Using the RECOMS Architecture for Controlling a Radio Telescope *

Alexander von Bülow Jürgen Stohr Georg Färber
Institute for Real–Time Computer Systems
Prof. Dr.–Ing. Georg Färber
Technische Universität München, Germany
{**Alexander.Buelow,Juergen.Stohr,Georg.Faerber**}**@rcs.ei.tum.de**
Peter Müller Johann B. Schraml
Max–Planck Institut für Radioastronomie, Bonn, Germany
{**peter,jschraml**}**@mpifr-bonn.mpg.de**

Abstract

General purpose computer systems are not designed to act as real–time systems. They are optimized to deliver a good performance in the average case, real–time constraints are not considered. Nevertheless these computer systems are cheap in price, offer a huge computing power and single components are usually downwards compatible. The RECOMS project deals with the task to make the computing power of general purpose computer systems available for real–time applications. In this paper we present the experiences we gained using a Dual–Xeon computer for controlling a radio telescope. This work evolved from a cooperation with the Max–Planck Institut für Radioastronomie. We describe the technical requirements of this application, the implementation using the RECOMS Software Architecture and the results of the studies we made on this system.

1. Introduction

General purpose computer systems are equipped with very fast processors like the Intel Pentium or AMD Athlon and with large and fast mass storage systems. They are very flexible in design and can be easily extended by a great variation of general purpose hardware. But they are not designed to act as real–time systems, they are optimized to deliver best performance in the average case. Worst case scenarios which are of great importance for real–time systems are not considered at all.

Nevertheless there are some aspects that make those general purpose computer systems interesting for use as real–time systems: They have a great computing power, can be designed individually for a special application and they are most of all downwards compatible, so often you can improve the computing power of your system by simply changing a few components without modifying your software. Furthermore they are cheap in price and there exist well known tool chains for developing software.

The RECOMS project [4] deals with the task to research the real–time capabilities of general purpose multiprocessor systems. The intention of this project is to study the runtime behavior of those systems and to develop new methods for using the computing power for the purposes of a real–time system.

In this paper, we present the use of a general purpose computer system for controlling a radio telescope. This task evolved from a cooperation from our institute with the *Max–Planck Institut für Radioastronomie* in Bonn, Germany. The idea behind this cooperation is to replace the so far used contolling system based on a VAX/VMS architecture by a general purpose computer system based on Linux. The main requirements are the lossless transfer of measurement data at a constant time rate over a long period and the reliable storage of the data on a hard disk within a dedicated span of time. This task is implemented using a Dual Intel Xeon processor computer together with Linux and the RECOMS Software Architecture developed by us.

This paper is organized as follows: Section 2 describes in short the characteristics of the RECOMS Software Architecture. In section 3 the scenario and the requirements of the application are described. Section 4 presents the results and experiences we made. In section 5 we summarize the paper and give a short view to future work.

2. RECOMS

Figure 1 gives an overview of the RECOMS Software Architecture. The system is separated into one processor called the GPU (General Purpose Unit, used for Linux exclusively)

*The work presented in this paper is supported by the *Deutsche Forschungsgemeinschaft* as part of a research programme on "Real-time with Commercial Off-the-Shelf Multiprocessor Systems" under Grant Fa 109/15-1.

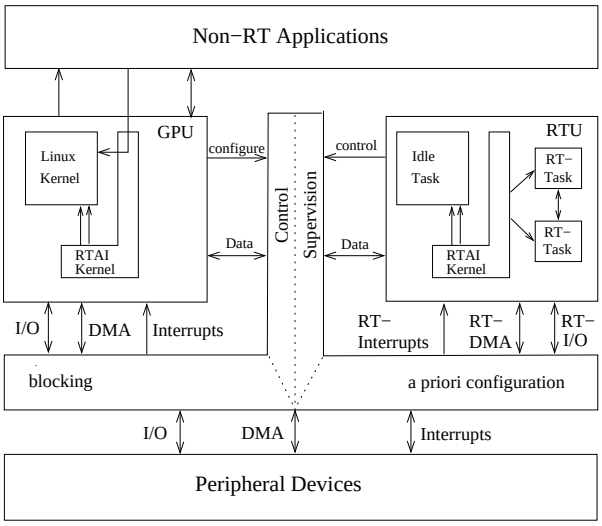

Figure 1: The RECOMS Software Architecture

and one ore more RTUs (Real Time Units, used for real–time tasks exclusively). This makes it possible to use standard Linux tasks and real–time tasks concurrently. The RECOMS Software Architecture enhances the RTAI [1] architecture by adding more options to the real–time operating system (RTOS) to control concurrent accesses to hardware. It is important to be able to route interrupts to different CPUs. This offers the option to route interrupts that are not used by any real–time components (e.g. keyboard) exclusively to the GPU so the RTUs are not burdened with that.

Another capability are the fine grained facilities to control accesses to I/O ports or I/O regions [3]. To control them is very important for accesses to hardware connected via the PCI–Bus from a real–time task. Those accesses can cause tremendous and nearly unpredictable blocking times when concurrent to those of the RTU. The RECOMS Software Architecture enables the user to synchronize them in a way that a real–time task is never blocked when accessing hardware connected to the PCI–Bus.

3. The Application

The objective is to replace a VAX/VMS based computer system which is used to control a radio telescope by a general purpose computer system based on Linux. The requirements are as follows:

- All tasks must run with accurate timing constraints.
- Measurement data has to be stored on hard disk together with an accurate time stamp obtained from the hardware.
- The real–time system should run for a long period of time (e.g. a few weeks) and no measured value must be lost.

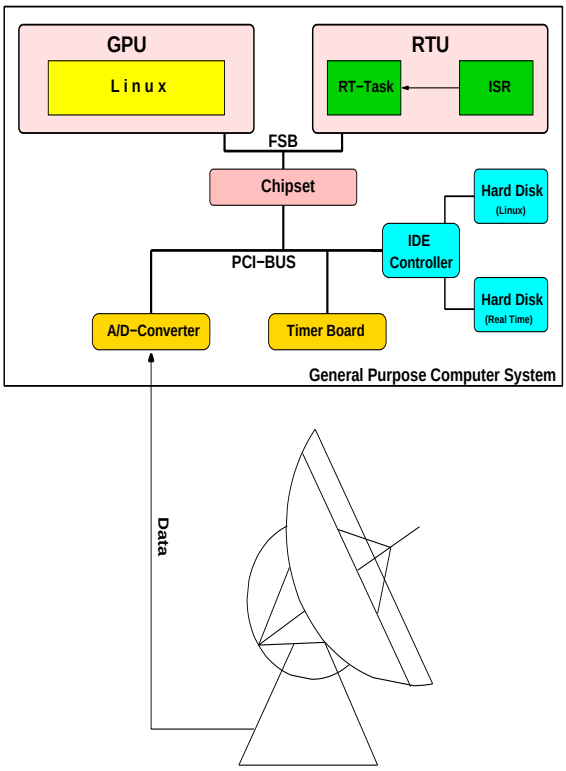

Figure 2: The application scenario

Figure 2 shows the application scenario. The measurement data is delivered from the backend of the antenna and must be converted by an analog-to-digital converter. Afterwards the data has to be stored on hard disk together with the refreshment time. It is essential that no measured value gets lost. One hard disk is used exclusively for the storage of the measurement data delivered from the analog-to-digital converter. The hardware and software structure is described in the following.

Hardware. For this case study we used a Dual Intel Xeon computer system with 2 GHz clock speed, FSB 533, 512 kb second–level cache and 512 MB DDR–RAM. The computer is equipped with two hard disks, one for Linux and one for the storage of the measurement data, both connected to the same IDE–Controller. The “real–time” hard disk has a size of 120 GB, 2 MB cache and a speed of 5400 RPM.

The timer of the computer system is realized by a Datum BC635 board. This board is connected to the PCI–Bus (33 MHz) and is able to produce an accurate time rate (PCI–Bus interrupt) up to a period of 100 ns. The timer can be synchronized with an external IRIG-B signal or can run in a battery-backed mode without any external synchronization. A time stamp can be obtained from the board by reading several registers.

Furthermore a PCI-AD/DA Analogue I/O board from Kolter Electronic is connected to the computer. This board comes with 16 AD/DA channels with a resolution of 12 bits, respectively. The time needed for converting one channel is about 25 microseconds. This converter is not able to deliver an interrupt signal when the conversion has finished so software needs to poll a status flag.

Software. For testing the run time behavior of the computer system, the following software structure was implemented:

- An interrupt service routine (ISR): This routine is triggered by the timer board and is used to measure the response time of the interrupt and to trigger a real–time task.
- A real–time task: This task reads one channel from the analog-to-digital converter, gets the time stamp from the timer board and stores this data on the hard disk. It must have completed its task until the next interrupt arrives. The period of this task is 100 microseconds.

The real–time task is implemented using the standard RTAI–API [1] [2]. The whole application (ISR and task) is implemented as a Linux kernel module. The drivers for the timer board, the analog-to-digital converter and the IDE hard disk were written by us to be able to guarantee a predictable real–time behavior of the driver API (especially for the hard disk).

4. System Analysis

To verify the real–time properties needed for the application described in section 3 we considered the following items:

- The interrupt response time for the timer interrupt triggered by the timer board when the computer system is not fully loaded and when it is working under full load.
- The response time for accesses to the analog-to-digital converter.
- The time needed by the hard disk to store the data.

First of all, let us consider the interrupt response times we observed.

Interrupt Response Time. The interrupt response time is the span of time when the interrupt source raises its edge until the first instruction of the corresponding interrupt service routine (ISR) is executed. To determine this span of time, the timer board is triggered by an external event. The point of time when the external signal arrives is held on in event time registers of the timer board. The first instruction of the ISR latches the refreshment time, then the time registers are read. The difference between the value obtained from the time registers and the value obtained from the event registers is the interrupt response time. The next instruction reads the timer register again, so we get the time needed for reading the registers as difference to the the time stamp read just before. For this span of time we measured 1.8 microseconds. The interrupt response time was calculated as 6.9 microseconds.

Furthermore we were interested in how the period of the real–time task varies. This real–time task runs on behalf of the ISR. That means, the ISR triggers the real–time task which should run just after the ISR has finished on the RTU (see section 3). The period of the real–time task was varied and we measured the span of time from when the period should begin to when the real–time task actually starts by reading the time from the timer board. The average value we got was 7.7 microseconds and the maximum value we obtained was 8.3 microseconds.

The values mentioned above apply to the unstressed case, that means that no extra work was added to the system. This is the situation the system should work when deployed for controlling the telescope. We made the same tests when the system was under a heavy work load. We observed that the worst case interrupt response time growed up to 13 microseconds.

We also studied the interrupt response time when using two interrupt sources in parallel. We used the external interrupt source as described above and a periodic internal interrupt source on the timer board. We observed no interference between these two interrupts.

AD/DA Converter. The analog-to-digital converter is connected to the PCI–Bus just as the timer board. It has 16 channels with a resolution of 12 bits, respectively. The focus of our interest was the time needed for the conversion itself (hardware dependent) and the time needed to read the converted value (PCI–Bus access). Our converter is not able to deliver an interrupt when the conversion has finished, so software must poll a status flag to determine the end of conversion. So the interesting time span is the time needed from start of conversion until the result is read. We measured a time of about 37 microseconds for that, without nameable variation. From these 37 microseconds about 25 microseconds were needed for the conversion.

Hard Disk Accesses. To store the gathered data we use a standard IDE hard disk. This hard disk is used by the RTU exclusively (see figure 2). It is connected to an IDE–Controller which is connected to the central PCI–Bus. The hard disk is invisible for Linux when the system is in real–time mode. The IDE–driver from Linux is not suitable for real–time purposes so we implemented our own driver. This driver only supports low–level data transfers and speaks directly with the IDE–Controller. This justifies our demands, we need no file system or other features because the evaluation of the stored data is made offline with standard Linux software. Remember that the hard disk is fully accessible from Linux when the system is not in real–time mode.

Hard disks can be accessed with small transfers of a few bytes, or, much better, with blocks of data, usually 4 kbytes. The implementation of our real–time driver is as follows: We have two regions of memory, each region has a size of 4 kbytes. The API–function provided by the driver writes to one of these two regions until this area is full. Then the next write access from software is deligated to the second memory region. In the meanwhile, the full memory area is written with a fast burst write to hard disk.

The advantage of this method is that following write accesses are not blocked because they are buffered in memory. Of course one has to take care of the frequency the data to be written comes in. If the data rate is too high, the second buffer runs full before the first is written to hard disk. For our application the data rates were far beyond this danger.

The storage of the data is performed by the real–time task. One run of this task reads one channel of the analog-to-digital converter, gets the time stamp from the timer board and saves the data to hard disk which is equivalent to a data rate of 12 bytes per run. The intended period of the real–time task is 100 microseconds so we get a data rate of about 120 $\frac{kb}{s}$. The access times we measured for writing one block of data to hard disk ranged from 40 microseconds to 50 microseconds.

5. Conclusions

The intention of the work presented in this paper is to evaluate the real–time properties of a general purpose computer system running Linux together with RTAI and the enhancements of the RECOMS Software Architecture. The application in mind is to use such a computer system to control a radio telescope. For this purpose we made a case study with a Dual–Xeon computer which was equipped with an AD/DA–Converter and a timer board. We wanted to know if such a system running with RTAI and RECOMS would be able to satisfy the real–time constraints.

The experiments we made show that a general purpose computer system can act as a real–time system with the help of a suitable operating system. The most important issue is the strict separation of Linux and RTAI/RECOMS. This includes the distribution to different processors (GPU and RTU), the routing of interrupts, the separation of the main memory between Linux and RTAI/RECOMS and the fine grained setting to control hardware accesses.

The time values we measured for the interrupt response time and the time needed for hard disk accesses showed that the computer system we used is able to satisfy the real–time constraints of our technical process. We are encouraged to continue our work to make the computing power of general purpose computer systems available for real–time applications.

In future, we will extend the real–time task we presented in this paper to do some astronomical calculations needed for the controlling, so the execution time of the software running on the RTU and therefore the processor architecture becomes more important. Furthermore the sending of control data to another computer via ethernet with real–time constraints should be investigated. Futher and deeper studies of the real–time behavior and the effect of certain arrangements to improve the predictability will be made.

References

[1] DIAPM, Dipartimento di Ingegneria Aerospaziale Politecnico di Milano. *A Hard Real Time support for LINUX*, 2002.

[2] Lineo, Inc. *RTAI Programming Guide 1.0*, September 2000.

[3] Jürgen Stohr, Alexander von Bülow, and Georg Färber. Controlling the Influence of PCI DMA Transfers on Worst Case Execution Times of Real–Time Software. In *Proceedings of the 4th International Workshop on Worst Case Execution Time Analysis in conjunction with the 16th Euromicro Conference on Real–Time Systems*, Catania, Italy, June 2004.

[4] Jürgen Stohr, Alexander von Bülow, and Georg Färber. Using State of the Art Multiprocessor Systems as Real-Time Systems – The RECOMS Software Architecture. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems – Work in Progress Session*, Catania, Italy, June 2004.