Virtual Processors as Kernel Interface

Adam Lackorzynski, Alexander Warg
Technische Universität Dresden
Department of Computer Science
Operating Systems Group
{adam, warg}@os.inf.tu-dresden.de

Michael Peter
Technische Universität Berlin
Deutsche Telekom Laboratories
Security in Telecommunications
peter@sec.t-labs.tu-berlin.de

Abstract

After virtualization has gained traction in a variety of fields ranging from the desktop computer to datacenter servers, it is likely to make inroads into embedded systems as well. The complexity of a VM implementation depends on the virtualization abilities of the processor used. Unfortunately, the instruction set architecture of many popular embedded CPUs is not virtualizable, which precludes efficient pure or faithful virtualization.

In this paper, we make the case for operating system (OS) rehosting, a flavor of virtualization that lends itself to implementations of low complexity and does not rely an CPU virtualization extensions. The feasibility of OS rehosting crucially depends on the traits of the interface of the underlying kernel. Our observation was that the ubiquitously used thread model is rather poorly suited to run an OS on top. As a solution, we propose the currently often employed threads be supplemented with virtual processors (vCPUs), an abstraction that is more aligned with the underlying hardware.

To evaluate our proposition, we ported the Linux kernel to a vCPU enhanced version of the Fiasco microkernel. Compared to a previous thread-based version, the vCPU version required much less development effort. The performance gains range from slight to well-pronounced depending on the workload.

1 Introduction

The market for embedded devices has undergone a fundamental transition in the recent past. Closed special purpose devices with a fairly limited resource budget have turned into general purpose gadgets that often exhibit performance characteristics of desktop machines of a couple of years ago. Such rapid strides in capabilities led to calls for new features, foremost the ability to customize devices by installing software according to personal preferences. However, the record of operating systems in the last years does not instill confidence when it comes to security. What is a nuisance on desktop systems, becomes an unacceptable risk for some embedded systems. For example, no network operator can tolerate smartphones that came under illegitimate control after a downloaded application subverted the handset and turned it into a jammer.

The situation is complicated by the presence of multiple parties who want their interests safeguarded. The owner wants his assets such as access codes protected. The network operator is concerned about stable network operations. Content providers insist on the enforcement of the consumption rules for their content. Unfortunately, current systems have inherent design and implementation flaws that prevent them from isolating multiple stakeholders on one machine reliably. The underlying reason is the lack of mechanisms to grant authority selectively following the *principle of least authority*. There are too many parts of the system that run with privileges sufficient to take over the whole system if subverted.

Microkernels have shown that they can contribute to making the trustworthiness problem more tractable. Their contribution is twofold: first, they allow for the construction of systems with very small trusted computing bases (TCB). That is achieved by minimizing the amount of code running in the most privileged execution mode, where it is, by definition, part of the trusted computing base of *any* application regardless of whether the code is actually needed by an application. In contrast, functionality resident in user-level tasks can only affect other applications if it was *explicitly* granted sufficient authority.

Second, not subject to backward compatibility requirements, microkernels can break new grounds regarding security features. A huge step forward was the adoption of capability-based security schemes [19][12]. Facilitating the *principle of least authority*, capabilities are regarded as superior to access-control lists based systems [16].

Their individual shortcomings notwithstanding, operating systems are valuable components that no non-trivial system can dispense with. Economic pressure mandates the deployment of existing systems as no single organisation can hope for developing a reasonable OS in an acceptable time frame. Virtualization allows leveraging their strengths while still enforcing isolation thereby limiting the potentially affected scope in case of a failure or subversion.

Commodity processors in desktop and server systems feature to a large extent virtualization extensions, which are missing from most embedded processors. Yet, previous work[9] has shown that small kernels can encapsulate operating systems on commodity processors that do not efficiently support virtualization.

For reasons that are partly historical, microkernel designs paid more attention to raw message passing performance than to the ease of OS rehosting. In this paper, we propose to augment the kernel interface with virtual CPUs, an execution abstraction that bears a close resemblance to physical CPUs. This addition to the kernel interface has the potential to reduce the porting effort while increasing the confidence in the correctness of the changes to the guest operating system.

1.1 Outline

We will proceed with a discussion of what the choices are regarding the implementation of the CPU part of virtual machines (Section 2) before we detail the design of virtual CPUs in Section 3. To prove the feasibility of the proposition, we report on our implementation (Section 5) and describe how the Linux kernel can be ported onto it (Section 6). Our conclusion (Section 9) is preceded by measurements (Section 7) that give an impression of the performance characteristics and related work (Section 8).

2 Virtualization

The term *virtualization* itself is used in two different contexts which occasionally gives rise to confusion. In a wider sense, virtualization is a technology whereby operating systems run on a machine without exercising absolute control over it. In that meaning it is left open whether the guest OS is modified and, if so, how intrusive that modification is. Apart from (faithful or pure) virtualization, paravirtualization and OS rehosting are possible.

In a stricter sense, virtualization denotes a method whereby software is provided with an environment that is a genuine replica of a physical machine. If used in that context, it contrasts with paravirtualization and OS rehosting as the two latter provide environments that only resemble a physical machine. Accordingly, virtualization is, in principle, compatible with all operating systems whereas paravirtualization and OS rehosting require more or less intrusive changes. Figure 1 illustrates the different virtualization approaches, which we will describe in more details in the following sections.

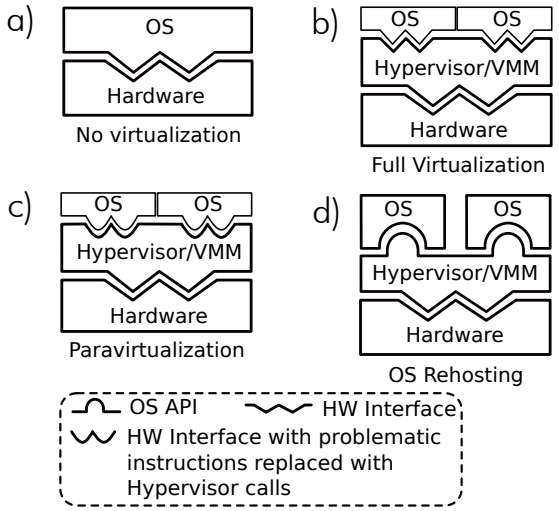

FIGURE 1: *Overview over different virtualization approaches. Virtualization duplicates the machine interface with high fidelity, whereas paravirtualization replaces problematic instructions with calls to the underlying kernel. OS rehosting goes further and provides a fairly abstracted kernel interface with many implementation specifics left out.*

2.1 Faithful Virtualization

Faithful virtualization – sometimes also referred to as *pure* or *full virtualization* – provides an environment that is a genuine copy of a physical machine. That allows for an existing operating system to be deployed without any modifications.

However, despite its merits, virtualization did not catch on in the commodity market because commodity processors did not have adequate support for

virtualization. For an instruction set architecture (ISA) to be virtualizable, it has to meet the Popek-Goldberg criterion[17]. Briefly, it mandates that each sensitive instruction is also privileged. A sensitive instruction is one that either reveals (privileged) execution state or affects the execution[1]. Many popular ISAs, though, are not virtualizable with x86[18] and ARM[6] the most prominent examples.

Although it is possible to implement faithful virtualization on CPUs with non-virtualizable ISAs, doing so efficiently involves complex technologies such as binary translation[3][1]. As such, this approach is of little appeal to security-concerned systems because their trustworthiness is negatively affected by complexity.

2.2 Paravirtualization

Faithful virtualization depends on the ability to efficiently intercept all of the guest's accesses to privileged resources such as control registers. That is necessary to retain control over the machine and present the guest with the behavior he would see if executing on a physical machine.

If not all accesses to privileged resources cause an exception or those exceptions are too expensive, it is reasonable to adopt a modified architecture, which is simpler to implement and yield better performance [20]. To that end, problematic privileged instructions are replaced with direct calls into the underlying software layer, which requires source code access. That is rather straight-forward task, which can even be fully automated[14].

The drawback of that approach is that it often draws on very specific ISA features for efficiency. For example, Xen [5]- a prominent representative for paravirtualization - makes use of three privilege rings on x86 and segmentation in order to use a single page table for both guest kernel and guest user. Such a close dependence on an ISA makes it non-portable. ARM, for example, features only two privilege levels and does not come with segmentation. Although an alternative implementation with dedicated pagetables for guest user and guest kernel might incur a lower performance penalty on ARM[2] than on x86, the diverging implementations would complicate the common code base.

2.3 OS Rehosting

Paravirtualization pays for the non-intrusive guest changes with the introduction of complexity into the kernel, which has to cover all quirks of a given ISA. That contrasts with OS rehosting, which aims to capture only those features that are necessary to host an operating system. The intention behind that change is to simplify the implementation of the kernel by deliberately dropping unused features that are of no importance for modern operating systems.

The simplification of the kernel comes at the cost of more intrusive changes required for the guest OS. Depending on the used system, it is likely possible that the features provided by the kernel such as threads or address spaces do not closely match the CPU model. In such cases, the effort required to port an OS may be significantly higher than that needed for paravirtualization. For example, User-mode Linux[2], a Linux kernel rehosted on itself, requires a fair amount of effort to emulate the behavior of a (physical) CPU with the mechanisms provided by Linux. *ptrace* was not designed for debugging processes, not hosting an OS kernel.

Microkernel systems provide a more light-weight, yet general CPU abstraction, allowing them to rehost existing operating systems more easily. Typical systems are L4Linux[9], Wombat[13], and MkLinux [8].

In this paper, we argue that OS rehosting can be superior to faithful virtualization and paravirtualization. Embedded processors often do not provide the virtualization features required for faithful virtualization, paravirtualization adds complexity to the kernel that does not pay off with added functionality. Our goal is it to facilitate OS rehosting by aligning the kernel interface more with the CPU model that is assumed by operating system kernels. We expect that the resulting solution is of lower complexity and simplifies the needed modifications of the guest kernel.

3 Synchronous Threading

Threads are a common way to run user-level activities. Restricting the system to only allow synchronous communication promises a simplified implementation in the kernel but also has consequences on the whole system design. In the synchronous model the thread state space is of low

[1] The *popf* instruction silently disregards bits when run at user level, which can change the control flow.
[2] ARM features tagged TLBs.

complexity as each thread can only be in either of four states: running, preempted-and-ready-to-run, blocked-on-sending-a-message, and blocked-on-receiving-a-message. Yet, this simple model is surprisingly versatile, provided that a method is available which allows threads to manipulate the state of other subordinate threads.

The problem of that approach is that a thread has to chose if it either waits for an incoming message or executes. Asynchronous activities can only be handled with dedicated threads that run tight event loops. Once an event has occurred, the event handling thread has to bring the event to the attention of the main service thread. This can be non-trivial if the latter engages, for example, in cross address space operations where it waits for the arrival of a message from a thread resident in another task. This scenario is the common setup for the hitherto Linux port on the Fiasco kernel.

The event thread may take different actions depending on whether that message has arrived or not, yet it has no efficient way to figure out the current situation. Current schemes often endow event threads with higher priorities than anyone else. The event thread then can dispense with explicit locking while inspecting the other thread's states because it can be sure that they do not execute. Still, the interaction with the main thread involves at least one more system call and even more, if the state of the sender thread needs to be recovered as well.

Apart from the performance aspect, having multiple threads also raises questions as to time provisioning. From an external perspective, it does not matter which thread consumes time. To that end, it would be expedient to assign the two threads different priorities but feed them from one budget. The different priorities ensure that the event handling thread takes precedence over the main thread while the shared budget guarantees that the two as group only consume a certain fraction of CPU time. However, we do not know of any microkernel that offers such functionality. It is not apparent if such a scheme would be compatible with other objectives, such as time donation along IPC dependency chains.

4 Design

The fundamental problem of synchronous schemes is that a thread has to decide whether it wants to make computational progress or be ready to receive messages as both cannot be done at the same time. It would be better if a thread could make computational progress but receive an asynchronous notification when an event requires its attention.

4.1 Kernel Interface

We extended the synchronous thread model with an asynchronous mode where events can be delivered without that the receiving thread has called an explicit blocking operation. To make this possible we extended the existing thread object with the following fields, which reside on a page shared between kernel and user.

State indicator. Comparable to a (virtual) interrupt flag, the state indicator controls whether events can be delivered. Having the state indicator set, a thread can execute *and* react to an event as soon as it occurs. An event is delivered by the kernel changing the control flow such that an event handling function is invoked.

The user can set and clear the flag at its discretion, for example, to ensure that code paths are executed atomically. An event delivery is also accompanied by clearing the flag.

State save area. The save state area holds the state of the interrupted context when an event is being delivered. The control transfer requires that part of the user visible CPU state is overwritten. At least the instruction pointer and additional registers required for the entry, for example, the stack pointer, need to be loaded with new values. The previous values are made available through the state save area. From there it is further transfered to local data structures such as thread control blocks.

The state save area corresponds to the kernel stack on IA32. Upon a kernel entry, the processor pushes the old values of instruction and stack pointer together with the flags register onto the kernel stack before it reloads them.

Pending event indicator. The event indicator facilitates fast transitions from delivery disabled to delivery enabled without going through the kernel or unduly delaying the delivery of events.

When an event occurs while the state indicator is clear, it cannot be delivered outright. Instead, the kernel queues it and sets the pending event indicator. When the vCPU eventually enables event delivery, it needs to check whether events are pending. Only in these cases it needs to trap into the kernel.

It depends on the application binary interface and the instruction set architecture specifics if a fast path for entering into the *event delivery enabled* state without kernel assistance is possible. More details on that will be furnished in Section 5.

The pending event indicator is the only field that has no direct correspondence in physical processors. Processors maintain their interrupt status internally and do not expose them through their ISA.

Entry vector. Delivering an event transfers the control flow of a vCPU to the event handling function. Such a disruption of the otherwise sequential control flow of the vCPU is referred to as *upcall*.

The entry point and the pointer to the stack of that function are stored in the entry vector fields. The handling function finds the parts of preempted execution state that had to be overwritten to resume the event handler function in the *state save area*.

Continuing the hardware analogy, on IA32, the processor retrieves the new instruction pointer from the interrupt descriptor table (IDT) and, if necessary, the stack pointer from the task state segement (TSS).

4.2 Virtual CPU API

The vCPU API consists of two functions: *disable* suppresses the delivery of events, which allows for atomically executed code paths. Of course, atomic execution is only guaranteed with respect to the vCPU. The microkernel can preempt the vCPU at any time irrespective of its event delivery status. However, if it finds the vCPU disabled, no upcalls will be generated and the execution will be resumed with the same state later. A thread library may make use of (local) atomic execution for, for example, thread switching or the implementation of synchronization primitives such as semaphores, mutexes, and condition variables.

By *enable*, the vCPU indicates that it is ready for receiving further events. When the microkernel preempts an enabled vCPU, it may resume its execution later with an upcall. An upcall indicates a condition that requires immediate attention.

Events can be either synchronous or asynchronous. If the vCPU encounters a condition that results in a processor fault, then this fault is reflected to the vCPU by the kernel. The vCPU has to ensure that it can handle the event without faulting again before the last fault has been resolved. The second group contains events that are triggered by external sources. For example, timer expirations, interrupts associated with the vCPU, and IPC messages are signalled through upcalls.

enable can be combined with an atomic change of the execution state, which is necessary to resume the execution of a previously preempted code path. While *disable* is a simple write operation to the *state indicator*, *enable* has to deal with the case that events that occurred while the state indicator was cleared are pending. A straight-forward implementation may use a syscall and leverage the kernels ability to atomically execute code paths. Depending on the traits of the ISA, it may be possible to atomically check for pending events and, provided none is present, enable event delivery and resume execution without entering into the kernel.

4.3 Comparision with Threading

Virtual CPUs provide a better foundation for the implementation of user threading packages than kernel threads. The reason is two-fold. First, transparent preemption of kernel threads raises some concerns with respect to concurrency handling. To illustrate the problem, assume a scenario where a critical section is protected by a lock. Threads try to grab the lock or donate computation time to the current lock owner to speed up its execution. Unfortunately, grabbing the lock, say with an atomic instruction, and announcing the lock ownership by writing one's own identity in a spot that is protected by the lock is non-atomic. If a preemption occurs in between, then a thread may find the lock taken but is unable to donate time. More complex situations are simple to conceive.

Second, switches between kernel threads almost always require kernel involvement[3], whereas vCPUs can switch directly between threads at user level as long as no events are pending.

4.4 Subdomains

A virtual CPU should be able to execute code in different address spaces. This feature is motivated by standard operating systems as envisioned workload where multiple user processes run and must be hindered from accessing the guest kernel memory as well as the memory of other guest processes.

[3]Some microkernel designs allow for user-level switching between kernel threads in a limited number of cases.

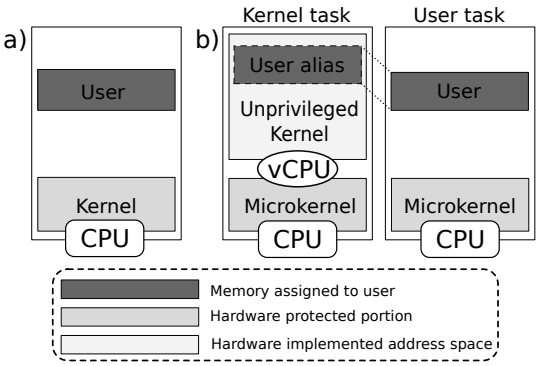

FIGURE 2: *Implemenation of a subdomain (a) on hardware, (b) on top of a microkernel*

Commodity processors allow to mark parts of an address space only accessible when running in a privileged mode. With the microkernel claiming exclusive rights on this privilege, this method cannot be used to protected a (guest) kernel from its subordinate domains. Instead, we use two address spaces as illustrated in Figure 2. The *resume* operation can take a reference to a second task, where the execution is to be resumed. Upon the occurrence of an event, the microkernel switches back to the kernel task, where the event is delivered to the vCPU. Using this mechanism the vCPU can execute in different address spaces depending on whether it is executing guest kernel or guest user code.

Using multiple tasks to emulate two privilege levels of a CPU has been used before[9]. With threads being the only execution abstraction, previous work had to populate each address space with at least one thread as those could not migrate between address spaces. Using multiple threads is not without pitfalls because scheduling attributes need to be found for each of them. The problem becomes more apparent if the system is to host more than one guest and each one shall be assigned a certain share of the available processor time. In contrast, in our system only one schedulable unit, the vCPU itself, is used. Subordinate address spaces act as environments but do not introduce new entities from the schedulers point of view.

5 Implementation

To validate the feasibility of the design, we implemented virtual CPUs in Fiasco.OC, the latest version of the Fiasco microkernel. Owing to its L4-ancestry, Fiasco.OC implements kernel threads, which we wanted to retain to ensure backward compatibility for workloads that do not immediately transition to virtual CPUs. Furthermore, much of the implementation of kernel threads can be reused so that we took the decision to add an additional mode of operation to threads. Switched into *vCPU mode*, they exhibit the behavior described in the design chapter before.

Fortunately Fiasco's kernel threads already have an associated memory region that is guaranteed to be resident. The user thread control block (UTCB) serves as an register extension, which, for example, allows to transfer payloads that do not fit into the register file. When running in vCPU mode, a UTCB conveniently accommodates the state indicator, the state save area and the pending event indicator.

Operations on the Event State Whereas *disable* is just a simple memory operation, *enable* is more involved. The reason for this disparity is that the enable operation has to pay attention to potentially pending, yet undelivered events that arrived while event delivery was disabled.

Event delivery can be disabled voluntarily, for example, to execute a critical section. When leaving the section, event delivery is enabled again. Events may have been arrived in the meantime. This condition is indicated by the pending flag in the vCPU state area. If the flag is set, the event enable code must perform an explicit message receive operation and handle the incoming message.

Another way event delivery gets disabled is when the vCPU is entered through its entry vector. At some later point, the execution should resume with the state that has been interrupted and saved in the state save area. Along with this operation, the event interrupt state must be set to the state from before the entry. In the general case this must be done through a system call since only the kernel is able to restore the state of the vCPU, even if no task switch is involved.

Whether control can be transfered to an arbitrary state without involving the kernel depends on the instruction set architecture of the processor and the employed application binary interface (ABI). Many RISC-like processors can restore arbitrary user states only from the kernel because only the kernel has access to special-purpose state save/restore registers. For such processors, *enable* invariably involves a system call.

[4] *ret* loads an new instruction pointer and changes the stack pointer.

The x86 architecture, though, allows for a pure user-level implementation as it features the *ret* instruction that changes two registers at once[4]. The resume operation first sets the state indicator. From that point events may arrive and disrupt the operation. Next it checks if events are marked pending. If that is the case, it clears the state indicator and returns back to the threading library, which will then invoke a syscall to have the pending event delivered. If no event is marked pending, the resumption proceeds with restoring the state of the preempted thread. This operation will load all but one general purpose register from the register state. At that point only *eip*, *esp*, and a general purpose register, say *eax*, need to be restored. To that end, *esp* is set to the value to be loaded. Then *eip* is pushed onto the stack. At that point, *eax* can be reloaded from the structure it itself points to. The operation is concluded by executing a *ret* instruction, which loads *eip* from the stack and adjusts *esp* to the expected value.

Special actions need to be taken if an event occurs after the state indicator has been set to *enabled* but before the last instruction of the sequence has completed. The vCPU entry path can detect this condition reliably as it knows about the start and end of the resume function. In that case, it does not save the preempted state, which is reported in the state save area, into the storage area maintained for the current (vCPU) thread. This is sound because at that point the user computation has not progressed so that the storage area still holds the latest state. The only modification happened on the users stacks *below* its stack pointer, which is of no concern because the ABI declares that space as undefined.

6 Linux on vCPU

For each logical Linux CPU we employ a vCPU. The events that a vCPU has to deal with are the same that a physical CPU encounters: exceptions including page faults, interrupts, and system calls.

For each user process a separate address space is used. The vCPU switches into it whenever user-level code is to be executed. Control is given to the address space in which the Linux kernel resides when an event occurs.

Fiasco features a kernel interrupt-object that allow asynchronous interaction between an event source and consumer. An event is marked pending and thus does not get lost if the receiver is not ready to consume it. Events on the interrupt object can be either triggered through messages or by device IRQs, which makes virtual and physical devices look the same. An interrupt object can be associated with a vCPU, where it raises an (vCPU) event as long as (interrupt) events are pending.

A special case is the timer interrupt, which is implemented with a thread that periodically triggers an interrupt on an interrupt object associated with the vCPU. The reason for that special treatment is that timer ticks, unlike device IRQs, are only accessible through IPC timeouts. Future versions of the Fiasco microkernel might expose timer ticks through interrupt objects as well.

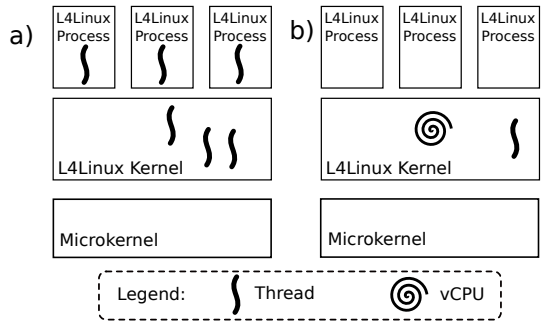

FIGURE 3: *(a) L4Linux implemented with threads and (b) L4Linux implemented with vCPUs.*

7 Evaluation

To evaluate our implementation of vCPUs and compare common operating system mechanisms with standard implementations we employed an AMD Phenom X3 8450 running at 2.1 GHz.

7.1 Exception Microbenchmarks

When a process that is under the control of a rehosted OS causes a hardware fault, then this fault transfers control to the microkernel, which, in turn, forwards it to the rehosted OS. Since faults may occur frequently, it is important that they are handled efficiently. Our benchmark compares Fiasco.OC using the vCPU and the IPC model of L4Linux. For comparison the same functionality has been modelled with Linux functionality to show the induced cost of the same operation on Linux, such as with User-Mode Linux[2] (UML).

	Linux Host	Fiasco Host	
		IPC	vCPU
Intra exception	2104	1594	870
Inter exception	7456	2433	1663
Inter PF with map	19833	3751	2833

TABLE 1: *Fault resolving methods in different environments, in CPU cycles. In* Inter *tests, the activity causing the fault and the fault handler reside in different address spaces, whereas in* intra *the use the same.*

The measured values are depicted in Figure 1. Despite involving the same number of CPU privilege transitions, fault reflection using the IPC mechanisms is slightly slower than with vCPUs. The reason is that sending an IPC is a rather complex operation because the IPC rendevouz has to be gone through, which also involves a capability lookup. Although the kernel will always find the fault handling thread ready to receive in our test that does not hold for the general case. Accordingly, the kernel has to check the validity of the capability. Furthermore, switching between threads involves some complexity with respect to scheduling. The thread switch path has to check if scheduling attributes are to be donated across the IPC operation. In comparison, fault reflection on vCPU is rather light-weight. The fault occurring in a subordinate domain can always be instantly delivered as switching into that sub-domain automatically set the state indicator flag. Moreover, from the schedulers point of view the fault reflection does not pose a relevant event, that means the scheduler is not involved at all.

When using Linux as the host system, such as with UML, handling faults for other processes is done with signal handlers and `ptrace` calls of which several are needed for a single fault. Establishing memory mappings in the faulting process must be done by forcing the faulting process to execute an `mmap`, which requires even more `ptrace` calls. In contrast, the mechanisms provided by Fiasco only require a single system call to receive and resolve the fault.

7.2 Thread switching

We implemented a basic thread library that does a purely user-level thread switch in 51 cycles. In contrary, a thread switch done through the Fiasco kernel is accounted with 335 cycles.

	User	Kernel
Thread switch	51	335

TABLE 2: *Thread switch durations, in CPU cycles.*

7.3 Compile Benchmark

A Linux kernel compile benchmark compares as seen in Figure 3. The vCPU version performs better due to the improved fault handling mechanisms, however those are not dominant.

L4Linux	Threads	vCPU
Linux compile	161.5	158.8

TABLE 3: *Linux kernel compile benchmark, in seconds, smaller is better.*

7.4 Fine-granular Timing

To keep the design simple, the threaded version of L4Linux picked up on events only under view conditions, specifically when being idle or before returning to user level. While this design choice avoided a complications regarding (L4) thread synchronisation, it made L4Linux non-preemptible in the kernel. Such a regimen is unacceptable for real-time workloads as worst-case event handling latency equals longest execution path in the kernel.

Switching the design to virtual CPUs opens up the opportunity to improve on the real-time properties of L4Linux. The adoption of vCPUs brings L4Linux more in line with the stock version, including the ability to preempt processes executing in the kernel.

We used `cyclictest` [7] to compare the event latency characteristics of L4Linux with those the mainline Linux. Both test candidates used the high precision event timer (HPET) as timer source. The results of the measurements are shown in figure 4.

L4Linux is hard pressed as the architecture puts it at a disadvantage. First, its kernel runs at user level where it cannot make use of the *global* feature for TLB entries. Such entries are not purged from the TLB during context switches and are used in Linux to keep kernel TLB entries resident. In contrast, L4Linux sustain a large number of TLB misses before it can dispatch `cyclictest`. Since `cyclictest` runs in its own address space, it will also suffer TLB misses. As such, L4Linux fares badly compared to mainline Linux which has good chances to suffer no TLB misses at all. There, `cyclictest` is likely to be the current process which obviates context switch.

Second, Fiasco does not support message-signalled interrupts (MSI) for non-PCI devices. It is quite plausible that interrupts incur additional delays if they go through the IO-APIC.

Finally, Fiasco has to mask the level-triggered interrupt of the HPET before it can acknowledge it. Masking the interrupt on the IO-APIC takes approximately 2.5μs alone. Since native Linux uses MSIs, interrupt masking is never required.

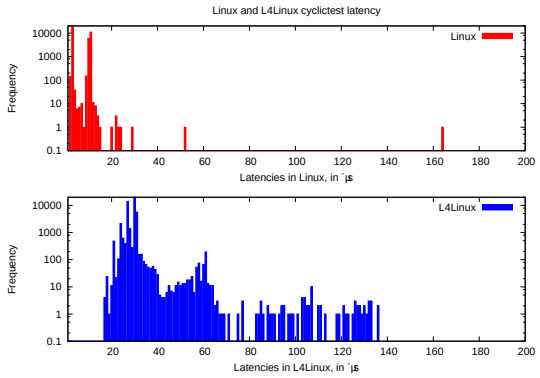

FIGURE 4: *Latencies in Linux and L4Linux running* `cyclictest`.

8 Related Work

L4Linux[9] was initially developed for the thread-based version of Fiasco. Delivering the proof that OS rehosting on microkernels can be implemented efficiently, it is nonetheless plagued with a number of issues regarding synchronization, performance and scheduling.

Scheduler activations [4] brought up the question of whether threading is the right abstraction for execution. It was observed that, at the time, neither kernel-level threads nor their user-level counterparts addressed the issue sufficiently. As a solution, an up-call mechanism more capable than traditional UNIX signals was proposed.

Psyche [15] investigated how affording threads first-class status benefits clarity, portability, and efficiency of parallel programs.

K42[11] employs an up-call based user interface that bears some resemblance to our solution. However, with a focus on scalability, K42's developer were not interested in deploying an (non-scalable) standard operating system. As such, K42 does not support switching into a subordinate protection domain. Another difference is the use of global namespaces, which contrasts with our capability based approach.

The merits of paravirtualization were most notably demonstrated by Xen[5]. As a VMM, Xen is geared towards the exclusive use of virtual machines, which invariably are rather heavy-weight. In contrast, Fiasco.OC supports heavy-weight VMs and light-weight tasks alike, giving the developer the choice how to meet his requirements best.

User-mode Linux[2] is an effort to port the Linux kernel onto itself. The difficulties encountered were numerous as the classical Unix functionality such as *ptrace*, *mmap*, and signals are not well suited to emulate a physical CPU. To mitigate these issues, the project even developed changes to the host kernel that provided functionality that otherwise could only be achieved with considerable performance impediments.

Rump[10] (Runnable Userspace Meta Programm) is an approach that allows to run a kernel components in a POSIX process for better development and debugging purposes. It is being development on the NetBSD platform.

9 Conclusion

In this paper we presented a design and implementation of a virtual processor approach that is well integrated into the microkernel infrastructure as well as it allows operating system rehosting and efficient user-level threading.

10 Acknowlegdements

We would like to thank the European Commission for their support through the Research Programme FP7 with the projects eMuCo[5] and TECOM[6].

References

[1] Qemu. URL: `http://www.qemu.org`.

[2] User-Mode Linux. URL: `http://user-mode-linux.sourceforge.net`.

[3] VMware. URL: `http://www.vmware.com`.

[4] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10:95–109, 1992.

[5] 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, 2003.

[5] `http://www.emuco.eu/`
[6] `http://www.tecom-project.eu/`

[6] R. Bhardwaj, P. Reames, R. Greenspan, V. S. Nori, and E. Ucan. A choices hypervisor on the arm architecture. Department of Computer Science, University of Illinois at Urbana-Champaign, 2006.

[7] cyclictest. URL:`https://rt.wiki.kernel.org/index.php/Cyclictest`.

[8] 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. Free Software Foundation, 59 Temple Place, Suite 330, Boston, MA 02111.

[9] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. pages 66–77.

[10] A. Kantee. Rump file systems: kernel code reborn. In *USENIX'09: Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 15–15, Berkeley, CA, USA, 2009. USENIX Association.

[11] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: building a complete operating system. *SIGOPS Oper. Syst. Rev.*, 40(4):133–145, 2006.

[12] A. Lackorzynski and A. Warg. Taming subsystems: capabilities as universal resource access control in l4. In *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, pages 25–30, New York, NY, USA, 2009. ACM.

[13] B. Leslie, C. van Schaik, and G. Heiser. Wombat: A portable user-mode Linux for embedded systems. In *Proceedings of the 6th Linux. Conf. Au, Canberra*, 2005.

[14] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser. Pre-virtualization: soft layering for virtual machines. Technical Report 2006-15, Fakultät für Informatik, Universität Karlsruhe (TH), July 2006.

[15] B. D. Marsh, M. L. Scott, T. J. Leblanc, and E. P. Markatos. First-class user-level threads. In *In Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 110–121, 1991.

[16] M. S. Miller, K.-P. Yee, and J. Shapiro. Capability Myths Demolished. Technical report, 2003.

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

[18] J. Robin and C. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor, 2000.

[19] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: a fast capability system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, New York, NY, USA, 1999. ACM.

[20] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, 2002.