Scheduling Server for Predictable Computing

—

an Experimental Evaluation*

Jan Richling

Department of Computer Science
Humboldt University of Berlin
D-10099 Berlin, Germany

Andreas Polze

Department of Computer Science
Humboldt University of Berlin
D-10099 Berlin, Germany

Abstract

Clusters of networked commercial, off-the-shelf workstations (COTS) are presently used for computation-intensive tasks that were often assigned to parallel computers in the past. However, since parallel computations and programs ran by interactive users have to share a workstation's resources (e.g., CPU cycles), mechanisms are needed which implement a fair policy for predictable resource sharing. In this paper we evaluate the scheduling server approach used in our SONiC (Shared Object Net-interconnected Computer) system which implements partitioning of CPU cycles between parallel tasks and interactive programs.

We present experiments which show the usability of the SONiC scheduling server for several types of applications. We evaluate the scheduling server's behaviour using a real-time multimedia application. Additionally, we discuss extensions to the scheduling server.

Although the current implementation of SONiC is based on the Mach operating system, we show applicability of the scheduling server's concept on other platforms. We briefly describe and evaluate a rtLinux-based implementation of the scheduling server principle.

Keywords: Predictable Execution, Resource Management, COTS-based Real-Time Services, Mach.

1 Introduction

Cluster computing on networked commercial, off-the-shelf (COTS) systems is becoming increasingly popular for performance reasons. However, a number of practical problems like cumbersome programming methods, uncoordinated resource management and unpredictable loads impede the performance gains. In this paper we evaluate the scheduling server approach, a method to overcome part of these practical problems.

Different cluster computing environments are available. Most of them assume sharing of workstations between parallel computations and interactive users. The main problem is to predictably partition the computing power of the workstation between user processes and the processes of the cluster computing environment. Both kinds of processes have different requirements: The interactive user wants to be undisturbed, and the cluster computing process needs a well determined, a priori known amount of computing power. Standard UNIX does not offer facilities for partitioning a workstation's resources among different tasks in a pre-determined fashion.

One solution to solve this problem is the scheduling server [1] developed for the SONiC cluster computing environment [2] [3]. This paper will evaluate the behaviour and practical usability of the scheduling server approach.

The remainder of the paper is organized as follows: Section 2 describes the main ideas of the scheduling server approach. Section 3 presents related work. Experimental scenarios as well as results of our evaluation can be found in Section 4, whereas Section 5 shows usability of the scheduling server for a real-world application. The paper is summarized and concluded in Section 6 which also describes future work.

2 The Scheduling Server Approach

2.1 Operating System Requirements

We have developed the "Shared Objects Net-interconnected Computer" (SONiC)[2][3] as a platform for execution of parallel programs in networked environments. SONiC provides an object-based distributed shared memory system together with syn-

*appeared in Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services, held in conjunction with 18th Real-Time Systems Symposium, San Francisco, USA, December 2-5, 1997.

chronization constructs and a remote execution service. SONiC's scheduling service allows execution of an application's sub-tasks in a pre-determined fashion. The current implementation of SONiC is based on the Mach operating system.

However, there are a number of general requirements which have to be fulfilled by an operating system in order of taking advantage of our scheduling server approach:

- support of kernel threads
- implementation of a fixed priority scheduling policy without aging
- support of explicit thread switches
- fixed priority threads which keep control over CPU until their quantum expires and a higher-priority thread becomes ready(adjustable quantum).

In the Mach operating system these qualities are implemented. Although there exist a number of different commercial and experimental Mach-based operating systems, experiments presented in this paper have been carried out on the NeXTSTEP 3.3 operating system.

2.2 The Scheduling Server and Time Slicing

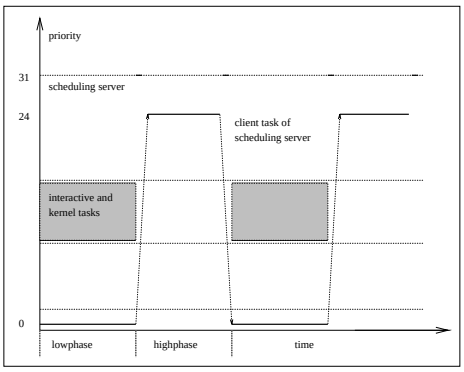

Figure 1: Scheduling Server

The scheduling server is a user-space task, which runs at the system-wide highest priority. Usually the scheduling server sleeps (voluntarily giving up control of the CPU) and allows for normal operation of the system. Tasks can register as clients with the scheduling server. Client tasks are scheduled using the fixed-priority policy. Normally they execute at a low priority (or at priority zero), leaving the rest of the system and interactive users' tasks undisturbed. From time to time the scheduling server picks one of its client tasks and raises the task's priority to the second highest value in the system. Then it suspends itself for a well known period of time — this way the scheduling server effectively dedicates a time slice to the selected client task. Later the scheduling server awakes and resets the priority of the client task, giving the rest of the system a chance to run again. Figure 1 demonstrates how priorities are manipulated by the scheduling server. Shaded rectangles represent the activity of the rest of the system.

2.3 Another Approach — the Load Partitioner under rtLinux

The operating system *rtLinux* [4] [5] is based on Linux and has the capability of running so called rt-tasks periodically. Based on the concept of rt-tasks, the idea described above can be implemented using periodic threads of different priorities.

We call our rtLinux-based implementation of the scheduling server idea *load partitioner* . The load partitioner runs at a period equivalent to the quantum in the scheduling server. It counts its runs modulo n (n is the number of periods which form a cycle — this corresponds to the period in the scheduling server approach) and enables or disables the client task - one per cycle. See figure 2 for details (n=8).

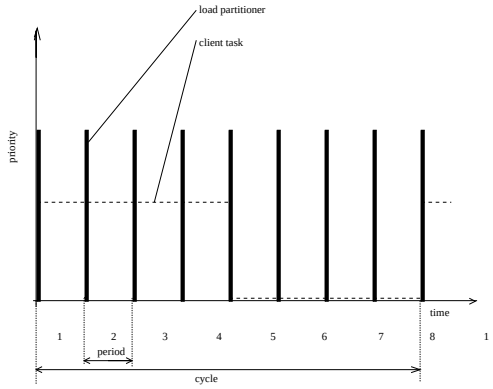

Figure 2: The Load Partitioner Idea

The main problem is that rtLinux is in a premature state and not fully implemented. Some workarounds are necessary to implement the load partitioner, and some restrictions apply. For instance the use of system calls in rt-tasks is nearly impossible, and the only method for communication between a rt-task and the rest of the system are the so called rt-fifos. Therefore currently only small experimental programs can be used as client tasks for the load partitioner.

3 Related Work

Experiments and measurements described throughout this paper are based on the SONiC scheduling

server [1] implementation. However, there exist several other papers which address the partitioning of computing power, mainly with the aspects *soft real-time* and *multimedia*.

Goyal, Guo and Vin [6] implemented the *Hierarchical CPU Scheduler* based on SUN Solaris 2.4. Here, the CPU resource is partitioned into a hierarchy of classes. Each class can be further partitioned into subclasses. Different schedulers can be assigned on a per-class basis. The scheduling on the classes' level is done by the *Start-time Fair Queuing* (SFQ) algorithm. The disadvantage of this approach is that the implementation requires modifications to the Solaris kernel scheduler. Furthermore, the scheduling overhead can be expected to rise proportionally with increasing depth and width of the class hierarchy.

Based on RT-Mach Mercer, Savage und Tokuda [7] implemented *Processor Capacity Reserves* for the RT-threads of this operating system. A newer version [8] supports also the dynamic adjustment of QoS (Quality of Service). A new thread must first request its CPU QoS (period and requested CPU usage in percentage). The time reserved for the thread has to include the timed needed by servers used by the thread. Implementation of the *Processor Capacity Reserves* approach requires non-trivial modifications inside the RT-Mach kernel.

Kamada, Yuhara and Ono [9] implemented a *User-level RT Scheduler* (URsched) in the SUN Solaris. URsched is based on the POSIX.4 fixed priority extension. The idea of scheduling is quite similar to [1]. URsched has some advantages (compared to *Hierarchical CPU Scheduler* and *Processor Capacity Reserves*), it does not require modifications to the existing OS kernel. The scheduler can be implemented as an user-level application. Furthermore, its computational overhead is low and arbitrary scheduling algorithms can be implemented in the user-level scheduler.

Chu and Nahrstedt [10] implemented a *Soft Real Time Scheduling Server* based on URsched. Here, new qualities are the *Soft RT Server Architecture*, the implementation of Rate Monotonic Scheduling as example scheduling algorithm and the introduction of protection mechanisms between concurrent RT-processes.

4 Experimental Evaluation

4.1 Scenarios

Overhead

The overhead is an important value to determine the cost of using our scheduling server. First we want to present our definitions of overhead. Two different points of view for the calculation of overhead imposed by the scheduling server are possible:

- calculation based on comparison between requested computing power (total computing power times requested factor) and measured computing power of the client task (of the scheduling server) — *internal overhead*.
- calculation of overhead based on comparison between theoretically and really remaining computing power (remaining to the rest of the system) — *external overhead*.

The following variables are used:
p - percentage of requested computing power
L_I - computing power of client task (measured)
G - total computing power of the processor
O_I - internal overhead
L_E - remaining computing power
O_E - external overhead.

internal overhead:	**external overhead:**
$L_i = pG$	$L_e = (1-p)G$
$L_I = (1-O_I)L_i$	$L_E = (1-O_E)L_e$
$O_I = 1 - \frac{L_I}{pG}$	$O_E = 1 - \frac{L_E}{(1-p)G}$

Stability

The measured computing power of the client task should be stable even under presence of background load. The stability value S can be defined based on a series of n measurements with different background loads — it is the variation coefficient of the series of measured values.

We define:
L_i - computing power of measurement number i, $i = 1, \ldots, n$
S - stability value (variation coefficient)

$$Var(L_i) = E((L_i - E(L_i))^2) = E(L_i^2) - E(L_i)^2$$

$$\Rightarrow \quad S = \frac{\sqrt{E(L_i^2) - E(L_i)^2}}{E(L_i)}$$

$$E(L_i) = \frac{1}{n}\sum_{i=1}^{n} L_i$$

$$S = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n} L_i^2 - \left(\frac{1}{n}\sum_{i=1}^{n} L_i\right)^2}}{\frac{1}{n}\sum_{i=1}^{n} L_i}$$

It is useful to distinguish two cases:

- unloaded CPU

  The calculation of $S_{unload}(p)$ for different CPU-percentages p delivers values which are independent from the load and can be used as basis for further considerations (loaded CPU)

- loaded CPU

  Different kinds of background load (computational load, disk-I/O, network traffic) and different numbers of background processes can influence the stability $S_{load}(p)$. These values have to be compared to values obtained from an unloaded system ($S_{unload}(p)$) because of influences of operating system and scheduling server itself.

Benchmarks

All measurements depend on a particular benchmark program. To abstract from the actual benchmark program, we focus on relative rather than absolute values. Therefore, two or more values measured with the same benchmark will be compared. Nearly any benchmark program can be used which fulfills the following set of requirements:

- the benchmark must have parameters which determine runtime and memory usage (it should not fit in the cache, and it should not use swap space)

- the measurement must be independent from values obtained by the benchmark program's *rusage* — other timing measures must be used.

In this work we use the well known linpack-benchmark [11] — with some small modification to fulfill the second requirement. Our extensions to linpack allow for registration with the scheduling server and collection of results. We call the new benchmark *lintime*.

For the measurement of stability (loaded CPU) a synchronization mechanism needed. Another extension to the benchmark program does this job - it can request the start of background load processes via signals. A *load server* implements signal handling and subsequently starts the background load processes. Our further extended *lintime* program is called *linload*. It can start up to 10 background load processes.

These load processes represent computational load (a loop doing calculation) and disk I/O (a loop writing a large file of random numbers over and over again). For experiments with network traffic the load is generated by small programs receiving UDP-packets. These packets are sent from a different host via an isolated network segment.

Our experiments regarding the Scheduling Server have been executed on a HP 715/33 workstation running NeXTSTEP 3.3. In contrast, experiments regarding the load partitioner on rtLinux were run on an Intel 386-based computer.

4.2 Measurements

Overhead

Figure 3 shows the difference in performance between two parallel *lintime* runs — the first one under control of the scheduling server, whereas the second one was executed as an ordinary Mach task. The difference between the two graphs is the absolute overhead (the left scale is for the scheduling server's client task, the right one for the remaining system).

Table 1 presents the calculated values for the external and internal overhead.

CPU perc.	mean value	meas. CPU perc. (int.)	int. overh.	ext. comp. power (MFlops)	meas. CPU perc. (ext.)	ext. overh.
10%	0.33	7.13	0.28	4.19	89.58	0.01
20%	0.72	15.52	0.22	3.77	80.63	-0.00
30%	1.11	23.81	0.20	3.33	71.33	-0.01
40%	1.51	32.40	0.18	2.93	62.67	-0.04
50%	1.93	41.30	0.17	2.49	53.27	-0.06
60%	2.30	49.26	0.17	2.07	44.32	-0.10
70%	2.70	57.72	0.17	1.63	34.84	-0.16
80%	3.10	66.27	0.17	1.20	25.76	-0.28
90%	3.49	74.73	0.16	0.77	16.51	-0.65

Table 1: Scheduling Server's overhead

The scheduling server's internal overhead decreases with increasing CPU-percentage. The number of cycles used by the benchmark is constant. This means, that the runtime decreases with increasing CPU-percentage, and this means that a smaller number of thread switches is needed (the scheduling server's period is fixed). Less thread switches require less time — resulting in a smaller overhead.

The external overhead is near zero in the 10% case and less than zero in the other cases. This means that the interactive user (the remaining system) is guaranteed to get at least the calculated part of the total computing power.

Both results indicate that the scheduling server and its client tasks use not more than the allowed part of the machine's CPU cycles. Thus, using the scheduling

server approach, we are able to guarantee a certain percentage of CPU cycles to the interactive user.

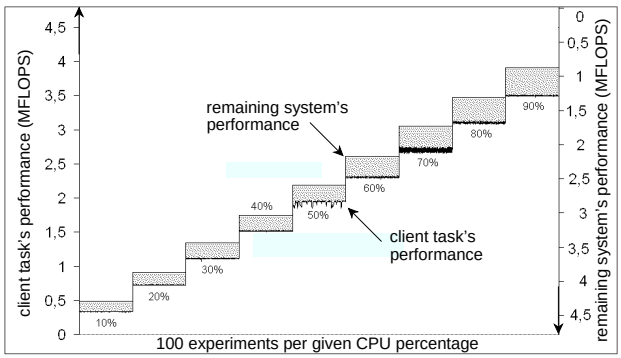

Figure 3: Scheduling Server's Overhead

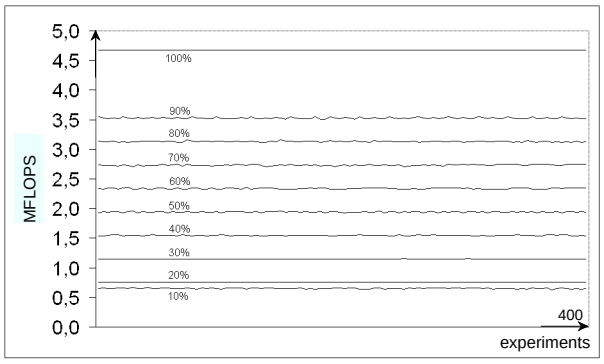

Figure 4: linload's performance without background load

Stability — unloaded

The stability is calculated using a serie of 400 measurements for each CPU-percentage. The values can be found together with the values for the loaded CPU in Table 2. Figure 4 shows the measurements in a graphical form.

In Figure 4, there exists a non-linearity at 10% which cannot be explained clearly. It appears only in some of the experiments without background load. We assume interferences between the scheduling server and NeXTSTEP's kernel scheduler which may well be hardware dependent.

Stability — loaded

The procedure for computing stability in the loaded case is quite similar to the one described above. However, we are now using the *linload* program, which generates a signal every 100 runs to initiate an increase in background load. A separate line in the diagrams shows the varying number of load processes. Figure 5 and figure 6 show the corresponding diagrams for varying computational and I/O background load.

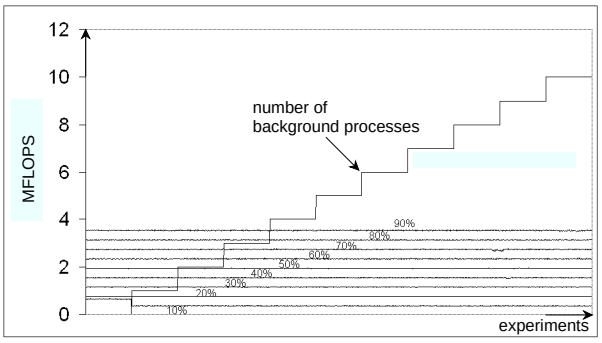

Figure 5: impact of varying computational background load

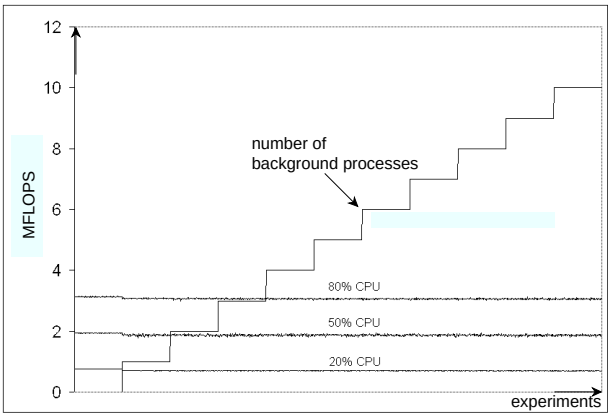

Figure 6: impact of varying disk I/O background load

CPU percentage	unloaded	comp. load	disk-I/O load
10 %	0.01250	0.20970	
20 %	0.00172	0.00301	0.02296
30 %	0.00211	0.00245	
40 %	0.00315	0.00361	
50 %	0.00405	0.00354	0.01676
60 %	0.00368	0.00538	
70 %	0.00412	0.00462	
80 %	0.00325	0.00270	0.00795
90 %	0.00333	0.00323	
100 %	0.00027		

Table 2: Stability values for Scheduling Server

The computing power obtained by the *linload* benchmark is nearly constant over time even in pres-

ence of large amounts of background processes. Except for the abnormal 10%-case, this is shown by our computations of stability in Table 2, which compares stability values for loaded and unloaded CPU. The stability is higher (smaller stability value) if the background processes do not perform any I/O operations.

4.3 rtLinux experiments

The same experiments as descriped above (with exception of the disk I/O) were made on a rtLinux machine to evaluate our alternative solution. In contrast to previous experiments, we have used an Intel 386-based computer for the rtLinux experiments. The following figures present the results:

Figure 7: Load Partitioner without background load

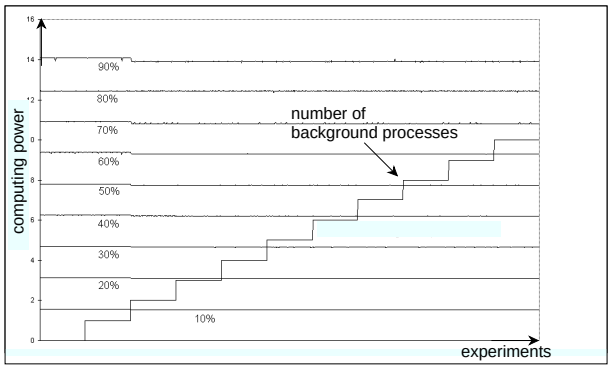

Figure 8: impact of varying background load

Without presence of background loads, the load partitioner is more stable than the scheduling server. However, the impact of varying background loads is more severe. This might be a result of the measurement method used in our scenario. Because system calls cannot be executed in the rt-thread, all communication has to be funneled through a *rt-fifo* and a non-rt-thread. However, this non-rt-thread is influenced by the changing background loads and can falsify the results.

This means also, that the load partitioner approach is only useful for threads with very low communication requirements - e.g. for computational intensive threads.

5 Proof of Concept – an Example Application

A MPEG player has been chosen as example application to prove suitability of our scheduling server for support of real-time multimedia applications. Furthermore, the MPEG player is an example for running a bigger piece of software under control of the scheduling server.

Multimedia applications have special requirements on several resources of a computer, including the CPU power. Multimedia applications are typically time-triggered and therefore have to run with a certain periodicity. Also, these applications require the available CPU power to be constantly above some threshold. These requirements seem to be suitable to the scheduling server.

One single change to the MPEG-player program was necessary to adapt it for our experiments: the inclusion of a function call for registration with the scheduling server. In fact, since task properties like scheduling policy are not affected by Mach's *exec()* system call, one could even imagine a more general *real time shell* which would allow for execution of arbitrary programs under control of the scheduling server.

Our example MPEG player has been programmed to achieve an average frame rate at a requested level. However, we have observed very low average frame rates (lower than requested) on highly loaded systems. Furthermore, changing background loads result in changing average frame rates. Assuming that the total computing power of the machine is high enough for the MPEG player, the scheduling server and the window server, our experiments show that the scheduling server can be used to dramatically increase the quality of a MPEG presentation on a loaded machine.

Measurements

During runtime the MPEG player computes the average, the maximum, and the minimum frame distance. In our case the test video was a 114 KB MPEG with 40 frames. We have displayed this video sequence three times in a loop — resulting in 120 single experiments. The requested frame-rate was 10 frames/sec so that an average frame distance of 100 ms was expected.

We have investigated a couple of different experimental settings which can be distinguished by the following parameters:

- execution with or without scheduling server

- CPU percentage and period length — parameters of scheduling server
- number of background load processes (only computational load is used)

Without scheduling server the MPEG-player runs with different numbers of background processes. With increasing load the average frame distance increases also. (see table 3).

Table 3 also shows that with increasing load the minimal distance between frames decreases. A possible explanation is that at higher load sometimes two frames were calculated directly one after the other without break — because the program attempts to achieve the average frame rate.

high phase	low phase	backg. proc.	min. dist.	max. dist.	avg. dist.
-	-	0	30	170	100
-	-	5	20	2380	293
-	-	10	10	2679	434

Table 3: MPEG-player without scheduling server — all times in ms

With scheduling server and a period length of 100 ms quality of the results does not increase (see table 4).

Although scheduling server period and frame distance have a value of 100 ms there exists no explicit synchronization between scheduling server and MPEG player. Result is a situation where the clock-driven MPEG player may get its activations just at the wrong points in time — wasting cycles while waiting for a clock pulse. The value zero for the minimal frame distance shown in table 4 is supposed to be a problem with time measurement inside the MPEG-player as client task of the scheduling server.

high phase	low phase	backg. proc.	min. dist.	max. dist.	avg. dist.
10	90	0	0	549	210
20	80	0	0	499	109
30	70	0	0	449	170
40	60	0	0	319	151
50	50	0	0	349	132

Table 4: MPEG-Player with scheduling server (100ms period) — all times in ms

In the next series of experiments we have varied the scheduling server's **period length** using high phases of 10 ms and 20 ms. As indicated in see Table 5 an assigned CPU percentage of 20% (10 ms high phase/40 ms low phase) is not sufficient for the MPEG player, but already 25% (10 ms high phase/30 ms low phase) are enough to reach the requested average frame distance. Not surprisingly, giving 50(20 ms high phase/20 ms low phase) yields the same average frame distance. However, Table 5 indicates an increase in minimal frame distance in this case (no two frames are processed directly one after the other).

high phase	low phase	backg. proc.	min. dist.	max. dist.	avg. dist.
10	40	0	0	299	113
10	30	0	0	200	100
20	20	0	40	160	100

Table 5: MPEG-player with scheduling server, short periods — all times in ms

Now, we have repeated the two experiments at CPU percentages of 25% and 50% under increasing background loads (see table 6). As our experiments show, although the average frame distance is not influenced by changing loads, the maximum distance varies.

high phase	low phase	backg. proc.	min. dist.	max. dist.	avg. dist.
10	30	5	50	200	100
10	30	10	0	300	100
20	20	5	40	200	100
20	20	10	40	280	100

Table 6: MPEG-player with scheduling server and background load — all times in ms

Our MPEG player employs the NeXTSTEP window server to display images on the screen. Since the window server is not run under control of the scheduling server, it may become a bottleneck on a highly loaded system — thus causing the increased maximum frame rate.

Lessons learned

Our experiments show that the scheduling server works well for predictable execution of the MPEG-player, a time- and resource-critical example applications. However, there are a number of design requirements which have to be taken into account for optimal results:

- All parts of a time-critical application should be run under control of the scheduling server. The application should avoid the use of shared system servers.

- The algorithms and timing behaviour of the application should take into account the scheduling server and its period.
- Synchronization within the application should rather be based on the scheduling server than on external clock events. For example the MPEG player could calculate a new frame at the beginning of every period. The time slices assigned to the MPEG player could be chosen such that they allow for worst case execution of a single frame.
- The scheduling server parameters need to be experimentally adopted to the requirements of the application.

6 Conclusions and Future Work

This paper has investigated the scheduling server approach which allows to partition the computing power of a workstation between an interactive user and time and resource critical applications. We have presented an experimental evaluation of the scheduling server. Furthermore, we have shown how a real-time multimedia application may take advantage of the scheduling server mechanism. The scheduling server is part of the SONiC (Shared Objects Net-interconnected Computer) framework for execution of shared-memory parallel programs in a cluster of workstations.

The current implementation of the scheduling server is based on the Mach operating system, however, we have shown applicability of the scheduling server principle to the rtLinux system which provides certain real time features. Additionally, we have performed promising experiments on the mkLinux [12] and HURD [13] platforms.

Future work will include extensions to the memory management system to allow static memory allocation and prohibit swapping for the scheduling server's client tasks. Also, we will implement additional scheduling algorithms [14] like *earliest deadline first* or *rate monotonic scheduling*. Finally, we will further investigate the synchronization of multiple, distributed scheduling servers. This will be extremely helpful for execution of parallel applications in cluster environments.

References

[1] A. Polze. How to Partition a Workstation. In *Proceedings of Eight IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems*, Oct 1996.

[2] A. Polze and M. Malek. Network Computing with SONiC. In *Journal on System Architecture (the Euromicro Journal), special issue on Cluster Computing, soon to be published.* Elsevier Science B.V., Netherlands, 1997.

[3] A. Polze, M. Werner, and M. Malek. High-Performance Responsive Computing with CORE and SONiC. *HUB Informatik-Berichte*, (75), 1996.

[4] V. Yodaiken and M. Barabanov. A Real-Time Linux. Technical report, New Mexico Institute of Technology, 1996.

[5] V. Yodaiken. Cheap operating systems research an teaching with Linux. Technical report, New Mexico Institute of Technology, 1996.

[6] P. Goyal, X. Guo, and H. Vin. A Hierarchical CPU Scheduler for Multimedia Operating System. In *The Proceedings of Second Usenix Symposium on Operating System Design and Implementation.*

[7] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[8] C. Lee, R. Rajkumar, and C. Mercer. Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach. In *Proceedings of Multimedia Japan 96*, April 1996.

[9] J. Kamada, M. Yuhara, and E. Ono. User-Level Realtime Scheduler Exploiting Kernel-Level Fixed Priority Scheduler. *In the proceedings of Multimedia Japan 96*, March 1996.

[10] H. Chu and K. Nahrstedt. A Soft Real Time Scheduling Server in UNIX Operating System. Technical report, University of Illinois, January 1997.

[11] J.J. Dongarra. Performance of Various Computers Using Standard Equations Software in a Fortran Environment. *Computer Architecture News*, 18(2):17–31, March 1990.

[12] mkLinux. *http://www.mklinux.apple.com/*, 1997.

[13] Free Software Foundation (FSF). GNU Hurd information. *http://www.gnu.ai.mit.edu/software/hurd/hurd.html*, 1997.

[14] A. Polze, M. Werner, and G. Fohler. Predictable Network Computing. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*, pages 335–343, 1997.