Taming Linux

Hermann Härtig, Michael Hohmuth, and Jean Wolter

Dresden University of Technology
Dept. of Computer Science
D-01062 Dresden, Germany
l4-linux@os.inf.tu-dresden.de
Tel.: +49 351 463-8282, Fax: +49 351 463-8284

Abstract. This paper describes the overall design, partial implementation and brief performance evaluation of a system in which Linux and its applications run besides real-time applications. The separation of the real-time and time-sharing subsystems is not restricted to the use of the CPU but enforced as well for other resources, namely main memory and caches. This paper details the changes needed for the original Linux to decouple it from real-time processes and analyzes the performance of the resulting system.

1 Introduction

During recent years a major change occurred in the use of computer systems that can be characterized by the coexistence of real-time and non-real-time (time-sharing) applications on the same computer. This coexistence is caused by the use of new media such as audio and video that have real-time requirements. So far, most of these systems deal with that coexistence by throwing enormous amounts of resources at these applications. Other systems, for example those based on QNX [9], deal with the situation by running non-real-time applications as low-priority processes on real-time operating systems. Another class of systems, for instance RT-Linux [18], extend time-sharing operating systems by means to run high priority processes besides the non-real-time applications. Both approaches do handle the real-time requirements with regard to the CPU as a needed resource, but neglect other resources such as caches and disk bandwidth.

That situation is paralleled by the observation that recent advances in computer architecture achieved enormous performance gains for the average case but added uncertainties with regard to the worst-case performance, which is a major concern in real-time systems. A prime example are caches that speed up applications with good locality behaviour. If locality cannot be preserved (*e. g.*, due to context switches), these gains cannot be maintained easily. Hence, worst-case analysis for systems with caches often leads to unacceptably high worst-case performance.

Both the coexistence of real-time and time-sharing applications and the properties of current computer architectures with regard to worst-case performance are taken as challenges by the Dresden Real-time Operating System project (DROPS). The architecture of DROPS is based on resource managers that are designed to separate real-time and time-sharing processes with regard to all resources that matter.

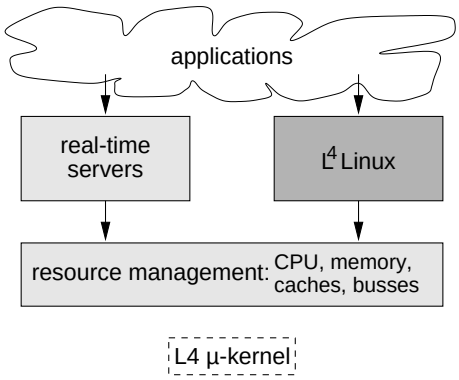


Fig. 1. Structure of the Dresden Real-Time Operating System (DROPS)

The main challenge here is to organize the coexistence in an efficient way, at least efficient enough to beat the "resource throwing" approach. To facilitate this, DROPS is based on the L4 microkernel (µ-kernel) [10], with the time-sharing and real-time subsystems running in user mode as separate operating-system personalities. The time-sharing subsystem is comprised of a Linux system using the L4Linux server, a port of the Linux kernel to the L4 µ-kernel [7]. The real-time subsystem consists of a set of servers which support applications with quality-of-service requirements.

These two subsystems use a common management layer for system resources. Managers for SCSI busses and ATM networking bandwidth and a file system have being designed and are being implemented [2, 14]. However, in this paper we present results for the management of CPU and caches.

In the rest of the paper, we first describe the overall structure of DROPS and the resource management. Then we discuss related work, especially the RT-Linux operating system from which DROPS borrowed some ideas. In Section 4 we discuss the changes needed to apply to L4Linux, an implementation of Linux running as a user process on the L4 µ-kernel [10]. At the end, we present some performance results and compare them to RT-Linux.

2 DROPS

The DROPS project focuses on µ-kernel-based real-time systems, for instance for continuous media presentation. Our vision is that real-time components can get all the system resources and processing power they need, and the rest can then be spent for a time-sharing subsystem running on the same machine. This is enforced by a system-wide resource management for all resources: not just for CPU time slices and main memory, but also for caches and I/O busses (Figure 1).

L4 is a lean µ-kernel featuring fast message-based synchronous inter-process communication (IPC), a simple-to-use external paging mechanism and a security mechanism based on secure domains. The kernel implements only a minimal set of abstrac-

tions upon which operating systems can be built flexibly [10]. There are versions of L4 for various x86 and for the Alpha and MIPS CPUs [11], but in this paper we solely refer to systems based on L4 for x86 (Pentium) processors.

In an earlier paper [7] we describe the time-sharing subsystem of DROPS we implemented by porting the Linux kernel to the L4 µ-kernel, creating an L^{4}Linux server running completely in user mode (including device drivers and interrupt handlers) as an application program. By using Linux as the base operating system, we have access to a vast library of development tools, creating a powerful development platform. L^{4}Linux is binary compatible with Linux for the x86 architecture (PCs), allowing any off-the-shelf Linux system distribution and software to be used without recompiling.

An important design criterion when designing L^{4}Linux was that the Linux kernel should be modified only minimally in order to allow easy integration of new Linux kernel versions into L^{4}Linux and to save development time. We didn't intend to optimize L^{4}Linux to the underlying L4 architecture in the way the BSD single servers have been optimized for the Mach µ-kernel [6, 8]. The simplest possible design matching this criterion is of course a single-server design where the Linux server and user applications run in separate tasks communicating using the L4 µ-kernel's IPC mechanism. Details can be found in [7].

Work is currently underway to build real-time components such as a real-time SCSI driver, a file system and a network protocol suite supporting applications with quality-of-service requirements.

In order to avoid real-time service disruptions by the time-sharing subsystem, we had to make some changes to the L^{4}Linux server so that instead of "taking over" the whole machine, it can run under a resource manager handling all system resources. Before we describe these changes in detail in Section 4, we'll cover RT-Linux and other related work in the next section.

3 Related Work

Real-time operating systems can be classified into three categories: The first one is comprised of systems which started as time-sharing operating systems and were retrofitted later for real-time needs. Examples are Real-Time Unix and many recent systems which implement the POSIX 1.b standard. Often such real-time extensions can only be added at considerable expense, and real-time guarantees that can be made by such systems are often constrained to CPU schedules only. The second class of real-time systems consists of those systems that have been specifically designed to support real-time. There are many successful commercial systems in this class, for instance VxWorks [16], and QNX [9]. However, due to their specialized nature these operating systems often support only a subset of the Unix API so that many popular applications aren't available for them.

We believe that DROPS belongs to a third class: Run an only minimally modified time-sharing kernel as an application on top of a real-time kernel. This is a clean design because it cleanly separates the real-time and time-sharing components: It requires less expenditure than making a time-sharing kernel fully preemptible, and it immediately opens the huge library of existing development and application software.

New Mexico Tech's Real-Time Linux (RT-Linux) [18] is similar to DROPS in that the time-sharing subsystem runs as an application on top of a small real-time executive. Both systems use a Linux kernel with minimal modifications as the time-sharing operating system.

In RT-Linux, the Linux kernel and all real-time tasks run along with the real-time executive in kernel mode. The real-time executive is responsible for CPU scheduling and also contains support for IPC between tasks using FIFOs. In order to be able to schedule real-time tasks with high precision—even though Linux has a habit of disabling interrupts for synchronization, and in the presence of interrupt-driven device drivers in the Linux kernel—, RT-Linux has been modified to enable and disable "soft" interrupts. All "hard" (machine) interrupts are caught by the real-time executive and only passed on to Linux if it is responsible for handling the interrupt and has enabled the corresponding soft interrupt; hard interrupts cannot be disabled by Linux.

DROPS' L^{4}Linux also uses soft interrupts to avoid disrupting the real-time subsystem; this is described in more detail in Section 4.2. L^{4}Linux differs from RT-Linux in that Linux runs completely in user space as a task on the L4 µ-kernel. The real-time subsystem consists of several separate L4 tasks, also running in user mode.

In comparison, DROPS has the advantage of fault separation (because all tasks run in separate address spaces) at the cost of a higher context-switching overhead. Furthermore, DROPS not only guarantees real-time tasks all the CPU time they need but can also manage several system resources including main memory, 2nd-level cache partitions, networking bandwidth, and in the future SCSI bus bandwidth. DROPS offers the additional feature that a real-time task can also be a Linux process. However, L^{4}Linux currently does not support symmetric multiprocessing (SMP), which RT-Linux does. We expect to add SMP support when upgrading L^{4}Linux to Linux 2.2 (whose concurrency model for interrupts better fits L^{4}Linux' multi-threaded design than the current base version, 2.0).

In the next section, we'll examine in detail the changes made to the L^{4}Linux server so that it cannot disrupt the real-time subsystem on the same machine.

4 Taming Linux

Because of our goal of making only minimal changes to the Linux kernel, we designed L^{4}Linux like yet another port of Linux to a new architecture: Only architecture-specific parts of the kernel were modified, and the port to the L4 µ-kernel is much like a port to for instance the Motorola 68k architecture. Therefore, the resulting L^{4}Linux system is basically just like any other Linux system, and as such it behaves like any typical monolithic kernel: It takes over the whole machine, controlling everything.

To fit nicely within the DROPS framework, however, L^{4}Linux had to be "tamed" such that system-wide global resource management would become possible. We have identified the following issues which need to be resolved to make L^{4}Linux a well-behaving member of the DROPS system: The L^{4}Linux kernel:

– has full control over all resources controlled by the L4 µ-kernel: task numbers, task priorities, main memory and the I/O address space, interrupt request lines, and the interrupt disable/enable privilege

– disables and enables hardware interrupt requests at will
– drives all devices
– doesn't support partitioning the cache (explained below)

In the following subsections, we'll address each of these issues, discussing their handling in the DROPS system and the modifications required to L^{4}Linux.

4.1 Controlling System Resources

The "L^{4}Linux controls all system resources" issue has been addressed by introducing an instance between L^{4}Linux and L4 from which the resources must be requested explicitly, a supervisor task called "Rmgr" (resource manager). This task is started as L4's first user task and magically "owns" all L4 µ-kernel resources enumerated in the previous subsection. System resources can be given to other user tasks on request using a special IPC protocol. Rmgr can be configured flexibly (but currently only at boot time) using a script language.

Using Rmgr it is possible to constrain the L^{4}Linux server in the system resources it can allocate. The allocations are static in nature[1], so the modifications required to L^{4}Linux can be restricted to the boot code where all system resources configured as "available for L^{4}Linux" are requested from Rmgr.

Rmgr also supports starting L4 tasks which are not also Linux programs—this wasn't possible previously with L^{4}Linux. This way, drivers or other real-time components independent from L^{4}Linux can be started at boot time or later.

4.2 Interrupt Request Enabling and Disabling

The Linux kernel uses the C interfaces `cli()` and `sti()` for entering respectively leaving critical code sections. In the x86 version, these interfaces are translated into the `cli` and `sti` assembly statements which disable respectively enable the propagation of hardware interrupt requests to the CPU. This implementation had been adopted unmodified in the L^{4}Linux port.

However, in DROPS it is unacceptable to disable interrupts for longer periods for synchronization purposes. Interrupts should remain enabled virtually at any time in order to prevent interference with L4's scheduler which depends on precise timer interrupts. The only legal use of the `cli` and `sti` assembly statements should be to protect code sections directly dealing with the PC's programmable interrupt controller (PIC), that is, in interrupt handlers.

We therefore modified the implementation of the `cli()` and `sti()` C interfaces to work without disabling interrupts while still ensuring their semantics (to protect a critical section). Our new implementation uses a lock which is acquired in `cli()`. When `cli()` detects that the lock is already held, the current context is enqueued in

[1] In fact, for main memory allocations it is desirable for the real-time subsystem to request more memory from L^{4}Linux when it is short on memory. This would be easy to implement in L^{4}Linux using the Rmgr protocol and using Linux' `get_free_page()` interface, but this hasn't been done yet. So, main memory allocation currently is static, too.

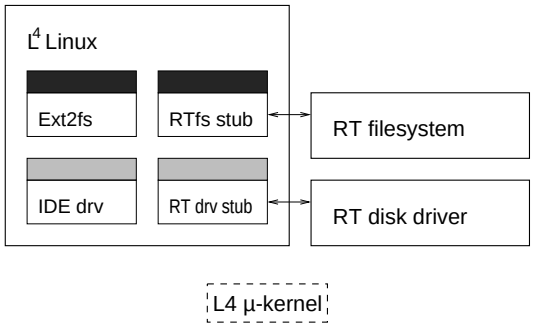


Fig. 2. L^{4}Linux can use external drivers at any level. In this example, we use two stub drivers: one at the file system level and one on the block device level.

the lock's wait queue and put to sleep. When leaving the critical section, `sti()` checks whether there is a context waiting in the queue and if so, wakes it up. In L^{4}Linux, all contexts which need to be synchronized—the kernel's main thread, the high-priority low-level interrupt handlers ("top halves") and the low-priority interrupt handlers ("bottom halves")—run in separate L4 kernel threads for which L4 handles all the context switching. L4 IPC primitives are used for putting threads to sleep and waking them up.

This way, interrupts can be delivered even if L^{4}Linux has entered a critical section. However, the thread executing the interrupt handler honors the `cli()` lock and adds itself to the lock's wait queue if it is currently held, and so the synchronization semantics are preserved.

Ideally, L^{4}Linux would not be allowed to execute the `cli` and `sti` assembly statements at all (this is a task attribute under L4 which can be granted or denied using Rmgr). However, because L^{4}Linux contains device drivers with interrupt handlers, it must be able to program the PIC and disable interrupts while doing so. Currently we feel this isn't much of a risk. Should we ever become convinced that L^{4}Linux must not disable interrupts at all, we either must externalize all device drivers with interrupt handlers into separate server tasks (we already have done this for some of them—see section 4.3) or we must externalize PIC programming.

The very few places where the use of the `cli()` C interface really means "I'm going to program the PIC, please *do* disable interrupts," were easily identified and changed not to use the new `cli()` implementation.

4.3 Device Drivers

The original L^{4}Linux controls all devices using the unmodified Linux-for-x86 drivers. However, for resources shared between the real-time and the time-sharing subsystems it is unacceptable to use a driver in the time-sharing subsystem because such a driver cannot give quality-of-service guarantees for the real-time subsystem.

Therefore we are creating drivers for the real-time subsystem which support resource planning and preallocation. We currently have drivers (ported from Linux re-

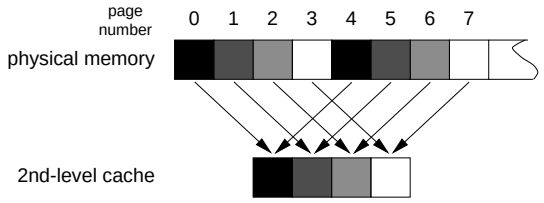


Fig. 3. Cache partitioning using main memory coloring. For a physically indexed 2nd-level cache, the main memory pages are mapped to different cache portions. By controlling the colors of main memory one task may allocate, the cache can be partitioned between the tasks.

spectively developed in-house) for SCSI controllers and Ethernet and ATM networking cards [5, 13, 15], and work is underway to make SCSI bus and networking bandwidth resources scheduled by a system-wide resource management. We are also developing a real-time file system using these drivers.

These drivers and the real-time file system should eventually be used by L^{4}Linux, too. The strategy here is to replace its native drivers with stubs that communicate with the real drivers via L4 IPC. Such stubs can be plugged in at any level (Figure 2). For instance, we currently have two stubs in the works: One uses an external ATM protocol server (which uses a low-level ATM driver in a separate task). The second uses an external SCSI block device driver (with embedded low-level drivers) which maintains two different request queues for requests from real-time and non-real-time tasks.

4.4 Main Memory Management and Cache Partitioning

We have implemented a virtual memory management server (called “VM server”) for L4 which allows separating the 2nd-level memory cache working sets of real-time and time-sharing tasks into separate partitions so that time-sharing applications cannot disrupt the cache working sets of real-time applications. This allows the worst-case execution times of real-time programs to be bound to a significantly lower level. The cache partitioning is accomplished by coloring the main memory pages and controlling which colors of pages can be allocated by a given task set (Figure 3) [12, 17].

The main obstacle in adopting Linux to running under this memory management policy was that Linux requires to be run in a virtual address space mapping the physical memory one-to-one; this is an important assumption of many Linux device drivers. Therefore, L^{4}Linux uses memory management tricks (using L4’s memory management primitives and information provided by the VM server) to map all physical memory pages it acquired from the VM server one-to-one, thus leaving holes in its own address space.

However, Linux assumes that its kernel code, data and bss segments as well as an initialization memory region (from which Linux subsystems can grab memory chunks at system initialization time) are mapped contiguously into its virtual address space—an assumption that clashes with the requirement that all pages be mapped in one-to-one from the physical space. Fortunately, we didn’t find any driver which mistakes its

initialization memory as physical memory; and so this contiguous memory region can be composed of non-contiguous physical memory pages.

5 Performance Measurements

The original L^{4}Linux as presented in [7] is about 2 % to 4 % slower than the original monolithic Linux kernel for application-level benchmarks on the same hardware[2].

In this section we'll answer the following questions:

- How does taming the L^{4}Linux server as described in sections 4.1 and 4.2 affect its performance?
- How does restricting L^{4}Linux to a cache partition comprised of half of the system's 2nd-level memory cache (128 KB of 256 KB) as described in section 4.4 affect its performance?
- How does taming L^{4}Linux affect the responsiveness of real-time tasks running on the same system?
- How can cache partitioning improve the performance and predictability of real-time tasks?

The last question has already been addressed in detail by Liedtke and colleagues [12], and therefore we skip the discussion here for brevity and just present the result: In one of the experiments, a 64×64 matrix multiplication, the slowdown induced by introducing a cache-intensive secondary workload could be reduced by 74 % when partitioning the 2nd-level cache.

The first three questions are discussed in the following subsections, which are are organized as follows:

We use two different benchmarking packages for the performance evaluation of the Linux systems. First, we will present a series of microbenchmark measurements executed using hbench:OS in Section 5.1. Then, Section 5.2 shows the results for running a macrobenchmark, the AIM Multiuser Benchmark Suite VII.

Finally, Section 5.3 evaluates the real-time responsiveness of of our system.

Where applicable, the performance of L^{4}Linux is compared with monolithic Linux and with RT-Linux. All measurements were conducted on a single machine.[3]

[2] Actually, [7] reports a slowdown of 5 %–10 %; however, in the meantime we've found and fixed a bug adversely affecting L^{4}Linux' performance; please see the slides accompanying [7] for more details.

[3] We used a 133 MHz Pentium PC based on an ASUS P55TP4N motherboard using Intel's 430FX chip set, equipped with a 256 KB pipeline-burst second-level cache and 64 MB of 60 ns Fast Page Mode RAM. We used version 2 of the L4 µ-kernel. L^{4}Linux is based on Linux version 2.0.21, RT-Linux on version 2.0.29; according to the 'Linux kernel change summaries' [4], only performance-neutral bug fixes were added to 2.0.29, mostly in device drivers. We consider both versions comparable.

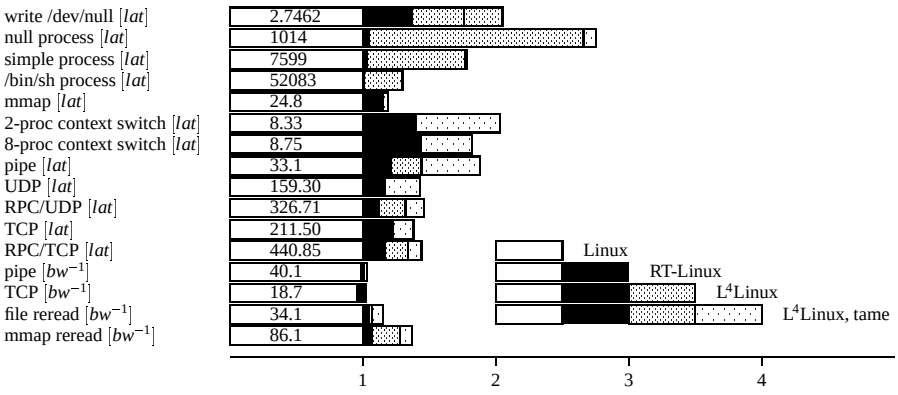


Fig. 4. *hbench:OS* results, normalized to native Linux. These are presented as slowdowns: A shorter bar is a better result. $[lat]$ is a latency measurement, $[bw^{-1}]$ the inverse of a bandwidth one. The numbers in the white boxes show the absolute values for native Linux in microseconds (for latencies) and in MB/s (for bandwidths), respectively. Hardware is a 133 MHz Pentium.

5.1 Microbenchmarks

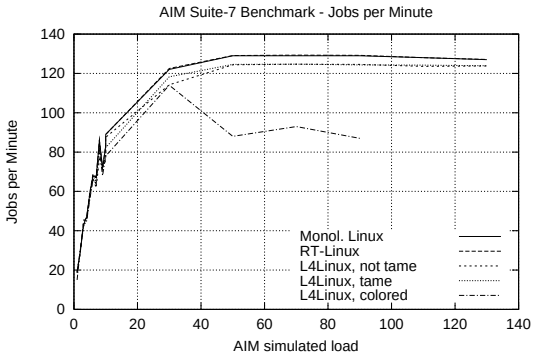
For a close look at the exact performance penalties involved we use the hbench:OS microbenchmark suite. It measures basic operations like system calls, context switches, memory accesses, and pipe operations. This benchmark has been developed to compare different hardware from the operating system's perspective and therefore also includes a variety of OS-independent benchmarks, in particular measuring the hardware memory system and the disk [3]. Since we always use the same hardware for our experiments, we only present selected operating-system dependent parts. The hardware-related measurements indeed reported the same results on all systems.

Figure 4 shows a chart of slowdowns for various operations for monolithic Linux, RT-Linux, L^{4}Linux, and tame L^{4}Linux (without cache partitioning). The exact benchmark results have been skipped here for brevity and can be found in the online version of the paper.[4]

Discussion. As can be seen from Figure 4, while some differences are lost in the noise, minor penalties for some microbenchmarks are introduced when adding soft interrupts in Linux (with RT-Linux) respectively L^{4}Linux (with the tamed version). These are due to the increased context switching overhead that occurs when a hard interrupt happens while soft interrupts are disabled. Also, the soft versions of `cli()` and `sti()` need significantly more cycles than their hard counterparts when more than one context tries to enter a critical section.

The generally larger number for the L^{4}Linux versions come from the higher number of user/kernel-boundary crossings in the µ-kernel based Linux variant. (Please see [7] for a breakdown of the penalties involved with one system call.)

[4] `http://os.inf.tu-dresden.de/pubs/#pub-part98`



kernel	**relative average throughput**	**maximum throughput** [jobs per minute]
monolithic Linux	100 %	129.6
RT-Linux	99.8 %	129.3
L^{4}Linux	96.2 %	124.7
L^{4}Linux, tame	96.1 %	123.7
L^{4}Linux, tame and with colored cache	80.7 %	114.8

Fig. 5. AIM Multiuser Benchmark Suite VII. Left: Jobs completed per minute depending on AIM load units. Right: Average throughput (normalized to monolithic Linux) and maximum throughput of the various Linux kernels.

5.2 Application Benchmarks

For an overall system performance test we use the commercial AIM Multiuser Benchmark Suite VII. It uses Load Mix Modeling to test how well multiuser systems perform under different application loads [1]. (The AIM benchmark results presented in this paper are not certified by AIM Technology.)

Figure 5 shows the achieved throughput (jobs per minute) depending on simulated load for monolithic Linux, RT-Linux, the original L^{4}Linux, tame L^{4}Linux, and tame L^{4}Linux running with a partitioned cache (128 KB of 256 KB). The AIM benchmark successively increases the load until the maximum throughput of the system is determined. From these results the average performance relative to monolithic Linux and the maximum throughput can be computed (displayed on right side in Figure 5).

Discussion. For macrobenchmarks, the introduction of soft interrupts doesn't seem to have much effect. The average slowdown of L^{4}Linux compared to monolithic Linux is 3.8 % while for a tame L^{4}Linux server it is 3.9 %. Similarly, the slowdown of RT-Linux against monolithic Linux is 0.2 %.

Again, the general performance penalty of the L^{4}Linux version compared to Linux and RT-Linux (about 3.8 %) is mainly due to the higher number of user/kernel-boundary crossings. One other factor which contributes to the performance loss is the large percentage of main memory the L4 µ-kernel reserves at boot time for its own internal use (14 MB of 64 MB); when comparing L^{4}Linux to a monolithic Linux kernel running with a similar handicap, the performance penalty decreases to below 2 %. We expect to lower L4's high memory footprint in its next release with a better kernel memory management scheme and with the advent of a memory management protocol which allows L4 to dynamically grow its kernel memory only when needed.

The version of L^{4}Linux running with a partitioned cache suffers an even larger performance degradation. The slowdown is acceptable (0 %–6.5 %) for loads smaller than 30, but after that point the penalty increases to 28 %–32 %. The additional penalty

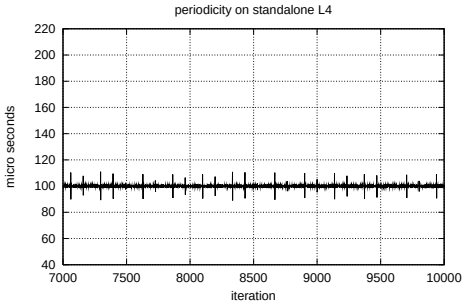


Fig. 6. Periodicity of a real-time task on stand-alone L4

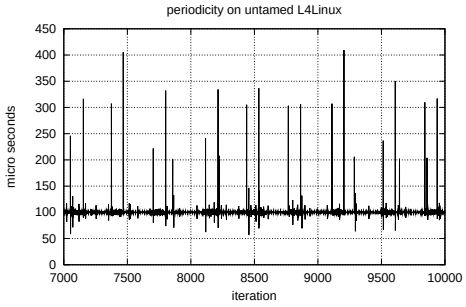


Fig. 7. Periodicity of a real-time task running besides original L^{4}Linux

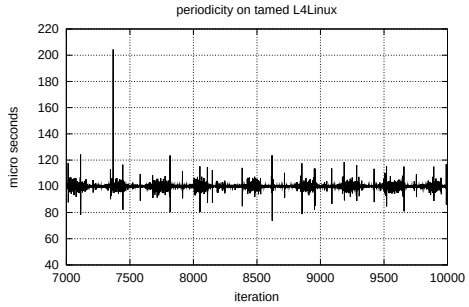


Fig. 8. Periodicity of a real-time task running besides the tamed version of L^{4}Linux

These diagrams show the latency of subsequent activations of two tasks alternately triggered by a 100-µs interrupt handler. The x-axis shows the number of the current activation, and the y-axis shows the elapsed time since the previous activation.
Please note that the y-axis has been scaled differently in the three diagrams.

originates not only from the limited part of the 2nd-level cache it can use but also from two other factors: First, and most importantly, due to the way the cache is partitioned, L^{4}Linux not only runs with only half the cache but also with only half the main memory—only 25 MB (the main memory must also be partitioned [12]); that is why the machine starts thrashing at a much lower load. Second, because a fragmented main memory is being used, L^{4}Linux can not any longer make use of the Pentium CPU's 4-MB memory pages (it hasn't mapped a contiguous 4-MB chunk of main memory), resulting in more TLB faults because L^{4}Linux now shares the TLB for 4-KB pages with its applications.

5.3 Real-Time Responsiveness

The real-time responsiveness has been tested by measuring the periodicity of two high-priority user tasks which are alternately triggered by a third user task using a 100-

µs timer interrupt routine as its activation source. The two periodic tasks record the Pentium CPU's cycle counter every time they are activated. The two cycle counter logs are coalesced later to a single long log where the difference of two adjacent elements is the time between two task activations.

This test has been conducted in three system environments: first running standalone on L4, then running besides the original L^{4}Linux server, and finally besides a tame L^{4}Linux server (without cache partitioning). During the last two measurements the L^{4}Linux server was stressed with a high system load: a 'dd' on a disk device and a 'find' running in parallel. The results are plotted in Figures 6, 7 and 8.

Discussion. Figure 6 shows that the periodicity that can be accomplished with the L4 µ-kernel is quite high. However, there are some small systematic errors: there is a small deviation of about 1 to 7 µs which happens systematically every 1.8 ms, L4's timer interrupt interval. These deviations stem from L4 not yet being fully preemptible.

Figure 7 shows that when the test program runs besides a non-tame L^{4}Linux server, a 100 µs periodicity cannot be guaranteed. A lot of interrupts are simply lost because Linux disables hardware interrupt propagation for too long.

Figure 8 shows that using a tame L^{4}Linux server, the 100 µs periodicity can be guaranteed just fine, but the deviation is larger (up to 24 µs). At one point the graph shows that an interrupt is being lost. Obviously under high load the L4 µ-kernel sometimes still disables interrupts longer than it should—clearly there is room for improvement here.

These results are similar to those presented for RT-Linux in [18]. The authors report a maximum variation of 15 µs for a task with a 100-µs period under RT-Linux.

6 Conclusion

In our experience, building a multi-personality µ-kernel-based system with a time-sharing and a real-time subsystem running on the same machine is relatively easy to accomplish. The performance of the time-sharing part isn't hampered significantly by the construction we have chosen. Also, we don't see a reason to give up on memory protection between real-time tasks.

Our experiments show that the L4 µ-kernel is not yet fully capable of real-time scheduling because it is not fully preemptible. We expect this to be fixed in the near future.

Acknowledgments

We'd like to thank our anonymous reviewers for their valuable comments. Robert Baumgartl, Martin Borriss, Sven Rudolph, and Sebastian Schönberg provided helpful feedback and commentary on earlier versions of this paper.

Many thanks to AIM Technology for providing us with the AIM Multiuser Benchmark Suite VII.

References

1. AIM Technology. *AIM Multiuser Benchmark, Suite VII*, 1996.
2. Martin Borriss and Hermann Härtig. Design and implementation of a real-time ATM-based protocol server. Technical Report SFB-G2-02/98, Sonderforschungsbereich der Deutschen Forschungsgemeinschaft 358, TU Dresden, June 1998. Available from URL: `http://os.inf.tu-dresden.de/pubs/#pub-rtatm`.
3. 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.
4. M. E. Chastain. Linux kernel change summaries. URL: `ftp://ftp.shout.net/pub/users/mec/kcs/`.
5. Uwe Dannowski. Portierung des Linux ATM-Treibers für FORE PCA-200E auf den Mikrokern L4. Term paper, TU Dresden, 1997. In German. Available from URL: `http://os.inf.tu-dresden.de/project/atm/`.
6. D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *USENIX 1990 Summer Conference*, pages 87–95, June 1990.
7. 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. Paper and slides available from URL: `http://os.inf.tu-dresden.de/L4/`.
8. Johannes Helander. Unix under Mach: The Lites server. Master's thesis, Helsinki University of Technology, 1994. Available from: `http://www.cs.hut.fi/~jvh/lites.MASTERS.ps`.
9. D. Hildebrand. An architectural overview of QNX. In *1st USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 1992.
10. J. Liedtke. On µ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.
11. 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.
12. 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.
13. Frank Mehnert. Portierung des SCSI-Gerätetreibers von Linux auf L3. Term paper, TU Dresden, November 1996. In German. Available from URL: `http://os.inf.tu-dresden.de/L4/`.
14. Sven Rudolph. Admission Control für ein echtzeitfähiges Dateisystem. Master's thesis, TU Dresden, May 1998. In German. Available from URL: `http://os.inf.tu-dresden.de/pubs/`.
15. René Stange. Systematische Übertragung von Gerätetreibern von einem monolithischen Betriebssystem auf eine mikrokernbasierte Architektur. Master's thesis, TU Dresden, May 1996. In German. Available from URL: `http://os.inf.tu-dresden.de/L4/`.
16. Wind River Systems, Inc., Alameda, CA. *VxWorks Programmer's Guide*, 1993.
17. A. Wolfe. Software-based cache partitioning for real-time applications. In *Third International Workshop on Responsive Computer Systems*, September 1993.
18. 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.