

# The LINC-NIRVANA Fringe and Flexure Tracker: Linux Real-Time Solutions

Yeping Wang, Thomas Bertram, Christian Straubmeier, Steffen Rost, Andreas Eckart

I. Physikalisches Institut, University of Cologne, Zùlpicher Str. 77, 50937 Cologne, Germany

## ABSTRACT

The correction of atmospheric differential piston and instrumental flexure effects is mandatory for optimum interferometric performance of the LBT NIR interferometric imaging camera LINC-NIRVANA. The task of the Fringe and Flexure Tracking System (FFTS) is to detect and correct these effects in a real-time closed loop. On a timescale of milliseconds, image data of the order of 4K bytes has to be retrieved from the FFTS detector, analyzed, and the results have to be sent to the control system. The need for a reliable communication between several processes within a confined period of time calls for solutions with good real-time performance. We investigated two soft real-time options for the Linux platform. The design we present takes advantage of several features that follow the POSIX standard with improved real-time performance, which were implemented in the new Linux kernel (2.6.12). Several concepts, such as synchronization, shared memory, and preemptive scheduling are considered and the performance of the most time-critical parts of the FFTS software is tested.

**Keywords:** Linux kernel, POSIX, RTAI, real-time, scheduler, FFTS

## 1. INTRODUCTION

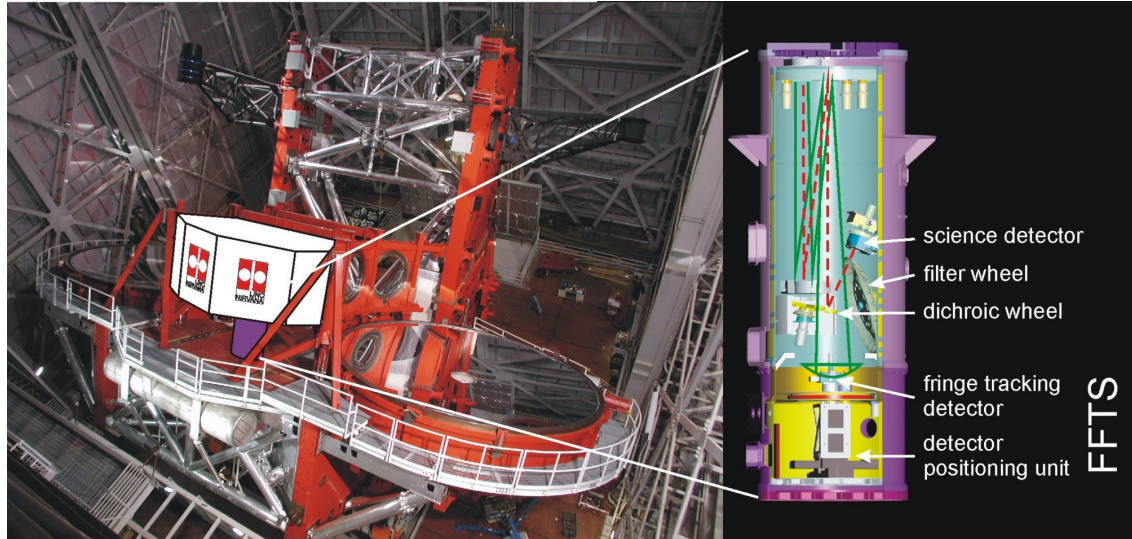
The Large Binocular Telescope,<sup>1</sup> with its two 8.4m telescopes on a common mount, allows for a unique combination of high angular resolution, large field of view (FoV), and the sensitivity corresponding to a light collecting area of 110m<sup>2</sup>. It forms the bridge from current 8-10 m class telescope to future extremely large telescope technology. LINC-NIRVANA<sup>2</sup> (Fig. 1), the NIR interferometric imaging camera for the LBT incorporates the advantages provided by the design of the LBT. Because it obeys specific geometric constraints that constitute a Fizeau interferometer, LINC-NIRVANA can profit from the resulting large FoV in two ways: It allows for a large interferometric science FoV (limited by the cost of NIR focal plane arrays). And it allows to exploit the FoV to choose from a large pool of off-axis reference stars for adaptive optics and fringe tracking. In its final level of implementation, LINC-NIRVANA will be equipped with a multi-conjugate adaptive optics system (MCAO) that can gather the light of reference stars in a 6 arcminute diameter field. In terms of angular resolution LINC-NIRVANA will outperform the imaging capabilities of exiting 8-10 m class telescopes by a factor of ~3.

The Fringe and Flexure Tracking System (FFTS) is an integral part of LINC-NIRVANA. Its purpose is the real-time detection and compensation of differential piston between the two interferometric channels and the correction of misalignment due to instrumental flexure. A general description of the FFTS can be found in Straubmeier et al.<sup>3</sup> Without continuous correction of the phase difference between the two channels, the high interferometric angular resolution is not achievable. The varying piston phase difference caused by the turbulent atmosphere requires a correction bandwidth of several 10 Hz.

In the FFTS, a fast sequence of two-dimensional images of a reference source is retrieved. These instantaneous interferometric point spread functions (PSFs) contain information on the piston phase difference. In a real-time control loop,<sup>4</sup> the PSF will be analyzed<sup>5</sup> and the derived piston phase offset will be compensated by a piezo actuator driven piston mirror.

---

Send correspondence to Y. Wang: E-mail: ywang@ph1.uni-koeln.de, Telephone: +49 (0)221 470-3548



**Figure 1.** LINC-NIRVANA will be installed at one of the interferometric focal stations of the LBT. The two beams of the interferometer will be combined by a cassegrain telescope within the dewar<sup>6</sup> of LINC-NIRVANA. The fringe tracker is located at the bottom end of the dewar, close to the science detector. A set of dichroic beam splitters will reflect part of the NIR spectrum in the center of the FoV to the science detector. The remaining NIR radiation in the center is transmitted to the fringe tracker. In the FoV exceeding the central 10 arcsec., the full NIR spectrum is available to the fringe tracker.

## 2. LATENCY REQUIREMENT

The LINC-NIRVANA instrument control system is a distributed, Linux based system consisting of several computers. The FFTS control software will be running on a dedicated multiprocessor Linux platform. In place of a kernel-space software solution or a monolithic, multi-threaded user-space solution, the FFTS control software consists of several separate user-space processes, some of which require fast interaction with each other. The FFTS image acquisition is done by one process, piston and flexure analyses are handled by other processes of the FFTS software. The objective is to enable fast and reliable data transport between processes with a minimum delay time. The timeliness of the frame data is essential for the fringe tracking performance.

The fringe tracking application requires a sampling of the PSF with a frame rate of up to 200 Hz. The PSF is measured in a subwindow (32x32 pixels) of the FFTS detector. Each frame, therefore, consists of about 4K bytes of data that has to be captured in each cycle of the FFTS control loop and has to be provided to the various analysis processes. The latencies caused by the inter-process data transport must be much smaller than the latencies caused by the other components in the control loop, i.e. they must not exceed 100  $\mu$ s.

## 3. TESTING LINUX REAL-TIME SOLUTIONS

A real-time system is a system in which the correctness includes both the response time and its functional correctness. In the FFTS control software the acceptance threshold for the inter-process data transport time is 100  $\mu$ s. If the transport time exceeds this limit in a cycle, it may lead to data loss and the fringe tracking control system will not be able to capture the piston phase difference correctly. This will not immediately cause the FFTS to fail. Recurring or continuous exceedance, however, will have an impact on the quality of the fringe tracking control loop. In this application, a response time greater than 100  $\mu$ s should be constrained to a predictable degree. It is a soft real-time solution with a hard real-time constraint.

### 3.1. Real-time with Linux

Linux, unlike traditional real-time operating systems such as QNX, was not designed to be a hard real-time operating system. It is a general purpose system in which the response requirements are rather soft. But

because of its open source philosophy, Linux is getting more popular and powerful also for many real-time applications. Many concepts to add real-time functionality to Linux are under development, not only within the official Linux kernel, but also as additional real-time kernel patches or as dual-kernel solutions. Since each application has its special requirements, no solution can serve all applications best. To fulfill the requirements for the FFTS control software stated in sect. 2, two of the Linux real-time solutions seem to be adequate: RTAI (Real Time Application Interface) with its user-space extension LXRT and the POSIX real-time functionality implemented in the recent Linux 2.6.12 kernel. In the following we focus on these two approaches.

RTAI is a dual-kernel approach to a real-time solution under Linux. It actually runs a small real-time kernel that is not Linux, but which runs Linux as its lowest-priority process. With RTAI, Linux only gets to run when the real-time operating system isn't running. RTAI runs only in kernel-space. Real-time tasks are specially written for the real-time kernel using the RTAI API<sup>7</sup> and therefore also run in kernel-space. They can exchange data with Linux applications. LXRT is an extension of RTAI. It allows real-time tasks to run in user-space using the RTAI API. The demands on the software development of a kernel-space solution are much more stringent than of a user-space solution. To be able to provide a stable and maintainable solution with a reasonable effort, it was decided to look for a user-space solution.

Linux 2.6.12 is the most recent release available at the time of investigation. The Linux kernels preceding version 2.6 did not provide sufficient real-time functionality. These features required the installation of special kernel patches. The implementation of POSIX real-time functionality in version 2.6 now provides improvements directly within the official Linux kernel. This change makes Linux much more interesting than in the past, when response time is a concern. The Linux 2.6 kernel is preemptive to some degree and has a more efficient scheduler. In previous Linux versions, a system call interrupts the execution of any process. The execution does not proceed before the system call returns, no matter how long that might take. This may cause important tasks to be delayed by an unpredictable amount of time. In Linux 2.6, the kernel code has been attached with preemption points that enable the scheduler to run and possibly block a current process in order to schedule a process with higher priority.<sup>8</sup> The Linux 2.6 kernel also introduces a new task scheduler with which the execution time is not affected by the number of tasks being scheduled. This is known as an  $O(1)$  scheduler compared to the earlier  $O(n)$ , in which the schedule algorithm is dependent on the number of tasks. With  $O(1)$ , the time required for task scheduling is predictable, therefore, the worst-case scheduling time can be predicted.

The Linux 2.6 kernel provides also improvements for SMP (Symmetric Multiprocessing) systems. It supports fast user-space Mutexes that check the user-space and perform system calls to block a thread only when race conditions happen. This avoids unneeded system calls and can improve the system's responsiveness when synchronizing processes. The Linux 2.6 scheduler was designed to reduce the bouncing of processes between CPUs. It switches tasks from one CPU to another only to resolve imbalances in the run queue. Linux 2.6 supports an efficient shared memory management in a multiprocessing system. A SMP system can benefit from this function when using shared memory to perform data communication between processes.

POSIX defines a standard way for an application to interface to the operating system. Subsequent releases of POSIX also included real-time extensions. POSIX 1003.1b provides functionality that supports the needs of real-time applications, such as enhanced Interprocess Communications (IPC), scheduling and memory management control.<sup>9</sup> In Linux 2.6, some functionality such as POSIX signals and POSIX high-resolution timers have been much improved.

### 3.2. Test setup

We report on various tests to compare the performance of the two alternative approaches that are mentioned above: RTAI LXRT and Linux 2.6 with POSIX. For the RTAI approach, we built RTAI-3.2, which was added to a normal kernel on an uniprocessor (UP) system. To be able to test the POSIX functionality, we built a separate 2.6.12 kernel on the same UP computer and, additionally, on a SMP system with 4 CPUs. The UP computer is based on a Intel<sup>®</sup> Pentium<sup>®</sup> 4 CPU with 2.40GHz, whereas the SMP system is based on 4 Intel<sup>®</sup> Xeon<sup>®</sup> MP CPUs, each with 2.80GHz. The important measure of the real-time performance in our FFTS system is the duration in which the image data, after it was captured by the sending process, is transferred to a receiving process for further analysis. We define this time segment as latency. We use shared memory to perform this kind of interprocess communication, and use a message queue to pass the data frame information

and to synchronize the two processes. A semaphore is implemented to avoid the two processes to address the same shared memory location at the same time.

In the RTAI approach, we use the RTAI API to initialize two tasks for data sending and receiving as hard real-time tasks in two separate processes. In each process we use RTAI IPC variants for every IPC call such as semaphore, message queue and shared memory. The RTAI timer is used to get the start and end time. A pipeline is used in the hard real-time task to place real-time stamps to a FIFO device. In a last step we use a normal Linux process to read the FIFO data out and get the latency between send and receive.

For the POSIX approach, we initialize two tasks for data sending and receiving in two separated processes with highest priority under SCHED\_FIFO policy. In the Linux 2.6 kernel, with preempt configuration, the tasks with SCHED\_FIFO scheduler will repeatedly preempt any other tasks that have the same or lower priority. Preemption during a system call is also possible, therefore, the time needed to schedule will be very low. In our POSIX test, the time resolution should also be noted. The POSIX function “gettimeofday()” uses Time Stamp Counter (TSC) ticks instead of the system timer, so it can provide at least a time resolution of 1  $\mu$ s.<sup>10</sup>

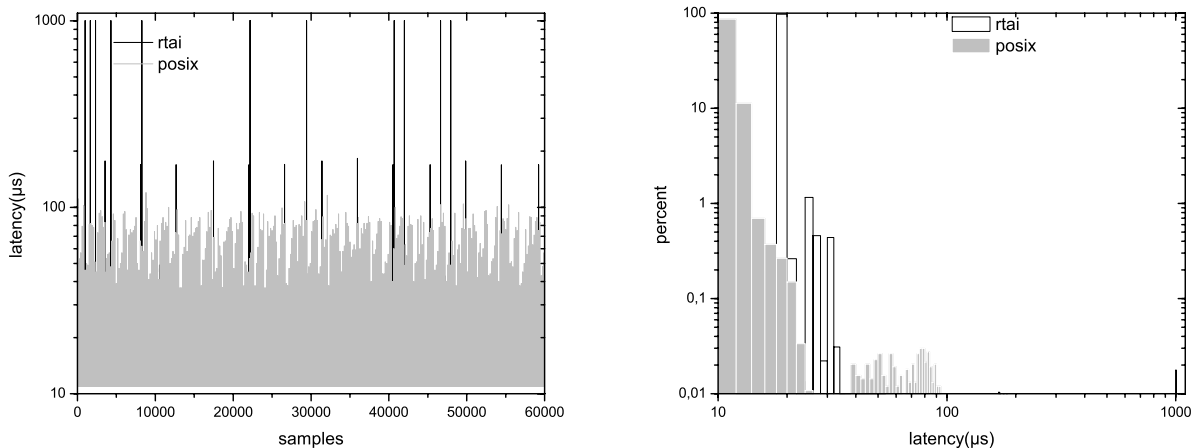
In order to test the latency under the different load conditions, we use simple tools to create 4 defined loads:

- increase the CPU load to an average of 100%
- simulate low memory conditions by 50% MEM usage
- perform disk I/O operations by continuously writing a 1Mbyte file
- create a network load by data transport through TCP sockets

### 3.3. RTAI LXRT versus POSIX features in the 2.6 kernel

We simulate the image data with 32x32 pixels. Each pixel has an integer value and is send it every 10 ms. Therefore, a latency measurement is obtained every 10 ms. The latency is measured in  $\mu$ s.

In the left plot of Fig. 2 the latency of a sequence of 60000 samples using the RTAI(LXRT) and POSIX approaches is shown. A comparison of the relative distributions of the time delays is shown in the right panel. The test is performed without additional system load. With RTAI, the latency distribution is more confined,



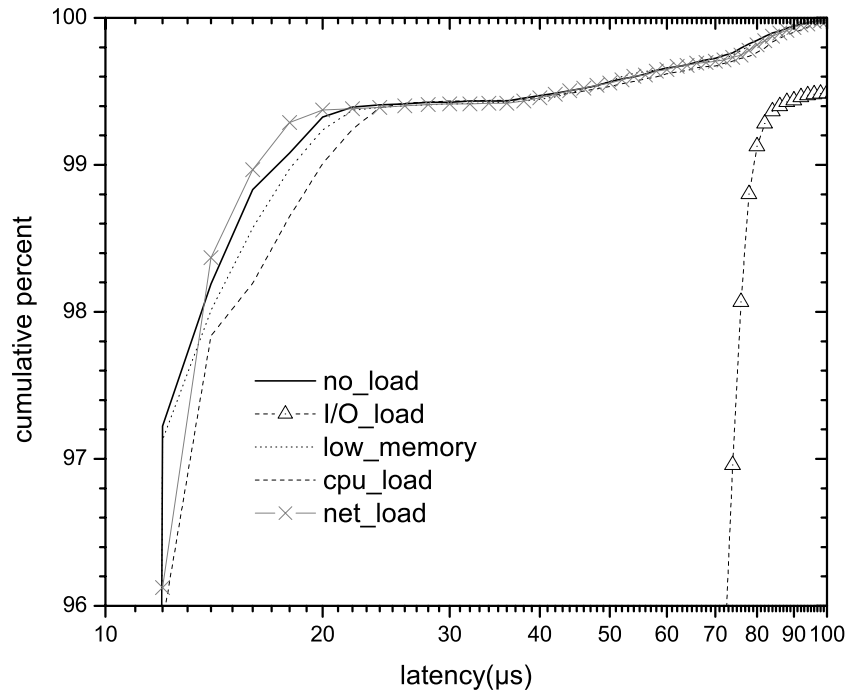
**Figure 2.** Latency comparisons between RTAI-3.2 LXRT and POSIX on a UP computer, without extra system load. Under our hardware environment, the POSIX approach has the lower average and worst-case latency.

but some events occur with a latency of about  $1000 \mu\text{s}$ . In the POSIX approach, on the other hand, the latency distribution shows a broader scatter, but with both a lower average and worst-case latency.

LXRT tasks can provide hard real-time response only as long as a Linux system call is excluded. In this case the latency can be kept at a very low value during the system operation in the RTAI kernel. However, when Linux system calls occur, the LXRT tasks have to deal with the kernel preemption, which in turn leads to an increased latency. In about every 5000th sample the latency exceeds the threshold of  $100 \mu\text{s}$ . In general, with the given test hardware environment, the POSIX approach meets our real-time requirement better than the RTAI LXRT approach.

### 3.4. System load tests

For the POSIX approach, several tests of the response time under different system loads were carried out. The results are shown in Fig. 3. In this and the following plot, the cumulative percentage of samples are plotted against their latency. In the ideal case the curve shows a steep inclination at low latencies. Compared to the behavior of the system with no extra system load, the tests reveals no strong impact of an increased CPU or network load or shortage of available memory on the system's real-time performance, while disk I/O operations degrade the performance significantly.



**Figure 3.** Comparison of the system response under different system loads, using the POSIX approach on a UP computer. Frequent disk I/O operation leads to a significantly increased latency. On the other hand, a high CPU load (100% CPU used), network load (5 ports for TCP file transport) or reduced free memory space (6Mbyte free) do not have a large impact on the real-time performance.

### 3.5. CPU affinity tests

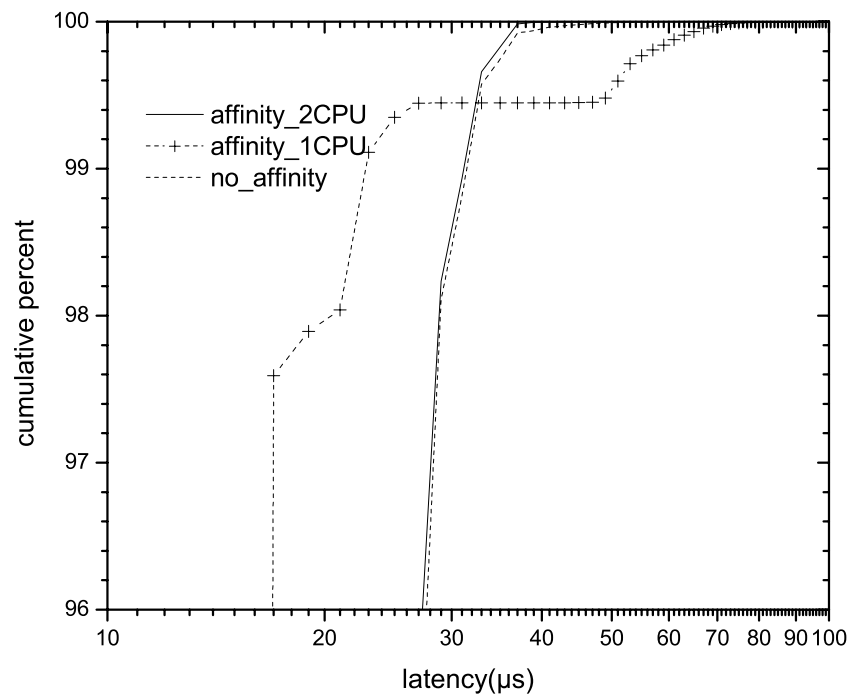
On the SMP computer, we test the response time of the system, when the sending and receiving processes are assigned to different processors. Due to CPU cache sharing advantages, it is expected that tasks running on the

same processor should show a better performance in the inter-process data exchange. The data does not have to be fetched and copied to the cache of a different processor. In Fig. 4 we show the performance using different CPU affinities. The curve “affinity\_2CPU” represents the cumulative percentage of samples versus latency in the case of the assignment of the processes to two different CPUs (here CPU1 and CPU2, in order to avoid the usage of CPU0 with its higher routine system load). For the curve “affinity\_1CPU” both processes are bound to one CPU (CPU 1), and for the curve “no\_affinity”, the processes are freely balanced by the scheduler without extra assignment to a specific CPU. Each test contains about 120000 samples.

Indeed, 99% of the samples in the single CPU affinity test show a  $\sim 8 \mu s$  reduction in their latency compared to the other two tests. Nevertheless, about 0.5% of the samples show a latency of more than  $50 \mu s$ .

Both other CPU affinity tests, “affinity\_2CPU” and “no\_affinity”, show a very similar behavior and allow for a guaranteed latency within  $50 \mu s$ . This similarity is a result of the scheduling algorithm, that is implemented in the 2.6 kernel. In the O(1) algorithm, the schedule time is constant regardless of the load on the system or the number of CPUs for which it is scheduling. Therefore, it can easily schedule the tasks between 2 CPUs with a determined time, plus the additional time needed to copy the data from the cache of the other CPU, compared to the single CPU situation. In fact, an improved SMP affinity scheduling is implemented in the 2.6 kernel. Each scheduler can group and run tasks on one CPU, and only if the distribution between the CPUs is not balanced, the tasks will migrate from one CPU to another one.

Even though the scheduler tries to keep the processes on the same CPU as long as possible, it still makes sense to enforce the affinity as a hard requirement to protect against cache invalidation when processes bounce. The “affinity\_2CPU” case, therefore, shows an improved performance with less than  $40 \mu s$  guaranteed latency. Similar tests under system loads show the same tendency, an assignment of the processes to two CPUs provides a better performance.



**Figure 4.** Comparison of different CPU affinity scenarios, using the POSIX approach on a SMP computer without extra system load. The assignment of both processes to one CPU reduces the latency by  $\sim 8 \mu s$  in 99% of all samples. The assignment to separate CPUs or free scheduling both allow for a guaranteed latency of less than  $50 \mu s$ .

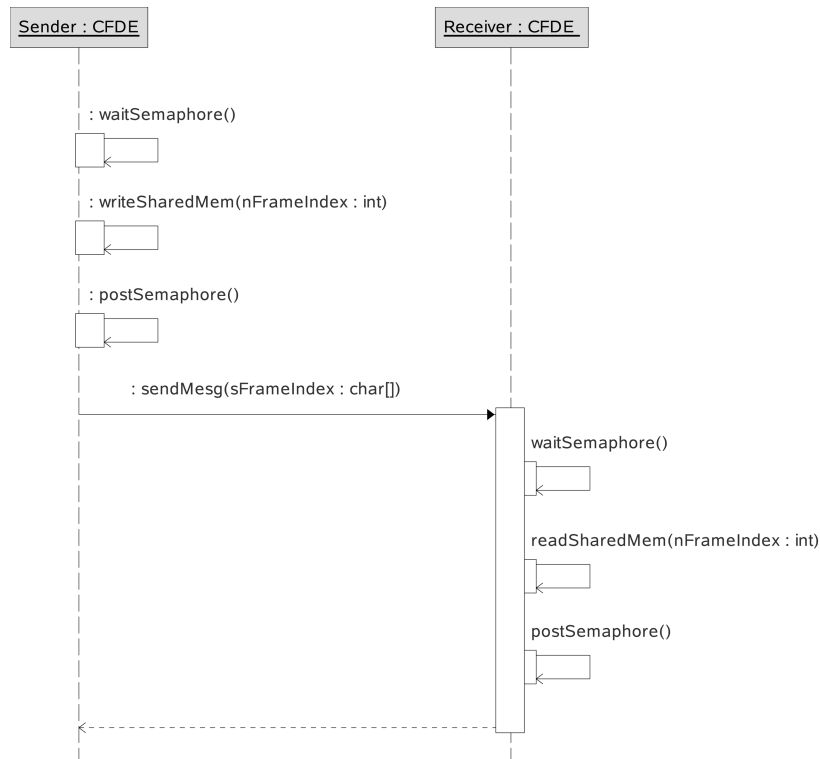
## 4. FFTS SOFTWARE IMPLEMENTATION

### 4.1. Fast Data Exchange (FDE)

As a result of the various tests and comparisons presented in the previous sections, the POSIX approach is identified as the favorite solution for the real-time image data transfer in the FFTS application. The concept is implemented in the “Fast Data Exchange (FDE)” class. In the FFTS software, two instances of the FDE class are generated, for the sending and the receiving process respectively. Both instances have to be initialized. A semaphore and a message queue has to be created and shared memory allocated in the sending process. Because these IPC objects are kernel-persistent, they can be used by both processes, as long as the kernel is not rebooted. After the initialization, the two processes interact with each other in the loop and continuously transport the image data. Fig. 5 depicts the sequence of the data exchange between the two processes via shared memory, whenever image data is acquired.

Before copying the data, the sending process first locks the access to the shared memory area with the help of a semaphore. After the image data has been copied to the shared memory, the sending process releases the semaphore and, by doing so, permits the access by the receiving process. The sending process then sends a message via the message queue, with the image frame index as the content of the message. This generates a signal, which the receiving process is registered to. Once the message is received by the receiving process, it blocks the access to the shared memory, reads the data at the position that corresponds to the frame index and releases the lock again.

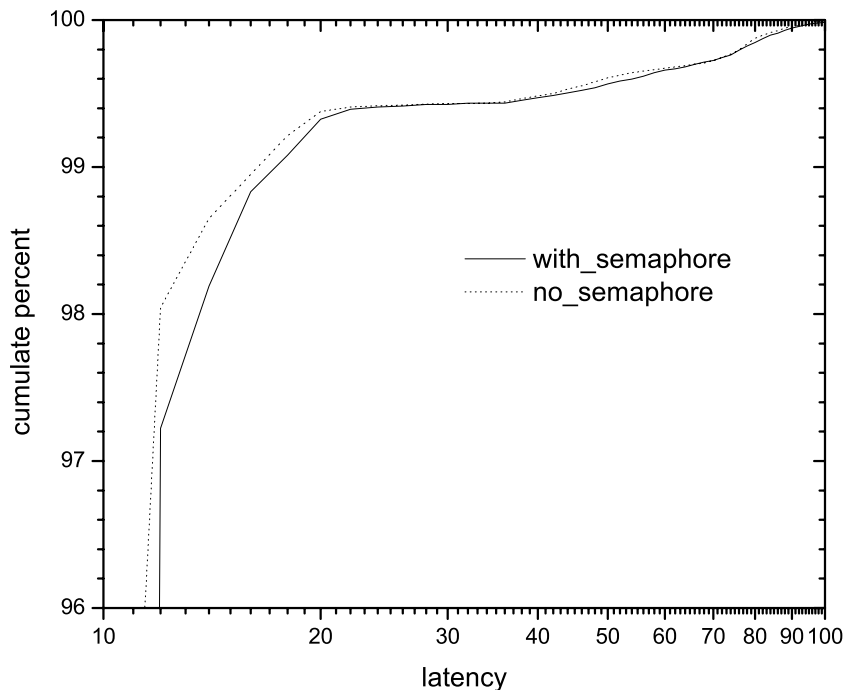
In the receiving process, a signal handler registers a change of the message queue from empty to non-empty. Once the signal is obtained, the signal handler is reset and ready for the next incoming message. In this way, the receiving task remains waiting until the sending task finishes writing the new data to shared memory. The signal, therefore, ensures the synchronization between the two processes.



**Figure 5.** Fast Data Exchange (FDE) sequence for each data acquisition cycle.

## 4.2. Ring buffer versus semaphore

In the FDE sequence shown in Fig. 5, the semaphore is used by the two interacting processes, to prevent the sending process to write to the shared memory before the receiving process finished reading previously stored data. This problem only occurs, if the sending process requests access to the shared memory in a frequency that is higher or comparable to the latency occurring in the FDE scheme. As long as the period between two captured frames remains large compared to the latency, a protection of the data in the shared memory is not necessary. In the FFTS application, the period between two frames is larger than 1ms, whereas the latencies remain shorter than 100  $\mu$ s. By omitting the semaphore, the average latency can be further reduced (cf. Fig. 6).

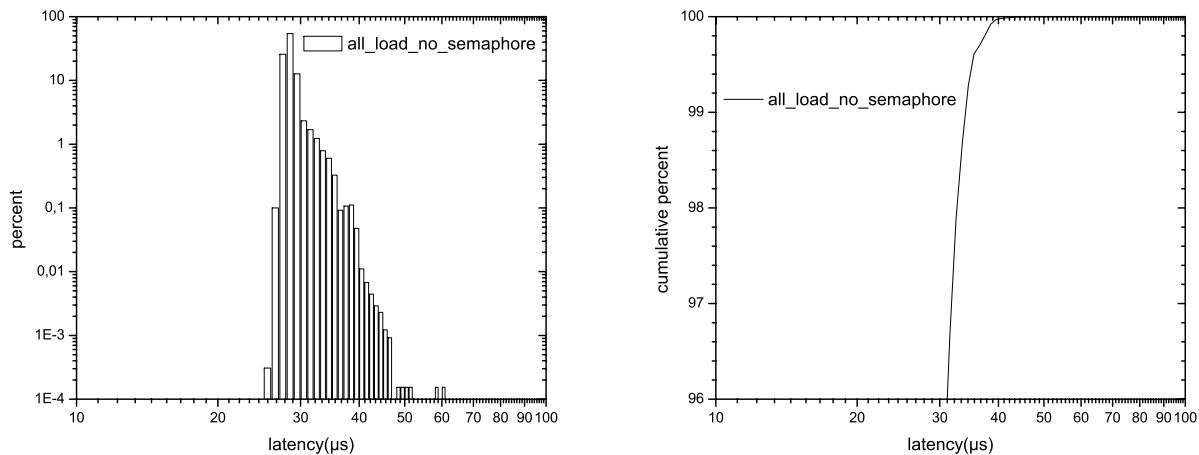


**Figure 6.** Performance comparison when semaphore is omitted, using POSIX solution without extra system loads on the same UP computer as in Fig. 3. In this environment, saving the semaphore operation can reduce the latency by  $\sim 2 \mu$ s at the 98% level.

To prevent simultaneous access of the same shared memory area by the two processes without using a semaphore, a cyclic storage scheme can be implemented. The use of a buffer ring with a size of  $N$  times the size of one frame increases the period by the factor  $N$ , in which the sending process has to address the same shared memory area twice. This approach is easily implemented, since the synchronization message sent to the receiving process contains the frame index, which indicates the position within the buffer ring.

## 4.3. FDE performance

In a final test, we use the FDE scheme on the SMP system, which will be used as FFTS computer, assign the processes to two processors and generate all 4 loads stated in section 3.4. The loads are freely balanced by the scheduler, therefore they have a negligible influence on the performance of the FDE. The test does not make use of a semaphore and consists of  $\sim 120000$  samples. The result is shown in Fig. 7 and proves that the latency caused by the FDE scheme remains within 60  $\mu$ s.



**Figure 7.** Final test using POSIX real-time solution, running on the SMP system under all 4 kinds of system loads, using CPU affinity, without the usage of a semaphore. The latency remains within 60  $\mu$ s

## 5. CONCLUSIONS

The FFTS control loop imposes a real-time requirement on the latency between the sending and the receiving process. The duration of the data transfer has to remain below 100  $\mu$ s. Since the FFTS software is executed on a Linux platform, this real-time requirement has to be fulfilled within a non real-time system. Several options to use real-time features within Linux are available. We tested the open source Linux real-time implementation RTAI with its user space extension LXRT. Furthermore, we tested the usage the POSIX real-time extensions provided by the Linux 2.6.12 kernel. The latter approach was identified as the favorite solution for the FFTS application. In our application environment and under various system loads we can achieve the real-time requirement.

## ACKNOWLEDGMENTS

This work is supported in parts by the Deutsche Forschungsgemeinschaft (DFG) via grants SFB 494, HBFG #111-519 & #111-520, and Verbundforschung 05AL2PLA/5 & 05AL5PKA/0.

## REFERENCES

1. J. M. Hill, R. F. Green, and H. J. Slagle, "The Large Binocular Telescope," in *Proceedings of the SPIE, paper [6267-18]*, 2006.
2. T. M. Herbst, A. Eckart, R. Ragazzoni, and G. P. Weigelt, "Beyond the fringe: an update on the construction of LINC-NIRVANA, a Fizeau imaging interferometer for the LBT," in *Proceedings of the SPIE, paper [6268-72]*, 2006.
3. C. Straubmeier, T. Bertram, A. Eckart, S. Rost, Y. Wang, T. M. Herbst, R. Ragazzoni, and G. P. Weigelt, "The imaging fringe and flexure tracker of LINC-NIRVANA: basic opto-mechanical design and principle of operation," in *Proceedings of the SPIE, paper [6268-55]*, 2006.
4. S. Rost, T. Bertram, C. Straubmeier, Y. Wang, and A. Eckart, "The LINC-NIRVANA fringe and flexure tracker: piston control strategies," in *Proceedings of the SPIE, paper [6274-66]*, 2006.
5. T. Bertram, C. Arcidiacono, C. Straubmeier, S. Rost, Y. Wang, and A. Eckart, "The LINC-NIRVANA fringe and flexure tracker: image analysis concept and fringe tracking performance," in *Proceedings of the SPIE, paper [6268-138]*, 2006.

6. W. Laun, H. Baumeister, and P. Bizenberger, "The LINC-NIRVANA IR cryostat," in *Proceedings of the SPIE, paper [6269-190]*, 2006.
7. P. Mantegazza, E. Bianchi, L. Dozio, S. Papacharalambous, S. Hughes, and D. Beal, "RTAI: Real-Time Application Interface," *Linux Journal* , 2000.
8. R. Love, "Introducing the 2.6 Kernel," *Linux Journal* , 2003.
9. K. M. Obenland, "POSIX in Real-Time," *EmbeddedSystems Programming* **14**, 2001.
10. S. Venkateswaran, "The Passage of Time," *Linux Magazine* , 2005.