L4 – Virtualization and Beyond

Hermann Härtig*,× Michael Roitzsch* Adam Lackorzynski×
Björn Döbel× Alexander Böttcher*

*Department of Computer Science
Technische Universität Dresden
01062 Dresden, Germany

×GWT-TUD GmbH
01187 Dresden, Germany

{haertig,mroi,adam,doebel,boettcher}@os.inf.tu-dresden.de

Abstract

After being introduced by IBM in the 1960s, virtualization has experienced a renaissance in recent years. It has become a major industry trend in the server context and is also popular on consumer desktops. In addition to the well-known benefits of server consolidation and legacy preservation, virtualization is now considered in embedded systems. In this paper, we want to look beyond the term to evaluate advantages and downsides of various virtualization approaches. We show how virtualization can be complemented or even superseded by modern operating system paradigms. Using L4 as the basis for virtualization and as an advanced microkernel provides a best-of-both-worlds combination. Microkernels can contribute proven real-time capabilities and small trusted computing bases for security-sensitive applications. We discuss L^{4}Linux and Symbian on L4 as virtualization examples and present several case studies where microkernels provide added value.

1. Introduction

Once a research topic, virtualization is a technology now easily available to end users. Thanks to industry efforts in product development, users can run Windows applications on their Linux desktop with seamlessly integrated window management. Others play DirectX-based Windows games within the OpenGL-based Mac OS X environment, using full hardware acceleration by the GPU. Virtual machines are used to test potentially malicious software without risking the host environment and they help developers in debugging crashes with back-in-time execution.

In the server world, virtual machines are used to consolidate multiple services previously located on dedicated machines. Running them within virtual machines on one physical server eases management and helps saving power by increasing utilization. In server farms, migration of virtual machines between servers is used to balance load with the potential of shutting down completely unloaded servers or adding more to increase capacity. Lastly, virtual machines also isolate different customers who can purchase virtual shares of a physical server in a data center.

We will revisit those use cases in Section 2, discussing their relationship to virtualization technology in more detail. They demonstrate that virtualization is a systems technology that is now widely adopted and has become a major industry force.

Another interesting systems technology that has not yet gained so much public attention is microkernels. Kernels of today's desktop operating systems are typically monolithic. Operating system services ranging from file systems to networking and from user management to paging are running in the CPU

privileged mode. This includes device drivers, which are a major source of operating system bugs. The monolithic design makes it hard to assure global system properties like real-time capability, security policies or robustness properties, because every component of the large code base running in privileged mode can interfere with these properties. Microkernels offer a way out of this dilemma by minimizing the code running in privileged mode and providing mechanisms for a well-structured user land particularly suited to the task at hand. Consequently, microkernels have been used successfully to implement security-sensitive and real-time systems. First generation microkernels were considered slow, but with the L4 family of microkernels, this drawback has been overcome. However, with the Hurd [7] still unfinished, microkernels miss a widely adopted native execution environment for general purpose applications. This lack can be mitigated by virtualization. Section 3 gives some background on microkernels, including a historical overview and their scientific achievements.

In Section 4, we will explore, how microkernel and virtualization technologies can be combined for mutual benefit. Virtualization can provide the required application support on top of a microkernel system, while the microkernel can contribute its unique system design, enabling the construction of platforms with properties beyond today's systems.

Section 5 concludes the paper and provides an outlook.

2. Virtualization

The term virtualization is being used inflationary to describe various different technologies. In its most general meaning, virtualization stands for an abstraction of resources that provides a logical rather than an actual physical incarnation of those resources. It typically involves a change in numbers and functionality, so actual resources are either multiplexed or aggregated to virtual resources. The differentiation now stems from the various levels of abstraction and the resource being abstracted.

The well-known term of a virtual machine (VM) describes a self-contained execution environment implemented by software rather than physical circuitry. Other resources like networks or file systems can be virtualized with the VPN or VPFS [27] technologies. Traversing the abstraction levels, we can single out different virtualization flavors also subsumed under the common name.

2.1 Flavors

To discuss advantages and shortcomings of the virtualization variants, we need criteria to judge them. The most common qualitative benchmark is provided by Popek and Goldberg's classical requirements for virtualization [22], which we briefly summarize in the following:

Efficiency requires the majority of operations to be performed on actual resources rather than being intercepted by the virtualization layer.

Resource Control requires the virtualization layer to be in complete control of the virtualized resources. There should be no uncontrolled way to bypass the virtualization.

Equivalence requires the program running on virtual resources to exhibit behavior identical to running on actual resources. This only includes temporal behavior to the degree demanded by efficiency.

Language Runtime Virtualization

Language runtimes provide an execution container including a virtual CPU and memory and also allows access to the file system and other peripherals. One of the most widely known such virtual machines is the JVM, the Java Virtual Machine. The VM executes Java byte code which is either interpreted or compiled just-in-time (JIT). This sandboxes the execution completely, so the JVM can exercise full resource control, with JNI as a controlled way out of the sandbox.

However, no byte code instructions can be executed directly on actual hardware, with the uncommon exception of Java processors. This gap between byte code and actual machine instructions enables the portability of binary Java programs, but also violates Goldberg's efficiency criterion.

Other runtime-level VMs include those of scripting languages like Python, that ship with library frameworks and a tightly integrated class hierarchy. A similar runtime-level VM technology that is decoupled from a specific development environment is LLVM [16]. This type of VM is typically implemented as a user mode application with marginal impact on kernel design. A notable exception is Singularity [13]. We will however not pursue this type of virtualization any further in this paper. They are often limited to execution of programs written in specific languages – the JVM only executes Java programs – and do not enable generic legacy support.

API and ABI Emulation

While only considered virtualization in a wider sense, it is clearly a related technology. Implementations of one operating system personality on top of another is a prominent example. This can be performed on an API level as in Cygwin, a UNIX personality for Windows. Because it requires recompilation of applications, this technique violates Goldberg's equivalence requirement. This disadvantage can be mitigated with ABI level emulation as in Wine, which implements a Windows personality on top of UNIX. However, even ABI level emulation does not necessarily provide full equivalence, because the ABI's functionality is a reimplementation of the original. In the case of the Windows ABI, this means aiming at a moving target with a huge set of functions, some of which are not fully documented.

Complete resource control by the virtualization layer is not provided either, because the underlying and neighboring APIs of the actual environment are still available, with no controlled way for the virtualization layer to intercept them. However, other than for language runtime VMs, the virtualization is Goldberg-efficient, because all computation is executed on the actual CPU.

Operating System Virtualization

The syscall interface of an operating system abstracts from the physical machine, so it can be considered a virtualization level by itself. Some OSes multiplex their syscall API to implement compartments. These compartments allow multiple, isolated instances of the user environment for improved security of services or machine partitioning. Two representatives in the UNIX world are FreeBSD jails and OpenVZ for Linux.

OS virtualization is efficient, because similar to regular syscalls, most instructions run directly on the physical CPU. Only the instructions to enter the kernel have side effects. Isolation between different compartments is guaranteed. This technology is appealing because it satisfies all three of the Goldberg criteria.

Paravirtualization

The virtualization layer performing paravirtualization provides a machine interface, that is abstract, but very close to real hardware and that can be implemented efficiently. Paravirtualization is used to run entire operating systems with their respective user mode applications inside virtual machines. As the virtual platform is not identical to a physically existing one, the operating system has to be ported to the abstract machine interface. On the one hand, this violates Goldberg's equivalence requirement, but only for the architecture-dependent part of the operating system kernel. The remaining architecture independent part plus the entire user mode software stack can then run unmodified.

As the operating system was originally expected to run in CPU privileged mode, it has to be

deprivileged to allow for resource control. This is achieved by replacing privileged instructions with calls to the virtualization layer, which can thus exercise control over the virtual machine. These virtual machines are also efficient, because all non-privileged operations run directly on the physical CPU. A representative of this technology is Xen [3], although current versions can also run unmodified operating systems by using full virtualization (see below).

Full Virtualization

Also called faithful virtualization, this technique allows running entire operating systems with their user environment completely unmodified in a virtual machine which mimics the interface of actual hardware. The operating system and applications within the virtual machine is called *guest*. The environment provided by full virtualization is designed to be indistinguishable from real hardware with the exception of temporal behavior. Communication with code inside a virtual machine is often implemented by means of a virtual network card. This strong isolation can be opened by means of special guest device drivers for communication purposes.

If the instruction set is not virtualizable, as with x86, full virtualization can be implemented using partial binary translation. This is the approach taken by VMware [2] or QEMU (with KQEMU). Although this limits the efficiency according to Goldberg, the user mode code of the guest OS can still run unmodified. In the latest versions of the x86 instruction set, the virtualization holes have been closed with the Intel VT and AMD-V instruction set extensions. So hardware-assisted virtualization without binary translation is finally possible with x86.

Instruction Set Emulation

Except for language runtime virtualization, all other types of virtualization reuse the instruction set of the host CPU and enrich it with virtualization functionality on either the API, ABI, syscall or platform level. However, running a guest with a different instruction set immediately prevents direct execution of instructions on the host CPU. Some form of binary translation, either interpretation or recompilation, is required. QEMU uses this approach to run for example ARM guests on x86 hosts.

2.2 Virtualization Use Cases

As shown by the diversity of the mentioned flavors, the term virtualization alone does not adequately describe a specific technology. However, the wide range of technologies enables a wide range of solutions. So it is essential to start with an analysis of the use cases to understand the problem and then pick the technology best suiting the needs. We will now revisit some of the popular use cases related to virtualization and evaluate the choice of solutions.

Application Integration

Probably the most common use case for virtualization on the desktop is running an application not available for your machine's native operating system. If neither the source code of the application nor a reliable ABI emulation is available, running the application together with the operating system it requires is a viable option. The application gets the environment it expects and no code has to be modified. Several full virtualization products on the market have successfully developed sophisticated solutions for this need with a slew of features from GPU acceleration to VM snapshots. Implementations use a hosted VMM running on top of the host operating system [26] as depicted in Figure 1. This provides users with the look and feel of a virtual computer within an application window.

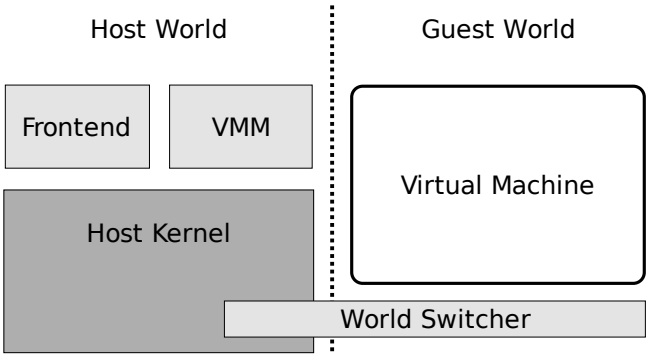

Figure 1: Hosted VMM Architecture

However, when it comes to integrating the guest and host world, things start to get difficult and the strict isolation of full virtualization becomes a hindrance. Features like VMware's Unity, that display guest application windows on the host desktop without rendering the guest desktop, are implemented with special drivers and helper applications in the guest.

Sandboxing

Secure containment of potentially malicious applications or entire operating systems or running different versions of the same application without mixing up configurations is another use case that is often addressed by full virtualization. However, if there is no need for a guest OS different from the host OS, this use case really only requires reliable isolation. This kind of isolation should be provided by the operating system. If the contained application should be able to interact with resources implemented outside the sandbox, full virtualization provides thick walls and a more lightweight isolation would be appropriate.

Server Consolidation

Running multiple servers on one physical machine is motivated by better utilization. Other than on desktops, the typical implementation is not a hosted VMM, but a native *hypervisor*, which runs directly on hardware and provides virtual machines as the only abstraction. Figure 2 shows the resulting design.

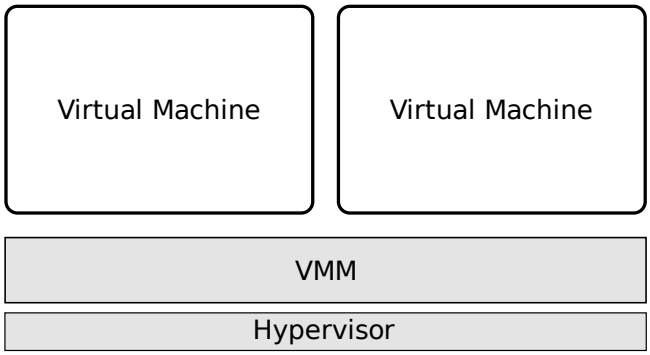

Figure 2: Hypervisor Architecture

If the consolidated servers all run the same operating system, the heterogeneity enabled by full virtualization is not required. In shared hosting environments, where customers buy virtual partitions of a physical server, the software stack is often homogeneous. Some providers have therefore opted to use operating system isolation features like BSD jails instead of full virtualization.

2.3 Decomposing Virtualization

If we are to design a system suitable for the use cases above, we can deduce three required features:

Isolation is required for sandboxed compartments.

Controlled communication must be provided. Compartments should be able to interact with high bandwidth and low latency, while a security policy is enforced.

Rehosting is the ability to run an operating system personality with its user environment on top of a host platform instead of on bare hardware.

Popular full virtualization implementations couple isolation and rehosting at the expense of fast communication. Looking at the use cases however, we can see that the sandboxing, consolidation and partitioning applications primarily require isolation. This is why operating system features like BSD jails are popular in this area. They provide lightweight isolation directly

built into the OS. But because UNIX kernels like BSD only received such features as an afterthought, the isolation is not very fine grained and communication control is rudimentary. We argue for a system with isolation as a first class property. Interaction between components should be controllable and fast.

Controlled ways to cross the isolation boundary can be devised. If permitted by system policy, communication primitives of the underlying platform can be made available to code running inside a virtual machine. Such code is then called *enlightened* and can easily use services implemented outside the virtual machine, with the system enforcing communication policies.

Only the desktop, where users want to mix programs from different operating systems, asks for rehosting. Of course, rehosting is also required for legacy support. While not a true use case per se, legacy support is important given the large amount of code available for existing commodity operating systems. We regard rehosting as a valuable bonus on top of isolation. Thus, we argue for a system design that is amenable to paravirtualization, so that rehosting can be implemented on top of it.

Other than full virtualization, which combines isolation and rehosting, but neglects the communication aspects, paravirtualization does not hinder communication. Enlightened guest applications can directly interact with outside components. By decoupling isolation – implemented by the kernel – and rehosting – implemented by a paravirtualized guest – we end up with a layered system design that offers many unique benefits. We think microkernels, which will be introduced in the following section, are a natural candidate to provide this substrate. In Section 4, we will then show the advantages of microkernel-based, paravirtualized systems.

3. Microkernels

Commodity desktop systems run large amounts of code including system services like networking or file systems, in the CPU's privileged mode. Such a design is typically called *monolithic*. Although even the monolithic kernels of today have evolved to modular, flexible designs, the components are only separated by convention, not by enforceable boundaries. One driver bug can crash the entire system. Unlike user level servers, it is also difficult to replace an in-kernel component at runtime with an alternative implementation or even just restart it, once it has failed. Kernel modules can help with that, but provide no enforced isolation.

Microkernels radically reduce the amount of code running in privileged mode. A microkernel provides no files, sockets or other higher level abstractions. The kernel only offers the necessary basis to implement functionality in user space. The idea of reduction to fundamental abstractions was already conceived in 1970 by Per Brinch Hansen [6] and was reformulated as the minimality criterion by Jochen Liedtke 25 years later [19]:

> A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.

Thus, a microkernel only provides mechanisms that cannot be implemented outside the kernel. But other than exokernels [14], the microkernel is not limited to secure multiplexing of resources, but actually abstracts from them. The three basic abstractions provided by microkernels are:

The Task is a container for resources, most notably memory pages. Thus, it abstracts from memory and provides spatial isolation amongst programs.

The Thread is an execution activity. It is an abstraction of the CPU and provides temporal isolation amongst programs.

Interprocess Communication (IPC) is the mechanism with which interaction amongst otherwise isolated programs is enabled.

Monolithic kernels employ a layered system structure. A file access will be handled by the kernel, which passes it through the VFS, file system, buffer cache and disk driver layers. Systems built on top of a microkernel transform this into a horizontal structure of cooperating servers. These servers can only communicate through well-defined interfaces via IPC.

3.1 History of Microkernels

A prominent representative of the first generation of microkernels is the Mach kernel [1]. It was targeted to be a kernel to replace UNIX, but with the services implemented outside the kernel. Mach was a communication kernel, providing ports as the mechanism to address communication partners. In an effort to provide a UNIX-like environment, the MkLinux project was started. It set out to port Linux to run as a paravirtualized single UNIX server on top of Mach. However, Mach's IPC performance slowed MkLinux down by about 30% compared to native Linux [11]. This and other Mach experiments led to the belief that the microkernel system design paradigm of collaborating user mode servers was fundamentally flawed, because the incurred communication overhead between the distributed components would slow the system down. To compensate, designers would colocate system services with the kernel.

This generalization of Mach's results was overcome, when Jochen Liedtke reexamined the design of the early generation of microkernels. Trying to prove that a minimal kernel can still provide a high overall system performance, he developed first L3, then the L4 kernel [19]. It was optimized for IPC performance [18] with the first implementation written in assembly code. Adhering strictly to the microkernel paradigm, only strictly required mechanisms are allowed in the kernel. Mach's concepts of message buffering, overwhelming rights validation and destructively interfering pagers were dropped. The only exception to this rule is scheduling, which is implemented in the kernel for performance reasons. Comparable to MkLinux, Linux was also ported to run on top of L4, this time with a mere 2.2% slow down [11]. This L^{4}Linux project will be described in more detail below.

L4 revived the microkernel idea, which gives us the opportunity to examine, how microkernels can enable new system designs that combine isolation, legacy rehosting and native microkernel services in an innovative way.

4. L4 as a Hypervisor

L4 provides a suitable substrate for a well-structured user environment based on collaboration of isolated server components. Moreover, the abstractions it provides are also lightweight enough to be amenable to paravirtualization. In this section, we will show, that existing legacy operating systems can be ported to the L4 interface. The L4 kernel will then act as a hypervisor, providing the required isolation. The paravirtualized OS will provide an operating system personality for an unmodified user environment to run on. At the same time, the system is still a microkernel platform, so all microkernel benefits presented in the previous section can now be combined with paravirtualization. This opens up a wealth of new and interesting system designs, of which we now describe some representative case studies.

4.1 L^{4}Linux

L^{4}Linux is a paravirtualized version of the Linux operating system kernel which is adapted to run within the L4 environment as a normal user mode application side by side with other L4 applications instead of on bare hardware. First

results of L4Linux have been published in 1997 [11] based on Linux 2.0 and have since been adapted to new Linux versions as well as new L4 systems. The current version of L4Linux is based on Linux 2.6.26.

The adaptations required to run the Linux kernel in an L4 environment are solely located in the platform-specific code of Linux. All other code, most notably all drivers, are left completely unmodified and even most of the platform specific code is reused. Currently, L4Linux has been ported to run on the x86-32 and ARM architectures. We now briefly explain the architectural parts that required changes.

Startup of L4Linux is much easier than with native Linux as L4Linux is loaded as a normal application on L4. It is not required to initialize the hardware, so most of the 16-bit startup code and BIOS interaction can be skipped.

Low-level memory management has to be adapted as application programs are not allowed to change page tables directly. Linux has to make use of the primitives provided by the L4 microkernel to modify the virtual address space of its processes. In other words L4Linux hooks into the page table manipulation functions and indirectly manipulates the real hardware page tables by calling the appropriate L4 kernel calls. All of the remaining general memory management in Linux stays unmodified.

Device handling, specifically the low-level interface to the hardware, particularly interrupts and I/O memory, has to be modified. For interrupts, an interrupt controller driver is supplied which uses the L4 microkernel way of accessing interrupts via IPC. Due to the internal structure of interrupts in Linux, we use separate L4 threads to run the interrupt top halves in. Interrupts and I/O memory in L4 are accessed through a device manager that provides clients access to those resources based on a user-defined policy.

Physical memory needs to be known by its real location to device drivers that communicate with their devices via DMA. The L4Linux kernel does not run on bare hardware, which means that the addresses it uses are not the same as the physical addresses. Fortunately, drivers in Linux use an interface to convert a virtual to a physical address, which L4Linux can hook into.

User processes in L4Linux are implemented with L4 tasks and L4 threads therein. Each Linux process gets its own address space and is thus isolated from other Linux processes and L4 programs. L4Linux is binary compatible to native Linux which means that it is able to run any unmodified Linux distribution, including the X Window System and popular desktop environments with their applications.

Running Linux on L4 becomes even more interesting when we look at the possibilities of interaction between L4Linux and the surrounding L4 environment. First to mention are special device drivers that connect L4Linux to services offered by L4 servers. Those are needed when a hardware resource should be shared amongst L4Linux and native L4 applications. All clients that want to use this resource must then access it by interfacing with a multiplexing service. L4Linux offers so called *enlightened drivers* for such resources.

L4Linux can also provide services to the L4 world. This is very useful, because a Linux system offers a rich set of features like thousands of drivers for all sorts of devices, mature and high-performance network stacks, file systems and a huge amount of available software. It is tempting to allow L4 applications to leverage such features. This of course raises security issues, so the benefits of reusing Linux code and the security concerns of entrusting such a large code base with critical data must be carefully balanced.

Generally, there are two ways of adding services to L4Linux. The first is providing a Linux kernel

driver that offers a service. An example might be an L4Linux that drives a network card and offers a network service to other L4 applications. This requires knowledge on writing Linux device drivers as well as a good understanding of L4Linux' internal workings but might be worthwhile for performance reasons. A much easier way of offering service is to make use of normal Linux programs, which can make use of all the available libraries and tools in the Linux environment. As described previously, each Linux process is running in its own L4 task and thus also on an L4 thread. This means that a Linux process is an L4 addressable entity and can be made visible to the L4 world. We call such tasks *hybrid*. L4Linux offers special support for such a mode of operation. L4Linux uses the same scheduling and execution behavior as native Linux, which means that execution is switching between the L4Linux kernel server and the currently active Linux process. The progress of the L4Linux system also depends on the fact that control is returned to the L4Linux server when an interrupt occurs. Now, if one Linux user process is enlightened and is implementing a service, the thread waiting for incoming service requests must not be scheduled. Instead it must stay in its IPC and wait for clients. A problem arises, when a request arrives. The L4Linux scheduler needs to schedule the thread again.

To be able to implement this behavior in the L4Linux system, we added an additional mode for L4 threads called *alien*. Alien threads are L4 threads that cannot invoke system calls directly but instead cause the microkernel to send an exception message to the exception handler of the alien thread. Thus, the exception handler is notified that one of his alien threads wants to execute an L4 system call. Using a special reply, the exception handler can approve this L4 system call or deny it. Once the L4 system call is finished, the exception handler receives a completion message. For both notifications the exception handler is free to modify the state of the alien thread in any respect.

The described mechanism is also used to implement the hybrid Linux/L4 programs in L4Linux. All Linux user processes are alien threads, which means that any attempt to call L4 system calls will lead to an exception message to the L4Linux server. It thus knows that it has to handle a hybrid Linux/L4 program. It will do so by setting the Linux internal state of the Linux process to *UNINTERRUPTIBLE* and allowing it go on executing the L4 system call. In this state, the L4Linux kernel never chooses this process for execution. Execution within L4Linux will continue as usual with other processes. When the L4 system call in the hybrid program returns, the L4Linux server will receive another notification. It then can clear the *UNINTERRUPTIBLE* flag and thus let the process return to normal execution. It will be scheduled according to the L4Linux internal scheduler. Using this approach, it is possible to write hybrid Linux/L4 programs to couple the infrastructures of Linux and L4 in order to implement applications that combine the benefits of both. Several case studies for such applications will be presented in the following.

4.2 Virtualization and Real-Time

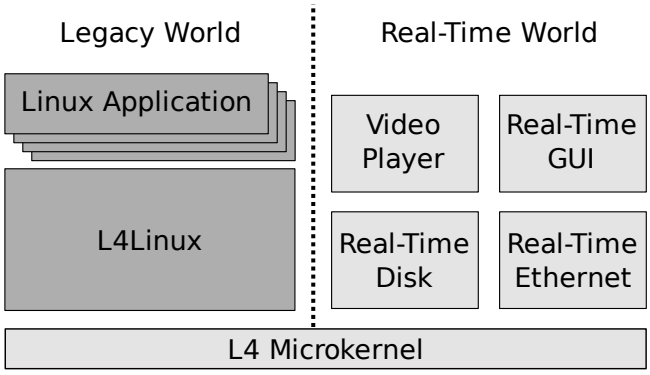

Figure 3: DROPS Architecture

A major challenge in computer systems is the coexistence of real-time and non-real-time applications on the same machine. A large variety of systems has to support a diverse set of such use cases: Multimedia applications have immediate

real-time requirements, because frames need to be delivered to the display at fixed time intervals. At the same time, non-real-time components like media management or editing components may be running next to the player core. Another example are mobile phones supporting a GSM stack with real-time requirements next to the typical set of calendar and address book applications. Running such applications on a commodity OS requires reworking the entire system including all drivers for real-time capability, which is prohibitively hard to accomplish.

On a real-time capable microkernel, all applications are temporally isolated and can execute with real-time guarantees. This property is not even disrupted by a paravirtualized legacy OS, because it is just another user mode application. Our L4 implementation, *Fiasco*, provides such real-time guarantees and is the foundation for the DROPS (Dresden Real-Time Operating System) architecture depicted in Figure 3. L^{4}Linux hosts the non-real-time legacy code, while a media player core and other real-time services run next to the paravirtualized environment in the real-time domain. Within this architecture, we also developed predictable resource managers to provide real-time guarantees for resources other than the CPU. Such resources include disk [23], network [20] and graphics [9].

The DROPS architecture enables real-time applications on a microkernel with predictable management for a variety of resources. Paravirtualization is employed to run the non-real-time part of such applications within the feature-rich Linux environment.

4.3 Virtualization and Security-Sensitive Applications

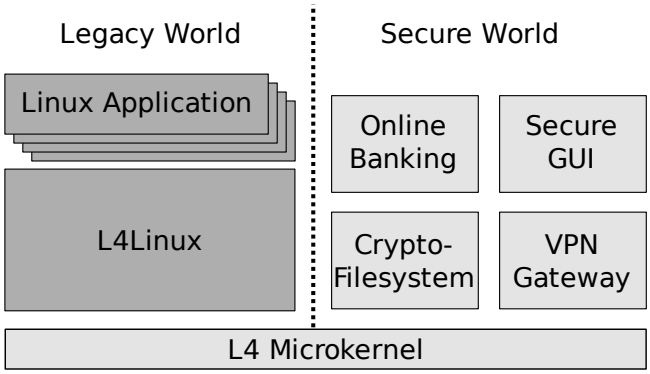

Figure 4: Nizza Architecture

Application correctness always relies on lower architectural layers like libraries used by the application, system services providing base resources and the operating system kernel itself. Some applications may handle security-sensitive data like cryptographic keys, logins, passwords or encrypted payload that resides temporarily unencrypted in memory. If memory content can potentially be spied upon by a lower level memory manager, it must be trusted not to do so. All system components on which the application must rely not to leak any information or maliciously disrupt its operation, form the *trusted computing base* (TCB) of an application. The operating system architecture has a strong influence on the size of the TCB. In monolithic systems, a large number of components are unnecessarily part of the TCB, because millions of lines of code are running in privileged mode or as privileged user tasks like the X server, with basically limitless possibilities to tap into or corrupt application data. For example, because the network stack in a monolithic system runs in privileged mode, it must be trusted by all applications, even those not using networking functionality. Ideally, the TCB should be tailored for each application, including only the services actually needed. This requires a system with a minimal kernel and a decomposed user environment, consisting of small, deprivileged servers.

In today's world, feature-rich applications seem to contradict this goal. An everyday example for many users is online banking in web browsers, where users enter security-critical data like logins, passwords and single-use transaction numbers. The browser runs on top of a large GUI framework and incorporates plugin code from sources of doubtful trustworthiness. On a monolithic operating system, it is also relying on functionally unrelated file system and device driver code. To reduce the TCB of such an online banking scenario while preserving the familiar browser user interface, we introduced the split-applications concept using paravirtualization [24].

Generally, paravirtualization is used to preserve a familiar environment, like running an unmodified web browser. The concept of split applications is used to separate security-critical parts into a dedicated protection domain. This security-critical part would not run in the paravirtualized environment, but directly on the microkernel and its services, relying only on code strictly required for the needed functionality. Between the parts, controlled communication exchanges data. If the user must be involved in the security-critical operation, it is necessary to provide a secure path to the user, otherwise malicious components can intercept sensitive data. A secure GUI multiplexer allows running windows of both the paravirtualized part and the security-critical part side by side, easily and safely distinguishable via unfakeable decoration. On L4, the Nitpicker secure GUI multiplexer [21] runs L^{4}Linux X11 applications alongside L4-based DOpE [9] GUI applications.

In the online banking scenario [24], the security-critical part comprises the SSL management and the analysis and display of security-related parts of web pages. SSL management includes certificate validation, session key handling and payload encryption and decryption. Pages that require input of sensitive data, like passwords or transaction numbers, are redirected to a small L4 DOpE client. All non-critical web pages are forwarded to the web browser in L^{4}Linux and displayed by X11.

A generalization of this approach is manifested in the Nizza architecture [10], which relies on a kernelized TCB and on the reuse of legacy code using trusted wrappers. The Nizza architecture was leveraged successfully to reduce the TCB of email signing [25], a VPN gateway [12], an encrypting file system [27] and an anonymity service [4]. Using the strong isolation provided by the microkernel to host the security-critical part separate from the large and untrusted legacy code allows to reduce the TCB of application scenarios compared to monolithic solutions by two orders of magnitude. Virtualization is used to provide an execution environment to run familiar legacy applications to minimize the impact on usability.

4.4 Virtualization and Device Drivers

Device drivers are a fundamental part of every operating system. Given the presence of an ever-growing number of devices to be supported, rewriting drivers for new operating systems can be a tedious amount of work. Fortunately, we can use virtualization to run device drivers from legacy operating systems within a microkernel-based system.

Another important fact is that drivers not only make up more than two thirds of the Linux kernel source code, but according to a study by Chou et al. [8] significantly contribute to the number of bugs in it. A promising approach to prevent faulty drivers from corrupting the rest of the operating system is to move them out of the kernel and run them as user-space applications. The problems arising herewith are similar to the problems of running paravirtualized OS kernels on top of a microkernel. Therefore, LeVasseur et al. [17] propose to use one instance of a paravirtualized kernel for every device driver and driver-related subsystem.

A major concern of our work in the area of microkernels has always been to reduce the trusted computing base needed by the system. Using a whole instance of the Linux kernel to run one device driver contradicts our concept of a small TCB. Most of the features needed by Linux device drivers, such as interrupt management, access to I/O ports and I/O memory, are already present in our L4-based system, the only difference being the interfaces to these services. Therefore, instead of incorporating a large software layer reduplicating this functionality, we chose to implement a library providing glue code to map Linux device subsystem calls to the services already available on L4. Our glue layer, called *Device Driver Environment* (DDE), consists of two parts. The fundamental *DDEKit* provides basic device driver mechanisms. This 1,500-LoC library is sufficient to construct user-level device drivers on top of the L4 base services. In addition to that, it is possible to construct glue code specific to legacy operating systems, such as Linux and FreeBSD. The DDE/Linux 2.6 layer consists of an additional 2,000LoC and is thus two orders of magnitude smaller than a minimally configured instance of the Linux kernel.

We implemented several user-level device servers using DDE, which at the moment provide access to network, hard disk, USB, TPM, and audio devices. Device servers virtualize hardware devices by providing an abstract hardware interface. This allows for multiplexing device resources between many clients. It is common for several instances of L^{4}Linux to access the same hardware network interface through the ORe software network switch. In addition, as a security feature, the driver servers can be used to enforce device access policies.

As an example, the ORe software network switch consists of approximately 92,000LoC, from which 89,000LoC are drivers that have been incorporated from Linux without any modifications. L^{4}Linux accesses the device servers using specific device drivers, so-called stubs. For the ORe network server, the Linux stub driver consists of 300LoC.

The DDE approach uses a thin API-level emulation layer to enable unmodified reuse of legacy device drivers in a microkernel-based system. The microkernel approach itself increases system robustness and fault isolation by moving drivers out of the kernel.

4.5 Symbian on L4

To enable advanced system designs like the case studies presented above to the embedded world, work has been undertaken to paravirtualize other operating systems on L4. In the scope of an embedded project within Nokia Research Center, Symbian has been ported to run side by side with L^{4}Linux on L4/Fiasco [5]. This proves operating system virtualization to be feasible even in embedded environments and can be used to bring different operating system worlds to one device while preserving security and real-time properties of neighboring software compartments.

5. Outlook and Conclusion

Current and next generation embedded systems pose new challenges for system designers. Mobile phone manufacturers consolidate the GSM stack with large legacy code bases of current embedded operating systems that can even run third party code. Thus, requirements concerning security, real-time and backward compatibility now appear combined in one handheld device. The problem gets even harder as the device is additionally limited by its energy budget.

The platform to build such devices must be able to run existing code, provide securely isolated components and drive the GSM hardware with real-time constraints. We believe that the L4 microkernel family is a suitable substrate to implement systems with such complex requirements: Paravirtualization can run the legacy OS with all available software. Next to it,

secure services can be placed and all tasks are globally scheduled according to a strict real-time regime. Our Symbian experiment proves the feasibility of such a system.

In contrast to paravirtualization, commodity virtualization technologies are an amalgamation of rehosting and isolation functionality. Popular use cases of virtualization mainly require isolation and thus use virtualization as a workaround for the lack of isolation in existing operating systems. By using the L4 microkernel, which provides proper isolation and a well-structured user environment with fast and controlled component interaction, we believe this deficiency can be overcome.

Virtualization is beneficial, when existing applications must run on the microkernel system unmodified. Porting is prohibitively expensive for large legacy applications that may only be available in binary form. However, when virtualization combines isolation and rehosting, the rehosting functionality is inert, when only isolation is required. A paravirtualizing facility on top of the microkernel solves this problem. Because the L4 API provides low-level abstractions, porting an existing legacy kernel to it is feasible, as demonstrated by the L^{4}Linux case study. This way, rehosting is implemented on top of the L4 microkernel, so it can be used optionally by only those legacy applications. All benefits of the microkernel are still available next to the paravirtualized environment.

Using these additional benefits in combination with paravirtualization allows for unique feature combinations. We presented examples for joining legacy code with real-time and strong security properties. Split applications on a microkernel can reduce the trusted computing base by two orders of magnitude without sacrificing functionality. We also demonstrated, how virtualization techniques help providing device support.

This design flexibility allows custom-tailored systems with just the right set of features, which is especially important for battery-limited embedded devices. ARM TrustZone is an emerging technology, which enables lightweight full virtualization in embedded systems. Together with ARM multicore ports of L4/Fiasco [15], microkernels and virtualization technology partner to enable new application spaces in the embedded domain.

Acknowledgments

We want to thank all our colleagues at TU Dresden who participated in the work presented. Most notably we thank Michael Hohmuth, Jean Wolter and Sebastian Schönberg for their work on Fiasco and the initial L^{4}Linux, Norman Feske for his work on the DOpE window server, Alexander Warg and Christian Helmuth for the Mikro-SINA experiment, Jork Löser for the real-time network theory and infrastructure, Martin Pohlack and Lars Reuther for the real-time disk scheduler and Bernhard Kauer and Carsten Weinhold for their work on the secure banking scenario. We want to extend our thanks to our friends in the L4 community: the L4 groups in Karlsruhe and in Sydney.

References

[1] 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, pages 93–113, Atlanta, GA, June 1986.

[2] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, pages 2–13, New York, NY, USA, 2006. ACM.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP), pages 164–177, Bolton Landing, NY, Oct. 2003.

[4] A. Böttcher, B. Kauer, and H. Härtig. Trusted computing serving an anonymity service. Springer Lecture Notes in Computer Science, 4968:143–154, 2008.

[5] J. Brakensiek, A. Dröge, H. Härtig, A. Lackorzynski, and M. 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, pages 17–22, Glasgow, Scotland, UK, April 2008.

[6] P. Brinch Hansen. The nucleus of a multiprogramming system. Commun. ACM, 13(4):238–241, Apr. 1970.

[7] M. Bushnell. Towards a New Strategy of OS Design. January 1994.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles, pages 73–88, New York, NY, USA, 2001. ACM.

[9] N. Feske and H. Härtig. Demonstration of DOpE – a Window Server for Real-Time and Embedded Systems. In 24th IEEE Real-Time Systems Symposium (RTSS), pages 74–77, Cancun, Mexico, Dec. 2003.

[10] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The nizza secure-system architecture. In In IEEE CollaborateCom 2005. IEEE Press, 2005.

[11] 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), pages 66–77, Saint-Malo, France, Oct. 1997.

[12] C. Helmuth, A. Warg, and N. Feske. Mikro-SINA – Hands-on Experiences with the Nizza Security Architecture. In Proceedings of the D.A.CH Security 2005, Darmstadt, Germany, Mar. 2005.

[13] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. SIGOPS Oper. Syst. Rev., 41(2):37–49, 2007.

[14] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, and T. Pinckney. Application performance and flexibility on exokernel systems. In Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP), Saint-Malo, France, Oct. 1997.

[15] A. Lackorzynksi. TUD:OS – the L4 SMP microkernel on ARM11 MPCore. http://www.youtube.com/watch?v=dWVfaZYkz-Y.

[16] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. Code Generation and Optimization, 2004. CGO 2004. International Symposium on, pages 75–86, March 2004.

[17] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, Berkeley, CA, USA, 2004. USENIX Association.

[18] J. Liedtke. Improving IPC by kernel design. In Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP), pages 175–188, Asheville, NC, Dec. 1993.

[19] J. Liedtke. On μ-kernel construction. In Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP), pages 237–250, Copper Mountain Resort, CO, Dec. 1995.

[20] J. Löser and H. Härtig. Real Time on Ethernet using off-the-shelf Hardware. In 1st Intl Workshop on Real-Time LANs in the Internet Age, Vienna, Austria, June 2002.

[21] Norman Feske and Christian Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC), 2005.

[22] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. Commun. ACM, 17(7):412–421, 1974.

[23] L. Reuther and M. Pohlack. Rotational-Position-Aware Real-Time Disk Scheduling Using a Dynamic Active Subset (DAS). In 24th IEEE Real-Time Systems Symposium (RTSS), pages 374–385, Cancun, Mexico, Dec. 2003.

[24] L. Singaravelu, B. Kauer, A. Boettcher, H. Haertig, C. Pu, G. Jung, and C. Weinhold. Enforcing Configurable Trust in Client-side Software Stacks by Splitting Information Flow. Technical report, GeorgiaTec University, Nov. 2007.

[25] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. SIGOPS Oper. Syst. Rev., 40(4):161–174, 2006.

[26] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In Proceedings of the 2001 USENIX Annual Technical Conference, General Track, pages 1–14, Boston, Massachusetts, USA, June 2001.

[27] C. Weinhold and H. Härtig. VPFS: building a virtual private file system with a small trusted computing base. SIGOPS Oper. Syst. Rev., 42(4):81–93, 2008.