Cost and benefit of separate address spaces
in real-time operating systems

Frank Mehnert   Michael Hohmuth   Hermann Härtig

Dresden University of Technology
Department of Computer Science
e-mail: drops@os.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.

This paper is a follow-up of an earlier publication at the 3rd Real-Time Linux workshop [17]. It is different to that paper in that we have further optimized our microkernel and examined more hardware.

1 Introduction

Hybrid operating systems—systems with both a real-time and a time-sharing subsystem—have two interesting benefits. First, they offer a comfortable, well-known interface with lots of existing applications. Second, they are able to run real-time and standard non-real-time applications at the same time on the same machine without impairing the predictability of the real-time applications.

Existing hybrid systems differ in the degree of separation between the real-time and the time-sharing subsystem.
They can be categorized into the following three categories.

Shared-space systems introduce a small real-time executive layer into the time-sharing 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. In this architecture, real-time applications are protected from errors in time-sharing applications.

Shared-kernel systems, like shared-space systems, embed a real-time executive in the time-sharing kernel. However, real-time applications run in user mode. This has the additional benefit that the time-sharing system can be protected from errors in real-time applications.

Separate-space systems move the whole time-sharing subsystem to user mode. In these systems, only a small real-time kernel runs in kernel mode. Real-time applications, time-sharing applications, and the time-sharing operating-system "kernel" run in separate address spaces in user mode. Separate-space systems have 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 corrupt the real-time subsystem, except if a device driven by the time-sharing subsystem locks up the machine or corrupts main memory.

The increased level of fault tolerance of separate-space systems 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 their stability. However, the undoubtable benefits of separate address spaces do not come for free.

In this paper, we determine the cost of separate-space systems relative to that of shared-space systems. We compare these particular two types of systems because we feel that separate-space systems lend themselves for the largest number of applications, and we expect the largest performance overhead relative to shared-space systems.

For our evaluation, we used RTLinux [27] from the

shared-space–systems category, and L4RTL, a reimplementation of the RTLinux API as a separate-space system based on a real-time microkernel and a user-level Linux server. We chose these systems because they are easily comparable—both are based on the Linux kernel—and because we had available the source code of these systems. Also, we knew from earlier experiments that RTLinux has excellent real-time–scheduling properties [16, 17].

We concentrated on predictability in worst-case situations rather than writing yet another instance of the notorious "who has the fastest IPC implementation" type of papers. For example, while the "fastest IPC" papers in general tried to secure optimal environments for their mechanisms, like warm caches of the communicating processes [12, 19], we looked at the predictability of base mechanisms under conditions such as flooded caches.

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.

The remainder of this paper is organized as follows. In Section 2, we consider related work. Section 3 introduces RTLinux and L4RTL, the two systems we have compared. In Section 4, we describe in detail our experiments and what we have learned. We conclude the paper in Section 6 with a summary and an outlook on future work.

2   Related work

There is a large body of work on microbenchmarking IPC or interrupt latencies (see for example [1, 12, 19]) and many worst-case evaluations for real-time operating systems (e. g., [23]). However, we are unaware of any work that specifically addresses the worst-case overhead of using address spaces in real-time systems.

In this paper, we concentrate on hybrid real-time and time-sharing operating systems:

KURT [21] and Linux/RK [18] are real-time extensions of the Linux kernel. These systems target real-time applications that make use of UNIX system services and that may run in separate address spaces (i. e., they are shared-kernel system). Additionally, KURT can also operate as a shared-space system. As both systems try to keep the amount of changes to Linux as small as possible, their real-time CPU-scheduling accuracy is limited by non-real-time parts of the Linux kernel, for example device drivers that disable interrupts for synchronization.

To our knowledge, RT-Mach [24] and LynxOS [20] also both belong to the shared-kernel system category. These systems have extended existing systems with real-time mechanisms. In RT-Mach's case, one could argue that it

is a separate-space system because it runs operating-system servers as user-level tasks. However, Mach (from which RT-Mach has evolved) still is a "fat kernel" because it is designed to contain everything that the designer of a user-level operating-system server could find useful, including device drivers and paging policies.

We have not used these shared-kernel systems in our study for two reasons. First, separate-space systems lend themselves to a superset of the possible applications of shared-kernel systems. Second, we believe that separate-space systems inherently have a higher overhead than shared-kernel systems when compared to shared-space systems. Therefore, our results can be meaningfully interpreted for shared-kernel systems as well: They represent an upper bound on the cost for shared-kernel systems.

We know of few systems in the separate-space systems category. Some, like QNX [8] and OnCore OS, have a real-time kernel and come with a set of non-real-time operating-system servers and programs.

TenAsys INtime [22] is a real-time extension of Windows NT and Windows 2000. While both the Windows-NT kernel and the INtime real-time executive run in kernel mode, they are physically protected from each other using the x86 CPU's task-switching mechanism [23]. It is not clear to us which level of fault tolerance this architecture provides.

We have not considered these separate-space systems because no comparable shared-space system exists and because we did not have access to their source code. Instead, we used the Fiasco real-time microkernel [9] and L^{4}Linux [6], a user-level Linux operating-system server that runs on top of the Fiasco microkernel.

There are several systems with a design similar to our L^{4}Linux. User-mode Linux [25], Linux/a386 [14], and Mk-Linux [4] are ports of the Linux kernel to user mode that run on top of various operating systems: The first two run on top of UNIX or Linux itself, while the latter runs on top of a (non-real-time) microkernel. Neither runs on top of a minimal real-time kernel.

Many ad-hoc UNIX and Windows modifications fit into the category of shared-space systems. Today the most prominent exponent of shared-space type is RTLinux [27]. RTLinux uses a systematic approach for adding real-time capabilities to an existing time-sharing kernel. It has excellent real-time–scheduling properties [16, 17]. We have used this system as a base for our experiments and discuss it in detail in the next section.

There exist two basic approaches for determining the worst-case execution time for an instruction path of a given system: static and dynamic program analysis. Static program analysis determines the maximum execution times by analyzing the code off-line [3]. This method requires a pre-
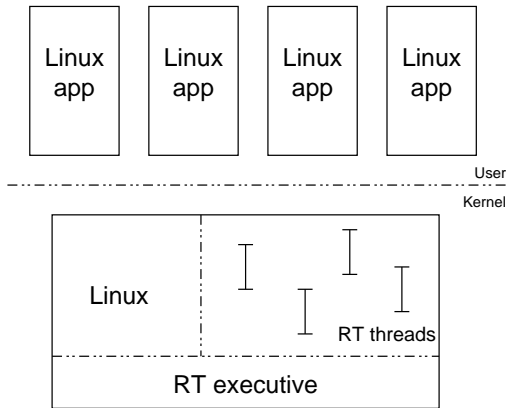
formerly RadiSys INtime

Figure 1. RTLinux structure

cise model of the hardware (including processors, caches, TLBs, buses, input–output subsystem) to calculate the execution time at instruction or block level. The other method, dynamic program analysis, works by executing code paths using different input patterns and determining the pattern that results in the longest execution time. However, as the input pattern and code paths that lead to worst-case execution times are not known, it is important to ensure realistic code and data coverage. In our study, we have used dynamic program analysis because it does not require a detailed hardware model and thus is easier to apply. Section 4.1 explains our methods for generating high coverage in detail.

3   RTLinux and L4RTL

For an accurate comparison of the cost of address spaces in real-time systems, we have partially reimplemented the RTLinux API in the context of the DROPS system. The resulting system, called L4RTL, can run unmodified RTLinux programs after re-compilation.

In this section, we provide a short overview over the architecture of both RTLinux and L4RTL. In Section 4, we will describe our approach for quantifying the worst-case overhead of introducing address spaces to systems using the RTLinux API.

3.1   RTLinux: A real-time Linux

FSMLabs' RTLinux [27] is a small real-time executive that works as an extension to the Linux kernel. The Linux kernel runs on top of the real-time executive as one low-priority application program. The RTLinux executive, the Linux kernel and all real-time threads share one kernel address space and all run in kernel mode. Figure 1 illustrates RTLinux' structure.

RTLinux supports applications with hard real-time CPU-scheduling requirements. The co-located Linux kernel con-

tains modifications so that it enables and disables "soft" interrupts for synchronization. This allows RTLinux to schedule real-time threads with high precision despite Linux's habit of disabling interrupts for synchronization and despite Linux's interrupt-driven device drivers.

The real-time executive includes a scheduler with static priorities that can schedule real-time threads in periodic and aperiodic mode, according to a number of different scheduling policies. It is independent from the Linux scheduler, which only schedules Linux processes but not RTLinux threads. RTLinux also offers an interface that allows real-time threads to attach to hardware interrupts.

RTLinux comes with facilities to exchange messages and bulk data with non-real-time Linux processes: FIFOs and Mbuffs. Real-time processes can use these IPC mechanisms in a nonblocking mode so that they can continue to work even if a Linux process does not manage to keep up with the amount of data a real-time thread sends it.

RTLinux implements FIFOs as ring buffers. Real-time threads can sleep, waiting for a FIFO condition variable, or can access the FIFO in a nonblocking mode. Linux processes access FIFOs using normal read and write system calls, which also trigger a signal for the condition variable. A write operation from a Linux process immediately restarts the waiting real-time thread.

Mbuffs are blocks of memory shared between Linux processes and real-time threads. Therefore, RTLinux provides only open and close operations on Mbuffs for Linux processes.

RTLinux is available for several hardware architectures. The x86 implementation supports as uniprocessors as well as SMP machines.

3.2   L4RTL: A real-time Linux—with address spaces

3.2.1   The DROPS system

We have developed L4RTL, a new implementation of the RTLinux API, in the context of the Dresden Real-Time Operating System, DROPS.

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 [5]. It uses the Fiasco microkernel as its base.

The Fiasco microkernel is a fully preemptible real-time kernel supporting static 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 [9].

For time-sharing applications, DROPS comes with L4Linux, a Linux server that runs as an application program on top of the Fiasco microkernel [6]. L4Linux supports standard, unmodified Linux programs. We have modified
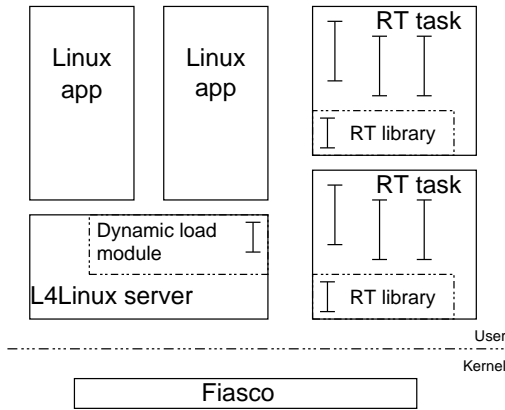
Figure 2. L4RTL structure

L^{4}Linux so that it uses locks instead of disabling interrupts for internal synchronization to prevent its device drivers from disabling interrupts and thereby inducing scheduling delays [7].

The current stable version of Fiasco only supports the x86 architecture in non-SMP mode. Development versions are available for x86-SMP and IA64 machines.

3.2.2 L4RTL implementation

We implemented L4RTL as a library for RTLinux application programs and a dynamic load module for L^{4}Linux. Figure 2 gives an overview of L4RTL's 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 *RT threads* and *RT tasks*, respectively. Each RT task can contain several RT threads, and these threads can cooperate using the RTLinux API.

There can be more than one RT 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* for L^{4}Linux implements the API for use by Linux programs to communicate with RT threads. It also creates a service thread in the L^{4}Linux task. RT 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.

In theory, the service thread in L^{4}Linux could be a bottleneck as several RT threads could wish to communicate with L^{4}Linux at the same time. However, this is not an issue

---

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.

for our tests since we do not use FIFOs for real-time communication but only for transferring measurement values to L^{4}Linux at the end of the run.

FIFOs and Mbuffs are implemented using shared-memory regions between RT threads and the L4RTL load module. RT 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 RT threads.

Besides initialization, more Fiasco-microkernel IPC is necessary only for FIFO signalling. For this purpose, the L4RTL library creates one thread in each L4RTL task (Figure 2). This thread handles signal messages from the L4RTL load module and forwards them to RT threads. RTLinux does not need to generate an IPC for FIFO signaling since RT tasks share the same address space as the Linux kernel and Linux tasks are woken up by calling the appropriate Linux-kernel function in a virtual interrupt context.

In contrast to RTLinux, L4RTL does not contain a scheduler. Instead, it relies on the Fiasco microkernel for scheduling. In our study we use the same scheduling policy for both systems—fixed-priority round-robin 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.

4 Cost of address spaces

Let us now consider the cost that we introduce with address spaces in real-time systems. More specifically, let us compare the latency of interrupts on the shared-space system RTLinux, and the the L4RTL system with separate address spaces. We conduct a minimal interrupt-latency benchmark under worst-case conditions. With this experiment we present a detailed breakdown of the overhead incurred by L4RTL compared to RTLinux and determine which cost can be attributed to the introduction of address spaces and to other implementation artifacts of the system. This experiment was meant to approximate the upper bound for the overhead that can be expected from providing address spaces.

We used two test machines for our measurements:

```
wake_up_interruptible()
```

1. a PC with a 200 MHz Pentium Pro with 64 MB EDO RAM and 256 KByte 2nd-level cache (4-way, 32 bytes/line)

2. a PC with an 1.6 GHz Pentium 4 CPU, 256 MB SDRAM and 256 KByte 2nd-level cache (8-way, 64 bytes/line)

Both RTLinux and L4RTL were based on Linux kernel version 2.2.20. We used the latest version of RTLinux, version 3.1.

In the experiment, we measured the time between the occurrence of a hardware event that triggers an interrupt and the reaction in an RTLinux real-time thread. We conducted the experiment for both original RTLinux and L4RTL.

4.1 Experimental setup

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

First, prior to triggering the hardware event, we make sure that the kernel's and the real-time thread's cache and TLB working sets needed to react to the event are completely swapped out and the corresponding 1st-level and 2nd-level cache lines are dirty.

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 under disabled interrupts. Additionally, they increase confidence that the DROPS system is indeed completely preemptible as we claimed in Section 3.2.1. To avoid missing critical code paths because of pathologic timer synchronization, we varied the time between triggering two interrupts in a range from 2 milliseconds to 30 milliseconds. Times smaller than 2 milliseconds are not practical because this time is required for rigorous cache and TLB flushing.

For cache and TLB dirtying purposes we have written a Linux program that invalidates the caches. It ensures that all 1st-level and 2nd-level cache lines are dirty and need to be written back to main memory when the hardware event occurs. Note that all of the first-level and the second-level cache lines mapping to a specific memory address can contain dirty data from different memory addresses. That means that in the worst case, a single memory read operation results in three memory accesses: write-back of a dirty first-level cache line, write-back of a dirty second-level cache line, and finally the read that was actually intended [15].

Because the cache-flooding program uses more memory pages than the number of page mappings that can be cached in the CPU, it has the side effect of flushing the TLB. One pass of the program needs up to 2.3 milliseconds on Pentium Pro and about 1.2 milliseconds on Pentium 4. Consid-

ering a maximum interrupt frequency of 500 Hz we therefore ensure that the cache and TLB is dirty before an interrupt is triggered.

As the 2nd-level cache is physically tagged on the x86 architecture, the cache-flooding program requires access to physically contiguous memory. We have modified the Mbuff driver for RTLinux such that it hands out physical contiguous memory pages which where reserved using the bigphysarea mechanism [2]. In DROPS, physical memory is offered by a memory server.

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 it makes use of almost all of the Fiasco microkernel's services. We can make this statement with confidence because of Fiasco's small size and limited amount of functionality it implements. For example, hbench heavily starts and deletes processes, resulting in address spaces being created and deleted, and executes L^{4}Linux system calls, resulting in microkernel IPC. As an exception, we have avoided the use of certain microkernel services altogether when these services where required neither for L^{4}Linux nor for L4RTL, for example, message-buffer–copying IPC. The fact that these services are not required indicates that they could have been absent in the first place, and their use would not contribute to our study of the impact of address spaces.

As a reliable and measurable interrupt source, we have used the x86 CPU's built-in interrupt controller (Local APIC). 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 use the Local APIC in periodic mode changing the interrupt rate after 100 interrupts are released.

The drawback of this interrupt source is that unlike other interrupt sources, it cannot be given a higher hardware priority than other interrupt sources. In other words, except for disabling *all* interrupts in the CPU, it is impossible to globally specify which other interrupts must not occur until this interrupt has been acknowledged. We have therefore simulated hardware-interrupt priorities by manually disabling interrupts in the external PIC (*not* in the CPU) immediately after entering the kernel until the user-level interrupt handler in the L4RTL measurement thread has acknowledged its interrupt. This adds overhead—1.5 μs on Pentium Pro, 1.8 μs on Pentium 4—which has to be considered later when we talk about worst-case interrupt latencies. Note that the kernel is entered through an interrupt gate, which automatically disables interrupts on the CPU prior to interrupt-

---

advanced programmable interrupt controller

handler invocation.

With this precaution in place, our interrupt source could not be blocked by any other interrupts, and no other interrupt can preempt our interrupt's handler—not even the system's timer interrupt. Our interrupt source could only be delayed when an interrupt handler servicing another interrupt did not enable the interrupts in the CPU.

4.2 Measurements

4.2.1 What we measured

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 while both the hbench and the cache-flooding load where active.

Additionally, we measured the time between the occurrence of the hardware event and the first instruction of the kernel-level interrupt handler in both RTLinux and the Fiasco microkernel. This measurement was intended to quantify the effect of critical code sections that disable interrupts within these kernels. 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 extended periods of execution with disabled interrupts.

4.2.2 Expectations

To estimate the costs for RTLinux und L4RTL under worst case conditions we have to compare the execution path of both systems when an interrupt is released:

RTLinux. First, there are costs of releasing the interrupt at the hardware and entering the kernel through the interrupt gate. These costs increase if the interrupts are blocked at the CPU for synchronization purposes. The Linux kernel is not an issue here because all `cli` and `sti` statements are emulated by the real-time executive preventing blocking of real-time tasks by time-sharing tasks. Second, there are costs of missing TLBs and caches. Then the kernel module code is executed which does only check if the interrupt is connected to a real-time task and then passes to the interrupt handler.

L4RTL. The costs to enter the kernel are expected to be in the same order of magnitude as under RTLinux. We use a similar mechanism to prevent blocking of interrupts by the Linux server (see 3.2.1). The costs for executing the kernel path will be somewhat more expensive because Fiasco has additional overhead for switching to the address space of the real-time task. Then we return to user mode and have

---

L⁴Linux is not an issue here, because it never disables interrupts for synchronization.

some additional overhead for missing cache lines and TLB entries.

4.2.3 Results and discussion

The diagrams in Figures 3 and 4 show the densities of the interrupt-response times under the load conditions.

The worst-case time we measured for RTLinux is 23 μs on the Pentium Pro and 24 μs on Pentium 4. From these values we have to subtract the overhead added by blocking the interrupts at kernel entry (refer to Section 4.1). The resulting worst-case interrupt latencies of about **22 μs** for Pentium 4 and **21 μs** for Pentium Pro are somewhat higher than the official values claimed for RTLinux.

For L4RTL on Fiasco we measured 58 μs on Pentium Pro versus 33 μs on Pentium 4. These values result in worst-case interrupt latencies of **56 μs** on Pentium Pro and **31 μs** on Pentium 4.
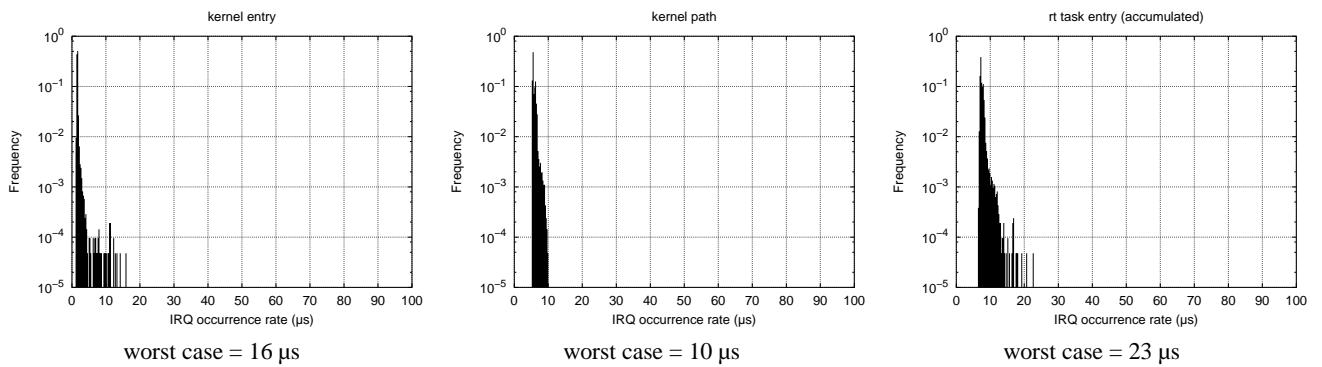
RTLinux achieves a higher level of kernel interruptibility than the Fiasco microkernel (first column in Figures 3 and 4). We believe that we can further reduce the kernel-entry cost of our microkernel in the future. As stated in Section 4.1, the interrupt source could be delayed when an interrupt handler servicing another interrupt did not enable the interrupts in the CPU. We have found that the timer interrupt of the current implementation of Fiasco disables the interrupts up to 23 μs on PPro 200 and up to 13 μs on P4.

The actual worst-case real-time–handler invocation times are shown in the center column of Figures 3 and 4. These results represent the cost needed to activate a real-time interrupt handler with or without address spaces. The main implementation-dependent artifacts, namely interrupt-blocking times, have been factored out from these times. The difference in handler-invocation time between RTLinux and L4RTL (on Fiasco) can be attributed to the introduction of address spaces. We observe that this extra worst-case cost is not significantly larger than uncertainties introduced by dirty caches or blocked interrupts, which designers of real-time systems seem to accept readily.
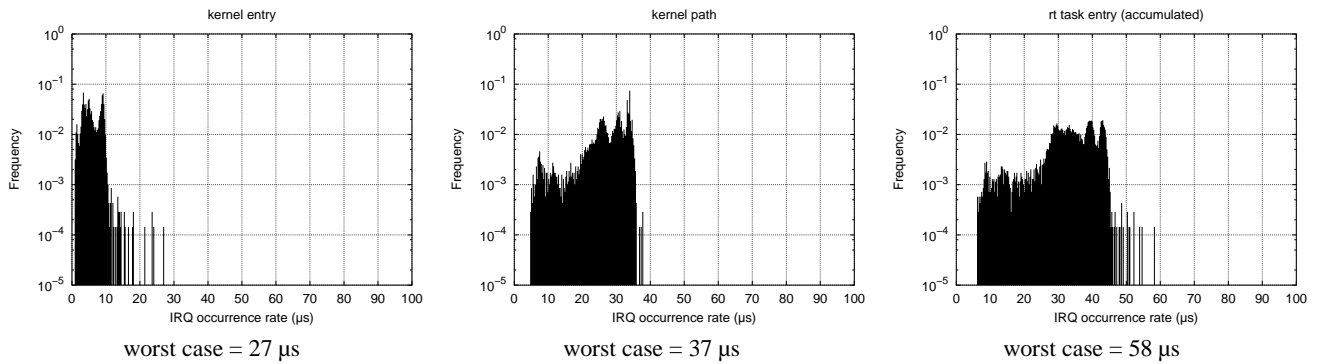
The difference between RTLinux and L4RTL on Pentium 4 is smaller than on PPro 200 (3 μs versus 27 μs). One reason could be that the Pentium 4 has doubled cache-line sizes which results in fewer cache misses.

In [17] our tests resulted in much higher costs for the kernel path of RTLinux on different hardware (800 MHz Pentium III Coppermine, 256 KByte 2nd-level cache, VIA Apollo Chipset). We have rechecked our test and got the same results for RTLinux (56 μs for the kernel path). With our improved version of Fiasco, we now get a worst-case execution time of 22 μs for the kernel path (corresponds to center column of Figures 3 and 4) and a worst-case total interrupt latency of 35 μs (corresponds to right column). We cannot explain why RTLinux exhibits much higher interrupt
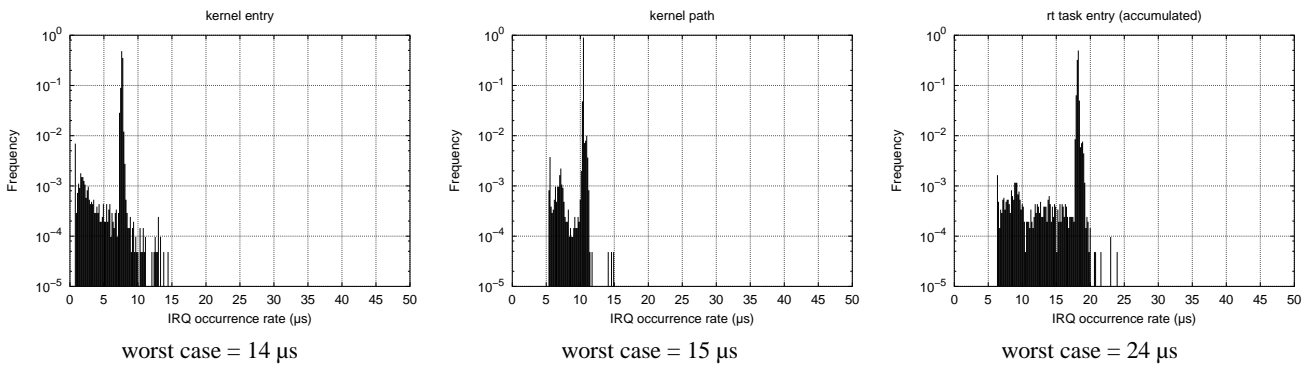
RTLinux

worst case = 16 µs   worst case = 10 µs   worst case = 23 µs

L4RTL / Fiasco

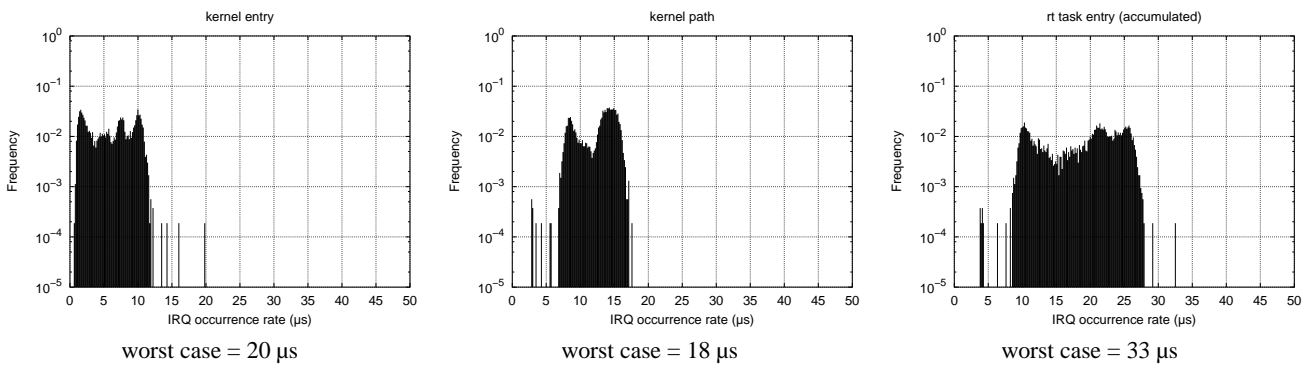worst case = 27 µs   worst case = 37 µs   worst case = 58 µs

Legend: X axis shows interrupt latency. Y axis shows density of occurrence of particular latencies. Note that the Y axis has a logarithmic scale. Left: Time to enter kernel mode. Center: Time to activate handler in RTLinux thread. Right: Accumulated, total time

Figure 3. Interrupt latencies measured on Intel Pentium Pro 200 MHz

RTLinux

worst case = 14 μs   worst case = 15 μs   worst case = 24 μs

L4RTL / Fiasco

worst case = 20 μs   worst case = 18 μs   worst case = 33 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. Left: Time to enter kernel mode. Center: Time to activate handler in RTLinux thread. Right: Accumulated, total time

Figure 4. Interrupt latencies measured on Intel Pentium 4 1600 MHz

response times on this hardware. One possible explanation would be that (RT)Linux enables some SMM feature of the hardware which Fiasco does not (for more information, refer to Section 5). Furthermore, the memory interface of this machine seems to be much slower than on other machines: While one pass of our cache-flooding program needs about 2.3 milliseconds on a Pentium Pro and about 1.2 milliseconds on a Pentium 4, it needs about 32 milliseconds on the 800 MHz Pentium III.

4.2.4 Worst-case execution time of the real-time application

Real-time applications always must deal with the worst case. For their worst-case execution time, they have to take into account invalidated caches and a flushed TLB. We have not measured these secondary (after-invocation) cache-reloading and TLB-reloading costs in the RTLinux real-time thread, because in a worst-case scenario, they are equivalent for both RTLinux and L4RTL. (In the *average* case, we estimate the cost to be somewhat higher for L4RTL: The TLB is always flushed when a task switch to a L4RTL task occurs, and L4RTL probably invalidates more cache lines than original RTLinux before invoking the interrupt handler.)

There are ways on x86 CPUs to guarantee real-time threads a fixed, nonrevocable share of cache and TLB entries to reduce the secondary worst-case cost, but neither original RTLinux nor L4RTL currently implement them. In the remainder of this section, we outline two of these techniques.

For TLB entries, a possible implementation would be to issue 4-MByte pages exclusively to real-time tasks. As 4-MByte pages use a TLB separate from the TLB for 4-KByte pages, TLB conflict misses cannot evict a 4-MByte–page TLB entry. To prevent the TLB entries from being flushed during an address-space switch, a tagged TLB can be emulated by marking the TLB entries as "sticky" (using the global flag in page-table entries) and by using the x86 CPU's segmentation hardware for address-space protection despite "sticky" virtual-memory regions (also referred to as the "small address-space trick") [11].

To avoid the flooding of cache lines and to thereby reduce worst-case memory-access times, cache partitioning can be used to guarantee tasks a fixed set of 2nd-level cache entries. This technique uses knowledge of the wiring of physical-memory addresses to 2nd-level–cache lines and assigns partitions of the cache to address spaces by reserving memory pages that map to specific cache lines for that address space [13, 26]. This approach has the drawback that if an application reserves a certain percentage of cache lines, then the same percentage of main memory needs to be wasted on it.

5 Caveats

Different scheduling behavior of RTLinux and L[4]Linux. L[4]Linux uses a different scheduling algorithm than RTLinux because it uses the scheduler of the underlaying microkernel. Therefore we cannot guarantee that the system load generated by the Linux tasks is exactly the same on both systems.

System Management Mode. There is one source of delay that we could neither measure nor filter out: All Intel Pentium processors implement a System Management Mode (SMM)—a special mode mainly used for emulating devices, to monitor and manage system resources for energy consumption, and to control system hardware [10]. The SMM is entered by the System Management Interrupt (SMI) which is a non-maskable interrupt that preempts and disables all other interrupts. The SMI is handled completely by the hardware and is invisible to the operating system and applications. We carefully tried to avoid actions that trigger an SMI.

DMA virtualization. Currently, we have not virtualized the direct memory access (DMA) controllers so that it is possible for the Linux server to destroy physical memory. This has no influence on our measurement results, but it does affect the system's robustness.

6 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. Furthermore in non-worst-case situations, they consume more cycles than solutions without address space separation, stealing them from time-sharing applications.

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.

Hence, systems with real-time and time-sharing subsystems should be built with address spaces as a basic separation technique.

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") and to apply a user-level memory-management server that provides cache partitioning. Further we plan to use static program analysis for important execution paths of both systems to determine more precise costs and to find potential performance flaws.

In this work, we have investigated the effect of address spaces on the guarantees that can be made to real-time applications. Another area of future work is determining the effects on the performance of microkernel services such as IPC and of time-sharing applications that run on the same system.

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] R. Butenuth. Managing big physical memory areas in linux. Available from URL: http://www.uni-paderborn.de/cs/heiss/linux/bigphysarea.html.

[3] A. Colin and I. Puaut. Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System. In *Proceedings of the 2nd Workshop on Worst Case Execution Time Analysis*, Vienna, Portugal, June 2001.

[4] 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.

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

[6] 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, Oct. 1997.

[7] H. Härtig, M. Hohmuth, and J. Wolter. Taming Linux. In *5th Annual Australasian Conference on Parallel And Real-Time Systems (PART)*, Adelaide, Australia, Sept. 1998.

[8] D. Hildebrand. An architectural overview of QNX. In *1st USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, Apr. 1992.

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

[10] Intel Corp. *Intel Architecture Software Developer's Manual, Volume 3: System Programming*, 1999.

[11] 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, Sept. 1995.

[12] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. Achieved IPC performance (still the foundation for extensibility). In *6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 28–31, Chatham (Cape Cod), MA, May 1997.

[13] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Third IEEE Real-time Technology and Applications Symposium (RTAS)*, pages 213–223, Montreal, Canada, June 1997.

[14] Linux/a386 website. URL: http://linux.a386.nocrew.org/.

[15] J. Löser and H. Härtig. Cache influence on worst case execution time of network stacks. Technical Report TUD-FI02-07-Juli-2002, TU Dresden, July 2001.

[16] F. Mehnert. L4RTL: Porting RTLinux API to L4/Fiasco. In *Workshop on a Common Microkernel System Platform*, Kiawah Island, SC, Dec. 1999. Available from URL: http://os.inf.tu-dresden.de/~fm3/l4rtl_l4ws.pdf.

[17] F. Mehnert, M. Hohmuth, S. Schönberg, and H. Härtig. RTLinux with address spaces. In *Proceedings of the Third Real-Time Linux Workshop*, Milano, Italy, Nov. 2001.

[18] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in Linux. In *Fourth IEEE Real-time Technology and Applications Symposium (RTAS)*, Denver, Colorado, June 1998.

[19] J. Shapiro, D. Farber, and J. M. Smith. The measured performance of a fast local IPC. In *5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 89–94, Seattle, WA, Oct. 1996.

[20] V. Sohal and M. Bunnell. A real OS for real time — LynxOS provides a good, portable environment for embedded applications. *Byte Magazine*, 21(9):51, Sept. 1996.

[21] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *Fourth IEEE Real-time Technology and Applications Symposium (RTAS)*, Denver, Colorado, June 1998.

[22] TenAsys INTime website. URL: http://www.TenAsys.com/intime/.

[23] M. Timmerman and B. V. Beneden. INtime 1.20 evaluation — executive summary. *Real-Time Magazine*, 99(2):9–13, 1999.

[24] H. Tokuda, T. Nakajima, and P. Rao. Real-time Mach: Towards a predictable real-time system. In USENIX, editor, *Mach Workshop Conference Proceedings, October 4–5, 1990. Burlington, VT*, pages 73–82, Berkeley, CA, USA, Oct. 1990. USENIX.

[25] User-mode Linux website. URL: http://user-mode-linux.sourceforge.net/.

[26] A. Wolfe. Software-based cache partitioning for real-time applications. In *Third International Workshop on Responsive Computer Systems*, Sept. 1993.

[27] V. Yodaiken and M. Barabanov. A Real-Time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, Anaheim, CA, Jan. 1997. The USENIX Association.