RTLinux with Address Spaces

Frank Mehnert **Michael Hohmuth** **Sebastian Schönberg** **Hermann Härtig**
Dresden University of Technology
Department of Computer Science,
D-01062 Dresden, Germany
drops@inf.tu-dresden.de

Abstract

The combination of a real-time executive and an off-the-shelf time-sharing operating system has the potential of providing both predictability and the comfort of a large application base. To isolate the real-time section from a significant class of faults in the (ever-growing) time-sharing operating system, address spaces can be used to encapsulate the time-sharing subsystem. However, in practice designers seldomly use address spaces for this purpose, fearing that extra cost induced thereby limits the system's predictability.

To analyze this cost, we compared in detail two systems with almost identical interfaces—both are a combination of the Linux operating system and a small real-time executive. Our analysis revealed that for interrupt-response times, the delay and jitter caused by address spaces are similar to or even smaller than those caused by caches and blocked interrupts. As a side effect of our analysis, we observed that published figures on predictability must be carefully checked whether or not such hardware features are included in the analysis.

1 Introduction

In this paper, we determine the cost of introducing address-space protection to RTLinux [8].

RTLinux is a real-time extension for Linux. It introduces a small real-time executive layer into the time-sharing Linux kernel. On top of that layer, the time-sharing kernel and the real-time applications all run in kernel mode. User mode is reserved for time-sharing applications. Real-time applications are protected from errors in time-sharing applications.

For our evaluation, we used L4RTL, a reimplementation of the RTLinux API based on a real-time microkernel and a user-level Linux server. That design has the property that the real-time subsystem is protected from a large class of errors: Crashes in the time-sharing subsystem (either in the operating system or in user code) will not bring down the real-time subsystem, except if a device driven by the time-sharing subsystem locks up the machine or corrupts main memory.

This increased level of fault tolerance is desirable for many real-time applications. In many ways, it is more important to protect the real-time subsystem from the time-sharing subsystem (both kernel and applications) than the other way round: For example, a real-time application may be safety-critical, but time-sharing operating systems have become so big and bloated that it is difficult to trust in their stability. However, the undoubtable benefits of separate address spaces do not come for free.

We observed RTLinux' worst-case response time to be much higher than "about 15 μs on a generic x86 PC" claimed by the system's authors [7]. In our experiments, RTLinux' worst-case interrupt latency was **68** μ**s**. The worst case for L4RTL was **85** μ**s**.[1]

We found that the cost induced by address-space switches to real-time applications does not significantly distort the predictability of the system. In general, most of the worst-case overhead we observed must be attributed to implementation artifacts of the microkernel we used, not to the use of address spaces.

2 L4RTL

For an accurate comparison of the cost of address spaces in real-time systems, we have reimplemented

[1] We carried out our measurements on a 800-MHz Pentium-III PC—more details in Section 3. We believe that this system qualifies as a "generic x86 PC" as presumed by the RTLinux statement.

the RTLinux API in the context of the DROPS system. The resulting system, called L4RTL, can run unmodified RTLinux programs (source-level compatibility).

2.1 The DROPS system

DROPS is an operating system that supports applications with real-time and quality-of-service requirements as well as non-real-time (time-sharing) applications [2]. It uses the Fiasco microkernel as its base. The Fiasco microkernel is a fully preemptible real-time kernel supporting hard priorities. It uses non-blocking synchronization for its kernel objects. This ensures that runnable high-priority threads never block waiting for lower-priority threads or the kernel [5].
For time-sharing applications, DROPS comes with L^{4}Linux, a Linux server that runs as an application program on top of the Fiasco microkernel [3]. L^{4}Linux supports standard, unmodified Linux programs (binary compatibility). The Linux server never disables interrupts for synchronization purposes. Instead, we modified the implementations of `cli()` and `sti()` to work without disabling interrupts while still ensuring their semantics (to protect a critical section) [4].

2.2 L4RTL implementation

We implemented L4RTL as a library for RTLinux application programs and a dynamic load module for L^{4}Linux. Figure 1 gives an overview of L4RTL's structure.

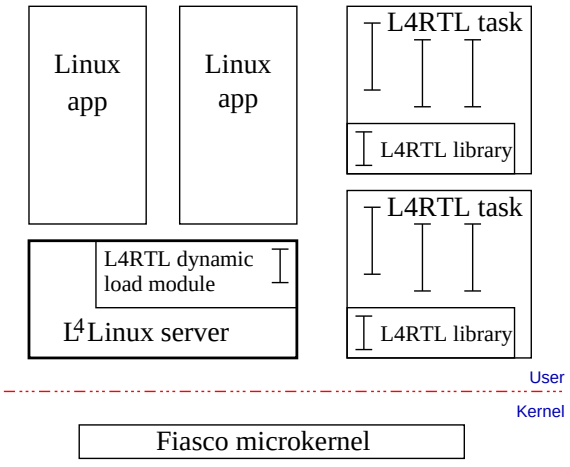


FIGURE 1: *L4RTL structure*

The *L4RTL library* implements the RTLinux API for real-time RTLinux applications. RTLinux applications run as user-mode threads in address spaces separate from L^{4}Linux's address space. We call these threads and address spaces *L4RTL threads* and *L4RTL tasks*, respectively. Each L4RTL task can contain several L4RTL threads, and these threads can cooperate using the RTLinux API.
There can be more than one L4RTL task. All of these tasks can use the same single L^{4}Linux server. However, L4RTL currently does not allow real-time threads in different tasks to communicate with each other using the RTLinux API.
The *L4RTL dynamic load module*[2] for L^{4}Linux implements the API for use by Linux programs to communicate with L4RTL threads. It also creates a service thread in the L^{4}Linux task. L4RTL threads use this thread as their only connection to L^{4}Linux, and only for communication with time-sharing Linux processes. Otherwise, L4RTL tasks can work completely independent from L^{4}Linux.
FIFOs and Mbuffs are implemented using shared-memory regions between L4RTL threads and the L4RTL load module. L4RTL threads allocate these memory regions and then transfer a mapping of these to the load module's service thread inside L^{4}Linux using microkernel IPC. The shared-memory regions contain all necessary control data. Once a L4RTL thread and the load module have established the mapping, they only communicate using their shared-memory region. L4RTL has been designed so that bugs in L^{4}Linux that corrupt the shared-memory data cannot crash L4RTL threads.
More Fiasco-microkernel IPC is necessary only for FIFO signalling. For this purpose, the L4RTL library creates one thread in each L4RTL task (Figure 1). This thread handles signal messages from the L4RTL load module and forwards them to L4RTL threads.
In contrast to RTLinux, L4RTL does not contain a scheduler. Instead, it relies on the Fiasco microkernel for scheduling.
L4RTL is the subject of ongoing work and research. Currently, it does not implement all of the very rich RTLinux API. However, everything relevant to the discussion in this paper (and more) has been implemented and works well, and we believe that the missing features have no influence on our measurements.

3 Cost of address spaces

Using original RTLinux and L4RTL, we ran a minimal interrupt-latency benchmark under worst-case conditions. This experiment was meant to establish an *upper bound* for the overhead that can be expected from providing address spaces.

[2] From L^{4}Linux's point of view, this is a "Linux kernel module," but of course L^{4}Linux runs as a user program and not in kernel mode.

3.1 Experimental setup

To induce worst-case system behavior, we have used two strategies.

First, prior to triggering the hardware event, we configure the system such that the kernel's and the real-time thread's cache and TLB working sets they need to react to the event are completely swapped out to main memory (and the corresponding 1st-level and 2nd-level cache lines are dirty). For this purpose we have written a Linux program that invalidates the caches and TLB entries (cache flooder)

Second, we exercise various code paths in RTLinux, L^{4}Linux, and the Fiasco microkernel. These coverage tests are a probabilistic way to reveal code paths with maximal execution time while interrupts are disabled. Additionally, they increase confidence that the DROPS system is indeed completely preemptible as we claimed in Section 2.1. To avoid missing critical code paths because of pathologic timer synchronization, we varied the time between triggering two interrupts.

For code coverage we use a benchmarking suite for UNIX, hbench [1]. This benchmark provides excellent coverage for Linux and, in our experience, also for L^{4}Linux and the Fiasco microkernel.

As a reliable and measurable interrupt source, we have used the x86 CPU's built-in interrupt controller (Local APIC[3]). This unit offers a timer interrupt that can be used to obtain the time between the hardware event and the reaction in kernel or user code. When the Local APIC is used in periodic mode, its overhead is close to zero because first, it does not require repeated reinitialization, and second, the elapsed time since the hardware trigger can be read directly from the chip.

We used RTLinux version 3.0 with Linux 2.2.18 in non-SMP mode. In this mode, the Local APIC is not used for system purposes. Linux has the *bigphysarea* patch applied to allow allocation of contiguous physiscal memory pages we need for our cache flooder. The version of L^{4}Linux we used is 2.2.18.

3.2 Measurements

For both RTLinux and L4RTL, we measured the exact time between the occurrence of the hardware event and the first instruction in the real-time thread. We measured this time under two conditions: Average case (no additional system load) and under a combination of the hbench and cache-flooding loads. Additionally, we measured the time between the occurrence of the hardware event and the first instruction of the kernel-level interrupt handler ("kernel entry"). This measurement was intended to quantify the effect of critical code sections that disable interrupts within the kernels[4]. By measuring both kernel-level and real-time–thread latencies, we were able to filter out overhead not induced by introducing address spaces but by artifacts of the kernels' implementations.

The diagrams in Figures 2 and 3 show the densities of the interrupt response times under the two load conditions. Table 1 gives an summarization of the worst-case measurement results.

	kernel entry	*kernel path*	*user entry*
L4RTL	53 μs	39 μs	**85 μs**
RTLinux	53 μs	56 μs	**68 μs**

TABLE 1: *Worst-case interrupt execution times on L4RTL and RTLinux*

The *maximal time* to invoke the Fiasco microkernel's handler was 53 μs, and the maximal time to start the corresponding L4RTL thread was 39 μs. As the sum of these separate maximums is close to the total maximum (86 μs), it seems that the code path that disables interrupts and the code path that starts the L4RTL thread use different code and data and are accounted for separately. We believe that the kernel-entry cost can be attributed to deficiencies in our microkernel's preemptability.

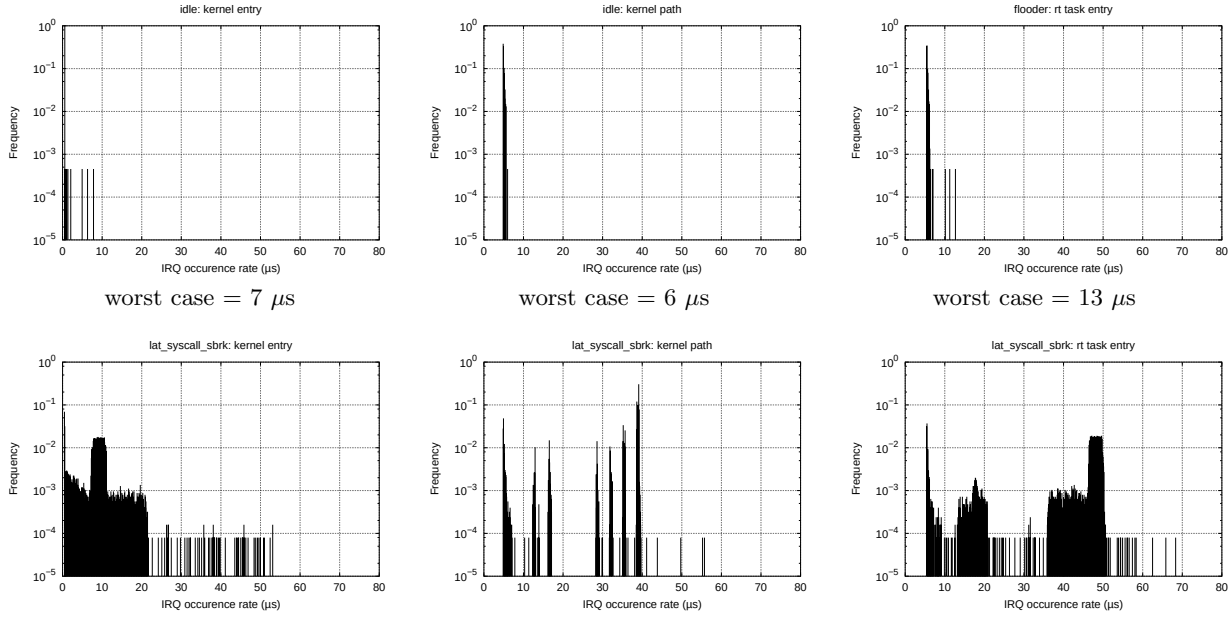
Perhaps the most interesting result is that the worst-case latency for RTLinux is 68 μs. RTLinux achieves a higher level of kernel interruptibility than the Fiasco microkernel. This fact leads to better worst-case response times in our code-coverage tests—39 μs versus 56 μs. However, RTLinux seems to have long periods of disabled interrupts, too. It takes up to 53 μs to invoke a kernel handler. The fact that this value is close to the total maximum of 68 μs suggests that the code that keeps interrupts disabled is close or identical to the code that is executed when an interrupt occurs (Figure 2, bottom row). In other words, it is probably RTLinux itself that impairs the system's interruptibility.

All in all, our measurements show that unrelated limitations in our microkernel, namely its disabling of interrupts for up to 53 μs, was more problematic for L4RTL's real-time performance than the introduction of address spaces for real-time tasks into the system.

We deduce that providing address spaces for real-time tasks does not lead to unacceptable worst-case scheduling delays. Moreover, the introduction of address spaces introduces less uncertainty than blocked interrupts and caches.

[3] APIC = advanced programmable interrupt controller

[4] L^{4}Linux is not an issue here, because it never disables interrupts for synchronization.

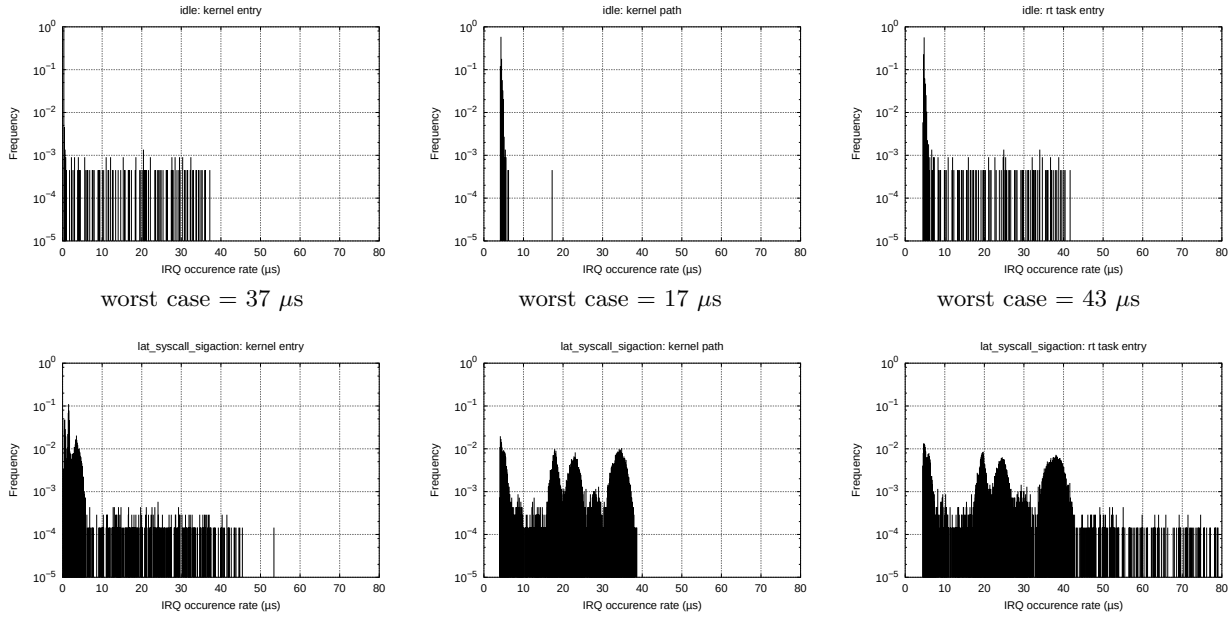


worst case = 7 μs | worst case = 6 μs | worst case = 13 μs

worst case = 53 μs | worst case = 56 μs | worst case = 68 μs

Legend: X axis shows interrupt latency. Y axis shows density of occurrence of particular latencies. Note that the Y axis has a logarithmic scale.

FIGURE 2: *RTLinux performance under no load (first row) and hbench + cache flooder combined (second row). Left: Time to enter kernel mode. Center: Time to activate handler in RTLinux thread. Right: Accumulated, total time*



worst case = 37 μs | worst case = 17 μs | worst case = 43 μs

worst case = 53 μs | worst case = 39 μs | worst case = 85 μs

Legend: X and Y axes have the same meaning as in Figure 2

FIGURE 3: *L4RTL performance under no load (first row) and hbench + cache flooder combined (second row). Left: Time to enter kernel mode. Center: Time to activate handler in RTLinux thread. Right: Accumulated, total time*

4 Conclusion and future work

In this paper, we have compared two Linux-based real-time operating systems: RTLinux, a shared-space system, and L4RTL, a separate-space system.

We learned that address spaces, when provided by a small real-time executive and used to protect critical real-time tasks from a shaky time-sharing subsystem, do not come for free. They increase worst-case response times, adding delays and jitter.

However, the good news is that the additional overhead in worst-case situations is comparable to costs introduced by blocked interrupts and by common hardware features of modern CPUs, such as caches. These costs seem to be well-accepted by designers.

So far, we have only claimed that separate address spaces are effective means to isolate the real-time section from faulty time-sharing subsystems. As future work, we plan to test this claim by injecting faults, for example arbitrary memory accesses, to the Linux section in kernel and user space and to add a rebooting facility for the time-sharing subsystem.

To improve worst-case execution times, we plan to extend the Fiasco microkernel with a facility for emulating tagged TLBs ("small address spaces" [6]) and to apply a user-level memory-management server that provides cache partitioning.

References

[1] A. B. Brown and M. I. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–224, Seattle, WA, June 1997.

[2] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[3] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.

[4] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*, Adelaide, Australia, September 1998.

[5] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference*, Boston, MA, June 2001.

[6] J. Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Arbeitspapiere der GMD No. 933, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1995.

[7] RTLinux FAQ. URL: `http://www.rtlinux.org/documents/faq.html`.

[8] Victor Yodaiken and Michael Barabanov. A Real-Time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, Anaheim, CA, January 1997. The USENIX Association.