DROPS
OS Support for Distributed Multimedia Applications

Hermann Härtig Robert Baumgartl Martin Borriss Claude-Joachim Hamann
Michael Hohmuth Frank Mehnert Lars Reuther Sebastian Schönberg Jean Wolter

Dresden University of Technology
Department of Computer Science
D-01062 Dresden, Germany
email: haertig@os.inf.tu-dresden.de

Abstract

The characterising new requirement for distributed multimedia applications is the coexistence of dynamic real-time and non-real-time applications on hosts and networks. While some networks (*e.g.,* ATM) in principle have the capability to reserve bandwidth on shared links, host systems usually do not. DROPS (Dresden Real-time OPerating System) is being built to remedy that situation by providing resource managers that allow the reservation of resources in advance and enforce that reservations. It allows the coexistence of timesharing applications (with no reservations) and real-time applications (with reservations). By outlining the principle architecture, some design decisions, and first results, the paper demonstrates how these objectives can be met using straightforward OS technology. It argues that *middleware for diverse platforms* cannot meet these objectives efficiently without proper core operating system support.

1 Introduction

The following observations and objectives are guiding the design of DROPS (Dresden Real-time OPerating System):

- Real-time systems used to be dedicated systems, hence had complete computing systems for themselves. With the advent of multimedia applications, real-time and non-real-time applications (we will use the term “time-sharing applications”) share computer cores, disks, the video subsystem, and networks. The real-time applications are dynamic in two aspects. On the one hand, new applications are started and others terminated from time to time. On the other hand, some application types can adapt to changing resource situations. Traditionally, the problems caused by this situation are either dealt with not at all, or by spending enormous amounts of resources. DROPS attempts to organise the coexistence of real-time and time-sharing applications by providing managers for all resources that can reserve them for real-time applications, enforce these reservations, and leave the rest to the time-sharing applications. Resource managers have means to notify applications about changes in the availability of resources.

- Advances in computer architecture achieved enormous performance improvements for time-sharing systems by exploiting observed locality of behaviour in such applications. Many of the employed techniques are not useful for real-time systems, where not the average case counts but the worst case (take caches as an example). Hence, many dedicated real-time systems use expensive and less powerful architectures. DROPS attempts to make simple powerful standard technology predictable.

- The system must achieve a degree of maturity that makes it usable on daily basis, for the builders as well as for outsiders. This causes a lot of “unscientific” work, but enforces honesty. Hence, a widely available time sharing API must be provided and its implementation must be efficient and maintainable. DROPS provides a Linux ABI as the standard time-sharing interface. Design and implementation decisions must be guided by their effect on achieving the objectives, not by novelty or extravagance.

Contrasting common belief in middleware technology, the key to meet the first two objectives in the authors’ opinion lies in the core of the host’s operating systems, since otherwise—that is, without support at the memory management or driver level—neither reservations nor efficient use of modern hardware architectures are possible.

The remainder of this paper points out how the mentioned objectives are pursued in the DROPS system using straightforward core OS technology. It describes the architecture and some components; performance results and other experiences are given. It concludes by arguing that none of these techniques can be properly applied at the middleware level and hence the objectives cannot be addressed there unless enormous amounts of resources are wasted. A proper discussion of related work is omitted due to lack of space.

2 The DROPS Architecture

DROPS borrows from the (almost ancient) principle to provide multiple “OS personalities” based on virtual machines or microkernels. It runs a time-sharing personality (a modified Linux kernel [8]) as a user level program side-by-side with a real-time personality. It has a manager for basic resources to provide CPU time, second-level cache and main memory to real-time and time-sharing system components and applications. Using the basic resource manager, real-time components, for instance the driver for the SCSI subsystem, provide interfaces for both L^{4}Linux and the real-time applications, one interface with reservations and guarantees, the other without.

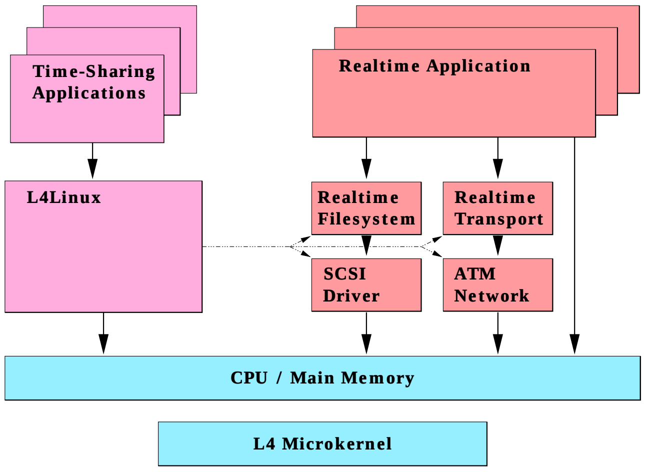

Figure 1: The DROPS Architecture

In the subsequent subsections, we show an example scenario and specify the general interface of the real-time components.

2.1 An Example Scenario

Figure 2 describes a scenario which is typical for a multimedia (real-time) application as those DROPS is being built for.

A SCSI-based file containing compressed video is fed into a decoder and then on to a presentation and a transport component. The presentation component controls the current position of the video, the transport component moves the video to another host. To start such an application, the resources needed for all components need to be reserved. During the runtime of the application, the reserved resources must be guaranteed to the components.

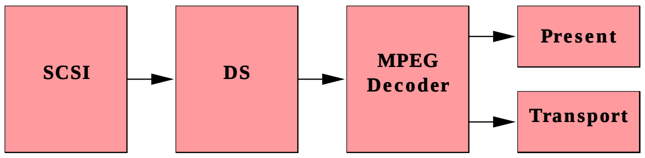

Figure 2: A Real-time Scenario

2.2 The Interface of RT Components

It is well established that physical copying operations are costly and must be avoided, at least for larger amounts of data. Hence, DROPS uses virtual memory management techniques similar to fbufs [5], extended however by time-interval-stamps for the validity of memory regions.

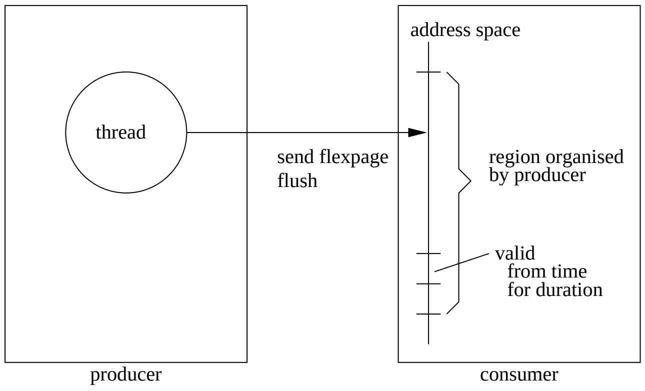

Figure 3: Timed Fbufs

An application producing a data stream for a consumer application organises a region of the consumer’s address space. For each part of that region a time-and-duration-stamp, hence a time-interval-stamp (TIS), specifies the period of time during which that part is valid. The producer is responsible to provide the data in time, the consumer to use them within the specified period. After the specified time interval the region is flushed. If the region offered by the consumer is not large enough the data stream is wrapped around.

To access a data stream, the consumer requests a reservation from the potential producer by providing as parameters

- the bandwidth needed,
- the duration of the complete stream,
- the duration of each unit of the stream as needed by the consumer,

- and the approximate starting time.

If the producer is capable of delivering the stream, admission is granted, otherwise it is rejected. Once an admission is granted, the data stream can be started. The start operation takes a requested starting time as an in parameter and the real starting time as an out parameter. Started streams may be requested to stop and restart. As in start operations, restart takes the requested restarting time as in, the real restarting time as out parameter.

After the start of a stream, the producer selects the size of the units in which the stream is provided to the consumer. The real starting time and the duration parameters are used to calculate the time-interval-stamp for each unit of the stream. Units are provided to the consumer before their time interval and flushed thereafter. The implementation is based on sending, receiving, and flushing “flexpages” [9]. Flexpages are roughly similar but much more efficient then Mach’s out-of-line messages. A producer acts as an external pager for its consumer.

A generalized feed back mechanism to allow notification of application programs regarding the availability of resources based on reservation priorities as described for processes is still under construction.

3 Coordination of Resource Scheduling

One of the generally hardest problem is the coordination of the reservation and scheduling of the various envolved resources. A commonly employed technique is to decouple components using buffers of messages that increase latency. Then, latency critical events need higher responsiveness than others. This section describes a framework to compute necessary buffer sizes based on a mathematical description of streams of events and a CPU scheduling model that allows to meet the responsiveness requirements.

3.1 Jitter Constrained Periodic Streams

The interface as roughly described in Section 2.2 allows to specify the duration that each unit of a stream is requested to be maintained in the consumer’s address space. This duration is determined by the requesting consumer based on the estimated processing time and on the jitter in processing the data. Jitter may be caused by application dependent interpretation of the content of the stream (e.g., differing frame sizes in MPEG encoded stream), differing processing times for units of the stream and the preciseness of CPU scheduling. The larger the jitter the larger the time interval needed.

The model used for DROPS describes such a stream as a sequence of events E_i. Beginning from a starting point t_0 the events occur principally with a period T, but they may vary over a given interval: events may occur τ time units too early or τ' time units too late as long as they obey a minimum distance $D < T$ (see Fig. 4).

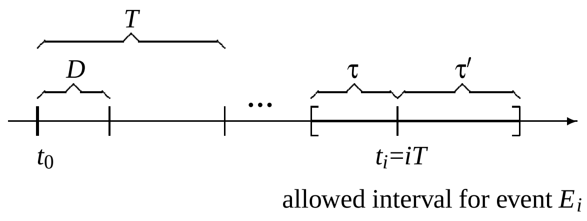

Figure 4: Jitter constrained periodic event stream

Based on a formal definition Hamann [7] proved the equivalence of numerous sets of parameters that are used to specify the jitter of streams and provided formulas to compute the duration parameters needed, mainly the maximum burst size L and the lower bound P of buffer size to avoid loss of data:

$$L = 1 + \left\lfloor \frac{\tau + \tau'}{T - D} \right\rfloor$$

$$P = 1 + \left\lceil \frac{(L-1)(T-D)}{T} \right\rceil$$

Furthermore, the model enables to transform different sets of parameters describing jitter constraint streams into each other.

3.2 CPU Scheduling

To process units of streams in time, the availability of CPU cycles must be guaranteed. However, jitter constrained streams have quite different requirements regarding the responsiveness (latency) with which the buffered units are processed.

Hence, these three requirements are to be met by a scheduling scheme:

- The number of cycles needed within a period is to be reserved for the process in question. A process exceeding its guaranteed CPU time is to be notified.

- Within the limits of their reserved cycles events needing higher responsiveness need to have priority over other processes. Prime examples are device driver processes, since precise prediction of I/O interrupts is not possible in general and—for some hardware—fast responsiveness is required.

- Processes need ways to reserve cycles in at least two levels, one expressing the resources that are guaranteed under all circumstances the other on *nice to have* basis. Processes need to be notified when *nice to have* cycles become available or unavailable to allow adaptation.

Static reservation of CPU time for a given process and time interval as employed in the MARS system [11] or in principle in [8] or EDF based scheduling is much to inflexible due to the impossibility to precisely predict I/O interrupts. Classic priority based systems do not offer enough flexibility to allow programmers to influence scheduling. Capacity Reserves [13] provide a notion to reserve processor shares for tasks subject to time constraints. Tasks with reservations are given priority over other tasks within the limits of their reservation. Capacity reserves don't provide a feedback mechanism allowing to allow applications to adjust to the current system load.

The scheme being currently investigated for DROPS employs reserved priorities to achieve the mentioned objectives. It allows a process to reserve a priority for a given number of cycles within a period. If a process has used up its priority cycles, its priority is decreased and the process is notified. Thus, the process is guaranteed to have the reserved priority for the specified time period, but not for longer as the specified number of cycles. If for instance an interrupt is known to appear in a certain period and the handler is known to run for n cycles, then a high priority is reserved for the driver. A lower priority can be reserved for another process within the same period and for m cycles. Admission control must check whether $n+m$ cycles are available for the processes during the period in question. The remaining cycles can still be used by lower priority time sharing processes.

An up-call mechanism is provided to notify processes in the case that they try to exceed the reserved number of cycles. This turns out to be useful to determine the numbers of cycles needed for certain processes and to enable scaling of applications.

A simplified MPEG player looks as follows. It first reserves a priority for a given number of cycles in a given period. Then it starts its computation. The begin (and implicitly the end) of each region is marked by calling the begin_region function. If the reserved number of cycles is exceeded a call back function is called to globally scale the application. By returning *false* from the begin_period function the application is notified of having exceeded the requested and granted number of cycles.

```
reserve(period, cycles, priority);
while (!end) {
    if (begin_period(call_back, event) {
        decode_picture();
    } else {
        skip_picture();
    }
}
release_reservation();

void call_back(reason) {
    if (reason == TIME) {
        reduce_quality();
    }
}
```

Figure 5: Example for the use of the process model

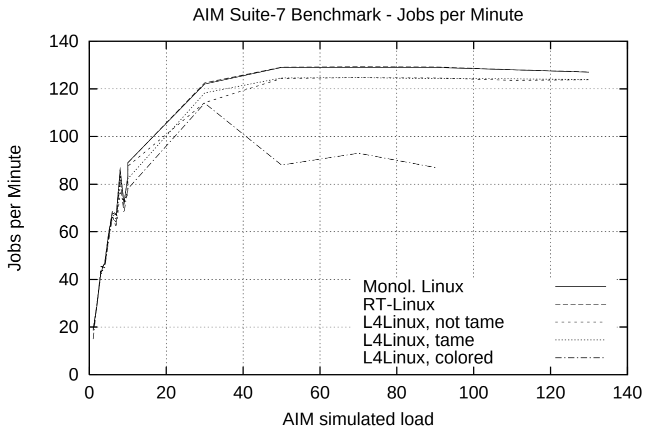

Figure 6: AIM Multiuser Benchmark Suite VII. Jobs completed per minute depending on AIM load units.

4 Some Components, Performance and Other Results

Some components of DROPS that have been built or are under construction are described next and performance data are given.

4.1 L^{4}Linux and its Taming

Linux has been ported to the L4 microkernel by modifying the scheduler and the architecture dependent part of the original Linux kernel. The design follows closely former attempts at Carnegie Mellon University [6] and at OSF [4]: System call traps are reflected to the calling process using a trampoline mechanism. The exception handler calls the Linux server process using IPC. Interrupts are mapped to IPC to top halves of device drivers running as threads. Driver threads are subject to standard L4 scheduling. Synchronisation of critical regions is done explicitly rather than by disabling interrupts [10]. Memory resources are divided between the L^{4}Linux subsystem and the real-time side.

To measure the effect of running Linux as a user level process and using explicit syncronisation we used 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.)

For macrobenchmarks like AIM, 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 %. Earlier measurements [8] showed a penalty of 8.3 %. While analyzing the exact penalties involved with L^{4}Linux we found a missing wakeup in our low-level interrupt handler leading to unnecessary idle times within the system. After fixing the bug the performance penalty went down to 2 %–4 %.

Tamed Linux reduces worst case interrupt latencies to about 20 μs on the Intel architecture [10].

4.2 ATM Subsystem

A prime example of a DROPS component is its native ATM protocol implementation. While ATM has been developed with strong emphasis on deterministic high-speed communication, current operating systems and networking protocols cannot yet fully utilize its potential. The DROPS component resolves this dilemma by providing operating-system-level mechanisms to allow for end-to-end guarantees on throughput, delay and jitter.

Enforcement and restriction of real-time guarantees essentially work as follows: Individual ATM connections have a dedicated associated *worker* thread. The worker has a certain number of credits per period and uses them up while handling incoming and outgoing packets. By suspending and rescheduling threads, strict separation of connections and its associated resources is achieved. The monitoring and policing scheme is based on dynamically configured measurement intervals and associated credits.

A crude approximation of the needed resources is based on the assumption that for a given packet size the CPU usage is proportional to the bandwidth and the that the maximum bandwidth on low end PCs is CPU limited. Then a given fraction of the maximum bandwidth is assumed to need the same fraction of the full CPU resource. Buffers for individual connection are sized depending on the requested class of service, application-requested delay variation tolerance and maximum packet size. Both, buffer space and estimated CPU-time are reserved during admission control.

To obtain a first indication for the performance, we measured the overhead implied by the real-time extensions resulted in a performance penalty of 1 %–2 %. For example, maximum achievable throughput dropped from 130.7 Mbps (traffic management disabled) to 128.5 Mbps (traffic management in effect).

Next, we looked at performance degradation imposed by a high bandwidth ATM connection on the AIM multiuser benchmark suite VII [1] running under L4Linux. Figure 7 shows the results. System performance degrades pretty close to the prediction using the crude resource estimation algorithm. Using its knowledge about CPU time needed for the 40 Mbps ATM connection, remaining system performance is estimated to be about 69 %.

Then, the ability of the system to shield guaranteed services from time-sharing services has been validated. Figure 8 shows that even under extreme load conditions, real-time guarantees are preserved. Total test duration was about 22 hours with about 5000 measurement points, each consisting of 10000 8kB-sized packets. After about 10 hours, AIM ran until cross-over, at which point the test machine started to run the RT-connection exclusively. In the example, throughput for a 40Mbps reserved transmit connection has been measured at a peer machine. Paradoxically, ATM throughput increases under high load. We account this effect to an anomaly in the L4 implementation we currently use.

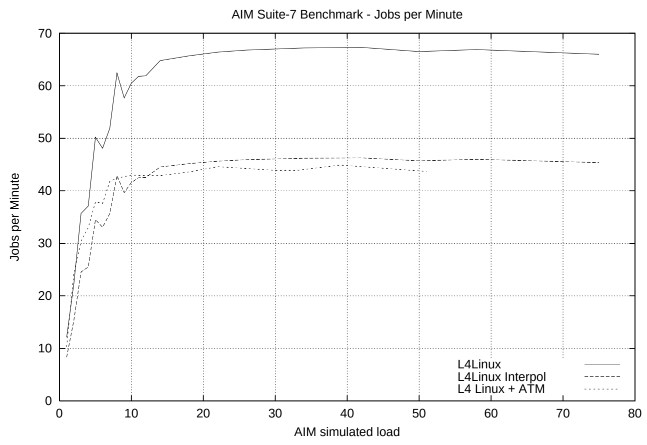

Figure 7: Achieved performance of the concurrently running time-sharing OS server.

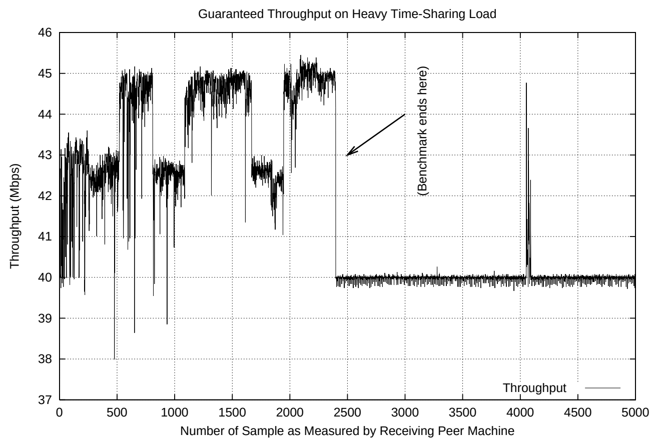

Figure 8: Measured throughput for guaranteed ATM connection under heavy concurrent load.

Besides an interface conforming to DROPS' real-time model (Section 2.2) it provides a standard BSD socket API for portability and acceptance, no assumptions on application's ability to adapt, transparent usage by the time-sharing subsystem and throughput preservation by the protocol. The protocol provides an API which is aligned to Linux' ATM implementation [2] and work of the ATM Forum. Implementation of the protocol server runs stable and is transparently being used by the time-sharing subsystem for all standard services (e.g., TCP/IP over ATM) .

The multi-threaded protocol server—as opposed to a single-threaded version with packet scheduler, implies several beneficial side-effects:

- Highly accurate rate control.
- Per-connection priorities to bound processing delays.
- Implicit flow control up to application level.
- Scalability of the traffic management schemes.
- High performance, since the techniques developed map well to the underlying microkernel's mechanisms.

Other design features include a zero-copy data transfer mechanism, based on DROPS' timed fbufs and cooperation with the ATM hardware driver's real-time capabilities.

Thus, it has been demonstrated that the mandatory operating-system-level support for guaranteed communication is possible. A novel approach using thread manipulation techniques allow for a efficient and low-overhead implementation in a microkernel environment. To the best of our knowledge, this yields the first end fully functional end system implementation of a predictable ATM-based protocol stack.

A detailed description of design and implementation of the DROPS ATM component is contained in [3].

4.3 The SCSI and File-System Components

The file system component offers three principle interfaces:

- a real-time interface as briefly described in section 2.2 to access a real-time file in real-time;
- a time-sharing interface to access a non-real-time file in a non-real-time fashion, i.e., without any known property with regard to bandwidth and allocation;
- a time-sharing interface to access a real-time file in a non real-time fashion, i.e., without guaranteed bandwidth but so that in write operations the file is allocated such that a later real-time access is enabled.

The SCSI component offers a real-time and a non-real-time interface.

To enable bandwidths greater than that of a single disk drive, files are striped across multiple drives. To enable reservations and guarantees for open files with a certain requested bandwidth the following scheme is employed:

Requests to the SCSI subsystem are issued such that the SCSI bandwidth can be fully exploited by dividing SCSI time into slots where the size of slots is determined by the worst case seek times of drives. To make a reservation for a file with a given requested bandwidth, the required number of slots are reserved. They may consist of either the slot belonging to a single drive or slots for several drives. If the requested bandwidth is low than every second or third slot or so (called subslots) belonging to drives is reserved.

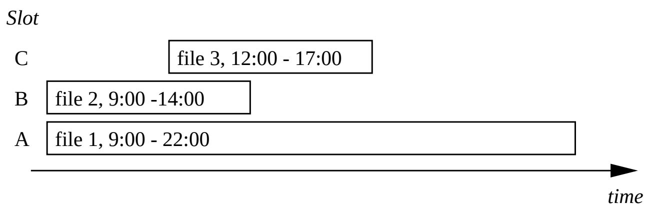

Figure 9: Reservation in the SCSI driver

Reservation is maintained by marking the period of time for which a slot is dedicated to an open file.

Reservation is enforced by maintaining a similar data structure for the actual I/O-requests. Slots that are reserved for a given file are marked as reserved and hence can only be occupied by I/O-requests that carry a capability for the slot. *Free* slots can be occupied by non-real-time I/O-requests.

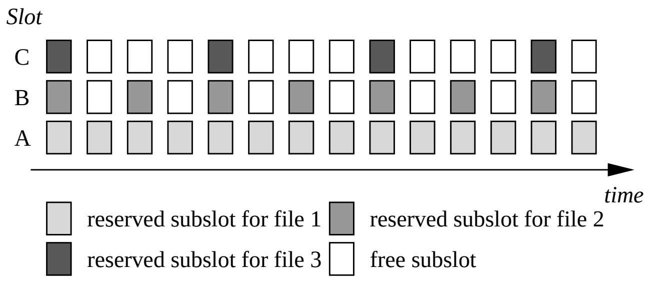
Slot
C
B
A
time
reserved subslot for file 1
reserved subslot for file 2
reserved subslot for file 3
free subslot

Figure 10: Scheduling of I/O-requests in the SCSI driver

The file systems maintains buffers to satisfy the requested duration for which the file must be present in the consumers address space.

4.4 A Resource Manager for 2nd-Level Caches

We have implemented a virtual memory management 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. [12, 14]

This leads to a significantly better predictable worst case behaviour for some applications. In one of our 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. [12]

The L^{4}Linux server running with a partitioned cache suffers a performance degradation. For instance, when using only one half of the cache, L^{4}Linux runs a simple compilation application at 9.8 % penalty. More details can be found in [10].

5 Conclusion

The essential techniques as described in the former sections are resource reservation that includes CPU cycles, memory management and the driver level. It has been demonstrated that realtime components can be effectively shielded from one another and from non realtime load. An example included a component to guarantee bandwidth for an ATM connection.

Neither reliable bandwidth reservation for file systems or network bandwidth nor CPU reservations for driver processes can be achieved without core-OS support. Hence, middleware needs to allocate enormous resources to achieve similar goals and it cannot achieve it reliably.

References

[1] AIM Technology. *AIM Multiuser Benchmark, Suite VII*, 1996.

[2] Werner Almesberger. Linux ATM API (Draft v. 0.4). Technical report, LRC Lausanne, 1996.

[3] 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`.

[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, February 1996. Free Software Foundation, 59 Temple Place, Suite 330, Boston, MA 02111. Available from URL: `http://www.gr.osf.org/~stephen/fsf96.ps`.

[5] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *14th ACM Symposium on Operating System Principles (SOSP)*, pages 189–202, Asheville, NC, December 1993.

[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] Cl.-J. Hamann. On the quantitative specification of jitter constrained periodic streams. In *MASCOTS*, Haifa, Israel, January 1997.

[8] 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/`.

[9] H. Härtig, J. Wolter, and J. Liedtke. Flexible-sized page-objects. In *5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 102–106, Seattle, WA, October 1996.

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

[11] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed fault-tolerant real-time systems: the Mars approach. *IEEE Micro*, 9(1):25–40, February 1989.

[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] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating systems support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, Boston, MA, May 1994.

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