Ten Years of Research on L4-Based Real-Time Systems

Hermann Härtig and Michael Roitzsch
Department of Computer Science
Technische Universität Dresden
01062 Dresden, Germany
{haertig,mroi}@os.inf.tu-dresden.de

Abstract

Microkernels are an intriguing technology for operating systems research in general and for real-time systems in particular. To gain experience and to explore new ground, the OS research group at Technische Universität Dresden has been developing L4/Fiasco, a real-time implementation of the L4 microkernel specification. Using this kernel, we built an architecture that supports legacy software and provides real-time guarantees. In this paper, we will describe and discuss the design decisions that led us to this architecture. Based on this system, we set out to explore interesting real-time research areas such as networking, disk scheduling and real-time graphics. The results have been published separately, but we will use this article to give a concise overview and present the rationale of our platform strategy as a whole.

1 Introduction

The ideas behind the L4 microkernel were born back in the mid-1990's when Jochen Liedtke reexamined the design of the earlier generation microkernels around Mach. Trying to prove that a minimal kernel can still provide a high system performance, he developed first L3, then L4. The fundamental principle of his microkernels is that a concept will only be allowed inside the kernel, if user-land implementations would be unable to achieve the required functionality. This leads to truly minimalist kernels supporting only address spaces, threads and interprocess communication. These basic services are enough to run isolated user-level processes on top of L4. Any additional functionality must be implemented as a server process. This includes components like file systems, networking and even device drivers, all of which are usually subsumed as an operating system personality.

1.1 Diverse Platforms

Roughly at the same time, multimedia applications were pushing forward into mainstream computing, because the required performance became increasingly available to consumers. The characterising new requirement of those systems was the coexistence of highly dynamic real-time and non-real-time workloads, sharing computer cores, disks, video subsystems and networks. Previously, real-time systems used to be dedicated, having the complete hardware for themselves. Now, both real-time and non-real-time applications are started and stopped side by side at the user's discretion. But although the requirements towards the system changed, the basic architecture of the underlying operating systems stayed the same. Instead, software vendors tried to solve the emerging problems in middleware. We believe this approach is misleading, because no middleware can isolate real-time from non-real-time tasks or reliably enforce resource guarantees for real-time applications without proper core operating system support.

1.2 Resource Throwing

Advancements in computer hardware achieved enormous performance improvements by using caches to exploit the locality of the applications' behavior. However, those techniques are not necessarily useful for real-time systems, because they tend to concentrate on improving the average case, whereas real-time applications must consider the worst case. For the longest time, dedicated real-time systems had been using less powerful, yet expensive specialized hardware to overcome this. On standard computer hardware, these problems were traditionally dealt with either not at all or by spending enormous amounts of resources. But all those overprovided resources are usually wasted, because the average case behavior is much more benign than the rare but devastating worst case situations, and this gap continues to widen as technology progresses. We intend to solve this problem by dealing with overload situations in ways other than the resource throwing approach. Making the powerful commodity hardware predictable, we can allocate resources much closer to the average case.

1.3 Overview

We do not want to explore our ideas just theoretically, but strive to build a system usable on a daily basis. To this end, we had to consider the maturity and stability of the system. This causes a lot of "unscientific" work, but enforces honesty. Thus, we first present our design decisions for the entire system architecture in Section 2. We include a thorough analysis of the costs caused by this design in Section 2.3. In Section 3, we discuss our solution to handling overload conditions. With those key elements in place, we continue to deal with the management of reservations for resources such as disk, network and graphics bandwidth in Section 4. In Section 5, we conclude by tying all the pieces together into a real-time component architecture that makes the research results readily available for the software development process.

2 Designing the System

A major change in computer systems was the emerging coexistence of real-time and non-real-time applications on the same machine. Today, a large variety of systems has to support a diverse set of such use cases:

- Multimedia applications are used on the average desktop. These applications have immediate real-time requirements, because frames need to be delivered to the display at fixed time intervals. Although deadline misses are not catastrophic, they diminish the user's media experience because they will be visible as motion judder or even frame drops.
  At the same time, non-real-time components may be running next to the player core. For example the media library and subscription management common to today's integrated client applications such as iTunes are clearly non-real-time tasks.
- Off-the-shelf computers are used for sound applications like multitrack editing and live recording of music instruments. Contrasting the media player scenario, enforcing a lower bound on throughput is not the only requirement, but a low latency is needed as well. Music artists can notice delay, if the sound from the speakers is more than 10 ms behind the key hit on the keyboard.
  But arranging the sound in the user interface into multiple tracks, choosing instruments, tweaking the sound and managing media assets has no real-time requirements.
- Mobile phones are being used increasingly for personal information management. Calendar and address book applications are running, as well as games. Sometimes an entire Java virtual machine is used. Alongside, the phone still needs to handle the GSM protocol in a timely manner and once a call is accepted, the speech encoder needs to deliver data from the microphone to the mobile network within certain bandwidth and latency bounds.

From these examples, we can observe that applications with real-time requirements often have small, isolated real-time core functionality surrounded by a large and complex non-real-time part. Current mainstream operating systems do not honor this separation but treat both parts equally, often they are even co-located in the same address space, so proper isolation of resource reservations is impossible. Consequently, current systems have to overprovide resources so that the requirements of the real-time part are satisfied even if it is treated as a best-effort task only.

2.1 Legacy Support

The obvious idea to derive from these findings is that we should reuse an existing operating system personality for the non-real-time parts. This will enable support for legacy software and will ease splitting applications into a real-time and a non-real-time part that can then be treated by the OS differently. Because of its availability in source code and its wide range of application software, the operating system personality of choice is the POSIX personality of the Linux kernel.

One way to implement this is by designing a real-time kernel from the ground up and reimplementing the interface of the legacy kernel in this system. This is the approach taken by the QNX [1] real-time operating system. However, matching the personality of an operating system by reimplementing it is very hard to do, because you are dealing with a moving target. Every change in the semantics of the legacy kernel must be followed to be fully compatible, which invariably introduces a delay. The state of the reimplementation will always be behind the original, which limits the compatibility.

A better solution is to reuse an existing implementation of the OS personality. This is the approach taken by RTLinux [2], which runs the legacy kernel next to high priority real-time processes on top of a small real-time executive. However, 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 supports interprocess communication between tasks. But since the legacy kernel is notoriously complex, it is difficult to enforce real-time properties reliably. Furthermore, in the design of RTLinux, there is no isolation between different real-time tasks, which makes it harder to rule out crashes. One real-time task alone can potentially take over the entire system.

Fortunately, with the L4 microkernel technology, we have another option: We can shift the legacy kernel into user space and have it run as an operating system personality server on top of our microkernel. Our implementation of the L4 specification is named Fiasco and it is fully real-time capable, because it guarantees upper bounds for the execution times of all critical operations. Moving Linux to a user space application is possible with only little modifications to the Linux kernel and without disrupting the real-time properties of Fiasco. Consequently, our port of Linux to L4 is called L^{4}Linux [3].

2.2 Real-Time Support

Next to the L^{4}Linux personality for non-real-time tasks, our system provides a real-time personality. Every basic resource such as CPU time and main memory is wrapped by a manager which provides the resource to real-time and non-real-time system components and applications. Using this manager, real-time components like a filesystem can provide an interface with reservations and guarantees for real-time applications and a best-effort interface for L^{4}Linux. This whole architecture on top of our Fiasco microkernel is named the "Dresden Real-time OPerating System", in short DROPS [4].

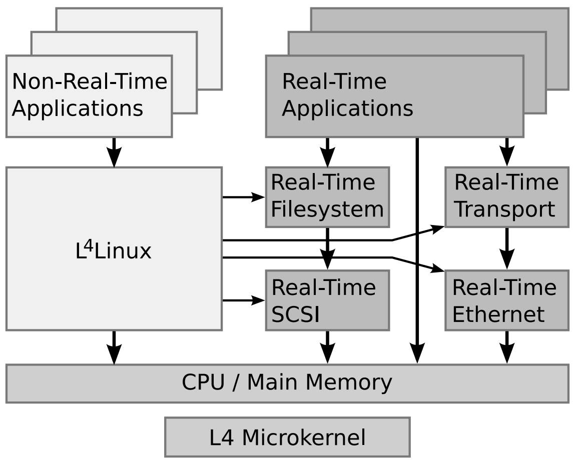

FIGURE 1: *The DROPS Architecture. [4]*

The key advantage of our design over the approach taken by QNX is that we can use the original Linux implementation with only few modifications. Thus, it is a lot easier for us to maintain a current and compatible version of our L^{4}Linux legacy OS personality. And other than RTLinux, our DROPS architecture allows for strong separation of real-time tasks from each other and of real-time tasks from the kernel, because those tasks all run in their own, isolated address space.

2.3 Paying the Price

Unfortunately, the separation of non-real-time tasks, real-time tasks and system servers into their own address spaces comes at a price: Every time an application wants to access a service from another application or server, interprocess communication across address spaces is required and this causes numerous additional context switches. This is one of the major reasons why many people in the OS research community argued that the layer of abstraction provided by a pure microkernel is too low to sustain acceptable performance. However, the non-real-time applications running on top of and therefore heavily communicating with the L^{4}Linux server are a good example to prove otherwise. A performance decrease of a Linux application running on L^{4}Linux instead of native Linux is expected, so we used the AIM multiuser benchmark suite VII to quantify the slowdown [3]. The benchmark tests, how well multiuser systems perform under different application loads. Figure 2 compares monolithic Linux with L^{4}Linux. To compare the performance of Linux on different microkernels, results for an in-kernel and a user-level version of MkLinux, a port of Linux to the Mach microkernel, are also listed. The numbers were obtained in 1997 on a 133 MHz Pentium.

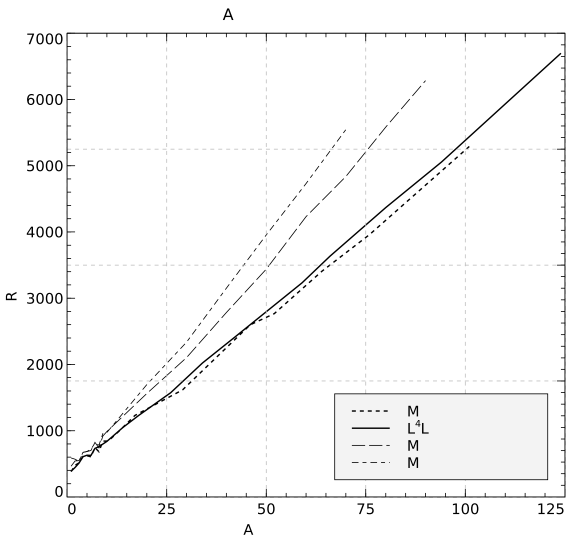

FIGURE 2: *Time per benchmark run depending on AIM load units. [3]*

Averaged over all loads, L4Linux is 2.2 % slower than native Linux. User-mode MkLinux is on average 29 % slower than native Linux, the co-located in-kernel version of MkLinux is 21 % slower. This demonstrates that L4Linux performs sufficiently close to native Linux, even under high load. Typical penalties range from 2 % to 10 %. The comparison with MkLinux shows, that the performance of the underlying microkernel has a profound influence on the performance of the applications.

One of the goals of our architecture was to ensure guarantees for the real-time tasks even when they are running next to L4Linux. Some additional modifications to L4Linux are necessary to achieve this [5]: Special measures must be taken to tame the locking mechanism of Linux so it cannot disable interrupts any more. We confirmed the effectiveness of our solution by measuring the actual periodicity of an L4 real-time tasks that requests a 100 ms period from the system. Running standalone on L4, the period length deviates by about 1 to 7 μs. With the untamed L4Linux, entire periods are lost because response times increase beyond the 100 ms bound. This is because untamed L4Linux can disable interrupts and thus prevent any scheduler activity for arbitrarily long intervals. Our tamed version of L4Linux however shows deviations of 24 ms, so the response times do increase compared to a real-time task running standalone on L4, but the periodicity of 100 ms can be supported. These results were obtained in 1998 on the original L4 implementation by Jochen Liedtke and motivated the development of Fiasco as a real-time time microkernel to improve the response times.

To further evaluate the real-time performance of our system, we wanted to compare it against RTLinux. With our DROPS system using a separate address space for each real-time task to increase fault-tolerance, a degradation of response times is expected compared to RTLinux, which runs all real-time tasks as kernel-level threads. To compare both systems, we developed the L4RTL library, which implements the RTLinux API on DROPS and measured interrupt response times on 1.6 GHz Pentium 4 [6]. To actually get worst case behavior, we ensured that caches and TLBs were always cold when an interrupt occurred. The measurements yield a worst case latency of 24 μs on RTLinux and 33 μs on DROPS. This shows that the cost of using address spaces for real-time tasks is not significantly larger than uncertainties introduced by dirty caches or blocked interrupts, which designers of real-time systems seem to accept readily.

We summarize that modifying Linux to run on top of our L4 microkernel Fiasco allows a performance close to native Linux without compromising the real-time properties of the system. Using separate address spaces for real-time tasks increases the fault-tolerance of the system without a significant impact on response times. This allows running real-time and non-real-time applications side by side, which we believe to be a key feature for today's computing requirements. With the presented system architecture, we can also support tasks that want to use both real-time and non-real-time services. Those hybrid tasks can communicate with the L4Linux part and with the real-time resource managers.

2.4 IPC Scheduling

Because a lot of system services are separated into dedicated servers, clients are required to communicate often. This drives the need for component interaction that is both fast and predictable. Such interaction is performed on L4 via synchronous interprocess communication (IPC). A successful message transfer requires a rendezvous between the sending and the receiving thread. However, synchronous message passing also introduces dependencies among components. Combined with fixed-priority scheduling of threads, this can lead to the infamous priority-inversion problem:

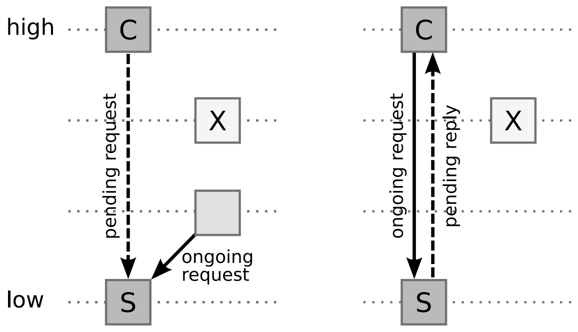

FIGURE 3: *Priority inversion during message passing. [7]*

The example shows a high-priority client C requesting a service from a low-priority server S. Because it is currently engaged in communication with another client, S cannot respond immediately. Before S can answer C, it is preempted by a medium-priority thread X, which leads to the high-priority client C waiting for X although X's priority is lower and although C has no communication relationship to X. Well known methods to avoid such a priority inversion are priority inheritance and stack-based priority ceiling. The basic message passing mechanism of our system must support those methods without sacrificing performance.

Our approach is to divide the thread context that is held in the kernel into two separate contexts:

- an execution context that keeps track of saved CPU registers and the thread state and
- a scheduling context which represents a time quantum coupled with a priority.

Thanks to this separation, the kernel can switch both contexts independently, which enables the implementation of a time donation scheme named capacity-reserve donation from one thread to another [7]: The kernel tracks references to the current execution context and the current scheduling context. On regular thread switches, the kernel will switch both contexts. However, when a request is sent from a client to a server thread, the kernel switches only the execution context, but keeps the client's scheduling context. In doing so, the client effectively donates its time quantum and priority to the server for the time the server executes the request on behalf of the client. When the server replies, the kernel switches back to the client's execution context and the client reobtains its donated scheduling context. If a server needs to contact another server to fulfill a request, the scheduling context donation is transitive. This scheme allows for fast message passing, because scheduling is eliminated from the critical path by directly switching from the sender to the receiver of a message with no priority changes. The consumed CPU time is always correctly accounted to the client that originated the request.

With capacity-reserve donation, the priority inheritance and stack-based priority ceiling protocols can be implemented efficiently. This elegantly solves the priority inversion problem and makes component interaction more predictable, while preserving the high performance of the interprocess communication primitive of the underlying microkernel.

3 Probabilistic Scheduling

As the basic architecture of the system is now in place, it is time to consider resource scheduling in real-time systems. The primary resource real-time system designers focus on is CPU time, because it provides the basis for all subsequent resource accesses. Scheduling the CPU is all about timeliness: Real-time applications provide a deadline and require sufficient CPU time to finish their job before the deadline is reached. The system scheduler must guarantee this property to all real-time tasks it admitted to run. However, the key problem here is how to determine, what "sufficient CPU time" means. Today's hardware makes a lot of effort to speedup applications in the average case by using caches to exploit locality in the application's behavior. Unfortunately, this widens the gap between the average case and the worst case time consumption. In addition, a common real-time application in desktop computing is video playback, which per se does not have a fixed execution time per job.

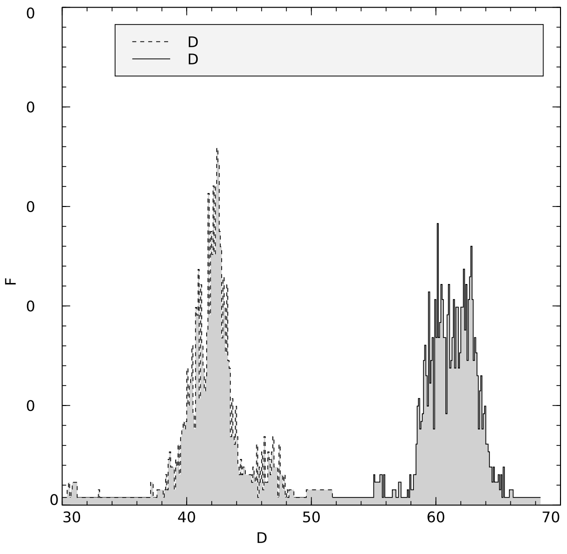

FIGURE 4: *Measured distribution of total decoding times per group of pictures. [8]*

These two factors cause execution time distributions to have a long tail. If the CPU resource was always allocated for the worst case, the system's utilization would be very low, because only few jobs can be admitted and large amounts of resources would be wasted. But media applications are an example for a class of real-time tasks that can tolerate occasional deadline misses, if this does not happen too frequently. With this observation, we devised a system that can handle overload predictably without dedicating enormous amounts of resources [8]. Our idea is to allow a percentage of deadlines to be missed; applications can configure this percentage as a quality level. Resource reservation is based on the distribution of the execution time instead of just the worst case value. Competing approaches in this area such as Imprecise Computation [9] and Statistic Rate Monotonic Scheduling [10] are either based on deterministic duration of resource usage or cannot guarantee a desired quality.

The task model of our Quality-Assuring Scheduling (QAS) allows each real-time task to be split into mandatory and optional parts. The mandatory parts are always guaranteed to be executed before their respective deadline, so worst case reservation is performed. Of the optional parts only the percentage requested with the quality parameter is guaranteed to be completely executed before the deadline. For these parts, admission and reservation is performed using the distribution of the execution time. Caused by the typical long tail of these distributions, even requested qualities only slightly below 100 % cause considerably less resources to be reserved, which greatly increases the overall utilization of the system. Although the scheduling is probabilistic, the requested quality levels are matched quite accurately, as the following table proves. The results were obtained for MPEG decoding with the I- and P-frames as mandatory parts and the B-frames as optional parts with the given quality.

Requested Quality	Reservation Time for Optional Parts	Achieved Quality
0.95	55 ms	0.9506
0.90	53 ms	0.8588
0.80	47 ms	0.7875
0.70	39 ms	0.6740
0.60	32 ms	0.5804
0.40	23 ms	0.4063
0.20	9 ms	0.2451

TABLE 1: *Requested quality, derived reservation time and measured quality of the optional parts. [8]*

Unfortunately QAS' applicability is limited because it only handles periodic tasks with uniform and harmonic periods and the admission is expensive, especially when the distributions are to be calculated with a high resolution. The model can handle arbitrary periods as well, but then the admission cost is increased beyond practical applicability. Therefore, the designated successor of QAS is QRMS, the Quality-Rate-Monotonic Scheduling. It simplifies QAS by assigning the mandatory and optional parts a unified reservation time, which is regarded as constant in the admission control. Thus, QRMS ignores situations where jobs do not completely consume their reservation. QRMS is therefore more pessimistic than QAS, but has a tremendously simpler admission even for arbitrary periods. The results convince of the accuracy and feasibility of the model:

Requested Quality	Quality Achieved with QAS	Quality Achieved with QRMS
0.70	0.7001	0.7024
0.50	0.5019	0.6742
0.7323	0.7326	0.7324

TABLE 2: *Requested and achieved quality of QAS and QRMS for a task system with three concurrent tasks.*

Another interesting property of both QAS and QRMS is that applications can be notified when the optional parts overrun their deadline. This way, an application can react, for example by reducing quality:

```
set_period(period);
reserve_time(mand_time, mand_priority);
reserve_time(opt_time, opt_priority);
do {
  begin_period();
  try {
    do_something();
  } catch {
  exceeded:
    adjust_quality();
  }
  next_reservation();
  try {
    do_something_else();
  } catch {
  exceeded:
    discard_result();
  }
} while (!end);
end_period();
```

With these unified admission and scheduling schemes, we can give probabilistic guarantees for periodic real-time tasks. Considering the variation of execution times allows us to admit far more applications and thus achieve better resource utilization than in systems based on worst-case admission.

4 Resource Management

The previous chapter dealt with scheduling the CPU, but this is not the only resource a real-time application might need. In our DROPS system architecture, all resources used concurrently by multiple tasks must be encapsulated and scheduled by a resource manager running as a server in user-land. In the following, the design of such managers for resources like disk, network and graphics bandwidth is presented.

4.1 Disk Requests

Disk usage in modern systems combines traditional best-effort file access with storage and retrieval of real-time streams, such as audio and video data. The former has relatively weak requirements while the latter must meet deadlines for individual disk requests. For good overall performance, the disk-request scheduler has to optimize the disk utilization as well. This is challenging, because the construction of disk drives causes a poor ratio of average and worst-case execution times. However, the same idea that has been successfully applied to CPU scheduling as discussed in the previous section also helps here: If an application can tolerate occasional deadline misses, probabilistic service guarantees can substantially improve the disk utilization compared to guarantees based on the worst case [11].

Thus, the basic idea is again to split real-time disk requests into mandatory and optional requests and to assign a quality parameter to the optional requests, which denotes the percentage of requests that must be completed. To optimize utilization, requests should be scheduled with the SATF (shortest access time first) algorithm, which is aware of the position of the drive's head on the disk. However, this scheduler does not know anything about deadlines. But instead of implementing a new scheduler, we devised a method to decouple the scheduling of the disk requests from the deadline and reservation enforcement [11]: The Dynamic Active Subset (DAS) always includes all pending requests that can be executed in any order without violating any deadline or reservation. This subset of disk requests is recalculated after every request completion and if enough time is available, the set even includes non-real-time requests to increase utilization. The SATF scheduler or any other scheduler can be run on this set to pick the request to execute next without having to know about deadlines.

With this technology, the disk request scheduler matches the desired quality levels of the tasks.

Bandwidth	Requested Quality	Achieved Quality	Achieved Bandwidth
640 KB/s	0.99	0.9973	638.54 KB/s
2560 KB/s	0.95	0.9798	2509.12 KB/s
1280 KB/s	0.90	0.9444	1209.36 KB/s
640 KB/s	0.85	0.9004	576.44 KB/s
1280 KB/s	0.60	0.6705	858.65 KB/s

TABLE 3: *Requested and achieved quality for a disk (IBM Ultrastar 36Z15) loaded with five concurrent streams. [11]*

Even quality levels only slightly below 100 % push the disk utilization close to the peak best-effort bandwidth.

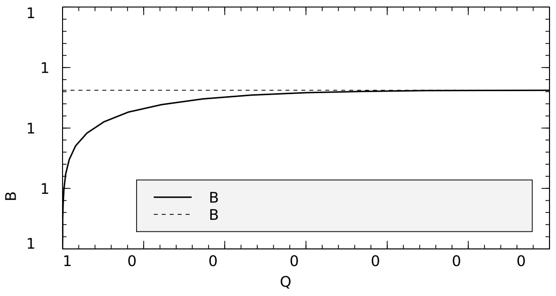

FIGURE 5: *Bandwidth that can be assigned to an optional stream. [11]*

4.2 Ethernet Transmission Delays

With the deployment of switches, Ethernet as the most widely used commodity network becomes interesting for real-time communication. Each port of a switch provides its own collision domain, so collisions do not occur in a star topology network. However, switches generally lack traffic policy features. Thus, if too many Ethernet frames are being sent to a machine that does not receive them fast enough, the switch will enqueue the frames internally, which causes transmission delays. If the internal queueing storage of the switch is depleted, it will even drop frames.

A mathematical model of the network traffic can be used to predict the buffer fill levels in the switch. If the nodes within the network cooperate, this model can be used to parametrize a traffic shaper running on each node that keeps buffer lengths and thus transmission delays within specified bounds [12].

Shaping Interval	Buffer Bound	Calculated Max. Delay	Observed Max. Delay
10 ms	111.8 KB	9357 μs	8759 μs
1 ms	15.7 KB	1380 μs	1300 μs
100 μs	6.1 KB	582 μs	438 μs

TABLE 4: *Buffer bounds in the switch and transmission delay bounds. [12]*

The achievable delay bound mainly depends on the granularity of the traffic shaping. This results in a trade-off between delay bound and CPU load.

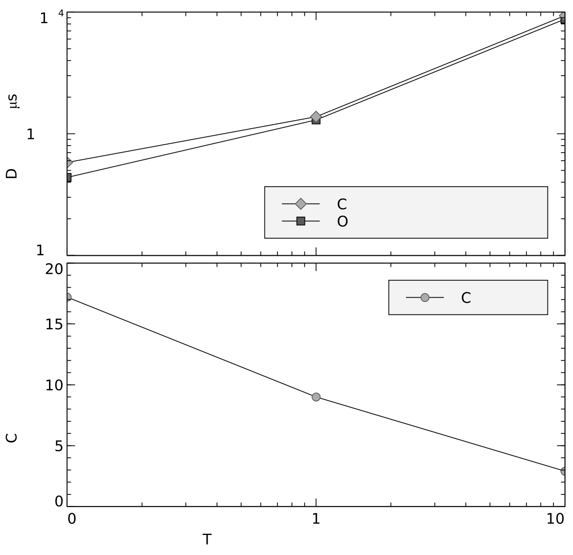

FIGURE 6: *Delay bound / CPU load trade-off. [12]*

Because all nodes on the network must cooperate to ensure the guaranteed delay bounds, each node must run an instance of the traffic shaper. However, not all nodes must run a real-time operating system. The shaping capabilities of the machines influence the delay bound, but we successfully shared the network with Linux machines while still observing predictable delays.

4.3 Screen Real Estate and Screen Updates

Today's modern desktops feature a graphical user interface. Furthermore, modern real-time applications like media players also feature a graphical output. This drives the need for a real-time capable window manager that can provide guaranteed redrawing rates for the real-time windows while providing best-effort services to the remaining non-real-time windows and auxiliary operations such as the user reordering windows. Therefore, the design goal of our DOpE (Desktop Operating Environment) window server [13] was to multiplex the singleton resource of physical screen real estate to client applications. For real-time clients, quality of service is guaranteed even in overload situations, which can be caused by massive screen updates of non-real-time applications. DOpE can therefore sustain real-time client windows running next to L^{4}Linux and X11 on the same desktop.

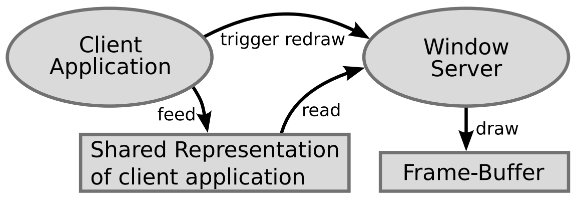

FIGURE 7: *Design of the DOpE window server. [13]*

The architecture of DOpE separates the client's updates to the user interface from the server updates of the representation on screen. The client and the server share a description of the layout and content of the client's user interface. This allows the client to update the shared description without interference of the server and then trigger a redraw operation. The server can then interpret the shared window description and perform the necessary updates to the on-screen representation independently of the client. Because the execution time of such a redraw is known beforehand, the window server can guarantee previously negotiated refresh rates to admitted real-time clients. A real-time client can subscribe to periodic notifications of completed redraw operations. Updating the shared representation in a timely manner is entirely the responsibility of the client.

This separation of cause and execution of redraw requests allows us to display real-time graphics and windows of non-real-time clients seamlessly side by side.

4.4 Second-Level Cache

One easily overlooked resource used concurrently by real-time and non-real-time tasks are CPU caches. They are an especially interesting resource for real-time applications, because every task switch potentially disrupts cache working sets and thus makes execution times unpredictable. To avoid this, the CPU caches should be managed like all the other resources discussed above to isolate the real-time tasks from cache interference by other tasks or the operating system. A well-known solution for this problem is cache partitioning: portions of the cache are dedicated exclusively to specific applications. For our system, we developed a cache partitioning technique that operates without any hardware modifications [14].

A page size of 2^p divides the cache in banks of 2^p bytes, if the cache is direct-mapped. The least significant p bits are used to index an element within such a bank. Assuming a cache size of 2^c, the next $c-p$ bits in the address select the cache bank. The remaining part of the address is compared against the tag. For an n-way set-associative cache, a cache size of $n2^c$ and a bank size of $n2^p$ are to be used. The division of the cache into banks also divides the main memory into classes, whose physical page frames all fall into the same cache bank. Those classes are called colors. Cache conflicts can only occur between page frames of the same color, so such conflicts can be avoided between any two tasks, if both tasks use disjoint colors. Since the L4 microkernel allows user level memory management, the mapping of physical to virtual addresses can be controlled by a memory server that assigns colors to tasks exclusively.

The problems with this approach are: Being based on the mapping of pages, it can only be applied to physically-indexed caches and only with page granularity. Additionally, if a certain percentage of the cache is to be dedicated to a task, the same percentage of the main memory is implicitly reserved for that task as well. On the other hand, the technique provides a way to close the gap between the average case and the worst case execution time for real-time tasks. This greatly helps when scheduling real-time tasks with hard deadlines, because less CPU resources need to be reserved.

5 Conclusion

With the Fiasco real-time L4 microkernel and the real-time enabled managers for various system resource, the DROPS system described in the previous sections provides all the building blocks for writing real-time applications. Legacy support for running non-real-time software and real-time tasks side by side is provided by the L^{4}Linux server. The Quality Assuring Scheduling and Quality-Rate-Monotonic Scheduling provide the mathematical foundation to handle overload situations

5.1 Real-Time in Software Development

However, what is still missing is a comprehensive way to open this technology to software developers. All the elegant solutions and advancements in real-time systems research are of limited use, if they are not accessible to the engineers in need. To this end, a joint team of members from our research group and from the software technology group of our department developed the COMQUAD component architecture. This architecture allows to specify non-functional properties like quality levels and resource usage of a component implementation in the component quality modelling language (CQML+). These properties are then used to derive contracts between components which are translated by the component runtime environment into resource reservations. Our real-time operating system and its resource managers enforce these reservations at runtime. This way, a component-based software development process was created, that supports adaptive real-time systems from specification all the way to the running system [15].

5.2 Current State and Outlook

Most of the software discussed here is available for download [16] under the terms of the GNU General Public License. The resource managers are still prototypes, but the foundation of the system is usable. A demo CD is available as well [17]. We hope to spark a wider interest amongst operating system enthusiasts for design, implementation and deployment of real-time systems on everyday computers. With the knowledge we gained from the DROPS architecture, we are currently exploring new ground in the areas of virtualization and hierarchical system design. We expect to present more fascinating research results and we will always ensure that our systems are designed to be useful beyond mere academic purposes.

Acknowledgements

The authors would like to thank the researchers who contributed all the findings that went into this overview paper, including but not limited to (list in alphabetical order): Ronald Aigner, Robert Baumgartl, Martin Borriss, Norman Feske, Claude-Joachim Hamann, Michael Hohmuth, Jochen Liedtke, Jork Löser, Frank Mehnert, Martin Pohlack, Lars Reuther, Sebastian Schönberg, Udo Steinberg, and Jean Wolter.

References

[1] D. Hildebrand: An architectural overview of QNX. In *1st USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, Seattle, Washington, April 1992.

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

[3] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, 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] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, J. Wolter: DROPS: OS Support for Distributed Multimedia Applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

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

[6] F. Mehnert, M. Hohmuth, H. Härtig: Cost and benefit of separate address spaces in real-time operating systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 124–133, Austin, Texas, December 2002.

[7] U. Steinberg, J. Wolter, H. Härtig: Fast Component Interaction for Real-Time Systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*, Palma de Mallorca, Balearic Islands, Spain, July 2005.

[8] C.-J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, H. Härtig: Quality-Assuring Scheduling – Using Stochastic Behavior to Improve Resource Utilization. In *Proceedings of the 22th IEEE Real-Time Systems Symposium (RTSS)*, London, UK, December 2001.

[9] K. J. Lin, S. Natarajan, J. W. S. Liu: Imprecise results: Utilizing partial computations in real-time systems. In *Proceedings of the IEEE Real-Time System Symposium*, 1987.

[10] A. Atlas, A. Bestavros: Statistical Rate Monotonic Scheduling. *Technical Report 98-010*, Boston University, May 1998.

[11] L. Reuther, M. Pohlack: Rotational-Position-Aware Real-Time Disk Scheduling Using a Dynamic Active Subset (DAS). In *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS),* Cancun, Mexico, December 2003.

[12] J. Löser, H. Härtig: Low-latency Hard Real-Time Communication over Switched Ethernet. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, Catania, Italy, June 2004.

[13] N. Feske, H. Härtig: DOpE – a Window Server for Real-Time and Embedded Systems. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, December 2003.

[14] J. Liedtke, H. Härtig, M. Hohmuth: OS-Controlled Cache Predictability for Real-Time Systems. In *Proceedings of the Third IEEE Real-time Technology and Applications Symposium (RTAS)*, pages 213–223, Montreal, Canada, June 1997.

[15] H. Härtig, S. Zschaler, M. Pohlack, R. Aigner, S. Göbel, C. Pohl, S. Röttger: Enforceable Component-Based Realtime Contracts – Supporting Realtime Properties from Software Development to Execution. To appear in the *Real-time Systems Journal.*

[16] `http://os.inf.tu-dresden.de/drops/`

[17] `http://demo.tudos.org/`