Real-Time Performance of L4Linux

Adam Lackorzynski
Technische Universität Dresden
Department of Computer Science
Operating Systems Group
adam@os.inf.tu-dresden.de

Janis Danisevskis, Jan Nordholz, Michael Peter
Technische Universität Berlin
Deutsche Telekom Laboratories
Security in Telecommunications
{janis,jnordholz,peter}@sec.t-labs.tu-berlin.de

Abstract

Lately, the greatly improved real-time properties of Linux piqued the interest of the automization industry. Linux holds a lot of promise as its support by a large active developer community ensures an array of supported platforms. At the same time, automization vendors can tap into a huge reservoir of sophisticated software and skilled developers. The open nature of Linux, however, raises questions as to their resilience against attacks, particularly in installations that are connected to the internet.

While Linux has a good security record in general, security vulnerabilities have been recurring and will do so for the foreseeable future. As such, it is expedient to supplement Linux' security mechanisms with the stronger isolation afforded by virtual machines. However, virtualization introduces an additional layer in the system software stack which may impair the responsiveness of its guest operating systems.

In this paper, we show that L4Linux — an encapsulated Linux running on a microkernel — can be improved on such that it exhibits a real-time behavior that falls very close to that of the corresponding mainline Linux version. In fact, we only measured a small constant increase of the response times, which should be small enough to be negligible for most applications. Our results show that it is practically possible to run security critical tasks and control applications in dedicated virtual machines, thus greatly improving the system's resilience against attackers.

1 Introduction

Traditionally, embedded systems follow the trend set by their siblings in the desktop and server arena. Rapid technological advances allow for ever more functionality. At the downside, the growing complexity poses a risk as software defects may badly impair the expected machine behavior. This risk is particularly threatening for embedded systems as these systems often have to meet stringent safety requirements.

Another trend carrying over from desktop and servers is the use of open source software, particularly Linux. The adoption of Linux is remarkable insofar as it started out as a hobbyist's system aimed at desktops. Although neither security nor scalability were considered in the beginning, Linux came to address either of them quite well, making it hugely popular as server operating system. Linux also got a significant foothold in the embedded market. Apart from the absence of licensing costs, its availability for a host of systems holds a lot of appeal for device manufacturers. Having ports of Linux for many platforms is due in no small part to an active community, which allows for improvements to circulate as open source. A third contributing factor for Linux' popularity is its maturity. For example, as embedded systems embrace network connectivity, they can draw on a mature network stack and a variety of complementing user level software. Originally not being a real-time operating system, the tireless effort of the community managed to evolve Linux to the point where it can run applications with real-time requirements. With that ability, Linux is set to make inroads into markets that used to be the preserve of dedicated real-time operating systems, most of them proprietary.

With regard to security, the operating system is of particular importance. It falls to it to keep applications running on them in check. Yet, prevalent operating systems have a rather poor record when it comes to isolation. The first problem is that they do poorly when it comes to the principle of least

authority. Some applications need special privileges to fulfill their task. Unfortunately, the granularity with which these privileges can be granted is too coarse, leaving many applications running with excessive rights. If such an application succumbs to an attacker, then he can wield that right for its malicious purposes. The point is corroborated by the observation that exploiting applications running with the `root` identity are in high numbers. However, today's operating systems cannot be easily phased out because porting their large number of applications and device drivers is a laborious endeavor with uncertain prospects of success.

Confronted with the wanting protection of operating systems, designers of security-critical systems have long turned to physical separation, that is, placing each application on its own machine. While ensuring excellent separation, this approach is not easily applicable to use cases where budget, space, and power limitations have to be met. That is where operating system encapsulation comes into play. While virtualization is currently the most prominent encapsulation technique, it is neither the only one nor the one best suited for all use cases. In the absence of hardware virtualization support, other approaches like OS rehosting might be the better choice.

While virtualization is a well-studied subject, we are not aware of research covering real-time operating systems in virtual machines. In this paper, we set out to investigate whether the real-time behavior of Linux can be preserved if it is executed in a container. Our results indicate that such a setup is feasible if moderately longer event response times can be tolerated.

We will proceed with a brief summary of OS encapsulation use cases followed by a description of L4Linux, the system we build on. The design chapter follows up with a discussion of the problems we faced when deriving an L4Linux version from a Linux with the `PREEMPT_RT` patch applied. To evaluate the real-time performance of L4Linux, we conducted a number of experiments. The results will be presented before we discuss related work and conclude.

2 Mitigating OS Shortcomings

In the past, monolithic operating system kernels have been perceived as an obstacle for the evolvement of secure, robust, and flexible systems. The underlying concern was that their growing complexity would inevitably introduce software defects. If such a defect triggers a fault, the consequences are most likely fatal and will crash the system, which is unacceptable for safety-critical systems.

As an alternative solution, the research community put forth designs based on a minimalistic kernel which is small enough to be thoroughly inspected. Most of the functionality that used to be part of monolithic kernels (device drivers, protocol stacks) would be moved into user level tasks. The rationale was that such an architecture would allow for easy customization by replacing or duplicating the relevant user level components.

As device driver development had been known to be the weak spot for any new system, there was the need to reuse as many device drivers as possible. That raised the question of how to provide the execution environment they were written for. The original vision was that monolithic kernels would be decomposed into small components, each executing in its own protection domain. Since faults would be contained, many system architect nourished the hope that the resulting system would be superior in terms of flexibility, fault tolerance, and security. This endeavor, though, proved complicated. Plans to move to a system completely devoid of legacy operating system code have not materialized so far and it is dubious if they will in the foreseeable future.

As set out above, the insufficiencies of prevalent operating systems pose a risk that may be unacceptable for security- and safety-critical applications. At the same time, they cannot be completely retired as many applications and device drivers depend on them. One solution might be to deploy multiple of them each dedicated to a specific task and ensure that no unsanctioned interference between them occurs. The most frequently cited arguments are the following:

Security. Embedded systems are ever more often connected to the internet whereby the attack surface is drastically enlarged. Given the checkered security record of existing operating system, it seems prudent to place exposed functionality in dedicated compartments which provide an additional line of defense.

Consolidation. High end embedded systems often employ multiple electronic control units. Not only does such a setup incur high material cost, each unit has also to be wired and adds to the overall energy consumption. Consolidating multiple of them onto one unit can yield significant cost, weight, and energy savings.

Certification. So far, safety-certified components could not be deployed alongside noncertified

ones on the same controller necessitating physical separation which incurs costs. A certifiable kernel holds the promise of providing isolation strong enough so that a coexistence on one controller becomes viable.

Development. The diversity of current software may be difficult to leverage if the development is constrained to one operating system. It would be much better if the respective class-leading solutions could be used while preserving the investments in existing software assets.

3 Design

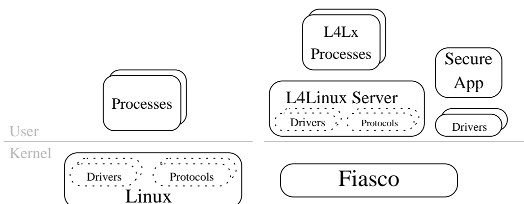

FIGURE 1: *Left: Linux running directly on a machine. Right: Linux running on top of Fiasco, an L4 kernel.*

Our design is based on L4Linux[?], a port of the Linux kernel to the L4[5] microkernel. When it was initially developed it set the standard regarding performance for encapsulated operating systems. Ever since it has evolved alongside with Fiasco as the latter improved on its security features (capability-based access control) and became available for multiprocessors and multiple architectures (IA32, AMD64, ARM).

Unlike previous versions of L4Linux, which relied on L4 threads, the current vCPU-based execution model[1] is very similar to the environment found on physical machines. As such, any improvement of the mainline Linux kernel with respect to interrupt latencies should in principle also be available in the corresponding version of L4Linux. The overhead due to the additional context switches and privilege transitions will have an impact but should lengthen the critical paths only by a constant delay.

That said, the previous versions of L4Linux were designed with an emphasis on throughput performance. Design decisions that yield good performance results might however prove problematic for real-time performance. In fact, we came up against three problems of that kind:

Timer. The current version of Fiasco uses a 1 KHz clock to drive its scheduler. Accordingly, timeouts are limited to that granularity as well, which is insufficient for high-resolution timers.

Communication. L4Linux operates in an environment where some services are provided by other tasks. Requesting them involves the platform communication means, in case of Fiasco primarily synchronous IPC. Waiting for an outstanding reply may delay the dispatch of an incoming event.

Memory virtualization. For security reasons, L4Linux cannot manage its page tables directly but has to use L4 tasks instead. It turned out that some applied performance optimizations have an adverse effect on responsiveness.

We will touch on each of these points in the following sections.

3.1 Communication

In our architecture, L4Linux draws on services provided by other tasks. Depending on the use case, communication may either by synchronous or asynchronous. Synchronous IPC has performance advantages for short operations, while asynchronous communication can better hide latencies.

An example where synchronous IPC was chosen is the graphics server. L4Linux maintains a shadow framebuffer and notifies the graphics servers when changes have occured. The graphics server, which has also access to the shadow framebuffer, then updates the device framebuffer taking into account the current visibility situation (as multiple L4Linux instances can have windows at various positions).

Considering performance, such an arrangement is reasonable. However, whenever a screen update is in progress, the L4Linux main thread cannot pick up on incoming events as it waits for the reply from the graphics server. While this situation is hardly noticeable in performance measurements, latency-sensitive applications are hurt badly. Our solution involves a second thread relieving the main thread from engaging directly in IPC. The two threads themselves interact through a ringbuffer in shared memory and use asynchronous notification. As a result, the main thread is never tied up and can always promptly respond to incoming events.

3.2 Timer

Unlike other peripheral devices, timers are not directly accessible by user level components. Where timeouts or delays are required they are typically implemented using IPC timeouts. IPC timeouts as implemented by the Fiasco microkernel have timer tick granularity—typically with a one millisecond period.

As this granularity is too coarse to use it as a timer source for a high-resolution timer, we decided to ditch the periodic IPC-timeout based timer. Instead, L4Linux was granted direct access to the HPET device, which can be used to trigger timer events with a high resolution.

3.3 Memory Virtualization

As a user-level task, L4Linux has no direct access to the page tables, which are under the exclusive control of the microkernel. To isolate its processes, L4Linux makes use of L4 tasks. L4 allows two tasks to share memory and provides the original owner with the means to later revoke that sharing. L4Linux uses that memory sharing facility to provision its processes with memory. Whenever Linux modifies the pagetable of one of its processes, this change is reflected in a corresponding change in the memory configuration in the process' task.

While under normal operations page table updates are propagated individually, the destruction of a process is handled differently. To avoid the overhead of a microkernel syscall for each single page table invalidation, the destruction is performed by a single system call. The destruction of a task address space requires the microkernel to iterate the page directory and return page table to its internal memory pools. Although task destruction can be preempted, it will not be aborted once started. As such, it is fully added to worst-case times of timing critical paths in L4Linux.

We added a thread to L4Linux, which disposes of tasks of perished processes. Not longer executing long-latency syscalls, the main L4Linux thread remains responsive to incoming events.

4 Evaluation

To evaluate our design we conducted a number of experiments. Our test machine contained a 2.7 Ghz Athlon 64 X2 5200+ processor, an nVidia-MCP78-based motherboard and 4 GB RAM. In order to increase the magnitude of the differences between our test scenarios, we reduced the memory available to the OS to 128 MB and disabled the second core of the CPU.

As base OS version we chose the recent Linux version 3.0.3 and the corresponding release of L4Linux. We then took Thomas Gleixner's Linux-RT patches and applied them to both code bases. L4Linux was supplemented with the current version of Fiasco and a minimal L4 runtime environment, which consisted of a framebuffer and console driver. Both resulting setups were finally booted directly from Grub.

We chose the following software components as our test suite:

- `cyclictest` is a simple benchmark to measure the accuracy of OS sleep primitives. The tool is part of the `kernel.org` "rt-tests" benchmark suite [1]. To achieve accurate results, we executed it at the highest realtime priority and chose `clock_nanosleep()` as sleep function, as it allows an absolute specification of the wakeup time and thus ignores the time spent setting up the actual timer event.

- `hackbench` transmits small messages between a fixed set of processes and thus exerts stress on the OS scheduler. Its current development version is also hosted through git [2].

- Finally, the compilation of a Linux kernel both causes a considerable amount of harddisk interrupt activity and creates and destroys a large number of processes. We used a standard 2.6 series Linux source tree and the default i386 configuration as baseline.

4.1 Throughput

To get a feeling for the setup we were about to experiment with, we started out with some throughput measurements. While these are by no means representative due to the restrictions to very little memory and only one core, they serve as a good starting point and already hint at some of the expected results.

As every switch between a userspace task and the L4Linux server involves a round-trip into the microkernel and a switch to a different address space, L4Linux suffers from frequent TLB and cache misses.

[1] RT-Tests Repository: git://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git
[2] Hackbench Repository: https://github.com/kosaki/hackbench

To highlight the effect of this disadvantage, we created an "intermediate" version of native Linux without support for global pages, large pages and with explicit TLB flushes on every kernel entry and exit path.

The `hackbench` benchmark (cf. Fig. 2) shows the stripped-down version of Linux almost halfway between native Linux and L4Linux, which demonstrates that the impact of repeated cache misses is quite severe when tasks are rapidly rescheduled. The compilation of a Linux kernel (cf. Fig. 3) displays as expected only a mild slowdown for the non-caching Linux variant.

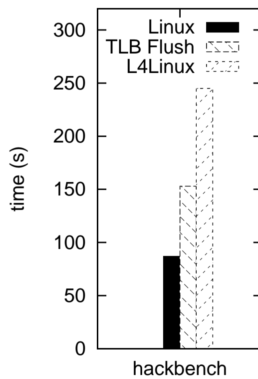

FIGURE 2: *Runtime of* `hackbench` *(40 processes, 100'000 iterations).*

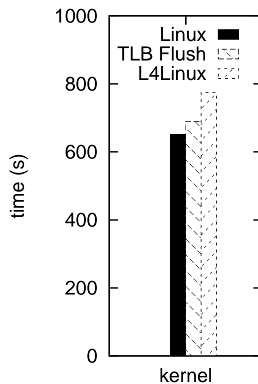

FIGURE 3: *Duration of* `kernel compilation`.

4.2 Memory Management

Another possible source of large latencies are long-running non-abortable system calls. One particularly long operation especially in the Fiasco-based setup is the destruction of an address space, as this requires assistance from the microkernel. To see this problem in effect, we created two latency histograms with `cyclictest` under concurrent operation of the `hackbench` benchmark (cf. Fig. 4) and a kernel compilation (cf. Fig. 5), respectively. While the maxima of both histograms are in the expected order of magnitude, the former shows outliers up to $100\mu s$ and the latter (due to its constant destruction of processes) even an almost constant distribution of "long" latencies beyond $40\mu s$.

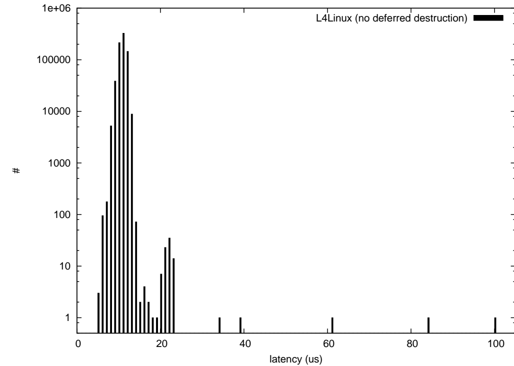

FIGURE 4: *Latencies measured with* `hackbench` *as load.*

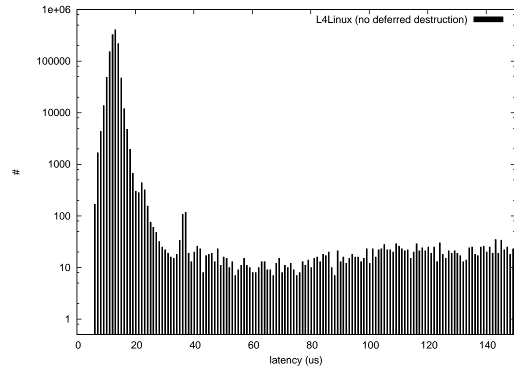

FIGURE 5: *Latencies measured with* `hackbench` *as load.*

As outlined in section 3.3, we therefore externalized the destruction of address spaces to a separate L4 thread. While the execution context issuing the destruction request still waits for the destruction to complete, L4Linux as a whole is then interruptible during the wait and can react to external events.

4.3 Latency under Load

Taking all these findings into consideration, we finally compared native Linux and our improved L4Linux implementation directly using our established benchmark combinations

cyclictest/hackbench and cyclictest/kernel-compile. The results are shown in Fig. 6 and Fig. 7.

The first obvious result is that the L4Linux distribution is offset from the native distribution by $5\mu s$. The explanation for this is pretty simple. As L4Linux is not allowed to receive interrupts directly, the microkernel must first handle the interrupt, determine the recipient attached to it and pass the interrupt on. This operation induces a very constant amount of overhead – we measured a delay of $3.65\mu s$ in our setup for the delivery of an edge-triggered interrupt. Handling level-triggered interrupts requires even more time, as L4Linux has to issue an extra syscall to unmask the interrupt line once it is done with the device in these cases.

As our timer is edge-triggered, there remains a delay of about $1.3\mu s$. We attribute this to the overhead induced by the microkernel syscalls involved in switching to the real-time task (restoring global descriptor table and execution state) as well as to the aforementioned additional address space switch.

Both distributions reach their maximum not immediately, which means that the ideal interrupt delivery path is not the most frequent. This effect is likely linked to execution paths during which Linux (just as L4Linux) have disabled interrupt reception. The delay incurred by this deferred delivery is more pronounced in the rehosted setup, because deferring the delivery causes additional round-trips to the microkernel once L4Linux has enabled interrupt reception again. Overall though, the difference is not exceptionally large.

Finally, both setups exhibit outliers well beyond the main part of their respective distribution. With hackbench as load-generating benchmark, these show the usual difference between native and rehosted and are therefore no specific effect of an implementation detail of Fiasco or L4Linux. The interrupt-heavy kernel compilation on the other hand demonstrates that additional interrupts (mostly generated by the hard drive controller) affect L4Linux much harder than native Linux due to the interrupt delivery overhead. This effect is even worsened by the fact that the cyclictest benchmark has no way to atomically determine the clock overrun once it is awoken from its sleep syscall: interrupts hitting between the sleep and the time lookup are accounted for with their full duration.

L4Linux even has to deal with another interrupt source which is not present in the native scenario: Fiasco employs the on-board PIT for its own scheduling needs and configures it to generate interrupts in regular intervals.

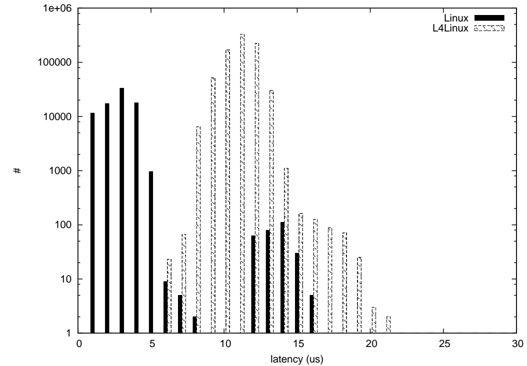

FIGURE 6: *Latencies measured with* hackbench *as load.*

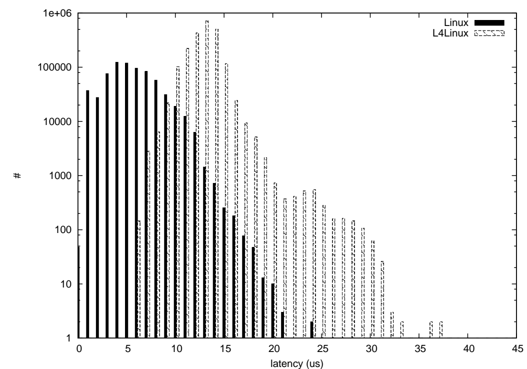

FIGURE 7: *Latencies measured with kernel compile as load.*

5 Related Work

Virtualization shares many goals with OS encapsulation, it can be even viewed as an alternative implementation. For a long time, the adoption of virtual machines was hampered by the lack of hardware support in processors. The construction of VMs that are efficient, equivalent, and properly controlled requires the instruction set architecture to meet certain requirements[6]. Unfortunately, many widely used processors, in particular those of IA32 provenance[7], do not, and among those which do meet the basic requirements are again many which lack support for virtualization of memory management structures. While this deficiency does not bar virtualization on these processors, performance is indeed severely hampered, as operations on a guest's page tables require extensive help by the hypervisor[2]. The performance degradation usually cancels the benefits provided by the comfortable hardware virtualization interface, so OS encapsulation remains a viable solution on these platforms.

Virtualization support has also been announced for ARM processors. It remains to be seen how long it takes until processors without it are fully supplanted.

Xen[3] provides for running multiple operating systems on one physical machine. As with L4Linux, guests have to be modified if the underlying machine does not provide virtualization support. Unlike Fiasco, Xen does not provide light-weight task and IPC. Although Xen was ported to the ARM architecture, this port has not found the echo of its siblings on the desktop or server.

Virtualization and microkernel architecture do not rule each other out, they are rather complementary[8, 9]. While thorough research of worst case execution times of mircokernels themselves exists[4], so far, neither of the encapsulation approaches has been examined as to their applicability for real-time applications.

6 Conclusion

In this paper we investigated the real-time execution characteristics of an encapsulated operating system. Our results indicted that run-time overhead is incurred in timing-critical paths. However, these overhead is in the order of the latency experienced with native Linux. As such, L4Linux is suitable for use cases where the security of real-time applications shall be bolstered by deploying them in dedicated OS capsules.

Future work will concentrate on reducing the current overheads. We see good chances that the influence of large overhead contributors such as the early timer expiration can be mitigated, e.g. by delaying the timer and rearming it until after the real-time task has finished executing.

Our choice of using OS rehosting as encapsulation technology was mainly motivated by its applicability to a host of processors currently deployed in embedded systems. As hardware virtualization support finds its way into more and more processors, the question comes up whether the software stack underneath a virtual machine can be designed in such a way that it allows for real-time operations in the virtual machine.

References

[1] A. W. Adam Lackorzynski and M. Peter. Generic Virtualization with Virtual Processors. In *Proceedings of the Twelfth Real-Time Linux Workshop, Nairobi*, 2010.

[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, 2003.

[4] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. Timing analysis of a protected operating system kernel. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, Vienna, Austria, Nov 2011.

[5] J. Liedtke. On micro-kernel construction. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

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

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

[8] A. L. Steffen Liebergeld, Michael Peter. Towards Modular Security-Conscious Virtual Machines. In *Proceedings of the Twelfth Real-Time Linux Workshop, Nairobi*, 2010.

[9] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 209–222, New York, NY, USA, 2010. ACM.