In the following we will show, using two examples, that a simple increase of the scheduling frequency in the time-slicing approach cannot cover a system with multiple tasks of differing importance in multiple VMs.

The first example works with two virtual machines that run two tasks each. Three distinct criticality levels are used to characterize the tasks within the virtual machines and all tasks have the same period. The second example also uses two virtual machines, which schedule task sets comprised of two respectively three sporadic tasks with two criticality levels and different periods.

For both examples the used units are normalized to the host system. Further we assume optimal mixed-criticality schedulers in the VMs and neglect all other times spent in the host or guest operating systems. Both examples assume that the hypervisor schedules virtual machines strictly hierarchically. That is, it assigns exactly one budget to each of the two virtual machines in the scenarios.

Example 1

Example 1 demonstrates the possibility of unfeasible schedules when integrating two task sets with two tasks each as seen in Figure 4.6. All tasks in the example share a single global strict period of 8 units of time. Table 4.1 contains the parameters of these task sets.

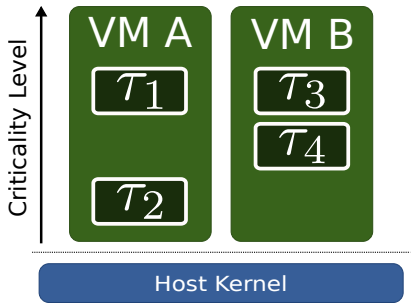

Figure 4.6: Two virtual machines running tasks with different criticality.

VM	Task	T_i	L_i	WCET		
A	τ_1	8	HI	$\mathbb{C}_1(HI) = 4$	$\mathbb{C}_1(MED) = 2$	$\mathbb{C}_1(LO) = 2$
	τ_2	8	LO			$\mathbb{C}_2(LO) = 1$
B	τ_3	8	HI	$\mathbb{C}_3(HI) = 4$	$\mathbb{C}_3(MED) = 2$	$\mathbb{C}_3(LO) = 2$
	τ_4	8	MED		$\mathbb{C}_4(MED) = 3$	$\mathbb{C}_4(LO) = 3$

Table 4.1: Task parameters for Example 1.

Let us first verify that the given WCET values in Table 4.1 are eligible for a valid schedule. Looking at the utilization at each criticality level shows that each task can be scheduled within the limits of the required budgets and the period of 8 time units:

$$\begin{aligned}
\mathbb{C}_1(HI) + \mathbb{C}_3(HI) &= 4 + 4 = 8 && \leq 8 \\
\mathbb{C}_1(MED) + \mathbb{C}_3(MED) + \mathbb{C}_4(MED) &= 2 + 2 + 3 = 7 && \leq 8 \\
\mathbb{C}_1(LO) + \mathbb{C}_2(LO) + \mathbb{C}_3(LO) + \mathbb{C}_4(LO) &= 2 + 1 + 2 + 3 = 8 && \leq 8
\end{aligned}$$

To ensure that all high-criticality tasks meet their deadlines, a minimum of four execution units per period must be allocated to each VM. Otherwise, τ_1^A, τ_3^B, or both may miss their

deadline if they fully need the execution time $\mathbb{C}_i(HI) = 4$ ($i \in \{1, 3\}$) as determined by the WCET analysis tools for the high criticality level.

However, if τ_1^A restricts its execution to the medium criticality regime and thus only executes $\mathbb{C}_1(MED) = 2$ units of time, the local scheduler of VM A can execute other work in the remaining 2 units that have been allocated to VM A. Consequently that will be τ_2^A because it has only a local view on its task set (see Isolation Requirement (R1)). If τ_2^A then uses more than its low-criticality execution time $\mathbb{C}_2(LO) = 1$, for example, because of an error or because the scheduler in VM A does not enforce $\mathbb{C}$ for low-criticality tasks, then the medium-criticality task τ_4^B may miss its deadline if both τ_3^B requires $\mathbb{C}_3(MED) = 2$ units and if τ_4^B requires the third unit as predicted by the WCET analysis tool for the medium criticality level ($\mathbb{C}_4(MED) = 3$). Note, the violation of the mixed-criticality scheduling criterion does not depend on a particular guest or host scheduler but merely on the assigned budgets. For as long as the host scheduler allows VM A to consume 4 units, τ_4^B may miss its deadline because the remaining 4 units do not suffice for $\mathbb{C}_3(MED) + \mathbb{C}_4(MED) = 5$.

This example shows that a time-slicing approach for scheduling virtual machines is not powerful enough to cover the execution of systems with differing criticality, such as real-time and best-effort work. The solution here is to introduce multiple budgets per virtual machine and have them scheduled by the host so that tasks can be scheduled following their criticality level. Table 4.2 provides configurations of global (i.e., host) priorities, parameters and tasks to run on these budgets. Following this assignment, we have to assume that the scheduler in VM A switches to its low budget if τ_1^A completes within 2 units of time.

a)

VM	Budget	Priority	Tasks to run on
A	4	1	τ_1^A
	1	3	τ_2^A
B	5	2	τ_3^B, τ_4^B

b)

VM	Budget	Priority	Tasks to run on
A	4	1	τ_1^A
	1	3	τ_2^A
B	4	1	τ_3^B
	3	2	τ_4^B

Table 4.2: Possible priority/budget allocations for Example 1, smaller numbers denote higher priority.

Figure 4.7 shows a potential sequence of budgets to be scheduled for the distribution shown in Table 4.2b.

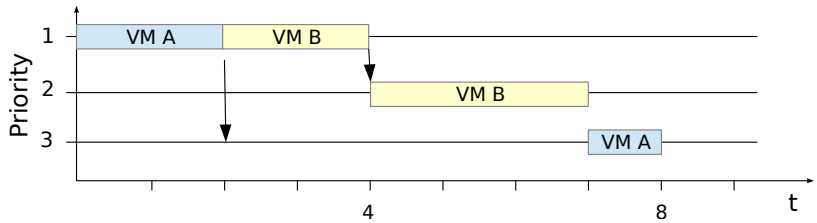

Figure 4.7: Possible schedule for budgets as shown in Table 4.2b for example 1. Down-pointing arrows indicate a switch of budget.

Example 2

Figure 4.8 shows a schedule for the simultaneous release of the task sets of Example 2 which has the task parameters shown in Table 4.3. Again, we want to verify first that the given

task set can be scheduled when assuming a global system view. As the tasks have differing periods, we normalize the budgets to the hyper-period 16 by using a multiplication factor $16/T$ with function $N(C,T) = (16/T) * C$.

$$N(\mathbb{C}_1(HI), T_1) + N(\mathbb{C}_4(HI), T_4) = 4 * 2 + 1 * 6 = 14 \leq 16$$
$$\sum_{i=1}^{5} N(\mathbb{C}_i(LO), T_i) = 1 * 2 + 1 * 4 + 4 * 1 + 2 * 1 + 1 * 4 = 16 \leq 16$$

With the shown results we see that the utilization of the system is 1 at a maximum for both criticality levels and thus the given task can be scheduled. A schedule for the low-criticality task set is shown in Figure 4.8.

VM	Task	T_i	L_i	WCET
A	τ_1	8	HI	$\mathbb{C}_1(HI) = 4$ $\mathbb{C}_1(LO) = 1$
	τ_2	4	LO	$\mathbb{C}_2(LO) = 1$
	τ_3	16	LO	$\mathbb{C}_3(LO) = 4$
B	τ_4	16	HI	$\mathbb{C}_4(HI) = 6$ $\mathbb{C}_4(LO) = 2$
	τ_5	4	LO	$\mathbb{C}_5(LO) = 1$

Table 4.3: Task parameters for Example 2.

Now we explore the case of running the tasks in their respective virtual machines in a time-sliced fashion. We can distinguish four phases which correspond to the period of 4 of the tasks τ_2^A and τ_5^B as seen in Table 4.3. Irrespective of when the hypervisor switches to VM A or VM B in the first phase, VM A and VM B cannot both execute τ_1^A for $\mathbb{C}_1(LO) = 1$ and τ_4^B for $\mathbb{C}_4(LO) = 2$ while meeting the low deadlines of τ_2^A and τ_5^B if both τ_1^A and τ_4^B would complete before their criticality decision points. For the same reason τ_1^A cannot completely be delayed to Phase II. As a consequence, A needs at least a budget of 2 in Phase I and B a budget of at least 1.

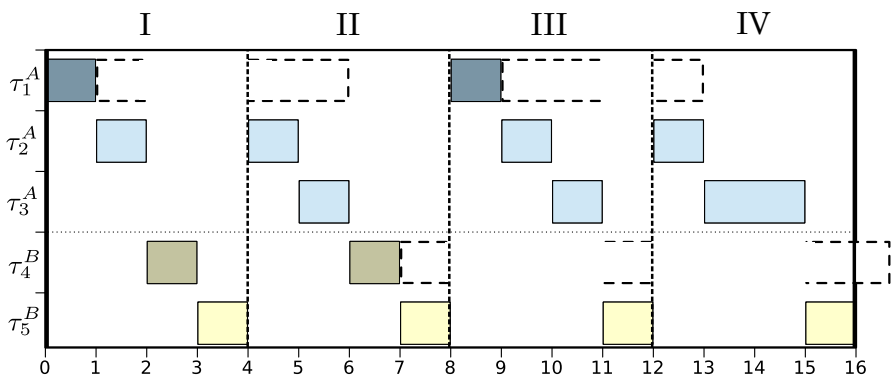

Figure 4.8: Schedule for the simultaneous release of the task set in Table 4.3 on top of a mixed-criticality system. Filled bars show low WCETs ($\mathbb{C}_i(LO)$), dashed bars show the time $\mathbb{C}_i(HI) - \mathbb{C}_i(LO)$ that is required to complete high tasks that do not complete before $\mathbb{C}_i(LO)$.

Figure 4.8 illustrates the case where both VMs receive the same budget of length 2. It is easy to see that the arguments that we give hold also for all other sensible budget assignments.

With a 2 : 2 budget assignment in Phase I, VM A can execute τ_1^A for one unit of time, which allows the scheduler in VM A to decide whether τ_2 is released or, if τ_1^A does not complete by $\mathbb{C}_1(LO)$, whether to allow further execution of τ_1^A. Remember the mixed-criticality scheduling rule gives no further guarantees to LO tasks if a HI task executes longer than its LO WCET. VM B has to execute τ_5^B because VM isolation (R1) prevents B's scheduler from knowing whether or not τ_1^A has already completed by $\mathbb{C}_1(LO)$. It may drop τ_5^B only after τ_4^B has executed longer than $\mathbb{C}_4(LO)$. Following Baruah et al. [BBD11a], we call the situation caused by τ_5^B *criticality inversion*. Following a 2 : 2 budget in Phase I, A needs at least a budget of 2 in Phase II to guarantee completion of τ_1^A in the situation when τ_1^A did not complete by $\mathbb{C}_1(LO)$. An assignment of a larger budget to A is counterproductive as this would result in a remaining LO utilization for the two remaining phases of 1 and a remaining HI utilization of 9/8 (i.e., > 1). If both τ_1^A and τ_4^B are not completed by their LO WCETs, a completion by their HI WCETs can therefore no longer be guaranteed. The key insight that completes this example is that any execution of τ_3^A for longer than 1 unit of time may result in τ_4^B missing its deadline if it has executed longer than $\mathbb{C}_4(LO)$. However, without knowing the progress of τ_4^B, VM A cannot decide whether or not to execute τ_3^A at time 14 in Phase IV. Following the same line of argumentation, it is easy to see that also for other budget assignments the taskset in Table 4.3 is not feasible for mixed-criticality hypervisors that assign only one budget per VM. A priority assignment π with $\pi(\tau_3^A) < \pi(\tau_4^B) < \pi(\tau_5^B) \leq \pi(\tau_2^A) \leq \pi(\tau_1^A)$, that is two budgets for VM A to execute τ_3^A and τ_2^A, τ_1^A interleaved with the tasks of VM B, however leads to a feasible (fixed-priority) schedule if we assume that VM A stops τ_2^A latest after $\mathbb{C}_2(LO) = 1$ and that it switches to its low budget if τ_1^A completes before $\mathbb{C}_1(LO)$. To fulfill the Isolation Requirement (R2'), we do not have to require a similar precaution for τ_3^A.

Conclusion We summarize that time-slicing multiple VMs is not a generally applicable method for running multiple virtual machines with real-time and best-effort tasks on one system. Although increasing the scheduling frequency in the host system improves on the achievable event latency in the guest, it cannot solve the problem of real-time tasks being blocked by less important tasks.

We propose an approach to enable the interleaved execution of virtual machines to flatten the hierarchical scheduling problem by exporting some parts of the guest scheduling to the host. This allows the host system to have a global view on the real-time requirements of the system without the need to know about all the best-effort threads running in all virtual machines. The approach requires that the guests are modified to export selected parts of their internal scheduling behavior to the host. In the next section we will detail on the mechanisms for this functionality.

4.4 Exporting Guest Scheduling to the Host Scheduler

As seen in our examples, we need to export scheduling information from the guest to the host. That information describes which type of work is running in the guest so that the host can adapt the scheduling of the virtual CPU accordingly. Additionally the host needs to know how this exported information translates to actual scheduling configurations on the host.

An important aspect with regard to security is how the responsibilities and possibilities

for interaction are distributed throughout the system. For example, while a guest virtual machine can expose knowledge of its internal scheduling behavior it shall not be able to set scheduling parameters in the host. Another component is entitled to combine exported scheduling decisions from a guest and actual settings for the host scheduler.

Generally, the following components in a system are involved with scheduling:

Guest Virtual Machines and Applications Guest virtual machines run operating systems that have their own scheduler and schedule according to their local view. Selected scheduling parameters can be exported to the host. Both VMs and applications need to be scheduled by the host and both need to express their resource requirements to the host.

Host Kernel The host kernel has a global view on the system and schedules all host-visible applications and VMs. Temporal isolation is ensured by appropriate scheduling methods.

Admission The admission component is a system application and responsible to manage scheduling configurations of its clients. Scheduling parameters of all the clients are checked to produce a valid schedule for the system and finally passed to the host kernel. So-called online admission systems can cope with arbitrary number of clients. They can deny a client's configuration because it demands resources beyond the capabilities of the system and they may negotiate applicable schedules with a client. Multiple admission components can exist in a system, while there must be one root admission service. In contrast to online admission with an active system component, offline admission defines all scheduling parameters statically at configuration time of the system.

Figure 4.9 depicts a schematic view on such a system with multiple admission components and clients. $Admission_1$ is the admission component with the global view, while $Admission_2$ is both an admission service to its clients as well as a client to $Admission_1$. An admission domain includes an admission component with its clients.

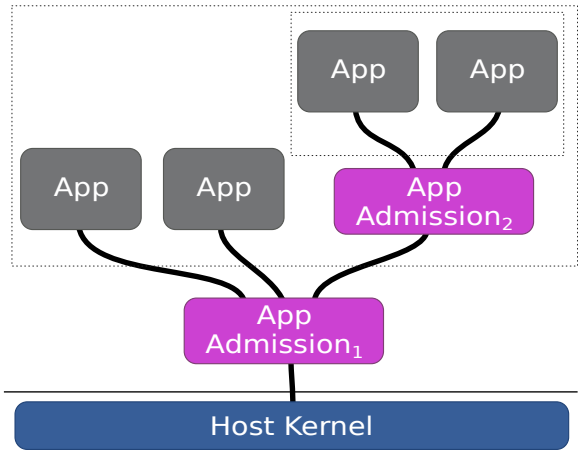

Figure 4.9: A schematic view of a system with multiple admission services. Each admission service is an application in the system itself. Dotted boxes denote the *admission domain* for each admission component.

Summarized, we need to provide a mechanism to securely export selected scheduling events from the guest to the host. The mechanism must be built such that the guest informs the

host about a change in scheduling, however, another independent component is responsible to configure global scheduling policy so that the untrusted guest is unable control global scheduling. The admission component is responsible to combine the information provided by the guest with the host's scheduling configuration.

Scheduling Contexts

Traditionally threads are represented in operating system kernels by a collection of thread-specific data, called *Thread Control Block* (TCB). A TCB also includes information required for scheduling. In this work we separate the information relevant for scheduling from the TCB and call the entity holding scheduling parameters a *Scheduling Context* (SC). The term '*scheduling context*' is derived from previous work on scheduling in the Fiasco microkernel [Ste04].

For the remainder of this work, we assume that the scheduling parameters represented by a SC are a budget, a period and a priority. As SCs are a mechanism of the host kernel, the priorities are global priorities.

Each thread has a default SC. Default SCs have an infinite budget. Additional SCs can be created and configured with a budget, period and priority. The default SC of a thread typically has the lowest priority of all SCs of a thread. The `set_sc()` function allows a thread to select among its available SCs. Threads running on the same priority are scheduled in a round-robin fashion. SCs are equally available for vCPUs and all statements for threads regarding SCs are equally valid for vCPUs.

For explaining the use of scheduling contexts, we assume they have been created and configured.

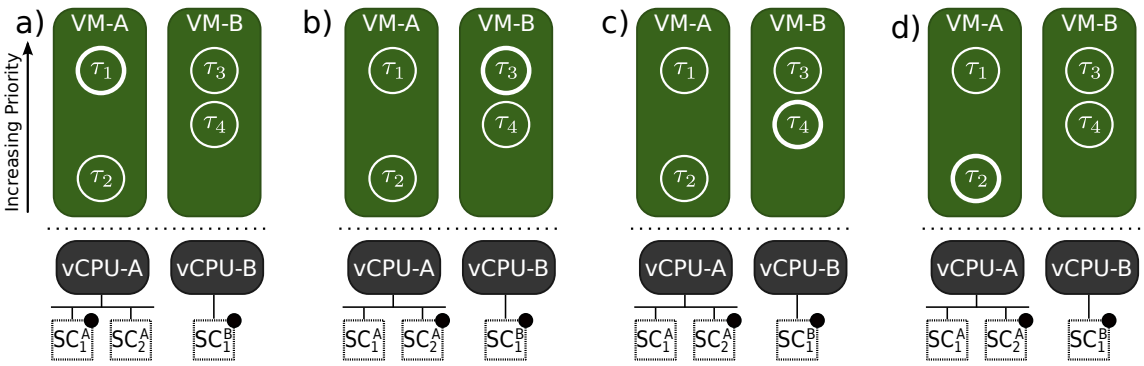

Figure 4.10: Step-by-step illustration of a sequence of scheduler context switches (`set_sc()`). Bold circles indicate the running task and the currently used and highest prioritized scheduling contexts.

Figure 4.10 illustrates the use of the `set_sc()` function, to resolve the mixed-criticality scheduling problem of Example 1. The three budgets (4, 1, 5) as shown in Table 4.2a already solve this problem if the host creates three SCs with the same budgets and if the guest scheduler in VM-A invokes `set_sc()` to switch from the SC_1^A to the SC_2^A in the event that τ_1^A completes before $\mathbb{C}_1(LO)$. The sequence starts with with τ_1^A using SC_1^A which has the highest priority, followed by a switch to SC_1^B to run τ_3^B and τ_4^B (Fig. 4.10b and 4.10c). Finally, τ_2^A will be run on SC_2^A in VM-A (Fig. 4.10d), concluding the sequence.

More insights in our approach can be drawn from a discussion of this scenario with two

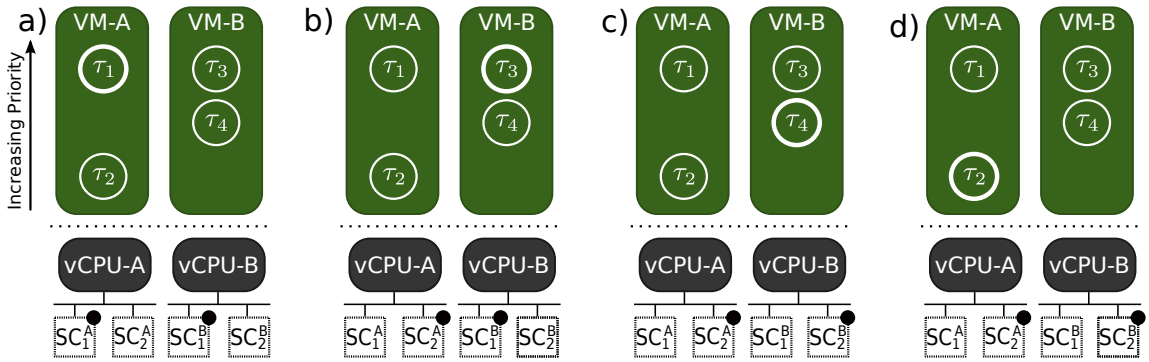

Figure 4.11: Modified sequence of Figure 4.10 using two SCs for VM-B.

budgets for each of the two VMs (i.e., one SC (SC_i) for each of the four tasks (τ_i) parametrized as described in Table 4.1). The priorities π of these SCs are $\pi(SC_1^A) = \pi(SC_1^B) = 1$, $\pi(SC_2^B) = 2$, and $\pi(SC_2^A) = 3$. Smaller numbers indicate higher priorities. We discuss Example 1 for the simultaneous release of all tasks. At time 0 (relative to this simultaneous release), both SC_1^A for τ_1^A and SC_1^B for τ_3^B are used. Irrespective of the host scheduling policy, both VMs receive a share of 4 units at the highest priority to complete τ_1^A and τ_3^B in the event that not both complete before $\mathbb{C}_1(LO)$, respectively before $\mathbb{C}_3(MED)$. If one of these tasks completes latest after 2 units, the corresponding guest scheduler invokes `set_sc`(SC_2^A) (for VM-A) or `set_sc`(SC_2^B) (for VM-B) to switch to the respective lower prioritized scheduling context. Fig. 4.11a and b depict this situation for the case where τ_1^A completes first. After both VMs have dropped to their lower prioritized budgets (Fig. 4.11c), SC_2^B has a higher priority than SC_2^A, which allows τ_4^B to complete even in the case that τ_1^A exceeds its budget. Finally τ_2^A runs (Fig. 4.11d), completing the sequence.

Conclusions The two examples and in particular the two solutions to Example 1 show that the number of exported SCs heavily depends on the host and guest scheduling policies and on the workload to be scheduled. For fixed-priority schedulers in all guests and in the host and for criticality monotonic priority assignment [BBD11b], a relatively easy mapping of guest tasks to host SCs is demonstrated in the 4 SC variant of Example 1: The scheduling parameters of every task are directly exported and the local priorities are interleaved in such as way that criticality levels are preserved. That is, the priorities of all high-criticality tasks are strictly higher than the priorities of all medium-criticality tasks and all low-criticality tasks, etc. The correct interleaving within these priority bands must of course be validated by the admission test performed by the host.

The two examples also show possibilities for reducing the number of SCs. For example, Table 4.2a shows a mapping for Example 1 with one SC per criticality level. A generic analysis of guest task to host SC mappings is a topic on its own and is left for future work. However, we have made a first step analyzing the applicability of SCs to various mixed-criticality scheduling algorithms, see Section 4.6 and Völp et al. [VLH13].

4.4.1 Interrupt Service Routines

So far, selecting a scheduling context is under control of the guest operating system. This raises the question how to trigger these operations from outside events.

Scheduling decisions in VMs are triggered by injecting interrupts such as timer or device interrupts. Upon receiving these interrupts, the guest runs the corresponding interrupt service routine (ISR) to decide how to react on these asynchronous events and how to adjust the VM internal scheduling. For example, a guest with one-shot timer may have programmed a timer to the minimum of the absolute deadline of the currently active task and to the point in time when the budget of this task will be depleted. When receiving this timer, the corresponding ISR invokes the scheduler to select the next task to run.

From the perspective of the VM, this service routine runs non-preemptively, that is, effectively at a priority above the priorities of all tasks in the VM. However, from the perspective of the host, the VM might run at a host priority that is lower than other runnable VMs or host threads and thus might be blocked from running. Consequently, injecting events, such as interrupts, to the VM will be deferred until the VM is eligible for execution in the host again. This also means that the VM cannot switch to a scheduling context that would allow it to change to a host priority that is above the currently runnable set of VMs and threads.

To resolve this situation, we invented a mechanism to boost the VM to a host priority sufficiently high to allow the VM to execute whenever an interrupt is injected that triggers work that must be run on an elevated host priority. The approach is to bind an SC with such an elevated priority to an interrupt. Whenever an interrupt is injected to a vCPU or thread, the host kernel will switch to the associated scheduling context.

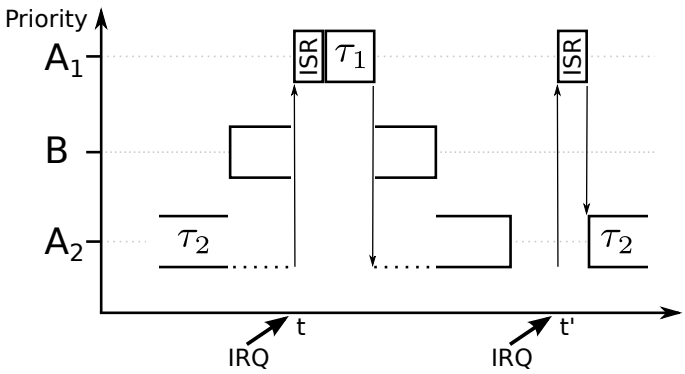

Figure 4.12: Scheduling context activation at the occurrence of asynchronous events such as interrupts or the expiration of a timer. The interrupt service routine always runs on priority A_1 until the guest scheduler, which it invokes, decides which task to run.

Figure 4.12 depicts a possible usage of the described mechanism by showing the release of τ_1 and the activation of the interrupt service routine that follows. Fig. 4.13 presents the same step-by-step illustration as Fig. 4.10 but for the scenario of Fig. 4.12, which includes interrupts. At time t, the host receives an interrupt (IRQ), which triggers the release of τ_1 and later (at time t') of τ_2 in VM A. The hypervisor therefore switches to the VMM of VM A, which in turn injects the interrupt into this VM. Because the interrupt is associated with SC_1^A (i.e., budget A_1), the interrupt service routine always runs on this highest prioritized scheduling context. In the first situation (at time t), the guest scheduler releases τ_1 and drops to SC_2^A (i.e., budget A_2) only if τ_1 completed before $\mathbb{C}_1(LO)$. In the second event (at time t'), the guest scheduler immediately switches to SC_2^A to release the second job shown for τ_2.

Notably the mechanism also allows to bind individual SCs to interrupts, thus for example allowing to run a VM at the host priority required to run the interrupt service routine for a

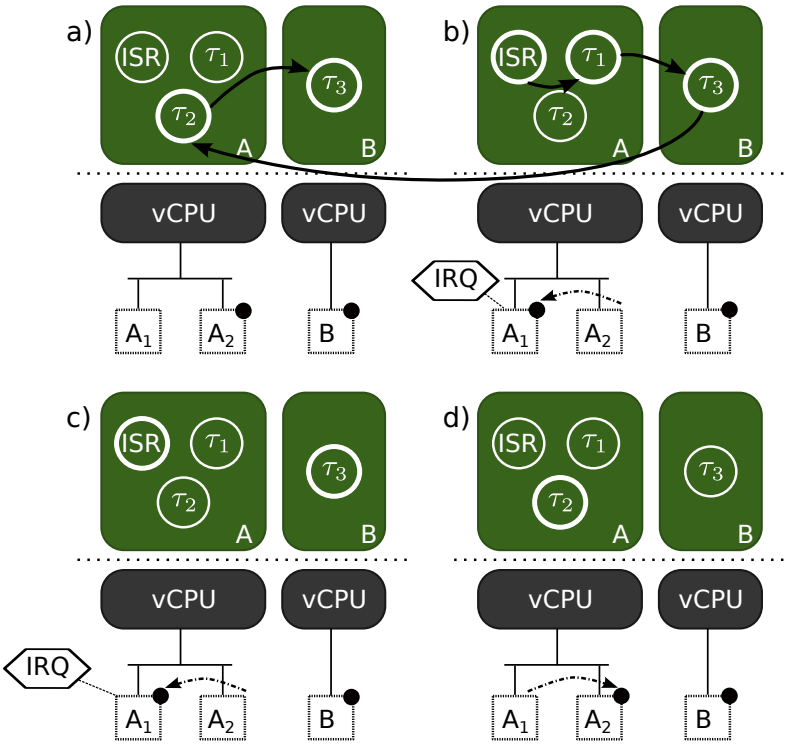

Figure 4.13: Example sequence of actions as depicted in Figure 4.12. a) and b) illustrate the first ISR invocation. After τ_3 has finished, the state of a) is restored. The second ISR invocation is depicted by c) and d).

specific device.

4.4.2 Guest Operating System Modifications

As described previously, guests must use the `set_sc()` function to inform the host of relevant scheduling decisions.

Generally, for an arbitrary guest, we have to add the following functionality:

- After every priority change in the guest, the corresponding SC must be activated by means of `set_sc()`, if this SC is not already active. A common place for this call to `set_sc()` is after the invocation of the scheduler and switching tasks.
- For every interrupt service routine that has an SC associated, the ISR-SC shall be deactivated and the previous one activated unless execution shall continue on that ISR-SC, for example, because a high-priority task has been selected by the internal dispatching decisions.

For specific guests a subset of these modifications may suffice depending on the features the guest provides.

Guests that are available in source code and can be recompiled can directly add the hooks into the code. In the implementation section we will discuss how the hooks are added to the two popular guest operating systems FreeRTOS and Linux.

For guests without source code availability adding the hooks is possible by instrumenting the binary code. However, finding the necessary locations to patch is challenging. I will describe possible options in the next section.

4.4.3 Blackbox Guests

Blackbox guests are not available as source code and thus cannot be modified by changing the source code and recompiling them. As described previously, SCs need to be changed at runtime of the guest at defined events. As the guest kernel cannot be directly modified, other techniques must be used to detect the described events and act accordingly.

Although the guest kernel might only be available in binary form, loading a kernel module at runtime is a common feature. This allows to add functionality to the kernel and is commonly used for device drivers, including third-party drivers. However, interfaces to hook into scheduling or thread handling are usually not offered to kernel modules. Nevertheless there is a use for SCs. A loaded driver can use a scheduling context to run its interrupt service routine at an elevated host priority. As the SC handling is in the sole responsibility of the driver, the ISR has to return the VM to the default SC at the end of its ISR.

A more sophisticated possibility is to detect the code paths in the guest where SC handling is required and to mark those locations in the binary. The marking can be done by replacing the relevant locations with instructions that trap into the VMM, so that the VMM can take the appropriate actions with regard to the SC handling and also make the guest execute the replaced code. If the architecture offers a sufficient number of hardware breakpoints those can be used to identify the execution of those locations without the need for modifying the guest kernel. For the techniques to work those locations must be identified. Depending on the usage scenario that can be done once, even manually, if the guest binary code is known. If the guest binary code is not available at system configuration time, the analysis of relevant locations in the guest must be done at runtime.

Analysis of the Linux kernel shows that two operations can be identified at which binary instrumentation is necessary to switch scheduling contexts: switching the stack from one kernel-internal thread to another, and returning from an interrupt context. They can be identified by analyzing the assembly instructions of the guest kernel. We will look at both the x86 and ARM architectures.

Stack switching On both ARM and the x86 architecture, the stack pointer is a register that can be written to and read from. As the C family of programming languages does not offer to directly set the stack pointer, operating system kernels must use assembly code to read and write it. Nevertheless, the compiler-generated assembly code contains instructions handling the stack pointer but those are limited to operations of increasing and decreasing it, as well as storing and reloading it to facilitate stack variables and function call nesting. Therefore the location where threads are switched in the guest kernel is detectable by finding a unique pattern of stack pointer handling operations.

On the x86 architecture, stack switching can be identified by locating the following pattern of instructions:

```
mov %esp, MEMORY
mov MEMORY, %esp
```

Only a single occurrence of this pattern can be found in a Linux kernel, including an L^{4}Linux kernel.

Linux for the x86 architecture may also switch stacks for calling interrupt routines onto an extra stack for interrupt handling. Those locations can be identified with the following pattern:

```
xchg REG1, %esp
call *REG2
mov REG1, %esp
```

However, not all interrupt handlers are called with different stacks, so that this location cannot be relied on to mark the end of an interrupt service routine. However, in those cases the interrupt return can be detected, as discussed in the following.

On the ARM architecture, the thread switching code can be identified with the following pattern when using ARM code:

```
stmia reg!, { ..., sp, lr}
...
ldmia reg, { ..., sp, pc}
```

The last `ldmia` instruction loads a set of registers from memory, including the new stack-pointer and the program counter. This instruction uniquely identifies the thread switching code throughout the whole kernel as this instruction is not generated by the compiler. It also only occurs once through the whole binary of the kernel. The ARM Linux kernel does not change stack for handling interrupt routines.

Interrupt handling Interrupts are asynchronous events that interrupt the currently running context by storing its state to a known location and branching to the interrupt service routine running in the kernel. Upon completion of the routine, the original context, or possibly another selected one, is resumed. As this operation possibly also involves a CPU mode switch, for example, when resuming to user code, CPUs provide specific instructions for that purpose. Those specific instructions can be found by analyzing the binary code to eventually handle the necessary SC work of switching away from the interrupt SC if required.

On the x86 architecture, the instruction for loading context state from memory into the CPU is called `iret` and easily detectable. The ARM architecture principally offers two instructions for that purpose. The `rfe` instruction (*return from exception)* has been introduced with the ARMv7 architecture revision, however, it is not used in Linux[1]. Linux uses an `ldm` instruction in its *exception return*:

```
ldmXX sp, {rX - pc}^
```

This exception return `ldm` can be uniquely identified as the `pc` register is given in the register list and the exception return is enabled as symbolized the caret sign at the end. Considering a binary-only L^{4}Linux, a `l4_vcpu_resume` system call is used to switch to another context which must be intercepted.

[1] An exception are kernels built for Thumb2 mode that use the `rfe` instruction as the `ldm` method is not available in that execution mode.

The return-from-interrupt code path is not only run after an interrupt service routine has finished but also for any other switch of context that involves a privilege mode change, i.e. it is called more often than required for SC-handling work. Especially it is also used when returning to a user program from a system call. Intercepting the resume instruction has thus a performance impact on normal user programs.

A possibility to avoid unnecessary detection of return-from-interrupt executions would be to only enable those after an interrupt has been injected and disable them again after a return-from-interrupt has been detected. However, it can be problematic to detect injections of interrupts.

Hardware-assisted virtualization provides the possibility to mark an interrupt for injection which is then carried out by the virtualization support in the CPU when appropriate, i.e. the virtual CPU is ready to take the interrupt. Thus, when the VMM receives a return-from-interrupt trap, this might not necessarily be from the interrupt service routine but could, for example, also be the final part of a system call processing. As the kernel uses the same exit paths for both functions, the VMM cannot differentiate between the two different events.

Also, the VMM might not be involved in injecting interrupts at all. As the injection method is uniform for any device that is directly passed to the virtual machine, the VMM might be surpassed to gain execution performance as the host kernel can inject a hardware interrupt directly into the VM. The same applies for paravirtualized guests where all interrupt are uniformly handled and thus an interrupt can be directly injected to the guest by the host kernel without any intermediate VMM or monitor component.

For handling scheduling contexts for the guest, this fast-path cannot be used and the VMM must inject interrupts, ensuring that the guest is in a state ready to take a interrupts. This way the VMM can ensure that the exception return following the injection of the interrupt is from the interrupt routine. However, this assumes that an interrupt handler does not cause any resolvable exceptions itself and does not cause any mode change before leaving the interrupt handling routine. When the VMM is injecting interrupts, it can also install a trap to exit the VM upon the exception return, and uninstall the trap again when the exit has happened. As described this allows to remove the penalty for system calls in the guest that use the same exit path when returning to user-mode. In case the number of system calls does not out-weight the number of SC-enhanced interrupts taken, the installation and removal of the trap instruction can just be omitted.

Analysis has shown that using a non-modifiable guest with scheduling contexts is possible but has trade-offs in engineering effort and runtime performance. Considering closed-source guests the analysis becomes more challenging as only the binary itself can be used instead of its source code. It is therefore advisable to use such techniques only as a last resort.

4.4.4 Further Use Cases of Scheduling Contexts

We introduced scheduling contexts by integrating them in mixed-criticality systems using two examples. There are more use cases where using SCs can be beneficial and which we want to introduce briefly.

Emergency Alarm System The "emergency-alarm" system includes two virtual machines: $VM1$ consists of τ_{alarm} and τ_{maint} and $VM2$ of τ^{M}_{video}. τ_{alarm} is a sporadic task and τ_{maint} is a best-effort task, which gathers statistics for maintenance purposes. τ_{alarm} has

a low minimum inter-arrival time and a worst-case execution time nearly as high as the inter-arrival time, leading to very high utilization. τ^M_{video} in $VM2$ is a high-utilization task, for example, doing periodic image processing of the connected video surveillance system. τ_{alarm} has an extremely low probability for high-frequency alarm showers. However, it is very important that deadlines are met in these rare situations. τ^M_{video} is much less important than τ_{alarm}, but more important than the maintenance task τ_{maint}.

Without the possibility to differentiate τ_{alarm} and τ_{maint} in $VM1$, the system would need to be set up in a way that $VM1$ gets a higher host priority as $VM2$ to fulfill the low-latency requirement of τ_{alarm}. However, τ_{maint} will then also be run under that host priority, potentially taking away CPU capacity from τ^M_{video} in $VM2$. Considering the availability of budgets would allow to give $VM1$ a considerably smaller CPU share than $VM2$, however, that leaves no room to run τ_{maint} in $VM1$.

With the availability of multiple SCs per virtual machine, the configuration for this described use case could look as show in the following table:

	$VM1$	$VM2$
τ_{alarm}	SC(1, 10, 3)	
τ^M_{video}		SC(2, 1000, 600)
τ_{maint}	SC(3, -, -)	

Table 4.4: Possible setting of SCs to virtual machines for the *emergency alarm use case*. The parameters for the SC configuration are *SC(host-priority, period, budget)*. τ_{alarm} has the highest host priority and can get a CPU share of 30% with a period of 10. τ^M_{video} gets 60% CPU share with a much higher period. The remaining CPU capacity is left for τ_{maint}.

Interactive Systems Another example for employing SCs are systems with interactive processes, such as desktop systems with graphical user interfaces (GUIs). Modern operating systems have means to discover and differentiate between interactive and background processes. Interactive processes get their priority boosted to improve the usability of the system. If several virtual machines with such interactive processes change focus, single-budget allocation schemes for the VMs cannot consider the priority adaptions without further knowledge of VM-internal task priorities. A way of solving this issue is to use a high switching frequency between the virtual machines. However, this increases the overhead in the system as described previously. The assignment of multiple interrupt triggered SCs for input devices (for example, keyboard and mouse) allows to minimize these switches to when they are needed.

Server and Cloud Systems Processing of latency-constrained workloads is also done in server and cloud systems. For example, handling real-time communication data, such as Voice-over-IP (VoIP) and video conferencing, or online gaming services, require timely processing. In a server infrastructure using virtualization, scheduling contexts can help to improve the latency of applications while still sharing a single physical system among multiple VMs. SCs can foster consolidation approaches for latency-constrained applications in the cloud by using fewer hardware resources.

4.5 Implementing Scheduling Contexts

After describing the mechanism of scheduling contexts we concentrate on their implementation in the Fiasco.OC microkernel now.

We will introduce use cases and then present two implementation variants. A very flexible approach which, however, raises implementation challenges, and a less flexible approach which still allows to cover the majority of use cases of scheduling contexts but significantly reduces implementation complexity.

4.5.1 Security and Performance Requirements

Before going into implementation details of scheduling contexts we first need to look at the security and performance characteristics of a possible implementation. The following has been published in [LVW13].

Security For understanding security implications we first need to understand the interaction of system components and their trustworthiness to other components in the system. Let us assume a typical microkernel-based system, consisting of the microkernel and multiple subsystems, for example, applications or virtual machines. Here, the microkernel must ensure isolation between those subsystems, both temporally and spatially. Concerning scheduling contexts, which an application or VM may want to use, the system cannot allow uncontrolled creation of SCs nor uncontrolled configuration of those. Uncontrolled creation can lead to exhaustion of system resources and arbitrary settings of scheduling parameters have an influence on the global scheduling, likely to be negative in a global view as applications try to monopolize the CPU. Consequently, it must be possible to control and limit the creation and configuration of SCs. Further, SCs must only be usable by the subsystems they have been assigned to so that only those can use the provided CPU time of the SC.

Performance Characteristics The most frequent operation that is used with scheduling contexts is activating an SC for a host thread or vCPU. This `set_sc` operation is called throughout the runtime of a vCPU. In comparison, the operations of creating and configuring scheduling contexts is muss less frequently. The focus is therefore twofold: the `set_sc` function must be optimized for performance while creation and configuration can focus flexibility.

4.5.2 Fiasco.OC Scheduling Interface

Fiasco.OC's `L4::Scheduler` interface is used to set scheduling parameters and CPU affinities of threads and vCPUs. In particular, the `run_thread()` function allows to initially start threads with scheduling parameters on a specific CPU as well as change this configuration while running. The scheduler of an application or VM is accessible through a capability. Calls to the kernel scheduler can be interposed by user-level components implementing the `L4::Scheduler` interface. Using interposition, user components can implement policies and strategies for running and controlling threads, for example, limiting priorities of their subsystems or doing load balancing over multiple cores. As each application in the system requires an assigned scheduler, an application setup can be configured to route the application's scheduler queries not to the kernel but to another user-level component.

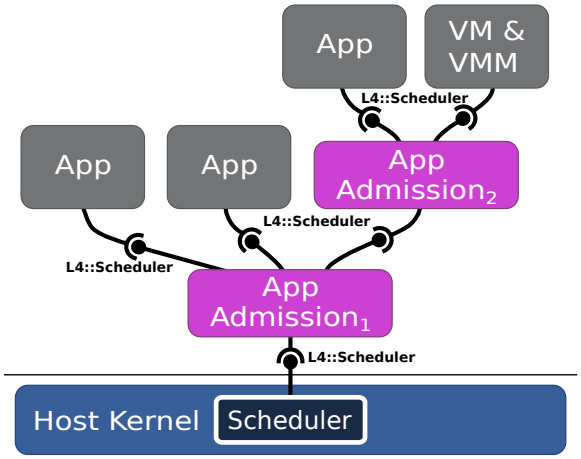

Figure 4.14: A schematic view on a hypothetical system with two *L4::Scheduler*-proxies, allowing to refine scheduler API invocations of clients. The proxies are service user and provider at the same time.

This component then receives any scheduler-related messages, can monitor or change the supplied scheduling parameters and pass it on to its scheduler interface. A schematic view of a possible composition is shown in Figure 4.14. This view is an implementation-focused view of the previously shown Figure 4.9.

Next, we focus on the integration of scheduling contexts into the system.

4.5.3 Managing Scheduling Contexts

As already outlined, additional scheduling contexts for a thread must be created and destroyed, configured with scheduling parameters and used by threads.

Creating and destroying scheduling contexts is by itself not security critical as non-configured SCs do not posses any CPU time and are thus of no use. However, the memory on which the scheduling context has been created must be accounted to a quota.

Principally, scheduling contexts can be part of some already existing object, or they can be a first-class object with capabilities referring to them. The first approach suggests a simpler implementation while the second approach promises more flexibility. Figure 4.15 illustrates the possibilities with the given variants.

In configuration A, the VMM creates vCPUs on behalf of the guest together with the scheduling contexts. For letting the vCPU run, the VMM invokes the scheduling interface of the admission component passing it the vCPU capability. The admission service will then apply its policy on the client's invocation and finally pass the request on to the kernel's scheduler. From this point on the vCPU can run. However, if the admission component requires to change the parameters it has assigned to the vCPU, for example, to lower its priority, it may be unable to change it. The reason is that the VMM might have revoked the vCPU capability from the admission component, leaving Admission no possibility to name the vCPU and thus change parameters. Therefore time once granted may never be reclaimed unless all clients of Admission are trustworthy, or are destroyed.

Scenario B shows a first solution to this problem. Rather than creating vCPUs in the VMM, the creation is done by Admission using a factory and the embedded quota that is part of

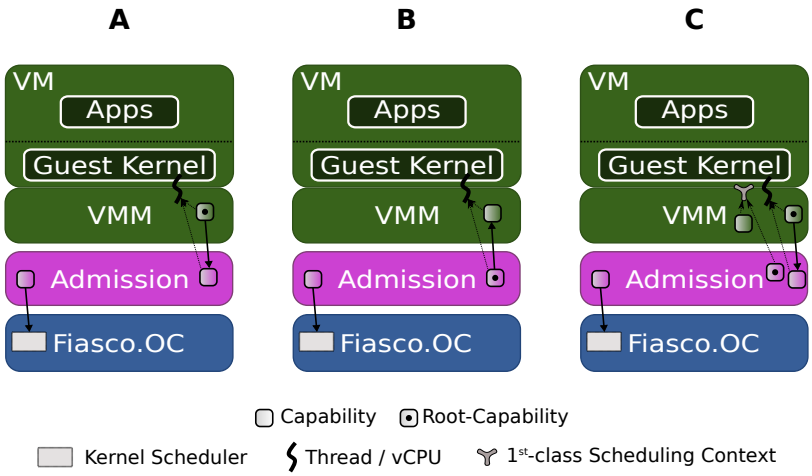

Figure 4.15: Different variants for creating and using scheduling contexts.

Admission's resources. The vCPU capability is then passed to the VMM which may create scheduling contexts. For configuring those SCs the VMM must go through the admission service again. Because Admission has created the vCPU, it has a non-revocable identifier of it for as long as the vCPU exists. For this reason Admission does not depend on the VMM as Admission has always access to the vCPU and can thus modify its scheduling configuration.

However, scenario B also burdens the vCPU and thread management onto the admission services instead of focusing on scheduling-related functionality. Scenario C offers an alternative approach where the admission component is only responsible for the scheduling contexts but not for vCPU and thread management. Like in scenario A, vCPUs and threads are created by the applications itself and passed down in a revocable fashion to Admission. However, instead of creating a second class scheduling context, which can only implicitly be addressed through threads, Admission now creates a first-class scheduling context with is own capability. Therefore, even if the VMM later on revokes the thread or vCPU capability, identifiers to the created SCs remain with the admission component and keep the possibility to change or reset scheduling parameters. This setup also enables further opportunities, for example, sharing a scheduling context for multiple threads or vCPUs.

4.5.4 Challenges with First-Class Scheduling Contexts

Scheduling contexts as first-class objects come with more flexibility but also with implementation challenges that are not present when scheduling contexts are embedded in threads. The following provides an overview on the required additions and added complexity:

- Principally, a stand-alone, first-class scheduling context can be bound to multiple threads. Allowing to bind the same scheduling context to multiple threads allows to use one budget for multiple threads. This gives an application the possibility to run multiple threads with one budget, without the need to redistribute budgets among multiple threads in dynamic workloads. However, the sharing also includes the other parameters in the scheduling context such as host priorities which will then be equal for all threads sharing the scheduling context. Depending on the workload, this might pose a problem, for example, virtualization systems may use a software timer that is

implemented as a host thread. Running the vCPU and the timer thread under one budget is attractive, however, the timer thread needs a higher host priority to timely interrupt the vCPU.

On single processor systems only one scheduling context can be in use at a time, avoiding additional complexity in the implementation. However, on multi processor systems, a scheduling context can be used for multiple threads running on different processors. This is problematic for several reasons:

- A priority value, as configured in a scheduling context, has a different meaning on different processors. Thus the single value stored in the scheduling context is not expressive enough.
- A scheduling context can be used on different processors at the same time. Concerning the budget in the scheduling context that means that all processors using the same scheduling context must know about the use of the same scheduling context on other processors to calculate their local remaining budget. In case one processor does not use up its share of the budget, other processors must check at the end of their budget whether budget is still available and coordinate with possible other processors.

This complexity is reduced by limiting SC usage to a single processor at a time.

- Access to objects can be revoked using the unmap operation (see 2.7). The revocation of access rights must happen immediately so that the unmapping task does not depend on the task that received the access rights earlier. This in turn means that a scheduling context can be revoked in any state, including being bound to a thread or an interrupt. The kernel tracks mapping relationships in its mapping database to know in which task a reference to an object exists. However, it also needs to find the object to which a scheduling context is bound to for disconnecting the two. A reference in the scheduling context to the bound kernel object (interrupt, thread) will solve this.

  When revoking a scheduling context from a thread, that thread must stop execution using the scheduling parameters of this particular scheduling context. By setting another scheduling context or stopping the thread, the original scheduling context is freed.

Overall, converting Fiasco.OC and L4Re to a system with first-class scheduling contexts requires a considerable effort in redesigning and re-implementing parts of the system. However, this is not required for showing the general applicability of the mechanism itself. For that reason the simpler approach with embedded scheduling contexts will be used in the following for the practical evaluation.

4.5.5 Integrated Scheduling Contexts

For implementation we choose the less complex variant of embedding scheduling contexts in another kernel object. This approach is less flexible, however, sufficient to implement the mechanism of scheduling contexts. Generally, the number of scheduling contexts used by a thread is dynamic and thus not known before starting the thread. The number might change during runtime so that more scheduling contexts need to be allocated, or unused ones can be freed. Although a scheduling context is not a first-class object, it is created by the kernel and this creation requires kernel memory. The allocation of that memory must be

bound to a task-specific quota, so that the creator of scheduling contexts can only allocate so many scheduling contexts that the limit is not exceeded.

Looking at the internals of the kernel, several kernel objects also store a reference to the quota of the object memory which they have been created from, for the purpose of returning the memory to the quota on their destruction. The availability of the quota is of benefit here as we also need the quota for allocating scheduling contexts.

As scheduling contexts are used with threads and the `L4::Thread` object also stores a reference to its quota object, it is natural to add allocation (`sc_add`) and release (`sc_del`) functionality for scheduling contexts to the thread object. For identifying a scheduling context an ID must be provided. The ID is an integer type, must be uniquely chosen by the creator of scheduling contexts and its scope is local to the thread.

After being allocated, the scheduling context is not yet usable because no scheduling parameters have been set yet. Configuration of a scheduling context is accomplished through an `L4::Scheduler` as the scheduler is the entity that manages the resource CPU. For that task, the scheduler provides a `sc_cfg` function, that takes an `L4::Thread`, the corresponding scheduling context ID and the scheduling parameters that shall be set for that scheduling context.

After configured, the scheduling context can be used. It can be activated explicitly via the thread's `sc_set` function, which puts the given scheduling context in use, or it is used when an interrupt triggers. The `L4::Irq` object is extended with an `attach_sc` function that behaves similar to the `L4::Irq::attach` function but also includes the ID of the scheduling context.

A scheduling context can only be destroyed when the scheduling context is not in use, not bound to any interrupt and when it has been unregistered at the corresponding scheduler. For that purpose the scheduler offers the `sc_release` function. Besides unregistering the scheduling context at the scheduler this operation also gives the scheduler the possibility to recalculate its CPU allocations, for example, for load balancing.

After being unregistered, the scheduling context can be freed with the `L4::Thread::sc_del` function. The kernel memory used for the scheduling context is freed and returned to the quota.

The following listings 4.1, 4.2 and 4.3 give an overview on the additional interfaces required for implementing scheduling contexts. Each class is derived from the original class to which the functionality is added. Due to the object-oriented design of Fiasco.OC and L4Re such enhancements can be added without modifying original files or functionality.

```
1  namespace L4
2  {
3    class Thread_SC : public Thread
4    {
5    public:
6      l4_msgtag_t sc_set(unsigned id, l4_utcb_t *utcb = l4_utcb());
7      l4_msgtag_t sc_add(unsigned id, l4_utcb_t *utcb = l4_utcb());
8      l4_msgtag_t sc_del(unsigned id, l4_utcb_t *utcb = l4_utcb());
9    };
10 }
```

Listing 4.1: Enhanced interface for L4::Thread

```
namespace L4
{
  class Scheduler_SC : public Scheduler
  {
  public:
    l4_msgtag_t sc_cfg(L4::Cap<L4::Thread> thread,
                       unsigned sc_id,
                       l4_sched_param_t *sp,
                       l4_utcb_t *utcb = l4_utcb());
    l4_msgtag_t sc_release(L4::Cap<L4::Thread> thread,
                           unsigned sc_id, l4_utcb_t *utcb = l4_utcb());
  };
}
```

Listing 4.2: Enhanced interface for L4::Scheduler

```
namespace L4
{
  class Irq_SC : Irq
  {
  public:
    l4_msgtag_t attach_sc(l4_umword_t label,
                          L4::Cap<L4::Thread> const thread,
                          unsigned sc_id,
                          l4_utcb_t *utcb = l4_utcb());
  };
}
```

Listing 4.3: Enhanced interface for L4::Irq

Usage Example

The following shall give a brief overview on a typical usage of threads or vCPUs with multiple scheduling contexts. The scenario consists of multiple virtual guest systems that have been stand-alone systems and that have been consolidated to run on a single host system. The operating system is para-virtualized, that is the operating system kernel is running as a native application on the host system and allows it to issue system calls as any other host application. The task of this operating system with its applications is to control a sorting machine that separates items that do not meet a particular weight. The items come by on a conveyor belt where a photoelectric beam triggers an interrupt when an item is about to reach the scale. The control program will be woken up by the interrupt of the beam and query the scale device for the weight of the unit on the belt. Depending on the measured weight the control program will trigger a pushing device that will redirect the item to a bucket and move back to its original position. Besides the control program, this system also offers a maintenance facility where operators can monitor the system status and query statistics information on the weights of the items and thus of the failure rate of the previous production steps.

The described scenario requires real-time execution of the control program because the belt is running at a constant speed and requires timely reaction of the pushing device to sort out any non-confirming items. The speed of the belt cannot be influenced without also influencing other task handling items on the belt, possibly the whole factory. Missing a timely reaction of the pushing device will let through non-confirming items for further processing, which can possibly hinder or even stop the whole production process. On the other hand, the maintenance and statistics tasks of the system do not require real-time execution and can be run whenever the system has otherwise unused processing capacity available.

The host system is required to setup the guest operating system and one part of this setup phase is to create an additional scheduling context for the single vCPU the guest is using. The first task is to create the scheduling context. As seen in Listing 4.1 of the extended `L4::Thread` interface, we require the thread object, thus adding the scheduling context is done after creating the vCPU for the guest. The variable `vcpu` is of type `L4::Thread_SC` and stores the capability to the vCPU to be used. To identify the scheduling context we choose the ID `1`:

```
L4::Cap<L4::Thread_SC> vcpu;

// Create / retrieve vCPU capability and store in vcpu variable

// Add scheduling context and check for errors
L4Re::chksys(vcpu->sc_add(1));
```

Listing 4.4: Creating a scheduling context

After successful execution of the `sc_add()` call the vCPU has one additional scheduling context available. Note that this call is only successful when the resource quota associated with the vCPU has enough resources available to create the scheduling context.

At this point the scheduling context is not yet usable because it has not been configured with proper scheduling parameters. Setting up scheduling parameters is done using the scheduling interface provided by the program environment. Prior analysis of the control program on the target platform concluded that a 10% share of the CPU with a period of 20ms is sufficient to run it. Therefore we choose the scheduling parameters of the scheduling context as seen the following listing 4.5. For the scenario the priority specified must be the highest for the guest system, that is the highest priority as seen by the local view of the guest. The scheduler proxies are then responsible for setting an appropriate host priority, considering a global view.

```
L4::Cap<L4::Scheduler> sched = L4Re::Env::env()->scheduler();
l4_sched_param_sc_t sched_params;

sched_params.prio = 5;
sched_params.budget = 2000;
sched_params.period = 20000;
L4Re::chksys(sched->sc_cfg(vcpu, 1, &sched_params));
```

Listing 4.5: Configuration of scheduling parameters of the scheduling context.

As a final configuration step, we need to bind the scheduling context to the device interrupt

of the photoelectric beam. For that we are required to supply the ID of the scheduling when attaching the IRQ to the vCPU.

```
1  L4Re::chksys(irq->attach_sc(label, vcpu, 1));
```

Listing 4.6: Binding the scheduling context to a device interrupt.

When all those described steps have been completed successfully the scheduling context can be used. Using it requires a single call on the vCPU to switch between the two available scheduling contexts:

```
1  // Switch to high priority scheduling context
2  L4Re::chksys(vcpu->sc_set(1));
3
4  // Switch to best-effort (default) scheduling context
5  L4Re::chksys(vcpu->sc_set(0));
```

Listing 4.7: Selecting a scheduling context as the active one.

The `sc_set` function must be invoked whenever the guest operating system kernel switches to the control program or away from it, so that the host priority of the virtual machine can be adapted accordingly.

4.5.6 Using Scheduling Contexts in Existing Systems

In the following we will show how scheduling contexts can be integrated and used in existing operating systems. We will cover paravirtualization and hardware-assisted virtualization, as well as a general purpose operating system, Linux, and the real-time operating system FreeRTOS.

For the following we assume that all required scheduling contexts have been set up and the guests are only required to use a single function to select among their available SCs. This function is named `set_sc(ID)` where ID identifies the SC.

The specific implementation of `set_sc(ID)` varies because of the different implementations of virtualization. With hardware-assisted virtualization guest operating systems invoke the hypervisor and hence the VMM through a special machine instruction (`vmmcall` on x86), which in turn the hypervisor forwards to the VMM. The VMM can then decode the VM-exit, extract the SC-selection request and pass it on to the hypervisor. More optimized implementations can also support special hypercalls in the hypervisor so that the forwarding through the VMM is not necessary.

Paravirtualized guests can directly invoke system-calls of the host kernel and do not require any specific event handling in the VMM.

To simplify the presentation of the guest operating system modifications, we restrict the examples to two scheduling contexts. A best-effort SC and one SC to handle high priority work. Elevation of a VM to a higher host priority, that is to the SC for the high priority work, is triggered by a single interrupt. Switching away from this SC targets the best-effort SC. An extension to multiple different SCs with different priorities is straightforward.

FreeRTOS

FreeRTOS is a typical representative of a real-time operating system. It runs on a variety of architectures and CPU variants and does not use a memory management unit, however, it can make use of a memory protection unit (MPU). The scheduler in FreeRTOS allows multiple tasks to run concurrently at static priorities. Preemptive and non-preemptive variants are available. In our implementation, we exclusively use the preemptive version, which calls the internal scheduler for each timer tick. This timer tick is associated with the high priority SC. After the new task to be run has been chosen, the function listed in Listing 4.8 is called, handing over the new priority. Referring to FreeRTOS v6 and v7, the function must be added as `xvPortPostSchedule(uxTopReadyPriority)` in the function `vTaskSwitchContext()` after the while loop, which calculates the new priority. The function uses a barrier priority `RT_BASE_PRIO` to split FreeRTOS tasks into a real-time and a time-sharing category. Based on this barrier priority, the FreeRTOS scheduler decides whether the selected task should continue to use the high priority SC or fall down to the best-effort one, using `set_sc(ID_SC_BE)`.

```
1 void xvPortPostSchedule(unsigned prio)
2 {
3   if (prio < RT_BASE_PRIO)
4     set_sc(ID_SC_BE);
5 }
```

Listing 4.8: SC switching function for FreeRTOS.

The function `vTaskSwitchContext()` is also called by FreeRTOS after interrupt processing so that a possible elevated priority is reset again when a non-high-priority task has been selected.

Linux

Linux is a widely used and popular general purpose operating system. With the ongoing work on improving the preemptiveness of the kernel and with the merge of a significant part of the Linux-RT patch-set, it is also increasingly used for real-time workloads. Linux priorities are divided into a range for time-sharing and an exclusive range for real-time processes. This distinction makes the implementation of the SC switching function straightforward as shown in Listing 4.9. Referring to Linux kernel version 3.12, the function `post_sched_sc(current)` is called within the function `finish_task_switch()` in `kernel/sched/core.c`.

```
1 void post_sched_sc(struct task_struct *p)
2 {
3         if (!rt_task(p))
4                 set_sc(ID_SC_BE);
5 }
```

Listing 4.9: Post scheduling function for Linux.

To switch back to the best-effort scheduling context in the case no scheduling decision will be made after an interrupt has occurred, we introduce the function `irq_no_sched()`. It is called in the code paths for exiting interrupts as shown in Listing 4.10. Referring to

Linux 3.12, a convenient location to call `irq_no_sched()` is the function `irq_exit()` in `kernel/softirq.c`. The synchronization in `irq_no_sched()` is required to atomically check the rescheduling condition with the actual operation. Otherwise, if an interrupt occurred meanwhile, a possible real-time task would be switched back to the best-effort scheduling context.

```
1 void irq_no_sched(void)
2 {
3         unsigned long flags;
4         local_irq_save(flags);
5         if (!need_resched() && !rt_task(current))
6                 set_sc(ID_SC_BE);
7         local_irq_restore(flags);
8 }
```

Listing 4.10: Function to be called in case no scheduling decision has been made upon interrupts.

Using and mapping multiple real-time tasks within Linux to different SCs is also possible by enhancing the two presented functions. For that the guest needs to know a mapping between its internal process priorities and scheduling context IDs. The contents of this map must be determined upon launching the guest system.

4.5.7 Scheduling Contexts and Hardware-Assisted Virtualization

In paravirtualized environments the kernel code can use host system calls like any other host program. The situation is different for hardware-assisted virtualization. Here, the guest must invoke a hyper-call which causes the VM to exit and switch control to the VMM. The VMM will identify the type of the hyper-call and invoke the appropriate function for handling the guest request. Thus for selecting scheduling contexts the guest needs to use a hyper-call, as schematically depicted by Figure 4.16. Overall, the used mechanism is more costly than the paravirtualized approach as the VM must first exit to the VMM, including saving the extended state for hardware-assisted virtual machines, and then issue the system call to the host kernel. Returning from the system-call back to the VM includes loading the extended state again to run the VM.

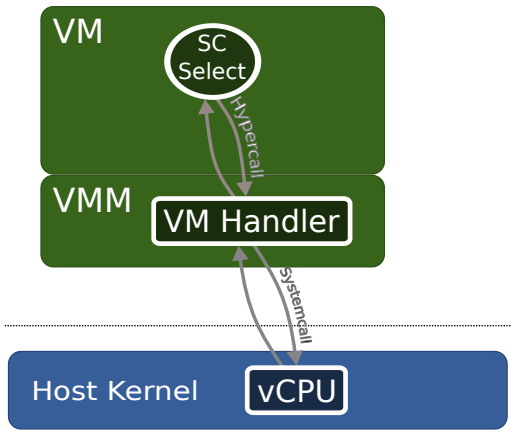

Figure 4.16: View on the control flow when selecting a scheduling context from within a hardware-assisted virtual machine.

With additional functionality in the host kernel, the execution performance in the described scenario can be improved. One part of the execution path is the handling of the VM guest state. When switching from a VM context to the hosting VMM, the host kernel saves the VM state from the physical VMCx memory page, only used internally in the kernel, to the extended state page of the vCPU so that it is available to the VMM. When returning to the VM, the host kernel loads the state from the vCPU state area to the physical VMCx so that the CPU loads the proper state when switching to the VM. In the case of handling a hyper-call, for example, to select a scheduling context, the VMM does not require the full VM state. Thus the host kernel could leave most of the state in the physical VMCx and invoke the VMM with that information. When switching back to the VM only the return information of the hyper-call needs to be updated in the physical VMCx and the VM can continue. This approach avoids state saves and restores when switching between VM and VMM. However, the kernel needs to implement lazy VMCx handling then and save the state of the physical VMCx whenever another VM, i.e. another vCPU with extended state, shall be run. As this is transparent to the VMM, the kernel needs to also load back the complete state when resuming a VM. Despite the effort in the host kernel, such an approach still improves on the fast path of switching between VM and VMM.

With more functionality in the host kernel, the VMM can be left out when handling a hyper-call. Generally, a hyper-call could invoke any capability available in its VM task, similar to invoking a capability with a system call in normal L4 task. Provided the host kernel offers a capability space for the VM, the VM can direct invoke the vCPU capability to select a scheduling context without requiring the VMM. For best performance this shall be combined with the lazy VMCx saving mechanism just described. With this implemented, the performance of selecting a scheduling context comes closer to the paravirtualized variant, however, as a VM exit and re-entry is still more costly than just a system call, it will still be more expensive.

We will evaluate the use of scheduling contexts with hardware-assisted virtualization in Section 5.

4.6 Applicability to Mixed-Criticality Scheduling Algorithms

Scheduling contexts are an operating systems mechanism and shall provide a reasonable generic interface to be able to implement a wide range of scheduling algorithms. We have investigated whether a range of mixed-criticality scheduling algorithms can be used on top of scheduling contexts and whether extensions are required to support a specific algorithm [VLH13]. In the following I will provide an overview on our findings.

Criticality-Monotonic and Static Fixed Task-Priority Algorithms require a deadline for the end of the budget of a scheduling context. The scheduling context will be refilled on the start of the next period. The required extensions for scheduling contexts are the deadline parameter in the SC configuration as well as a second timeout in the kernel for the deadline. We expect only minimal overhead for this extension.

Own-Criticality Based Priority Ordering and Static Fixed Job-Priority Algorithms can make use of multiple scheduling contexts. Examples are systems where work can be split in mandatory and optional parts, for example, video decoding where decoding

the frame is mandatory and additional quality enhancing picture post-processing is optional. The parts of the work can be modeled with scheduling contexts, so that the SC for the mandatory part has a high priority and the SCs for the optional parts have lower priorities.

There are two design choices for implementing the necessary SC switching: controlled by user-space or done in the kernel. When user-space shall handle the switching to the next scheduling context, the kernel has to notify user-space when a scheduling context ran out of budget. When done by the kernel, the scheduling contexts must be linked so that the kernel can select the SC in a defined order. A possibility to drop the remaining budget must also be available so that the kernel can switch to the next scheduling context in the queue.

Adaptive Mixed-Criticality Algorithms require to know the time a thread has run on an SC. This allows a scheduler to decide on further execution and which mixed-criticality level to use. For example, an overrun of a low-criticality WCET triggers a criticality change and thus means that low-criticality jobs must not be executed and consequently disabled.

On the implementation side this requires that the scheduler gets a signal when the kernel is switching to the next scheduling context so it can act accordingly and a mechanism to disable scheduling contexts. The signaling mechanism is the same as already described in the first case.

Disabling a group of scheduling contexts requires the next extension. This extension can be implemented with an *enabled* token, which the scheduler can toggle to disable at once all scheduling contexts that refer to this token. To cover multiple criticality levels, we suggest implementing the enabled token as a bit-field complemented with a mask inside each SC. The mask is then used to determine which bits are significant for this SC.

Earliest Deadline First with Virtual Deadlines requires the possibility to enable and disabled groups of scheduling contexts. To implement this requirement we can built on the group enable and disable functionality as described in the previous case. Groups are formed among scheduling contexts. Once a criticality change happens, all low-criticality SCs are disabled by clearing their significant low flag and high criticality SCs are enabled by setting the formerly disabled high criticality flag in the enabled token.

.

4.7 Further Directions

Scheduling contexts are a promising approach for combining several latency-constrained environments into a single physical machine. Their usage potential goes beyond what has been presented in this work, however, I will outline ideas for possible future directions in the following.

The property of scheduling contexts to be standalone offers interesting possibilities. Their budget feature could not only be used for a single thread but also for a group of threads. That way a subsystem, usually consisting of multiple threads, can be limited in their CPU time usage without the need to force a budget limitation on each individual thread. However, in the current form scheduling contexts also have other per-CPU scheduling settings, such as a priority. With a group of threads the priority of each thread still needs to be configurable individually. The challenge here is to allow the grouping while also still allow individual

thread parameters. Options could be different type of scheduling contexts which are linked together for a particular thread. When additionally considering that the group of threads is running on different CPUs, the grouping becomes more challenging. The reason is that a budget can be used by multiple threads at a time, which requires cross-CPU arbitration that needs to be redone whenever one involved thread blocks or gets running so that the potential end-of-budget point in time can be reevaluated. Whether such an approach is practically reasonable or is impractical due to the required cross-CPU communication overhead remains to be evaluated.

Other requirements need minor extensions to the model, such as notifications on end of deadlines or budgets so that a controlling software components can adapt workloads accordingly. The concept of scheduling contexts might also be of useful for other resources, such as memory buses, disks or network resources. Their applicability in such scenarios remains to be evaluated.

5 Evaluation

In this chapter we will evaluate how the presented virtualization techniques perform, including in the context of real-time execution. We will also look at scheduling contexts and evaluate the amount of modifications needed for the Fiasco.OC kernel as well as potential guest operating systems.

In the previous chapters we already introduced and used two different hardware architectures: x86 and ARM, showing the broad applicability of our virtualization approach. For evaluation, we will use three different type of systems, covering both architectures:

x86 The x86 architecture is the dominant platform in the desktop, laptop and server segment and thus represents a broad share of practically used systems.

ARM is the dominant architecture in the embedded area and available in many different configurations. The range of Cortex-A CPUs, which support virtual memory and privilege levels, offers CPUs with different execution characteristics. They are instruction set compatible and can thus run the same binaries without modification despite their differences in the architectural design of the processor. By optimizing for different design criteria, energy-efficiency or performance-centric, the implementer can chose between the contradictory goals. By putting both configurations into a single system a wider range of utility can be covered. Such a configuration is called big.LITTLE™ and the performance-centric design is called *big* core while the energy-efficiency-centric core is called *little* core. A big.LITTLE™ setup uses Cortex-A15 CPUs as big and Cortex-A7 CPUs as little cores respectively.

We use the following three systems as seen in Figure 5.1 for benchmarking throughout this chapter and refer to them by their short `x3`, `arm-B` and `arm-L`.

5.1 Performance Characteristics of the Systems

First, we will run generic benchmarks to get an overview on the performance characteristics of the three systems and their relative performance among each other.

5.1.1 Memory Bandwidth

Figure 5.1 shows the achievable memory bandwidth on each system for reading, writing and copying large chunks of memory.

Label	Architecture	System	Description
`x3`	x86	AMD Phenom-X3 8450	x86-based CPU, clocked at 2111MHz.
`arm-B`	ARM (big)	Odroid-XU	System-on-Chip based on the Samsung Exynos 5410, using a quad-core Cortex-A15, running at its default frequency of 900MHz.
`arm-L`	ARM (little)	Cubieboard2	Allwinner A20 SoC, running a dual-core Cortex-A7, with a clock speed of 912MHz.

Table 5.1: Systems used in evaluation.

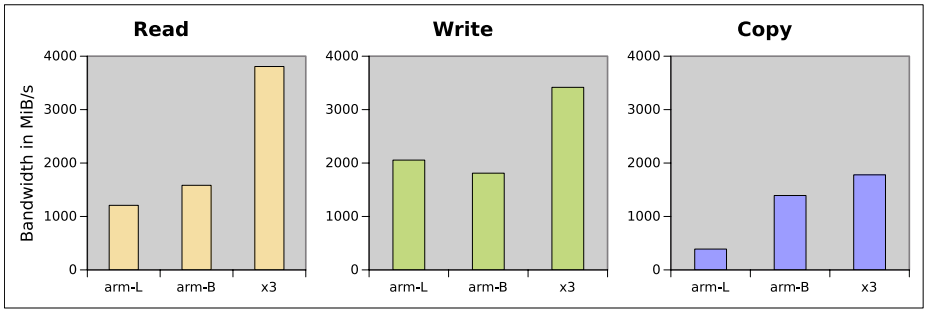

Figure 5.1: Achievable memory bandwidth for reading, writing and copying memory.

The x86-based system is the fastest for all operations, however, the ARM-based system are not far behind. While the read and write performance is about the same for the little and big ARM systems, the big ARM core can gain advantage when copying data compared to the little core. The advantage of the little core over the big core for memory writes is likely to be attributed to the different systems used.

5.1.2 CPU-Bound Application

For evaluating the CPU-performance of the systems, I have run a series of matrix multiplications. The program uses the naïve algorithm with $O(n^3)$ complexity for square matrix multiplications using `int` as a data storage type for the matrix elements. Figure 5.2 shows the performance on each of the three systems when running the calculation as an L4Re program. The execution times have been normalized to 1000MHz to allow for a better comparison between the systems independent of the particular clock speed of each system.

The results show that the x86 system offers the best performance per clock cycle in the trio, followed by the ARM big core. As expected, the ARM big core offers more performance than the ARM little core.

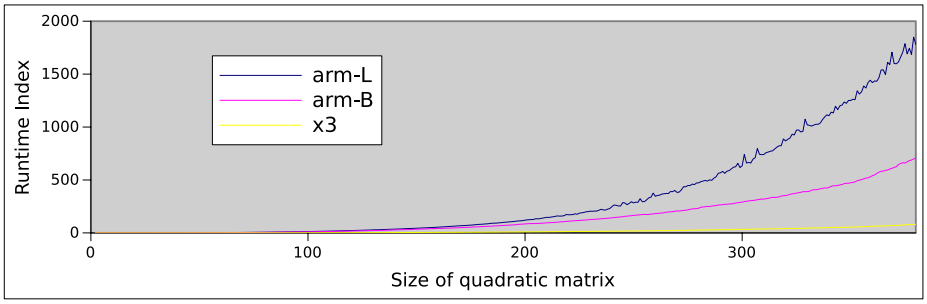

Figure 5.2: Performance of a series of matrix multiplications on the three evaluation systems, normalized to a clock speed of 1000MHz.

5.1.3 CPU-Bound Application within L4Linux

When running the matrix multiplication as a Linux program inside an L4Linux virtual machine, the results are as shown in Figure 5.3 together with the results from the previous Section 5.1.2.

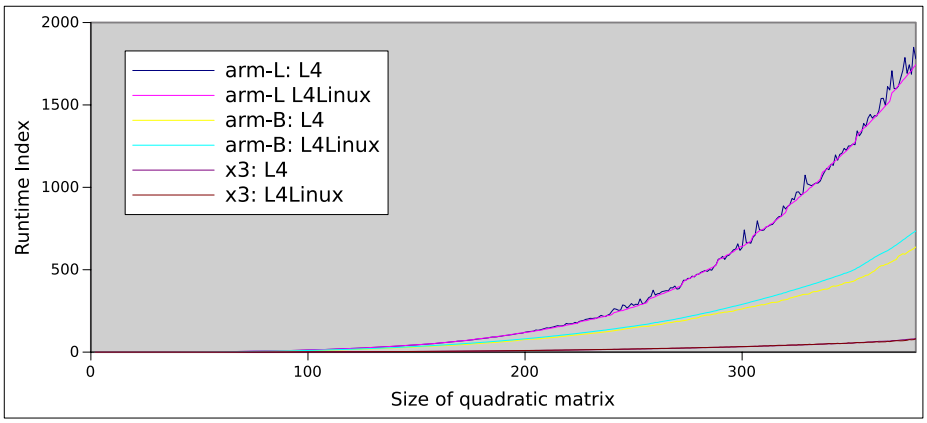

Figure 5.3: Performance of a series of matrix multiplications on the three evaluation systems, run each on L4Re as well as virtualized. Results are normalized to a clock speed of 1000MHz.

The results show that the virtualization infrastructure has no significant influence on the execution time of the matrix multiplications on all three platforms. This is to be expected as CPU-bound tasks shall not put load on the virtualization functionality.

5.2 L4Linux Application Performance

During this work, L4Linux was converted from the previous thread-based execution model to the vCPU execution model. The different execution model also implies different execution behavior and thus execution performance. I used the AIM Multiuser Benchmark Suite VII to evaluate the performance of the system, the same benchmark that has already been used for the first L4Linux system [Här+97]. The benchmarks simulates application workload and increases the jobs it runs until the system is saturated.

The figures show several configurations of Linux and L4Linux, running on the `x3` system. All

variants have 1500MB of memory available and a standard SATA hard-disk drive is used to run the benchmark.

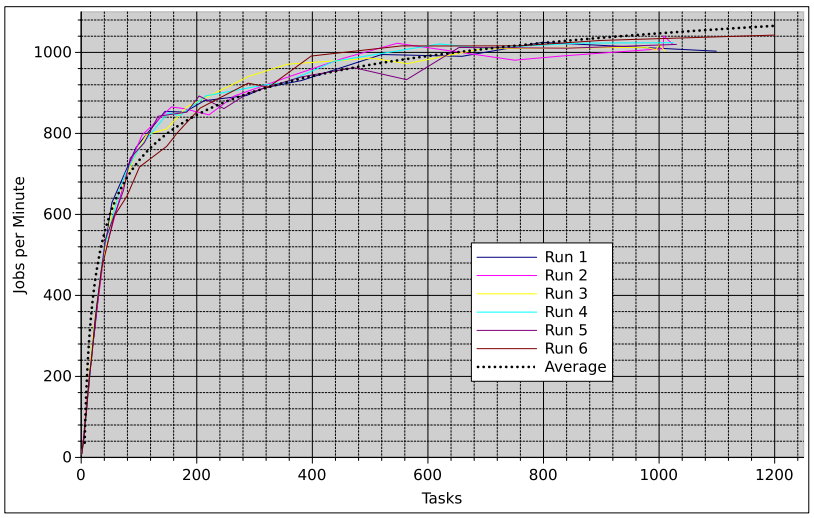

Figure 5.4: Each individual AIM XII benchmark run of *Linux standard* configuration with the fluctuating results and the average of all runs.

Four variants have been evaluated, two Linux and two L^{4}Linux configurations. For L^{4}Linux the vCPU-based (*L4Linux vCPU*) and thread-based model (*L4Linux thread*) have been run. For native Linux, the *Linux standard* configuration is an unmodified Linux. The *Linux TLB-flush* version uses a modified Linux kernel that flushes the TLB on each entry and exit of the kernel for page faults and system calls for two reasons. First, this effectively emulates the address space switches that are architecturally required by L^{4}Linux within Linux and thus gives an insight on the share of the address space switching on the performance difference between Linux and L^{4}Linux. Second, this also allows to judge on the potential benefit of the small address space optimization [Hof02] that is not available anymore.

All configurations were executed at least 5 times. Each run takes about 12 hours on the given setup. The Linux version used is 3.14 and all Linux and L^{4}Linux setups use the same base-line configuration. As the results of the same configuration showed to be fluctuating (see Figure 5.4 for an example) I computed an average graph of all benchmark runs of the same configuration using least squares fitting, as shown in Figure 5.5.

Overall, the results are close. For the averaged results the slowest configuration (*L4Linux thread*) is just 2.8% slower than the fastest setup (*Linux standard*). Experimental runs where storage has been placed in a `tmpfs` in-memory file-system instead on the hard-disk showed that the system can handle a magnitude more load, however, the system also runs into resource exhaustion in all configurations and cannot finish the benchmark run. This shows that file-system operations have a significant impact in the benchmark. For reference, the graph also shows the results from the first L^{4}Linux publication [Här+97] (*L4Linux '97 SOSP* and *Linux '97 SOSP*). Although the CPU performance has increased by magnitudes, the benchmark performance only increased by about one magnitude, giving further indication that the access performance of the hard-disk has significant influence in the benchmark.

The results show that the application performance of L^{4}Linux is in range of native Linux. Further, vCPU-based L^{4}Linux shows a slight improvement in application performance over thread-based L^{4}Linux. The modified Linux with TLB-flushing shows a performance degradation, however, in this application benchmark the share on the difference between

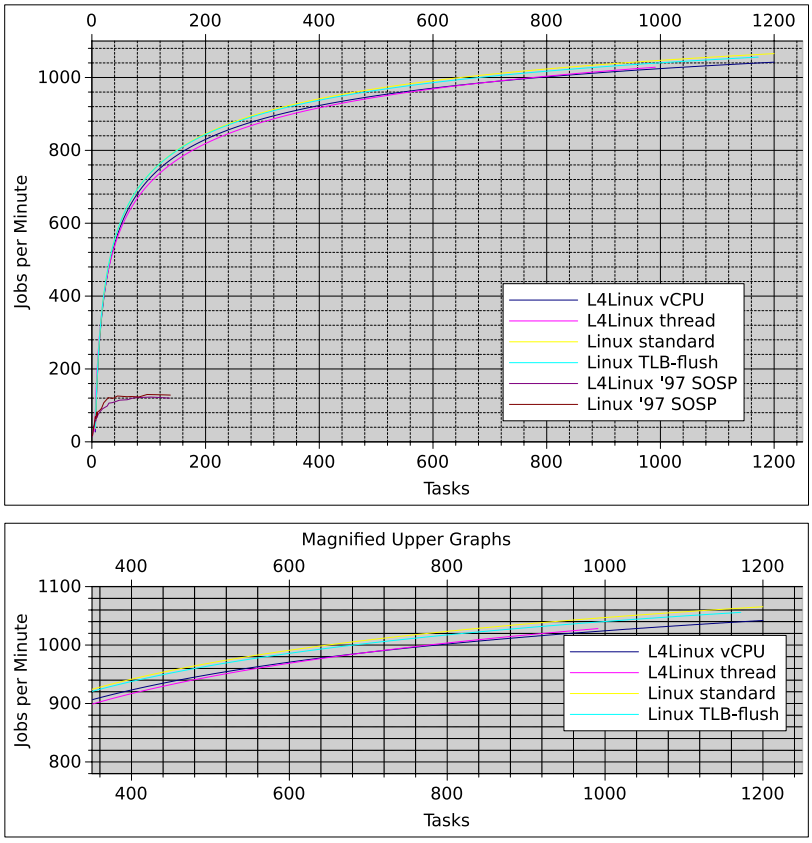

Figure 5.5: AIM Multiuser Benchmark Suite VII jobs per minute results. Each graph shows the average of multiple benchmark runs of different configurations of native Linux and L^{4}Linux, as well as a comparison against the benchmark results of the '97 SOSP publication on L^{4}Linux. The lower graph shows a magnified version for better visibility of the differences between the configurations.

Linux and L^{4}Linux is minimal.

5.3 Linux Compile Benchmark

The Linux kernel compile is a classical and popular system benchmark. It is mainly a CPU throughput benchmark but also exercising operating system functionality such as paging as well as storage and file-system subsystems.

The benchmark records the time required to unpack a Linux kernel source tree, compile a defined configuration and delete the source tree and generated files again. Both the source files and generated files are stored on a hard disk.

Figure 5.6 shows the results of the benchmark that has been run on the `x3` system. Again, as with the AIM benchmark in 5.2, four configurations have been used. *Linux standard* and *Linux TLB-Flush* for native Linux and *L^{4}Linux vCPU* and *L^{4}Linux thread* for L^{4}Linux.

The order of results are in line with the AIM benchmark. The fastest configuration is the *Linux standard* setup, followed by the *Linux TLB-Flush* setup. For the L^{4}Linux setups, the vCPU version shows improved performance compared to the thread-mode configuration. With this benchmark the native Linux with TLB flushing is about 5% slower compared to

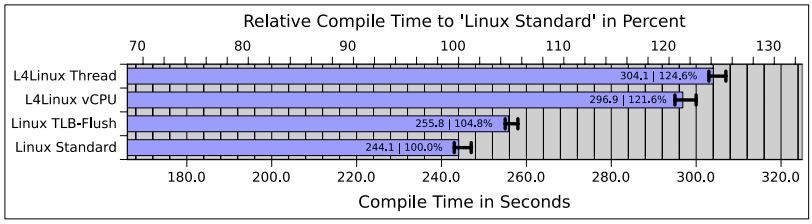

Figure 5.6: Linux and L^{4}Linux Linux kernel compile benchmark. Smaller numbers indicate better performance. The upper values show relative performance to the *Linux standard* setup, while lower values show absolute benchmark runs in seconds.

standard Linux. L^{4}Linux is 21.6% slower than standard Linux.

5.4 Influence of Host Timer Frequency

As described earlier, in a system with fixed time-slices the frequency of the host scheduler is the base of scheduling guests on the system. In such a system with a partitioning scheduler, the higher the frequency, the more accurate guests can be scheduled. However, increasing the host timer frequency also increases the overhead of the overall system. First, servicing a timer interrupt requires work in the host system. Secondly, switching guests more frequently means that the guests' performance will degrade because their cache working-set is evicted more frequently.

To demonstrate the effect of increased host timer interrupt activity, I ran matrix multiplications with varying host timer interrupt frequencies. The main reason for choosing this application is that it is both CPU-consuming as well as memory demanding and thus can represent a busy subsystem, including virtual machines. By adjusting the size of the matrices, the amount of work in the test can also be easily adapted to work practically well on any platform.

When running the multiplication in just one task, its runtime will only be influenced by the overhead of the timer interrupt processing in the host. In other words, by increasing the timer interrupt frequency, the matrix multiplication will slow down. When using matrix multiplications in different tasks, the effect of voided cache working sets will also be seen.

Figure 5.7 shows the results of running the matrix multiplication on the `x3` system. The size of the matrices is 886x886, resulting in memory usage of each task of about 9MiB. The x-axis shows the period of each task in µs, the y-axis shows the relative performance compared to a configuration with a 10ms period and a single task.

The results show that running a host system with a timer frequency of 1kHz instead of a lower frequency results only in a hardly measurable overhead. However, increasing the frequency further, considerably impacts the runtime of the workload. For example, running with 10kHz leads to about 20% more runtime. The impact of using multiple tasks concurrently is also clearly visible and further increases the overhead for each single matrix multiplication.

Considering a virtualization system that uses time partitioning and is running virtual machines with real-time latency requirements, we can conclude that a period of 1ms per VM is manageable from an overhead point of view. However, it still must be considered that the achievable latency for event reactions within a virtual machines depends on the length of the time slices in the host as well as the number of virtual machines running. Taking an

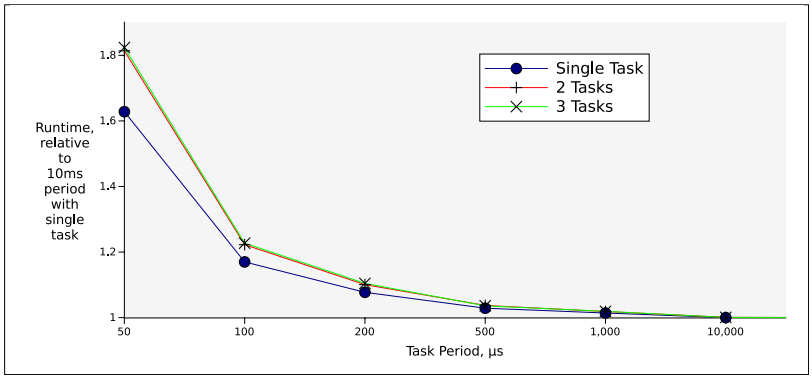

Figure 5.7: Relative time required to run the same workload, using one to three tasks and timer frequencies from 100Hz (period of 10000µs) up to 20kHz (period of 50µs).

arbitrary example of 5 running VMs and a length of a period of 1ms, a VM has to wait up to 4ms until it is dispatched again and can handle events. For practical real-time systems such a waiting time might be too long. Increasing the host scheduling frequency improves that situation but also increases the overhead, as demonstrated with the matrix multiplications. These results are in line with the results reported for RT-Xen [Xi+11]. Section 5.7 shows an evaluation on how scheduling context improve on the event reaction latency and avoid the problem of otherwise required high switching frequencies.

5.5 Added Source Code

Scheduling contexts are an enhancement that has been implemented in the host kernel. The host kernel is part of the Trusted Computing Base (TCB) of all software running on the system and thus we need to pay attention to a possible code increase as more code is also likely to introduce more bugs.

Fiasco.OC Fiasco.OC implements scheduling policies in modules, allowing us to compare the standard fixed-priority scheduler with our extended version supporting scheduling contexts. However, generic code has also been extended to support additional scheduling contexts.

Overall, the added scheduling module and the further added code, such as the internal interface extensions required for multiple scheduling contexts, sum up to 626 source lines of code as measured with the *sloccount* tool [Whe]. The already existing fixed-priority scheduling module was adapted to the extended internal interface for scheduling modules and grew from 102 source lines to 151 source lines. The scheduler module for multiple scheduling contexts, an extension of the fixed-priority scheduler modules, requires 311 source lines. The ready-queue implementation module for the fixed-priority scheduler stays constant in size with 73 lines of source code, while the ready-queue implementation for multiple scheduling contexts is 76 lines of source code.

Guests Virtual Machines Guests must be at least equipped with calls to the host to select their used scheduling context during runtime. Manual effort is likely to correlate with

the amount of code required to be added to the guest, thus small and compact additions are preferred. Table 5.2 shows the source lines of code that need to be added to Linux and FreeRTOS, the two systems introduced in Section 4.5.6.

Guest	Added Source Code Lines
FreeRTOS	10
Linux	22

Table 5.2: Required code addition for Linux and FreeRTOS to enhance them with scheduling context selection calls.

The implementation of a system call can be attributed with 8 lines of code, however, those might already be available in the guest for other reasons, for example, because the guest is paravirtualized.

Overall, the required additions are small and compact, allowing them be added to guest systems easily, either by modifying the sources or even patching the binary at the corresponding positions if possible.

5.6 Performance of the Scheduling Functionality

In this section we look at the performance characteristics of the scheduling context functionality.

5.6.1 Passive Runtime Overhead

When scheduling contexts are not used within a guest operating system, the applied modifications to the guest system incur no measurable overhead. Referring to 4.4.2 and assuming that the `set_sc()` function will call out only when the SC must actually be changed, the overhead is negligible because in this situation `set_sc()` boils down to a simple check for equality between the IDs of the current and the targeted SC.

5.6.2 Scheduler Call Latencies

The following test evaluates the latency of calling the kernel's scheduler. This is interesting because, as already explained previously, schedulers can be nested and thus calling a scheduler capability might go through proxies until it finally hits the kernel. Figure 5.8 shows the layout of the conducted tests that have been run on the three systems. Each of the proxies and the client run in their own address space.

The results are shown in Figure 5.9. The *"Direct"* column shows the call duration for directly calling a kernel's scheduler function in CPU cycles. The *"1"*, *"2"* and *"3"* columns show the CPU cycles it takes to call the same function through the specified number of proxies. All systems uniformly add about 2000 CPU cycles per proxy layer.

Taking about 2000 CPU cycles or multiples of that for calling a scheduler function in a typical setup is in a reasonable time frame. However, for the frequently used operation of setting the scheduling context of a thread or vCPU, such a call duration might be too long.

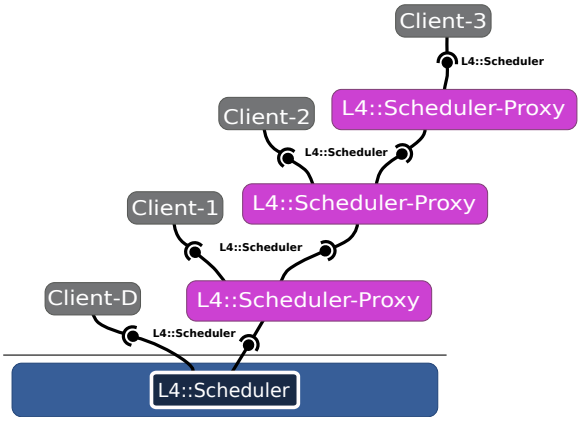

Figure 5.8: Measurement setup.

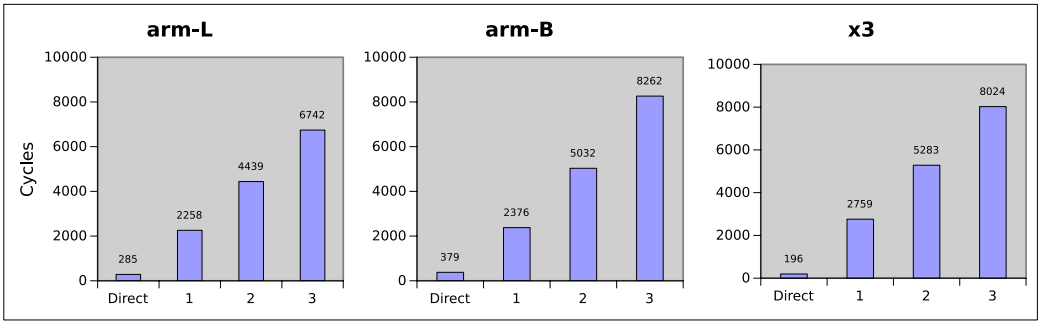

Figure 5.9: Results of calling a scheduler's function directly and via up to three proxies.

Additionally, by calling through the proxies, other address spaces are activated, leading to usage of caches and TLBs that evict entries from the current application. This lead to the design decision that switching a scheduling context should not be done through an intermediate component, such as with the Scheduler interface. Instead, a more direct way has been chosen by using the Thread interface for the selection operation.

Hardware-Assisted Virtualization With hardware-assisted virtualization, exiting a virtual machine is more costly because the CPU needs to save and restore more state. This also affects the *vmmcall* operation, the instruction that is used by guests to voluntarily exit the virtual machine, for example, to call host functionality. In virtualization context, this operation is typically called a *hypercall*.

Figure 5.10 shows the duration of calling out of a virtual machine, to the VMM and further to the scheduler in the kernel, including up to three proxies, on the `x3` x86-based system.

The results show that calling a VMM function from within a hardware-assisted virtual machine must be attributed with 3075 cycles on the `x3` system. This is also the baseline for further calling the scheduling interface, either directly or via up to three proxies. Calling from a hardware-assisted virtual machine is more expensive than using a paravirtualized guest or native applications, however, the about 3000 cycles are still a practically acceptable value. The added runtime for each added proxy correspond to the measured values as seen in Figure 5.9 and thus it makes no difference from where the call originates.

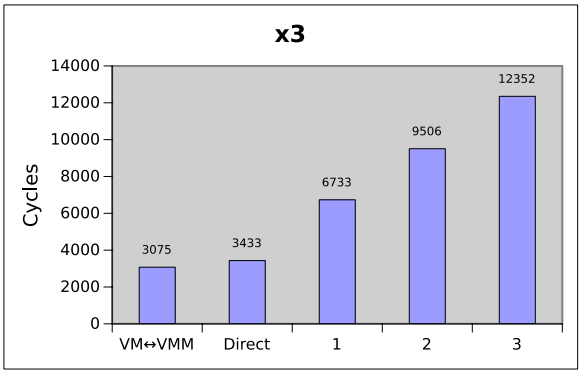

Figure 5.10: Results of calling a scheduler's function from within a hardware-assisted virtual machine directly and via up to three proxies. The column *VM↔VMM* denotes the time it takes to call an empty VMM function. Other columns perform scheduler invocations.

5.7 Effect of Fixed-Partition Scheduling on Event Latency

In this section we study the effect of fixed-partition scheduling on the event delivery latency into a vCPU, as being part of virtual machines, and compare it to a variant using scheduling contexts.

Figure 5.11 shows the results of a program that runs two vCPUs which have a period of 10ms each, that is the vCPUs are switched, ignoring other system activity, every 10ms. Both vCPUs execute a spinning loop to occupy the CPU. Interrupts are injected to one of the vCPUs by a third thread, which waits for the vCPU to acknowledge its reception until again waiting 1ms for injecting the next interrupt. The time between injection of 4000 interrupts and their reception is depicted in the histograms of Figure 5.11.

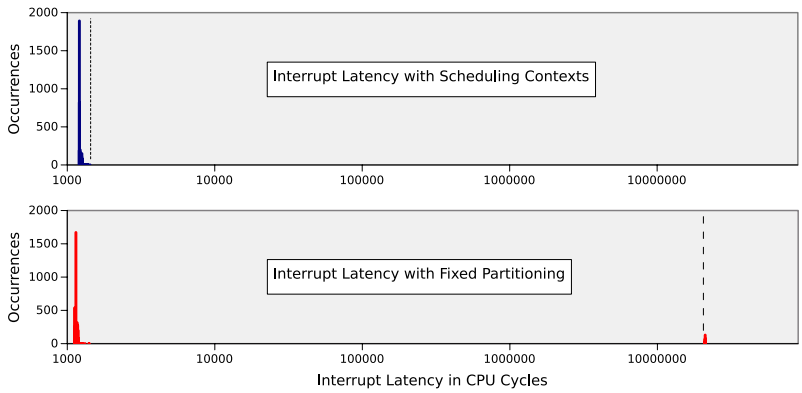

Figure 5.11: Plot of the interrupt latency using a fixed-partitioning scheme (lower graph) and an approach employing scheduling contexts (upper graph). In the bottom graph the large latency of the event delivery for non-active vCPUs is clearly visible at about 21 million cycles.

The lower graph shows the latencies observed when using a fixed-partition approach. While the "partition" is active, the latency is in the range of immediate delivery to the vCPU and can be seen in the left part of the graph. However, when the vCPU is not active, delivery is deferred until the vCPU is activated again. The experiment has been executed on the `x3` system which is using a processor speed of 2.1GHz, thus a 10ms period takes about 21

million cycles, as indicated by the dashed vertical line. Parts of the event delivery suffer from this behavior and have a latency over 21 million cycles. As the program is waiting for the interrupt to arrive, the injector has to wait until the period is over and thus only one interrupt per period is experiencing this large latency in this test, leading to less occurrences of large latencies compared to immediate deliveries.

In the upper graph, the vCPU receiving the interrupts employs a scheduling context that is bound to the interrupt and that raises its host priority compared to the other vCPU on reception of that interrupt. The budget for the interrupt scheduling context is a fraction of the overall budget for the vCPU. In our particular example a budget of 3ms for the whole execution of the experiment suffices. The maximum observed latency value in the experiment is 1426 cycles, including periods where the vCPU was not active when the interrupt was triggered, and is marked in the graph with the dotted vertical line.

For reference, Table 5.3 lists the times required for running the ISR in this scenario for all three systems. The listed figures are measured values and thus do not represent the values that shall be put into an SC budget configuration for running the ISR. Those values must be determined with an appropriate WCET analysis for the given system. The three measured budgets reflect the relative performance of the three systems. The `arm-L` is the slowest among the three and needs to allocate the biggest budget for running the ISR. The `x3` system is the fastest and required the least budget for the ISR.

System	Budget for ISR
`arm-L`	12 ms
`arm-B`	10.5 ms
`x3`	3 ms

Table 5.3: Measured times required for the ISR handling in the three evaluation systems for the complete runtime of the experiment.

In this experiment we see that the event delivery latency, for example for interrupts, in fixed-partitioned setups depends on the configured period for the vCPU. Thus the WCET of an interrupt service routine is at least a period. Picking up the findings of Section 5.4 and the example therein, practically achievable latencies are in the range of at least a millisecond, and increasing with more vCPUs running. Such latencies might be acceptable in scenarios with only soft real-time requirements, such as with user interaction. In scenarios with more stringent requirements, such as control applications, such latencies are not acceptable and require a solution that provides better event reaction latencies. Further, an influence on other running activities in the system is also not acceptable and thus requires a different approach such as with the proposed scheduling contexts.

5.8 Summary

The evaluation assesses key aspects of the vCPU execution model and scheduling contexts. Using the vCPU execution model within L^{4}Linux yields a slightly improved application performance and, due to behaving more closely to the native execution behavior of Linux, allows to add more features, such as multi-processor support.

Experiments using scheduling contexts have shown that the presented mechanisms allow low latency handling of real-time events in scenarios where other approaches fail. Furthermore,

we evaluated the performance impact of increasing the host timer frequency. Results indicate that a frequency of 1kHz still has barely any impact on execution time. Measurements have shown that with a frequency of 10kHz an overhead of about 20% for the scheduled applications must be accepted. By using scheduling contexts we can avoid the high switching frequency and avoid long event latency.

Further, scheduling contexts are only used when an event occurs while an increased scheduling frequency must be run permanently. This reduces the system load and thus gives potential to reduce the energy consumption of the system. Schönherr et al. [Sch+10] report that a system with a periodic timer tick consumes more energy compared to a tickless system. The amount of timer interrupts is therefore visible in the energy consumption of a system. We can therefore assume that using scheduling contexts also results to a lower energy consumption of the system.

Finally, we also evaluated the source code lines that have been added to the Fiasco.OC microkernel as well as the source code lines that need to be added to guest operating systems to enhance them to use scheduling contexts. We conclude that the required additions are small and manageable for any guest operating system.

6 Conclusions and Outlook

This chapter concludes the thesis by summarizing the contributions of this work and giving directions for future work.

6.1 Contributions

This thesis addresses the challenges raised by the ongoing demand of system consolidation in the area of real-time systems. Through the use of virtualization, systems, formerly distributed to multiple physical systems, can be merged into a single system, exploiting increased hardware performance and less overall resource usage.

The main contributions of this work are:

A portable, generic and latency-aware virtualization solution that allows to run a variety of guest systems on a wide range of systems and platforms. Through a natural enhancement of the existing microkernel-based system it allows to use virtualization techniques not only to run legacy applications but also as an integral part of microkernel-based applications.

As hardware platforms have a different range of features regarding their virtualization capabilities, different virtualization solutions are required. However, for ease of applicability the software interface to the different virtualization solutions shall be a generic as possible. The vCPU-based virtualization interface in the Fiasco.OC microkernel allows to use paravirtualization and hardware-assisted virtualization features under the hood of a common interface that only exhibits differences where enforced by hardware.

For running latency-constrained systems through virtualization, the virtualization layer must be designed in a way to allow guests to maintain their latency requirements. First, that requires that the virtualization layer only adds minimal overhead: The overhead that is required by the hardware to ensure isolation in the whole system. Otherwise it shall provide access as direct, and thus as fast, as possible. Second, the mechanisms must be designed in a way that the guest can react as direct as possible to incoming events. The vCPU mechanism provides the direct interface to the virtualization functionality and improves over previous paravirtualization approaches regarding event reaction. The mechanism has been implemented for x86 and ARM architectures.

Virtualization of real-time guests through scheduling contexts. Virtual machine guests may run both best-effort and real-time jobs. However, this distinction is not made at the hypervisor level that only schedules VM, rather than individual jobs

inside the VM. The introduction of scheduling contexts, a mechanism to export the necessary scheduling information to the hypervisor, allows to handle different types of jobs inside the VM. The responsibility for selecting the appropriate scheduling context is up to the VM while the host system ensures that the VM only uses assigned resources. Scheduling contexts also do away with the need to run at high scheduling frequencies for low event reaction latencies and thus save computing resources and energy.

Further, mixed-criticality systems have shown to map onto the mechanism of scheduling contexts, including mixed-criticality systems that run inside virtual machines. That allows to extend mixed-criticality system with virtualization, giving those system a broader applicability.

Generic and Hierarchical Scheduling Interface The proposed scheduling mechanism also shows to be applicable to for a multitude of possible mixed-criticality scheduling algorithms across a variety of platforms. Its hierarchical design further allows to stack several independent components in one system and provide required isolation and resource separation.

Both the vCPU-based virtualization and scheduling contexts have been implemented with the TUD:OS framework, showing feasibility of the approaches. vCPU-based virtualization has been provided on both the x86 and ARM architectures, including hardware-assisted virtualization. Scheduling contexts have shown practicability on both architectures and multiple platforms. In the introduction I claimed that this work addresses particular challenges which we can reflect on this work now:

- The **generality** of the approach has been accomplished by using different hardware architectures, namely x86 and ARM. Thus, use on other architectures is assumed to be straightforward.

- The approach uses **virtualization** on all platforms through the uniform *vCPU* interface. Hardware-provided features for virtualization are used to improve the virtualization performance.

- **Latency-constrained** applications and guests are supported by the preemptive Fiasco.OC microkernel and the asynchronous exception model provided by the introduced vCPU functionality.

- The **overhead of virtualization and isolation** as well as **resource** requirements has been shown to be minimal.

- By using a third-generation, capability-based microkernel, Fiasco.OC, the system uses a state-of-the-art **security** architecture.

- The **applicability** of the presented extensions to existing guest operating systems have shown to be small, clear and easy to integrate.

As the listing shows, all points could be covered. Overall my hope is that this work can give a valuable contribution and help to bring computing systems forward.

6.2 Outlook

This thesis opens up possibilities for future work. Further directions regarding scheduling contexts have already been discussed in Section 4.7.

On a more general view, the system can be extended in several directions. The proposed mechanisms allow for a configuration and reconfiguration of the system during runtime, thus a flexible admission system can manage changing system settings. Such a reconfiguration of a system might be beneficial for saving resources, for example, when a service is temporarily not required, less computing cores can be sufficient to provide necessary execution environment. The scope of the admission system can be from a simple approach of switching between different profiles up to a system that can dynamically manage new applications that are put on the system as well as their disappearance.

The applicability of the proposed mechanism to existing environments in virtualization settings might also be of interest. I have shown how scheduling contexts can be integrated in existing operating system kernels and how they can be virtualized. However, there are also domain-specific software architectures that require real-time execution. Examples are AUTOSAR, Real-Time Java and communication stacks such as for UMTS and its successor Long Term Evolution (LTE). When consolidating such systems it is of interesting how the proposed mechanisms can be applied, possibly reaching out to the application programming interface of that environment.

Bibliography

[AA06] Keith Adams and Ole Agesen. "A Comparison of Software and Hardware Techniques for x86 Virtualization." In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA, 2006, pp. 2–13. ISBN: 1-59593-451-0. DOI: 10.1145/1168857.1168860 (cit. on p. 33).

[AB98] L. Abeni and G. Buttazzo. "Integrating Multimedia Applications in Hard Real-Time Systems." In: *Proceedings of the IEEE Real-Time Systems Symposium*. RTSS '98. Washington, DC, USA, Dec. 1998, pp. 4–13. ISBN: 0-8186-9212-X. DOI: 10.1109/REAL.1998.739726 (cit. on p. 41).

[Acc+86] M. J. Accetta, R. V Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. "Mach: A New Kernel Foundation for Unix Development." In: *USENIX Summer Conference*. Atlanta, GA, June 1986, pp. 93–113 (cit. on pp. 43, 48).

[Adv08] Inc. Advanced Micro Devices. "White Paper: AMD-V™ Nested Paging." In: (July 2008) (cit. on p. 34).

[And+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*. SOSP '11. Cascais, Portugal, 2011, pp. 173–187. ISBN: 978-1-4503-0977-6. DOI: http://doi.acm.org/10.1145/2043556.2043574 (cit. on p. 40).

[ARM] ARM. *Cortex-M0 Processor*. URL: http://www.arm.com/products/processors/cortex-m/cortex-m0.php (visited on 09/2014) (cit. on p. 29).

[Aud+93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings. "Applying new scheduling theory to static priority pre-emptive scheduling." In: *Software Engineering Journal* 8.5 (Sept. 1993), pp. 284–292. ISSN: 0268-6961. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=238595 (cit. on p. 41).

[Bar+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the art of virtualization." In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. SOSP '03. Bolton Landing, NY, USA, 2003, pp. 164–177. ISBN: 1-58113-757-5. DOI: http://doi.acm.org/10.1145/945445.945462 (cit. on pp. 33, 37, 38, 60).

[Bar+10] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. "The VMware mobile virtualization platform: is that a hypervisor in your pocket?" In: *SIGOPS Oper. Syst. Rev.* 44

(4 Dec. 2010), pp. 124–135. ISSN: 0163-5980. DOI: http://doi.acm.org/10.1145/1899928.1899945 (cit. on p. 39).

[Bar+11] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. "Scheduling Real-time Mixed-criticality Jobs." In: *Computers, IEEE Transactions on* PP.99 (2011), p. 1. ISSN: 0018-9340. DOI: 10.1109/TC.2011.142 (cit. on pp. 29, 85).

[Bau+98] Robert Baumgartl, Martin Borriss, Hermann Härtig, Claude-Joachim Hamann, Michael Hohmuth, Lars Reuther, Sebastian Schönberg, and Jean Wolter. "Dresden Realtime Operating System." In: *Proceedings of the First Workshop on System Design Automation (SDA'98)*. Dresden, Mar. 1998, pp. 205–212 (cit. on pp. 43, 44).

[BBD11a] Sanjoy K Baruah, Alan Burns, and Robert I Davis. "Response-time analysis for mixed criticality systems." In: *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*. IEEE. 2011, pp. 34–43 (cit. on p. 90).

[BBD11b] S.K. Baruah, A. Burns, and R.I. Davis. "Response-Time Analysis for Mixed Criticality Systems." In: *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*. 29 2011-dec. 2 2011, pp. 34–43. DOI: 10.1109/RTSS.2011.12 (cit. on p. 93).

[BDR97] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. "Disco: running commodity operating systems on scalable multiprocessors." In: *Proceedings of the sixteenth ACM symposium on Operating systems principles*. SOSP '97. Saint Malo, France, 1997, pp. 143–156. ISBN: 0-89791-916-5. DOI: http://doi.acm.org/10.1145/268998.266672 (cit. on pp. 33, 60).

[Bel05] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator." In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA, 2005, pp. 41–41. URL: http://dl.acm.org/citation.cfm?id=1247360.1247401 (cit. on p. 39).

[BK09] Daniel Baldin and Timo Kerstan. "Proteus, a hybrid Virtualization Platform for Embedded Systems." In: *Analysis, Architectures and Modelling of Embedded Systems*. IFIP WG 10.5. Sept. 2009 (cit. on p. 41).

[Bra+08] Jörg Brakensiek, Axel Dröge, Hermann Härtig, Adam Lackorzynski, and Martin Botteck. "Virtualization as an Enabler for Security in Mobile Devices." In: *Proceedings of the First Workshop on Isolation and Integration in Embedded Systems (IIES 2008), EuroSys 2008 Affiliated Workshop*. Glasgow, Scotland, UK, Apr. 2008, pp. 17–22 (cit. on p. 56).

[Bug+12] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. "Bringing Virtualization to the x86 Architecture with the Original VMware Workstation." In: *ACM Trans. Comput. Syst.* 30.4 (Nov. 2012), 12:1–12:51. ISSN: 0734-2071. DOI: 10.1145/2382553.2382554 (cit. on pp. 33, 34).

[Bug+97] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. "Disco: running commodity operating systems on scalable multiprocessors." In: *ACM Trans. Comput. Syst.* 15 (4 Nov. 1997), pp. 412–447. ISSN: 0734-2071. DOI: http://doi.acm.org/10.1145/265924.265930 (cit. on p. 35).

[CC09] Gilles Chanteperdrix and Richard Cochran. "The ARM Fast Context Switch Extension for Linux." In: *Proceedings of the Eleventh Real-Time Linux Workshop*. Dresden, Germany, 2009 (cit. on p. 24).

[CGV07] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. "Comparison of the three CPU schedulers in Xen." In: *SIGMETRICS Perform. Eval. Rev.* 35.2 (Sept. 2007), pp. 42–51. ISSN: 0163-5999. DOI: 10.1145/1330555.1330556. URL: http://doi.acm.org/10.1145/1330555.1330556 (cit. on p. 82).

[Cha+12] Samarjit Chakraborty, Martin Lukasiewycz, Christian Buckl, Suhaib Fahmy, Naehyuck Chang, Sangyoung Park, Younghyun Kim, Patrick Leteinturier, and Hans Adlkofer. "Embedded Systems and Software Challenges in Electric Vehicles." In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '12. Dresden, Germany, 2012, pp. 424–429. ISBN: 978-3-9810801-8-6. URL: http://dl.acm.org/citation.cfm?id=2492708.2492815 (cit. on p. 15).

[Cho+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. "An empirical study of operating systems errors." In: *Proceedings of the eighteenth ACM symposium on Operating systems principles*. SOSP '01. Banff, Alberta, Canada, 2001, pp. 73–88. ISBN: 1-58113-389-8. DOI: 10.1145/502034.502042 (cit. on p. 32).

[Coma] Coreboot Community. *The coreboot Project*. URL: http://coreboot.org/ (cit. on p. 25).

[Comb] Linux Community. *Linux Operating System Kernel*. URL: http://www.kernel.org/ (visited on 12/2014) (cit. on p. 38).

[Coo] Intel Cooperation. *Extensible Firmware Interface (EFI) and Unified EFI (UEFI)*. URL: http://www.intel.com/technology/efi/ (cit. on p. 25).

[Coo14] Oracle Cooperation. *Java Card Platform Specification*. 2014. URL: http://www.oracle.com/technetwork/java/javame/javacard/download/platformspec/index.html (cit. on p. 19).

[Cor11] Intel Corporation. *Intel Virtualization Technology for Directed I/O*. Revision 1.3. Feb. 2011 (cit. on pp. 26, 37).

[Cor14] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*. 325462-052US, September 2014. Sept. 2014 (cit. on pp. 24, 25, 36, 59, 60, 70).

[CRM10] A. Crespo, I. Ripoll, and M. Masmano. "Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach." In: *Proceedings of the 2010 European Dependable Computing Conference*. EDCC '10. Washington, DC, USA, 2010, pp. 67–72. ISBN: 978-0-7695-4007-8. DOI: http://dx.doi.org/10.1109/EDCC.2010.18 (cit. on p. 42).

[DB05] R. I. Davis and A. Burns. "Hierarchical Fixed Priority Pre-Emptive Scheduling." In: *Proceedings of the 26th IEEE International Real-Time Systems Symposium*. Washington, DC, USA, 2005, pp. 389–398. ISBN: 0-7695-2490-7. DOI: 10.1109/RTSS.2005.25 (cit. on p. 41).

[Dev09] Advanced Micro Devices. *AMD I/O Virtualization Technology (IOMMU) Specification*. Rev 1.26. Feb. 2009 (cit. on p. 37).

[Dev11] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Rev 3.20. Dec. 2011 (cit. on pp. 36, 59, 70).

[Döb14] Björn Döbel. "Operating System Support for Redundant Multithreading." PhD thesis. Technische Universität Dresden, 2014 (cit. on p. 60).

[FPT14] Hasan Fayyad-Kazan, Luc Perneel, and Martin Timmerman. "Linux PREEMPT-RT V2.6.33 Versus V3.6.6: Better or Worse for Real-time Applications?" In: *SIGBED Rev.* 11.1 (Feb. 2014), pp. 26–31. ISSN: 1551-3688. DOI: 10.1145/2597457.2597460. URL: http://doi.acm.org/10.1145/2597457.2597460 (cit. on p. 53).

[Fre+10] Torsten Frenzel, Adam Lackorzynski, Alexander Warg, and Hermann Härtig. "ARM TrustZone as a Virtualization Technique in Embedded Systems." In: *Proceedings of Twelfth Real-Time Linux Workshop.* Nairobi, Kenya, Oct. 2010 (cit. on p. 36).

[Gum83] P. H. Gum. "System/370 Extended Architecture: Facilities for Virtual Machines." In: *IBM J. Res. Dev.* 27.6 (Nov. 1983), pp. 530–544. ISSN: 0018-8646. DOI: 10.1147/rd.276.0530 (cit. on p. 33).

[Här+97] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. "The Performance of μ-Kernel-based Systems." In: *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP).* Saint-Malo, France, Oct. 1997, pp. 66–77 (cit. on pp. 44, 48, 51, 54, 115, 116).

[Här+98] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. "DROPS: OS Support for Distributed Multimedia Applications." In: *Proceedings of the Eighth ACM SIGOPS European Workshop.* Sintra, Portugal, Sept. 1998 (cit. on pp. 17, 48).

[Hes+08] S. Hessel, F. Bruns, A. Bilgic, A. Lackorzynski, H. Härtig, and J. Hausner. "Acceleration of the L4/Fiasco microkernel using scratchpad memory." In: *Proceedings of the First Workshop on Virtualization in Mobile Computing.* MobiVirt '08. Breckenridge, Colorado, 2008, pp. 6–10. ISBN: 978-1-60558-328-0. DOI: http://doi.acm.org/10.1145/1622103.1622106 (cit. on p. 23).

[HHW98] Hermann Härtig, Michael Hohmuth, and Jean Wolter. "Taming Linux." In: *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98).* Adelaide, Australia, Sept. 1998 (cit. on pp. 49, 56).

[HL07] Galen C. Hunt and James R. Larus. "Singularity: rethinking the software stack." In: *SIGOPS Oper. Syst. Rev.* 41.2 (Apr. 2007), pp. 37–49. ISSN: 0163-5980. DOI: 10.1145/1243418.1243424 (cit. on pp. 19, 32).

[Hof02] Sarah Hoffmann. *Kleine Adressräume für FIASCO.* German. Großer Beleg, Technische Universität Dresden, Fakultät Informatik, Institut für Systemarchitektur, Betriebssysteme. Aug. 2002 (cit. on p. 116).

[Hoh+04] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. "Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors." In: *Proceedings of the Eleventh ACM SIGOPS European Workshop.* Leuven, Belgium, Sept. 2004 (cit. on p. 43).

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X, 9780123838728 (cit. on p. 22).

[HTS02] Michael Hohmuth, Hendrik Tews, and Shane G. Stephens. "Applying source-code verification to a microkernel: the VFiasco project." In: *Proceedings of the 10th workshop on ACM SIGOPS European workshop.* EW 10. Saint-Emilion, France, 2002, pp. 165–169. DOI: 10.1145/1133373.1133405 (cit. on p. 32).

[Ins] Texas Instruments. *OMAP Platform*. URL: http://ti.com/omap/ (visited on 09/2014) (cit. on p. 26).

[IO08a] Megumi Ito and Shuichi Oikawa. "Improving Real-Time Performance of a Virtual Machine Monitor Based System." In: *Proceedings of the 6th IFIP WG 10.2 international workshop on Software Technologies for Embedded and Ubiquitous Systems*. SEUS '08. Anacarpi, Capri Island, Italy, 2008, pp. 114–125. ISBN: 978-3-540-87784-4. DOI: http://dx.doi.org/10.1007/978-3-540-87785-1_11 (cit. on p. 41).

[IO08b] Megumi Ito and Shuichi Oikawa. "Lightweight Shadow Paging for Efficient Memory Isolation in Gandalf VMM." In: *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*. Washington, DC, USA, 2008, pp. 508–515. ISBN: 978-0-7695-3132-8. DOI: 10.1109/ISORC.2008.60 (cit. on p. 41).

[KBG10] Timo Kerstan, Daniel Baldin, and Stefan Grösbrink. "Full Virtualization of Real-Time Systems by Temporal Partitioning." In: *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. in conjunction with the 22nd Euromicro Intl Conference on Real-Time Systems Brussels, Belgium, July 7-9, 2010. July 2010, pp. 24–32 (cit. on p. 41).

[Kin+] Yuki Kinebuchi, Kazuo Makijima, Takushi Morita, Alexandre Courbot, and Tatsuo Nakajima. "Composition Kernel: A Software Solution for Constructing a Multi-OS Embedded System." In: *EURASIP Journal on Embedded Systems* 2010 (). DOI: 10.1155/2010/458085 (cit. on p. 42).

[Kin+08] Yuki Kinebuchi, Midori Sugaya, Shuichi Oikawa, and Tatsuo Nakajima. "Task Grain Scheduling for Hypervisor-Based Embedded System." In: *Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*. Washington, DC, USA, 2008, pp. 190–197. ISBN: 978-0-7695-3352-0. DOI: 10.1109/HPCC.2008.144 (cit. on p. 41).

[Kiv+07] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. "KVM: the Linux virtual machine monitor." In: *Proceedings of the 2007 Ottawa Linux Symposium (OLS '07)*. Ottawa, Canada, July 2007, pp. 225–230 (cit. on pp. 33, 37, 38, 71).

[Kle+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: formal verification of an OS kernel." In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP '09. Big Sky, Montana, USA, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596 (cit. on pp. 32, 43).

[Lac+12] Adam Lackorzyński, Alexander Warg, Marcus Völp, and Hermann Härtig. "Flattening hierarchical scheduling." In: *Proceedings of the tenth ACM international conference on Embedded software*. EMSOFT '12. Tampere, Finland, 2012, pp. 93–102. ISBN: 978-1-4503-1425-1. DOI: 10.1145/2380356.2380376. URL: http://doi.acm.org/10.1145/2380356.2380376 (cit. on p. 85).

[Lac04] Adam Lackorzynski. "L^{4}Linux Porting Optimizations." MA thesis. TU Dresden, Mar. 2004 (cit. on p. 48).

[Lac12] Adam Lackorzynski. "Managing Low Latency in Paravirtualized Virtual Machines." In: *Proceedings of the 14th Real-Time Linux Workshop*. Chapell Hill, North Carolina, US, 2012 (cit. on p. 77).

[Lan+il] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Zheng Cui, Lei Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. "Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing." In: *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. April, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470482 (cit. on p. 72).

[Law96] Kevin P. Lawton. "Bochs: A Portable PC Emulator for Unix/X." In: *Linux J.* 1996.29es (Sept. 1996). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=326350.326357 (cit. on p. 33).

[LD] J. Lange and P. Dinda. *An Introduction to the Palacios Virtual Machine Monitor—Release 1.0*. Tech. rep. NWU-EECS-08-11. Department of Electrical Engineering and Computer Science, Northwestern University (cit. on p. 72).

[LH04] Jork Loeser and Hermann Härtig. "Low-latency Hard Real-Time Communication over Switched Ethernet." In: *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*. Catania, Italy, June 2004, pp. 13–22 (cit. on p. 49).

[LHH97] Jochen Liedtke, Hermann Haertig, and Michael Hohmuth. "OS-Controlled Cache Predictability for Real-Time Systems." In: *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*. RTAS '97. Montreal, Canada, June 1997, pp. 213–223. ISBN: 0-8186-8016-4. URL: http://dl.acm.org/citation.cfm?id=523983.828369 (cit. on p. 23).

[Lie10] Steffen Liebergeld. "Lightweight Virtualization on Microkernel-based Systems." MA thesis. TU Dresden, Feb. 2010 (cit. on p. 71).

[Lie95] J. Liedtke. "On micro-kernel construction." In: *Proceedings of the fifteenth ACM symposium on Operating systems principles*. SOSP '95. Copper Mountain, Colorado, United States, 1995, pp. 237–250. ISBN: 0-89791-715-4. DOI: http://doi.acm.org/10.1145/224056.224075 (cit. on p. 43).

[Lie96] J. Liedtke. *L4 Reference Manual (486, Pentium, PPro)*. Arbeitspapiere der GMD No. 1021. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996. Sankt Augustin: GMD — German National Research Center for Information Technology, Sept. 1996 (cit. on p. 48).

[Lim14] ARM Limited. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*. ARM DDI 0406C.c. 2014 (cit. on pp. 25, 28, 36, 60).

[Lös06] Jork Löser. "Low-Latency Hard Real-Time Communication over Switched Ethernet." PhD thesis. Fakultät Informatik: TU Dresden, Oct. 2006 (cit. on pp. 28, 31).

[LPL10] Steffen Liebergeld, Michael Peter, and Adam Lackorzynski. "Towards Modular Security-Conscious Virtual Machines." In: *Proceedings of Twelfth Real-Time Linux Workshop*. Nairobi, Kenya, Oct. 2010 (cit. on p. 71).

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. "The rate monotonic scheduling algorithm: exact characterization and average case behavior." In: *Real Time Systems Symposium, 1989., Proceedings.* Dec. 1989, pp. 166–171. DOI: 10.1109/REAL.1989.63567 (cit. on p. 41).

[LVW13] Adam Lackorzyński, Marcus Völp, and Alexander Warg. "Flat but Trustworthy: Security Aspects in Flattened Hierarchical Scheduling." In: *Proceedings of the Workshop on Virtualization for Real-Time Embedded Systems.* VtRES 2013. Taipei, Taiwan, 2013 (cit. on p. 100).

[LW09] Adam Lackorzynski and Alexander Warg. "Taming subsystems: Capabilities as Universal Resource Access Control in L4." In: *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems, Eurosys affiliated workshop.* IIES '09. Nuremburg, Germany, Mar. 2009, pp. 25–30. ISBN: 978-1-60558-464-5. DOI: http://doi.acm.org/10.1145/1519130.1519135 (cit. on pp. 44, 45).

[LWP10] Adam Lackorzynski, Alexander Warg, and Michael Peter. "Generic Virtualization with Virtual Processors." In: *Proceedings of Twelfth Real-Time Linux Workshop.* Nairobi, Kenya, Oct. 2010 (cit. on p. 61).

[Mck04] Paul E. Mckenney. "Exploiting Deferred Destruction: An Analysis of Read-copy-update Techniques in Operating System Kernels." AAI3139819. PhD thesis. 2004 (cit. on p. 77).

[Meh+01] Frank Mehnert, Michael Hohmuth, Sebastian Schönberg, and Hermann Härtig. "RTLinux with Address Spaces." In: *Proceedings of the Third Real-Time Linux Workshop.* Milano, Italy, Nov. 2001 (cit. on p. 22).

[Meh05] Frank Mehnert. "Kapselung von Standard-Betriebssytemen." PhD thesis. TU Dresden, July 2005 (cit. on pp. 35, 49).

[Mic] Microsoft. *Hyper-V.* URL: http://technet.microsoft.com/en-us/windowsserver/dd448604.aspx (visited on 09/2014) (cit. on p. 33).

[MN10] Roberto Mijat and Andy Nightingale. "Virtualization is Coming to a Platform Near You." In: *ARM White Paper* (2010). URL: http://www.arm.com/files/pdf/System-MMU-Whitepaper-v7.0.pdf (cit. on p. 20).

[MS70] R. A. Meyer and L. H. Seawright. "A virtual machine time-sharing system." In: *IBM Systems Journal* 9.3 (1970), pp. 199–218. ISSN: 0018-8670. DOI: 10.1147/sj.93.0199 (cit. on p. 33).

[MYS03] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. *Capability Myths Demolished.* Tech. rep. Combex, Inc.; University of California, Berkley; John Hopkins University, 2003 (cit. on p. 32).

[Nak+11] Tatsuo Nakajima, Yuki Kinebuchi, Hiromasa Shimada, Alexandre Courbot, and Tsung-Han Lin. "Temporal and spatial isolation in a virtualization layer for multi-core processor based information appliances." In: *Proceedings of the 16th Asia and South Pacific Design Automation Conference.* ASPDAC '11. Yokohama, Japan, 2011, pp. 645–652. ISBN: 978-1-4244-7516-2. URL: http://dl.acm.org/citation.cfm?id=1950815.1950942 (cit. on p. 40).

[Ora] Oracle Corporation. *Virtualbox.* URL: http://www.virtualbox.org/ (visited on 12/2014) (cit. on p. 34).

[Pal+11a] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. "Faults in linux: ten years later." In: *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. ASPLOS '11. Newport Beach, California, USA, 2011, pp. 305–318. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950401 (cit. on p. 32).

[Pal+11b] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. "Faults in linux: ten years later." In: *SIGPLAN Not.* 46.3 (Mar. 2011), pp. 305–318. ISSN: 0362-1340. DOI: 10.1145/1961296.1950401 (cit. on p. 43).

[Pet+09] Michael Peter, Henning Schild, Adam Lackorzynski, and Alexander Warg. "Virtual Machines Jailed: Virtualization in Systems with Small Trusted Computing Bases." In: *VDTS '09: Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*. Nuremberg, Germany, 2009, pp. 18–23. ISBN: 978-1-60558-473-7. DOI: http://doi.acm.org/10.1145/1518684.1518688 (cit. on pp. 70, 71).

[PG74] Gerald J. Popek and Robert P. Goldberg. "Formal requirements for virtualizable third generation architectures." In: *Commun. ACM* 17 (7 July 1974), pp. 412–421. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/361011.361073 (cit. on p. 33).

[PSR96] F. B. des Places, N. Stephen, and F. D. Reynolds. "Linux on the OSF Mach3 Microkernel." In: *Conference on Freely Distributable Software*. Boston, MA, Feb. 1996 (cit. on pp. 43, 48).

[Reu05] Lars Reuther. "Disk Storage and File Systems with Quality-of-Service Guarantees." PhD thesis. Fakultät Informatik: TU Dresden, Nov. 2005 (cit. on pp. 28, 31, 49).

[SB96] Marco Spuri and Giorgio Buttazzo. "Scheduling aperiodic tasks in dynamic priority systems." In: *Real-Time Systems* 10 (2 1996). 10.1007/BF00360340, pp. 179–210. ISSN: 0922-6443 (cit. on p. 41).

[SBK10] Udo Steinberg, Alexander Böttcher, and Bernhard Kauer. "Timeslice Donation in Component-Based Systems." In: *Proceedings of the OSPERT 2010 Workshop (in conjunction with ECRTS 2010)*. Brussels, Belgium, July 2010 (cit. on p. 76).

[Sch+10] Jan Hendrik Schönherr, Jan Richling, Matthias Werner, and Gero Mühl. "Event-driven Processor Power Management." In: *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*. e-Energy '10. Passau, Germany, 2010, pp. 61–70. ISBN: 978-1-4503-0042-1. DOI: 10.1145/1791314.1791323. URL: http://doi.acm.org/10.1145/1791314.1791323 (cit. on p. 124).

[Sch02] S. Schönberg. "Using PCI-Bus Systems in Real-Time Environments." PhD thesis. Fakultät Informatik: TU Dresden, Sept. 2002 (cit. on p. 28).

[Sha99] J. S. Shapiro. "EROS: A Capability System." PhD thesis. University of Pennsylvania, Apr. 1999 (cit. on p. 32).

[SIG] PCI SIG. *Single Root I/O Virtualization*. URL: https://www.pcisig.com/specifications/iov/single_root/ (visited on 09/2014) (cit. on p. 35).

[Sin+06] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. "Reducing TCB complexity for security-sensitive applications: three case studies." In: *SIGOPS Oper. Syst. Rev.* 40.4 (2006), pp. 161–174. ISSN: 0163-5980. DOI: http://doi.acm.org/10.1145/1218063.1217951 (cit. on p. 16).

[SK10] Udo Steinberg and Bernhard Kauer. "NOVA: a microhypervisor-based secure virtualization architecture." In: *EuroSys '10: Proceedings of the 5th European conference on Computer systems*. Paris, France, 2010, pp. 209–222. ISBN: 978-1-60558-577-2. DOI: http://doi.acm.org/10.1145/1755913.1755935 (cit. on pp. 34, 41, 71).

[SLW09] Henning Schild, Adam Lackorzynski, and Alexander Warg. "Faithful Virtualization on a Real-Time Operating System." In: *Proceedings of the Eleventh Real-Time Linux Workshop*. Dresden, Germany, 2009, pp. 237–243 (cit. on pp. 28, 39).

[SS75] Jerome H. Saltzer and Michael D. Schroeder. "The protection of Information in Computer Systems." In: *Proceedings of the IEEE* 63.9 (Sept. 1975), pp. 1278–1308. ISSN: 0018-9219. DOI: http://dx.doi.org/10.1109/PROC.1975.9939 (cit. on pp. 32, 45).

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. "EROS: a fast capability system." In: *Proceedings of the seventeenth ACM symposium on Operating systems principles*. SOSP '99. Charleston, South Carolina, United States, 1999, pp. 170–185. ISBN: 1-58113-140-2. DOI: http://doi.acm.org/10.1145/319151.319163 (cit. on p. 45).

[Ste04] Udo Steinberg. "Quality-Assuring Scheduling in the Fiasco Microkernel." MA thesis. TU Dresden, Mar. 2004. URL: http://os.inf.tu-dresden.de/papers_ps/steinberg-diplom.pdf (cit. on p. 92).

[Sto07] Jan Stoess. "Towards Effective User-controlled Scheduling for Microkernel-based Systems." In: *SIGOPS Oper. Syst. Rev.* 41.4 (July 2007), pp. 59–68. ISSN: 0163-5980. DOI: 10.1145/1278901.1278910. URL: http://doi.acm.org/10.1145/1278901.1278910 (cit. on p. 46).

[Sze+11] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. "Eliminating the hypervisor attack surface for a more secure cloud." In: *Proceedings of the 18th ACM conference on Computer and communications security*. CCS '11. Chicago, Illinois, USA, 2011, pp. 401–412. ISBN: 978-1-4503-0948-6. DOI: 10.1145/2046707.2046754 (cit. on p. 39).

[Tan08] Andrew S. Tanenbaum. *Modern operating systems (3. ed.)* Pearson Education, 2008, pp. I–XXVII, 1–1076. ISBN: 978-0-13-600663-3 (cit. on p. 32).

[Tur10] Chris Turner. *Cortex-R4 processor*. May 2010. URL: http://www.arm.com/files/pdf/Cortex-R4-white-paper.pdf (visited on 12/2014) (cit. on p. 29).

[VLH13] Marcus Völp, Adam Lackorzynski, and Hermann Härtig. "On the Expressiveness of Fixed-Priority Scheduling Contexts for Mixed-Criticality Scheduling." In: *Proceedings of the 1st International Workshop on Mixed Criticality Systems*. IEEE. Vancouver, Canada, Dec. 2013, pp. 13–18. URL: http://www.cs.york.ac.uk/ftpdir/reports/2013/YCS/486/YCS-2013-486.pdf (cit. on pp. 93, 110).

[Whe] David Wheeler. *SLOCCount Website*. URL: http://www.dwheeler.com/sloccount/ (visited on 12/2014) (cit. on p. 119).

[Wig+03] Adam Wiggins, Harvey Tuch, Volkmar Uhlig, and Gernot Heiser. "Implementation of Fast Address-Space Switching and TLB Sharing on the StrongARM Processor." In: *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*. Aizu-Wakamatsu City, Japan, Sept. 2003 (cit. on p. 24).

[WL11] Alexander Warg and Adam Lackorzynski. "Rounding pointers: type safe capabilities with C++ meta programming." In: *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*. PLOS '11. Cascais, Portugal, 2011, 3:1–3:5. ISBN: 978-1-4503-0979-0. DOI: 10.1145/2039239.2039244 (cit. on p. 47).

[Xi+] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. *RT-Xen: Real-Time Virtualization Based on Fixed-Priority Hierarchical Scheduling*. Tech. rep. WUCSE-2010-38. Department of Computer Science and Engineering, Washington University in St. Louis (cit. on p. 82).

[Xi+11] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. "RT-Xen: towards real-time hypervisor scheduling in xen." In: *Proceedings of the ninth ACM international conference on Embedded software*. EMSOFT '11. Taipei, Taiwan, 2011, pp. 39–48. ISBN: 978-1-4503-0714-7. DOI: http://doi.acm.org/10.1145/2038642.2038651 (cit. on pp. 42, 119).

[ZB07] Fengxiang Zhang and A. Burns. "Analysis of Hierarchical EDF Pre-emptive Scheduling." In: *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*. Dec. 2007, pp. 423–434. DOI: 10.1109/RTSS.2007.12 (cit. on p. 41).

[Zha+11] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. "CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal, 2011, pp. 203–216. ISBN: 978-1-4503-0977-6. DOI: http://doi.acm.org/10.1145/2043556.2043576 (cit. on p. 40).