Task Jitter Measurement in Multi-purpose Real-time Systems

Pavel Moryc and Jindrich Cernohorsky
Technical University of Ostrava
Czech Republic

1. Introduction. Scope and problem definition

1.1 Real-time task and its physical realization.

A real-time task is a task, which meets its prescribed deadline. (More precisely, it is a task, which benefit depends on its execution time.)
A control system is a computing system used to control a production technology (fig. 1). Usually, a production technology includes many interacting sub-processes, which all need to be controlled. A real-time task runs in parallel with other real-time tasks and interacts with them. This type of parallelism can be called predictable.

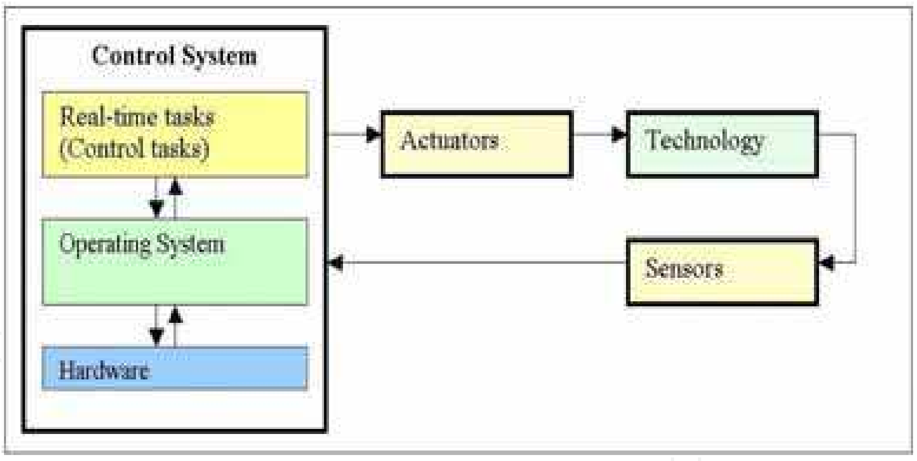


Fig. 1. Technology control

However, another type of parallelism can be seen in the controlled technology as well. The nature of the controlled technology is not fully known, and can only be described with statistical methods. Hence, it is not possible to fully plan, how the technology will be controlled, but from time to time it is necessary to interrupt the currently running control task, and to react on an unplanned urgent condition.
A production system works for years, and during that time, it enters its various conditions repeatedly. Thus, planned system operation means periodicity, while required reactions on unplanned conditions (errors, hazards, etc.) present aperiodicity. The aperiodicity naturally

stems from noise, disturbances, delays, and all other unpredictable phenomena in the real world.
A very simple control system can be realized as a monolithic block having no vertical layers, but usually, a layered structure involving both hardware and software layers is applied. Typically, three domains of control can be recognized in the layered control system structure:

- real-time task. It is a realization of a control algorithm, which controls the technology. Its precise execution is a goal of the control system. But, the control task itself is a pure design abstraction, which has to use services provided by underlying system layers.
- operating system. The real-time operating system provides resources that support control algorithm realization, and make possible that the control algorithm meets its deadline. The operating system is a design abstraction as well.
- hardware. The hardware physically realizes the control task algorithm and the operating system algorithm.

The real-time task directly uses not the basic resources provided by the hardware, but the virtual resources provided by the operating system. Hence, the operating system can significantly mitigate or amplify adverse effects of the hardware used. The quality of the control system is assessed as its ability to support specified accuracy of control, which first of all implies the ability to keep specified deadlines.
A real-time operating system is an operating system, which supplies resources that support the execution of a given real-time task on the specified hardware this way, the real-time task meets deadlines required by the controlled technology.
A real-time operating system should not enforce strict rules that nature cannot meet, but rather provide resources that help to smooth the conflicts. As these resources are supplied by the operating system, they are called virtual. Their quality depends not only on quality of the operating system itself, but also on the quality of the underlying hardware resources. The periodic task is designed to handle deterministic events. Hence, its basic characteristics are its starting time and its deadline.
The aperiodic task is designed to interrupt the periodic task and handle an unplanned, urgent, or even emergency event. Thus, its basic and most important characteristic is the delay between the time when action is required and the time when it is started. This delay is called latency.
Jitter is a variable deviation from ideal timing event. Scheduling jitter is the delay between the time when a periodic task is scheduled to be started, and the time when the task is started. Interrupt latency is the delay between the time when the interrupt is triggered and the time when the interrupt service routine is started. The interrupt latency varies, and therefore, it shows jitter. Kernel latency is defined as the delay between a nominal time when a periodic task shall be switched to the running state, and the actual time when the periodic task is switched to the running state (see Ripoll, 2001).
Both the virtual and hardware resources that support real-time task realization are standardized in the POSIX 1003.13 specification, which defines four resource profiles that can be used to design control systems. A higher profile extends the lower one (see tab. 1).

Profile No.	**Profile Name**
PSE 51	Minimal
PSE 52	Controller
PSE 53	Dedicated
PSE 54	Multi-purpose

Table 1. POSIX 1003.13 profiles

1.2 Multi-purpose real-time system

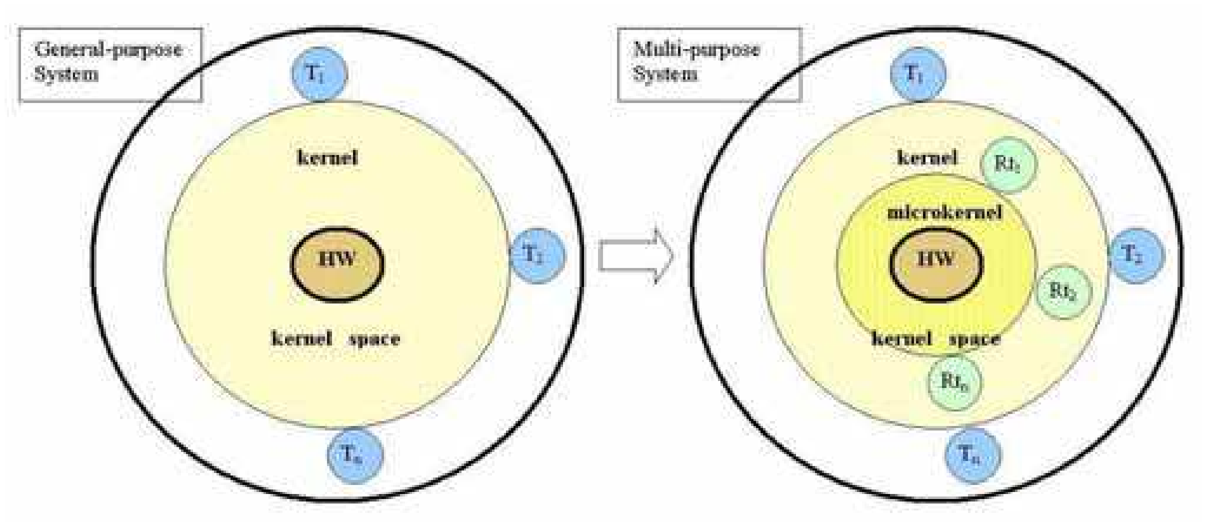
A multi-purpose real-time system can be defined as a resource set specified by the POSIX 1003.13 PSE 54 profile. It can be realized as an embedded PC computer, running multi-purpose operating system. The multi-purpose operating system can be based on a general-purpose operating system (e.g. Linux or Windows).
It has been measured (Ripoll, 2001), that the Linux kernel latency can reach up to several hundreds of milliseconds. It is evident, that such latency makes the system unusable in most technology control applications. This problem is well known, and basically, there solutions were proposed:

- low-latency kernel,
- preemptible kernel,
- hardware abstraction layer.

Low-latency kernel cannot be preempted by task. It is designed as a non-preemptible (monolithic) block, but with the intention to minimize latencies and jitters of API services typically used by real-time tasks.
Preemptible kernel can be preempted by task. Preemptivity by task means, that while a task is being serviced by the kernel, another task can successfully call a kernel API service and preempt the current flow of the kernel operation. This type of preemptivity is called reentrancy. However, at least some parts of the kernel cannot be made reentrant. Moreover, the idea of preemptivity itself does not imply, that the kernel is deterministically preemptible, i.e. the kernel latency and its jitter are kept acceptably low.
Hardware abstraction layer (HAL, fig. 2) is a software layer installed between the hardware and the operating system kernel. It is built as a module that is plugged into the operating system kernel. It receives timer interrupts and sends virtual (software) interrupts to the general-purpose operating system kernel, thus providing a virtual (slower) clock for the general-purpose operating system. This can be seen as a cycle stealing. Because the HAL microkernel can postpone the general-purpose kernel operation, it can make spare time to run real-time tasks. Obviously, latencies and jitters of the HAL layer must be kept low enough by design. In this publication, we will focus on the HAL based solutions only.

1.3 Multi-purpose real-time system based on Hardware Abstraction Layer

The HAL-based multi-purpose real-time system architecture is obtained from the general-purpose operating system by installing the HAL layer into the general-purpose operating system kernel space (see figure 2). In the Linux environment, the HAL layer is called RTLinux microkernel.



T_1 to T_n – non real-time tasks Rt_1 to Rt_n – real-time tasks

Fig. 2. POSIX PSE-54 multi-purpose system constructed as real-time extension of general-purpose system by adding hardware abstraction layer.

As the RTLinux Free source code is available, it can be analyzed. Basically, it catches interrupts from hardware, and sends them to the Linux kernel as software interrupts. The software interrupts are special signals with low latencies, i.e. with very short times of delivery. However, other software interrupts can preempt and postpone their delivery. If the microkernel is heavily loaded, many internal and external signals can wait in queues for delivery, and their delivery times could significantly vary.
The microkernel runs in the kernel space, as a kernel module. It shares the kernel space with the rest of the kernel and with the real-time tasks. The safety barriers between the kernel, the microkernel, and the real-time tasks are thin. It can be supposed, that a more safe solution would adversely affect both the microkernel and the kernel real-time characteristics, their overhead, latency and jitter.
It can be concluded, that the system must be forethoughtfully tested. However, a number of possible iterations in an industrial technology design process is rather limited, and each such iteration can be quite expensive. Hence, a model architecture that makes possible to simulate and verify system behavior in various design stages is needed.

2. Existing measurement methods

First, existing measurement methods were studied. (Ripoll, 2001) is only partially focused on tasks jitter measurement under RTLinux, but it can provide at least a starting point. More details on yet existing approaches were found in (Proctor, 2001), and (Dougan, 2004). Taking these approaches and experience into consideration, a model architecture design has been approached.

3. Designed measurement methods

As it has been yet mentioned before, the real-time task and the operating system resources are mere abstractions, and the overall architecture including hardware must be taken into consideration. Hence, the effort has been focused not on the formal approach only, but on its applicability as well. A measurement architecture (a control system model) has been designed and realized. The design has been intended tightly connected with an application study. Therefore, it can be assumed, that obtained results will be useful for engineering practice.

3.1 Generalized data acquisition program

The multi-purpose real-time system is typically used in applications, where both real-time and non-real-time tasks co-exist in one computing system. Many of these applications are interfaces between dedicated real-time and non-real-time IT levels. As a representative case, a data acquisition system has been chosen. Based on that case, a model architecture has been created. The data acquisition system has been reasonably simplified, generalized, and extended with additional diagnostic timestamp outputs.
The obtained model architecture (fig. 3) contains and integrates resources, which make possible to apply a defined workload to the system, as well as to measure, how the workload task is executed on the system. Its core part is a model of a real-time task, named RT-golem.

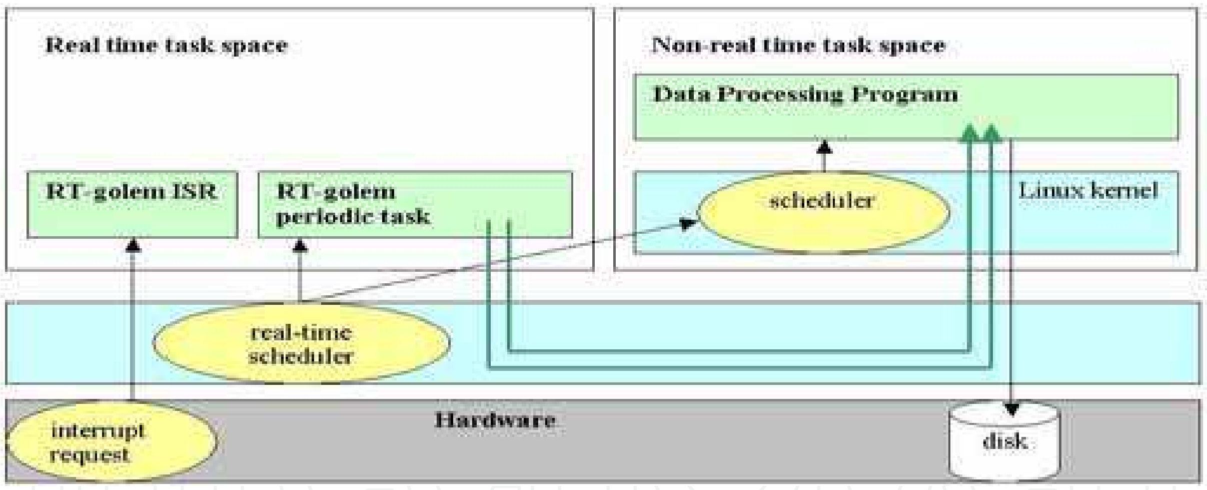


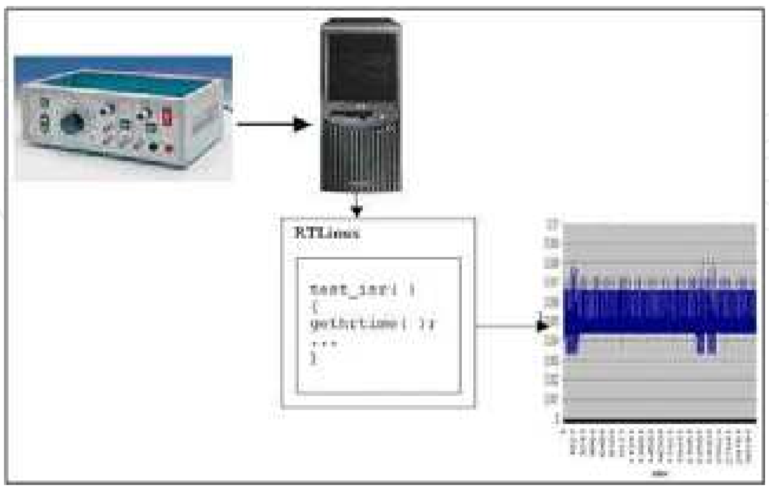
Fig. 3. RT-golem architecture

The RTLinux microkernel creates two separate software areas. First, it is a real-time space (compare fig. 2), in which real-time tasks run. Second, it is a non-real time task space, which contains the non-real-time Linux kernel space, and the Linux application space. (As it has been yet noted (see section 1), the real-time task space is realized inside the kernel space, and barriers between the two task spaces are realized by software means only). The RT-golem contains a periodic task, and an interrupt service routine. The periodic task represents the generalized data acquisition task, which in each instance measures its starting time and its finishing time, and sends the time stamps to a mass storage device (a hard disk).

In general, it would be possible to process the values either online, at the time in which they were being produced, or offline, after the simulation has been finished. The RT-golem operation creates a basic load in the system. Both theory and experience require testing under load, however, on certain PC hardware architectures, more jitter has been observed under less workload (Proctor, 2001). Hence, it has been decided to minimize the inevitable basic load caused by the measurement device as possible, and the offline data processing has been used. RTLinux itself does not provide a resource, which makes possible to access hard disk directly. Therefore, data has to be transferred from the real-time space to the non-real-time space first, and it can be stored to the hard disk using standard Linux API kernel calls then.
The interrupt service routine (ISR) is designed to support the ISR latency measurement. For the ISR latency measurement, a new method has been designed (Moryc, 2007). It is called the saturation method.

3.2 Interrupt latency measurement

Interrupt latency is one of the most important real-time system characteristics, as it describes the system ability to react to (and upon) unplanned conditions. A saturation method has been designed for measuring interrupt latency (see fig. 4). A rectangle-wave signal from a precise X-tal driven signal generator enters the system hardware input and triggers an interrupt request. The request is further processed by the system, and at the end, the control is given to the interrupt service routine. As the input signal is periodic and its period is stable, the interrupt service routine is run periodically, and it is possible to observe a time interval between two successive ISR routine starts, and its jitter. When the incoming interrupt rate is boosted, the time between two successive ISR routine starts decreases, until the saturation point is reached. Then, the minimum time between two successive ISR starts equals to the interrupt latency. It consists of latency times caused both by hardware and software resources. Since the saturation presents the maximum IRQ rate load on the measured system that it is capable to accept, the method is expected to provide comparable results across various hardware platforms.



```
test_isr( )
{
gethrtime( );
...
}
```

Fig. 4. ISR latency saturation method

3.3 RT-golem development and evaluation

The model architecture design and development process had been iterative, and the model had been verified after each step had been finished. Analyzing the verification results, it has been recognized, that a measurement tool, which encompasses the whole range of typical RTLinux resources and provides a deeper insight, is needed. Based on the RTLinux analysis, the following important RTLinux characteristics have been identified:

- scheduler precision (measured as task starting time jitter),
- interrupt latency time,
- execution time of typically used API services,
- pipe write and read operations,
- shared memory write and read operations,
- thread switching time, and
- I/O port read and write access time.

(The I/O access is included here, as it presents the basic (and sometimes the only) method of communication with many I/O cards. Nonetheless, it shall be avoided as possible, as it bypasses the operating system, which causes a potential for significant problems.) During the iterative design and development process, the RT-golem has been evolved from a mere data acquisition task to an advanced measurement tool. The advanced version of RT-golem consists of a periodic task, and an interrupt service routine. The periodic task includes two threads. It is possible to set priority and period of both threads, as well as to disable one or more parts of the task. This way, it is possible to balance the workload that the RT-golem causes inside of the model architecture.

4. Experimental setup

A set of measurements has been designed and performed. The periodic task starting time and finishing time jitters have been measured, under different system workload conditions:

- basic load (operating system kernel, standard daemons, RT-golem test task),
- basic load and additional non-real-time load (routine that copies short files),
- basic load and additional real-time load (additional 15 RT-golem tasks).

on two test systems (see tab. 2) :

- PC Dell GX 280,
- PC no name.

The source code of the RTLinux scheduler contains a comment suggesting that this scheduler should not be used for scheduling more than 10 tasks. For the verification of the scheduler behavior, an experiment was designed, where the system is heavily loaded with additional 15 RT-golem workload tasks, and the RT-golem test task jitter is measured. The additional 15 RT-golem tasks have been configured that way, they have created maximum acceptable load for the system, that is, the highest load at which the Linux kernel yet does not report lost timer interrupts.

Experimental results are given and discussed in details later (see section 6, Experimental results discussion, and section 9, Conclusion and future work). Briefly, they have shown, that:

- The implemented simple RTLinux scheduler manages to handle more tasks than it was suggested by its authors.

- The Linux kernel operation increases the amount of internal RTLinux microkernel signals, which in turn increases the RTLinux microkernel latency. This influence is measurable and significant.
- The influence of the hardware on the real-time characteristics of the operating system virtual resources is measurable and significant.
- The jitter values can be sorted by their amplitude into classes (Proctor, 2001).

Operating system

Linux kernel	v. 2.2.19
RTLinux microkernel	RTLinux Free v. 3.1

Hardware

PC DELL GX 280

CPU	Intel P4 at 3.0 GHz, 1 MB of L2 cache
RAM	1024 MB (only 64 MB used by Linux)
HDD	SAMSUNG SV0842D, 75 GB, SATA
	WDC WD800JD-75JNC0, 8 GB, ATA-66

PC no name

CPU	Intel P4 at 2.4 GHz, 32 K of L1 cache
mainboard	MSI 865 PE Neo2-P
RAM	256 MB (only 64 MB used by Linux)
HDD	Seagate Barracuda ST380011A, 80 GB, ATA-100
	Maxtor WDC WD100EB-00BHF0, 10 GB, ATA-66

Table 2. Test systems details

5. Win-golem, a RT-golem port to MS-Windows

After the RT-golem method has been evaluated by experiments, it has been required to port it to another target environment, based on Microsoft Windows and RTX for Microsoft Windows Real-time Extension, and to perform comparison of both real-time environments (RTLinux and RTX for Microsoft Windows).

First, the new intended target platform had been analyzed. The RTX microkernel is similar to the RTLinux microkernel in functionality, but different in realization. RTX is a proprietary solution, hence we have to rely on its description supplied by the vendor (Cherepov, 2002). The RTX microkernel supports real-time tasks, which are called RTSS (real-time subsystem) threads. It is interconnected with the Windows kernel via two access paths. The first access path is created in the Windows HAL, and the second one is the interface for connecting device drivers. Basically, it is necessary to modify the Windows HAL, because:

- interrupt isolation between the Windows kernel and the RTSS threads has to be added,
- high-speed clocks and timers need to be supported,
- shutdown handlers need to be provided.

The two interconnection points between Windows kernel and RTX microkernel mentioned above make possible to realize connection between the RTX microkernel and the Windows kernel, providing the same functionality as the interface between the RTLinux microkernel

and the Linux kernel (compare fig. 4, fig. 5). The communication interface between Windows and the RTX kernel implements a low-latency client-server mechanism, which includes both buffers and Service Request Interrupts (SRI). Due to the communication interface, a subset of Windows application programming interfaces (API) services is callable from within a RTSS thread. It includes APIs for storing data to file, thus real-time pipes are neither needed nor available in the API services set. However, we can reasonably suppose, that similar IPC mechanisms are necessary to provide similar functionality, no matter whether they are hidden for the programmer. Some of the Windows APIs available from the RTSS threads are listed as non-deterministic, i.e. they can cause significant jitter when called from a RTSS thread. High-speed (and high resolution) clocks are needed for realization of precise real-time timers. Within a RTSS thread, time is recognized with 100 ns step and resolution. Shutdown handler is a mechanism delivering more robustness to the real-time RTSS subsystem when the Windows subsystem is crashed or regularly shut down.
It can be summarized, that following significant differences from RTLinux exist:

- no real-time pipes or their equivalents are available in the API service set,
- it is possible to call a subset of Windows API services directly from a RTX (RTSS) real-time task,
- time is recognized with 100 ns resolution,
- Windows-controlled interrupts are masked while the RTX (RTSS) real-time task runs,
- a real-time interrupt routine has two mandatory parts, which can be seen as the upper and bottom ISR part,
- it is possible to implement a shutdown handler as the last resort resource.

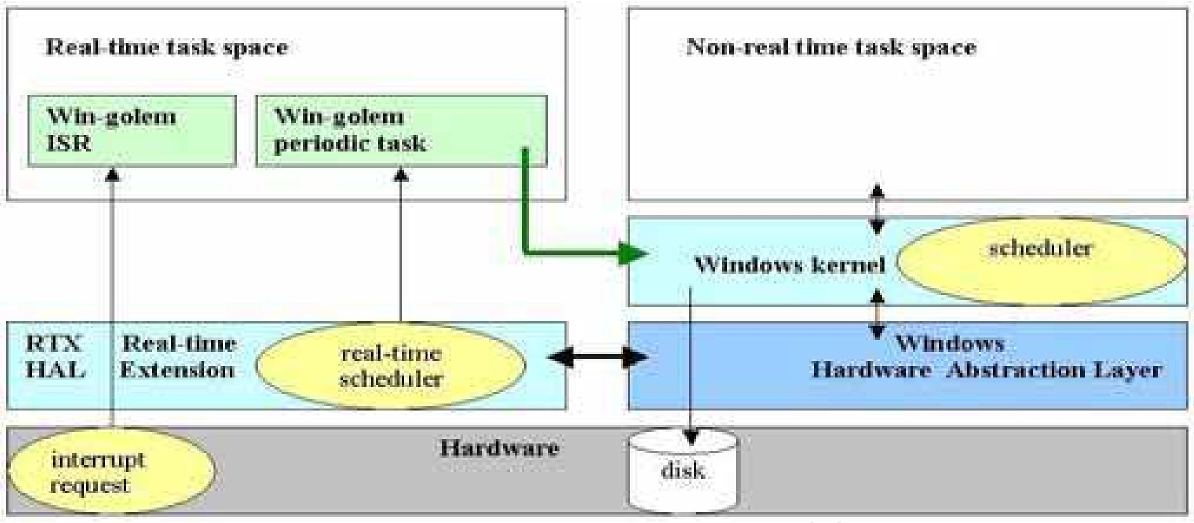
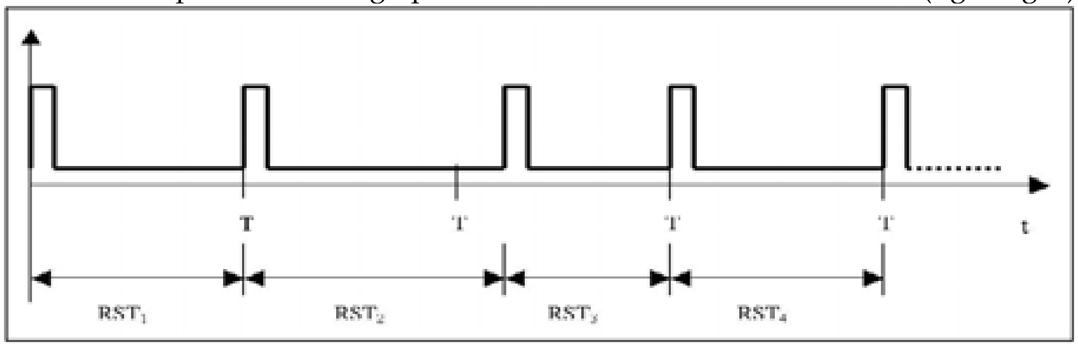


Fig. 5. Win-golem architecture

As the RTLinux and RTX operating environments are functionally similar and their key resources are almost the same, it can be supposed, that the RT-golem architecture design can be adapted for the jitter measurement in the RTX operating environment as well. After the target environment analysis had been performed, the realization issues have been approached. The RT-golem is written in C language, thus it should be easily portable. Nonetheless, both the environments contain many non-portable extensions, and as a result, the design had to be partially re-written. The RT-golem task re-written for the RTX environment is called Win-golem. For the RTX environment properties evaluation, it has been used the same experimental setup as described in section 4 (Experimental setup).

6. Experimental results

Experimental results are given below. For the evaluation, hundreds of data sets and graphs have been used, but here only a handful of them, which contains typical or interesting cases, is meaningful to be presented. The graphs are often arranged in pairs for easy comparison. (In the pair, the left graph is referred to as fig. x a, while the right graph is referred to as fig. x b.) There are presented relative starting time (RST) graphs, relative finishing time (RFT) graphs, and statistical evaluation graphs. Relative starting time is the time when the given instance of a periodic task starts, measured from the previous instance starting time. Because of the relative starting time definition given above, the relative starting time jitter graph consists of spikes oscillating up and down around the nominal value (fig.6, fig. 7).



T – period (relative starting time (RST) nominal value
RST1,..RSTn – relative starting time of the 1-st,..n-th instance

Fig. 6. Relative starting time measurement

The instance's relative finishing time is measured from this instance's relative starting time. The statistical evaluation graphs present statistical measures comparison bar graphs, showing mean vs. median comparisons, and standard deviation vs. interquartile range (IQR) comparisons. The graphs provide useful information on typical values as well as outlying values importance in the sample, but further statistical evaluation could be needed (see section 7.2).

7. Experimental results evaluation

7.1 Intuitive evaluation

It can be concluded from the presented experimental results, that the benchmark periodic task relative starting time jitter (kernel latency jitter) is significantly lesser on the RTX-based measurement architectures, than on the RTLinux Free-based architectures (fig. 7, 9, 13). The median of the task finishing time is approximately the same within the applied range of test architectures and workloads. Under RTLinux Free, the port write time median value is ca. 15% less than the port read time median value, but under RTX both medians are the same (fig. 11). Similarly to the task starting time, task finishing time shows lesser jitter on the RTX-based architectures (fig. 8, 10, 14). Evaluating the RTLinux Free-based measurement architecture, it has been observed on the graphs presented above, that most of the jitter instances are near the best-case values, but sometimes significantly higher spikes occur.

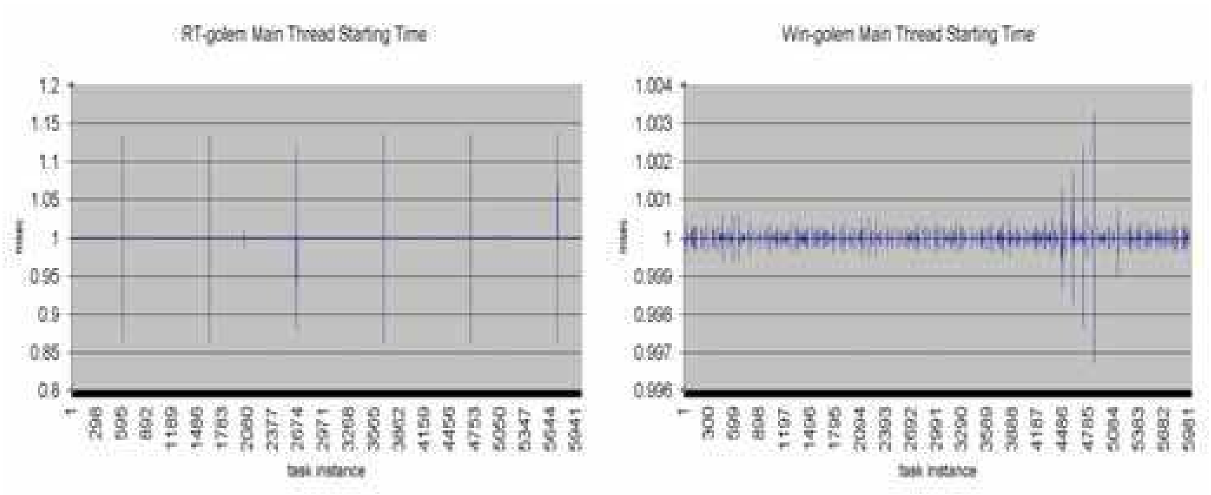


Fig. 7. RT-golem and Win-golem results comparison: periodic task starting time, PC Dell, basic workload only

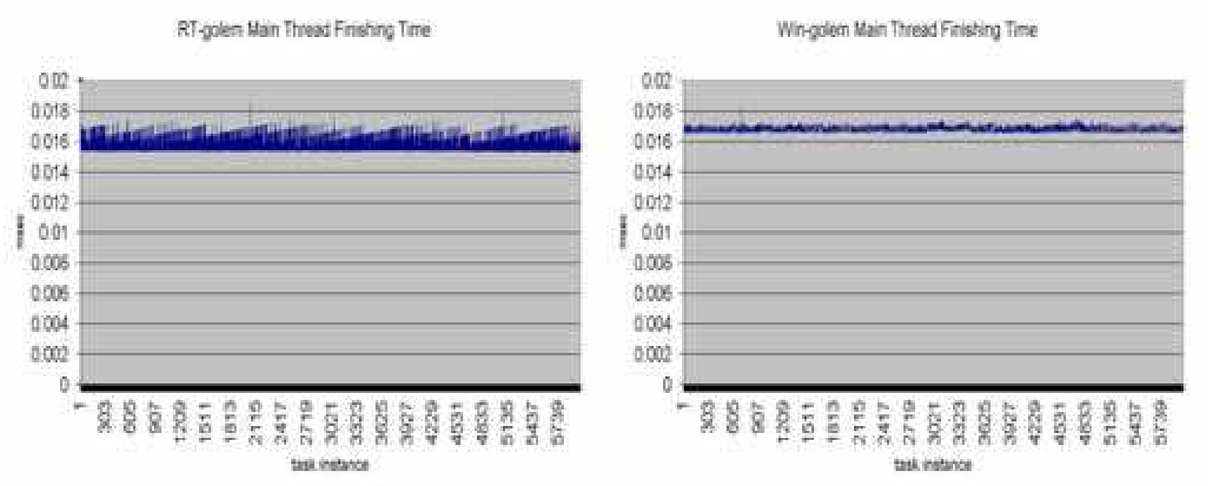


Fig. 8. RT-golem and Win-golem results comparison: periodic task finishing time, PC Dell, basic workload only

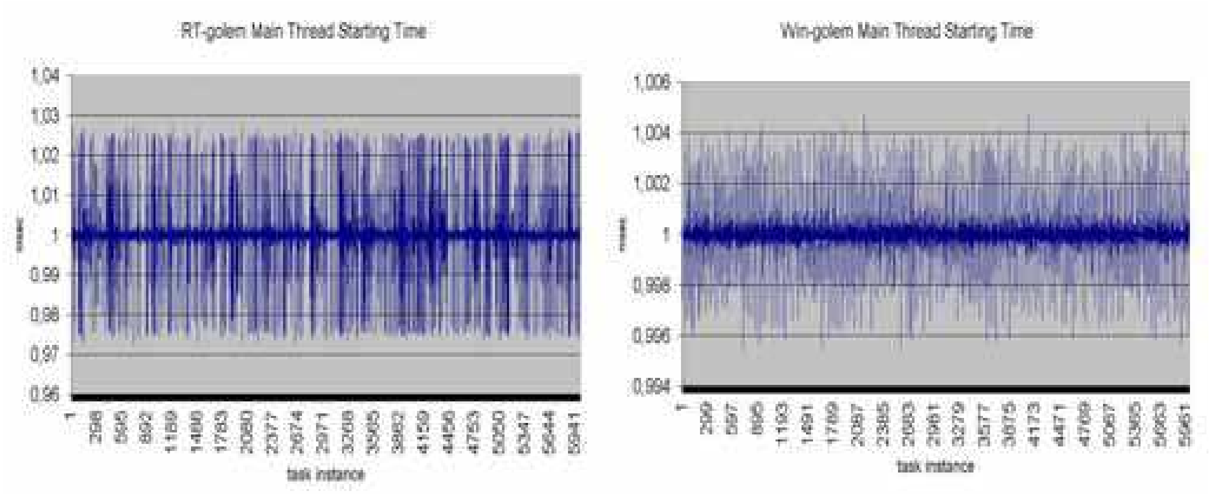


Fig. 9. RT-golem and Win-golem results comparison: periodic task starting time, PC Dell, basic and additional workload (copying files)

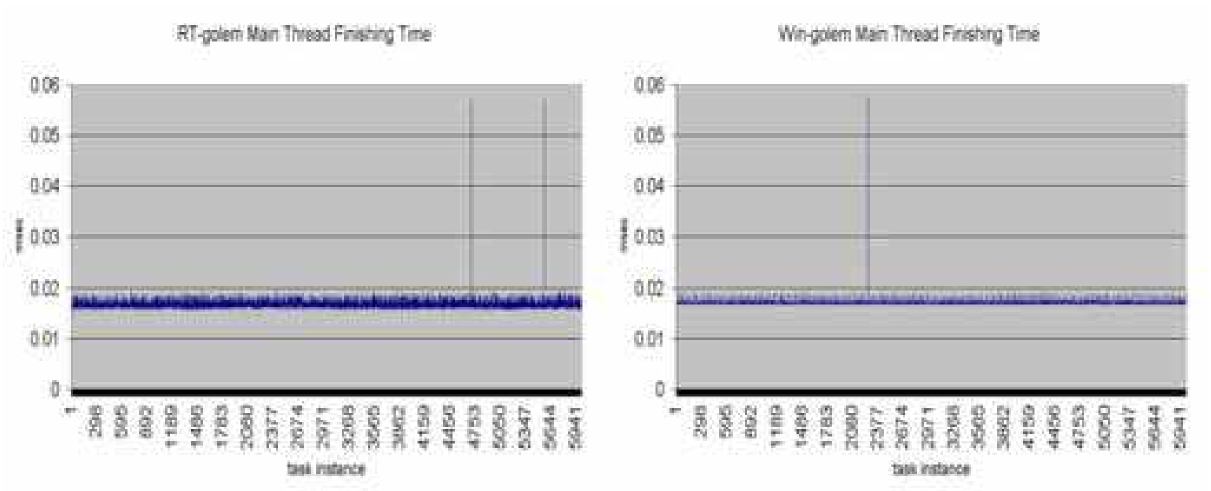


Fig. 10. RT-golem and Win-golem results comparison: periodic task finishing time, PC Dell, basic and additional workload (copying files)

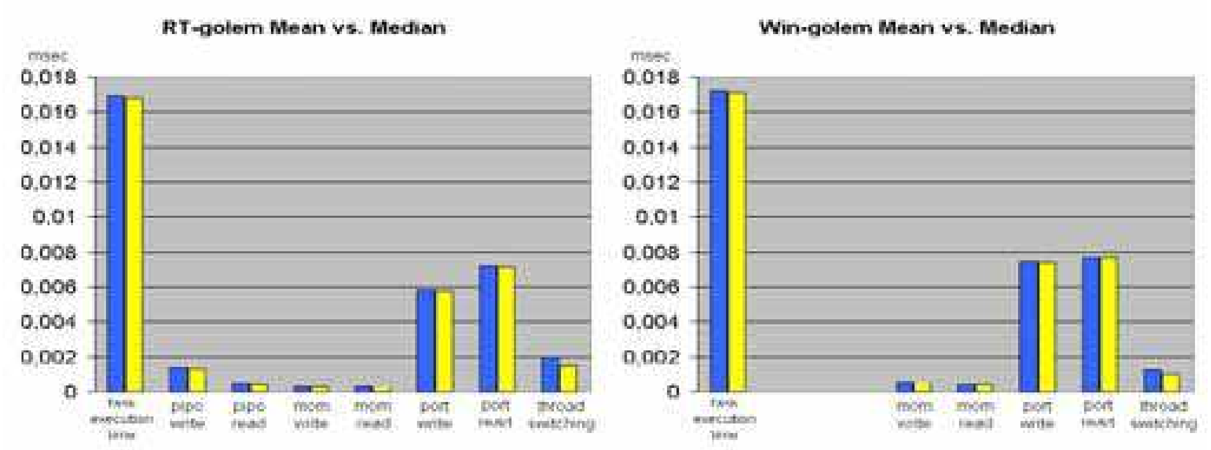


Fig. 11. RT-golem and Win-golem results comparison: execution time means vs. medians, PC Dell, basic and additional workload (copying files)

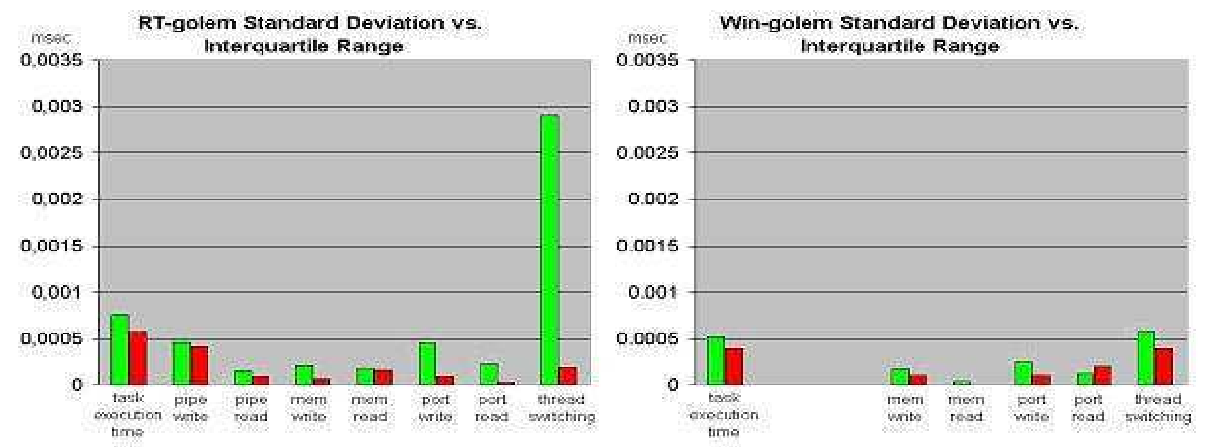


Fig. 12. RT-golem and Win-golem results comparison: execution time standard deviations vs. interquartile ranges, PC Dell, basic and additional workload (copying files)

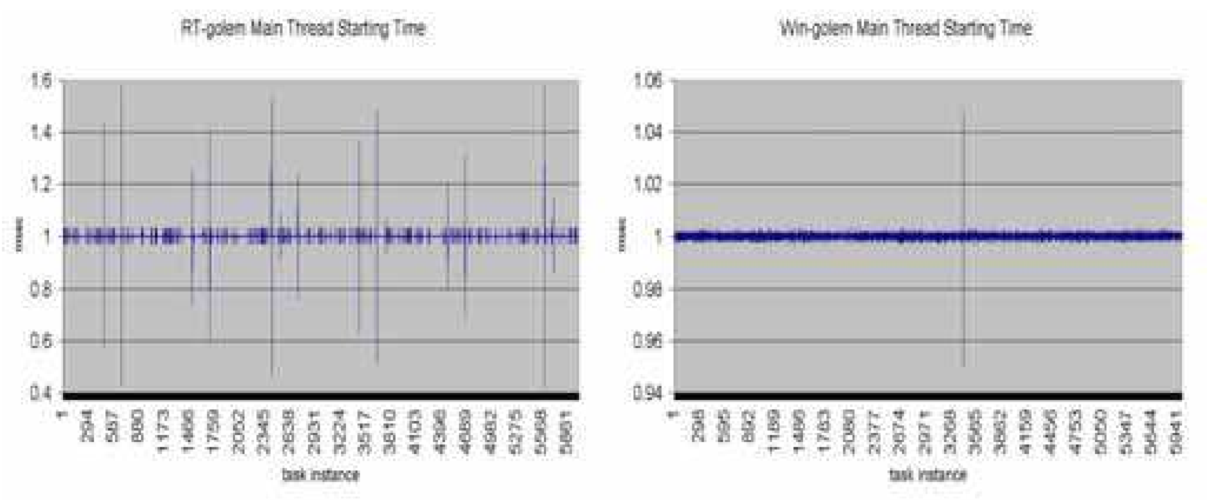


Fig. 13. RT-golem and Win-golem results comparison: periodic task starting time, PC no name, basic and additional workload (copying files)

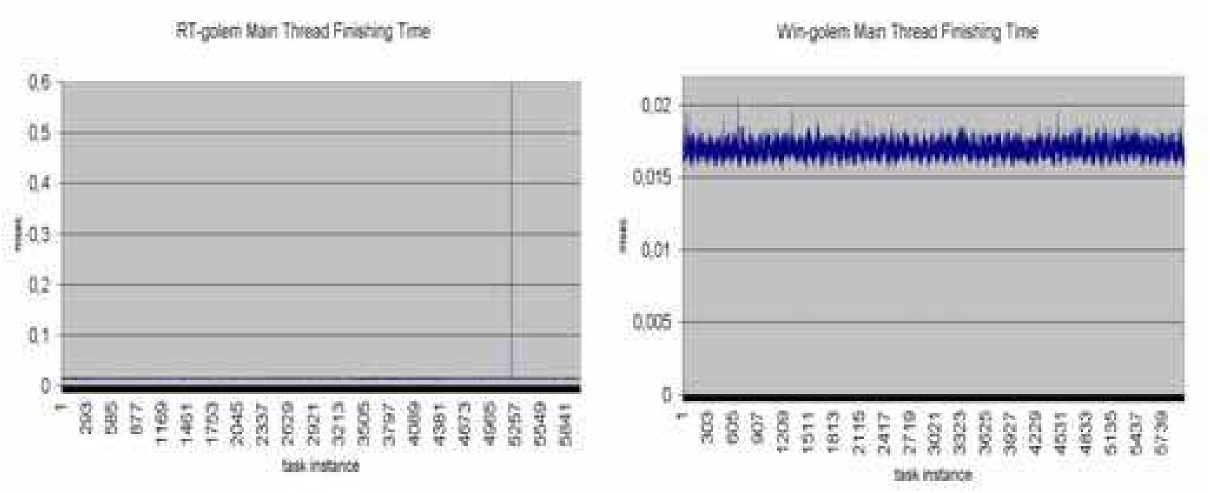


Fig. 14. RT-golem and Win-golem results comparison: periodic task finishing time, PC no name, basic and additional workload (copying files)

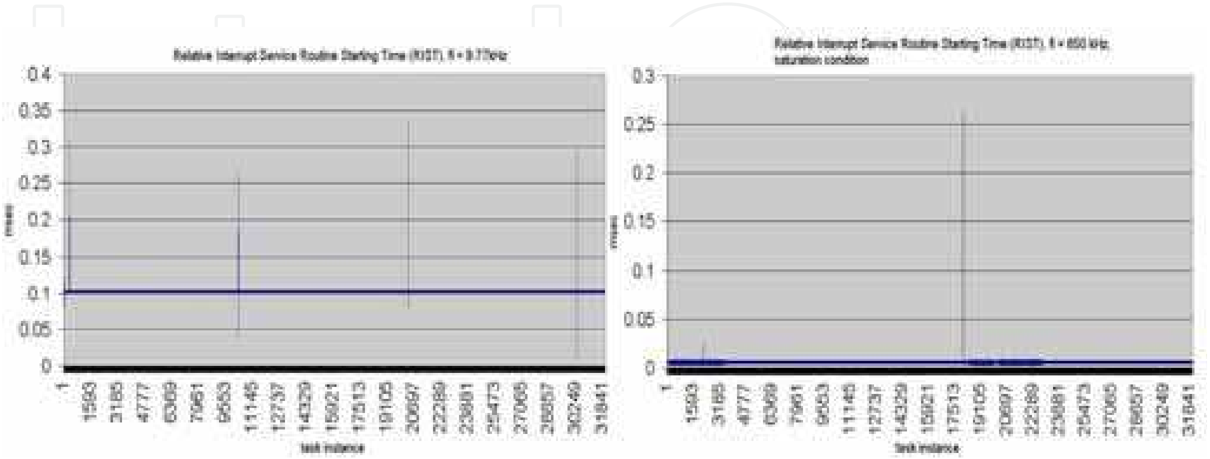


Fig. 15. Interrupt service routine starting time, RT-golem, PC Dell

These spikes can form typical patterns (relative starting time measurement, fig. 7a, 13a), or can be observed randomly (fig. 10a, fig 14a), but in any case their amplitude is typical for the underlying system layers. Evaluating the RTX and Windows-based architecture, significantly less spikes have been found (fig. 7b compared to fig. 7a, fig. 13b compared to fig. 13a), or none at all.
Last but not least, interrupt latency measurements have been evaluated (see fig. 15). On the left frame (fig. 15 a) can be seen the ISR starting time jitter at input signal frequency of 10kHz, while the right frame (fig. 15 b) provides the ISR starting time graph at input signal frequency of 650 kHz, when the saturation condition is reached. In both cases, the same hardware (PC Dell) as well as the same operating system (RTLinux) has been used. Apparently, the load conditions are very different, but the observed spikes have the same amplitude of ca. 25 msecs. It can be concluded, the spikes are independent on the workload, and they characterize the measurement architecture.

7.2 Statistical evaluation

Background
All task jitter instances that occurred or will occur in given measurement architecture under given load are considered a population, and measured task jitter instances represent a known sample drawn from this population. It can be formulated a null hypothesis about the population, that the population can be described with given distribution, as well as the alternative hypothesis to the aforementioned null hypothesis. Then, it can be tested, whether it is reasonable to reject the alternative hypothesis in favor of the null hypothesis. If it has been not possible to reject the alternative hypothesis, it could be expected, that the null hypothesis is valid and true, but it is not possible to be ever sure of that, because of type I and type II errors existence. The type I error (α, significance level) means the probability that the true null hypothesis is rejected in favor of the false alternative hypothesis, while the type II error (β) means the probability that the false null hypothesis is not rejected in favor of the true alternative hypothesis. Unfortunately, it is not possible to minimize both the type I and type II errors simultaneously. In engineering applications, usually it is chosen $\alpha=0.05$. When the resulting β value is not sufficiently low, it can be lowered by changing the test statistic used, or by drawing larger sample from the population.

Application
The instances of a real-time task shall be started in intervals specified by the task period, and not affected by previous system state or by previous system activity. Thus, they shall be independent to each other. If they can be described with Poisson distribution, they are sufficiently independent to each other. It has been formulated the hypothesis, that the task instance starting times follow Poisson distribution, and the hypothesis has been tested using χ^2 test. Results are given in table 3.

workload applied	χ^2 test result
basic load and additional RT load	H_A rejected in favor to H_o
basic load and additional non-RT load	H_A not rejected

H_o : The task instance starting times follow Poisson distribution.
H_A : The task instance starting times do not follow Poisson distribution.
Significance level: $\alpha=0.05$.
Measurement architecture: PC Dell GX 280 (see tab 2.), RT Linux

Table 3. χ^2 test results

8. Feasibility Study

Based on the outcomes, it has been designed and realized a data acquisition system for physical model of a steel flow in an interladle.

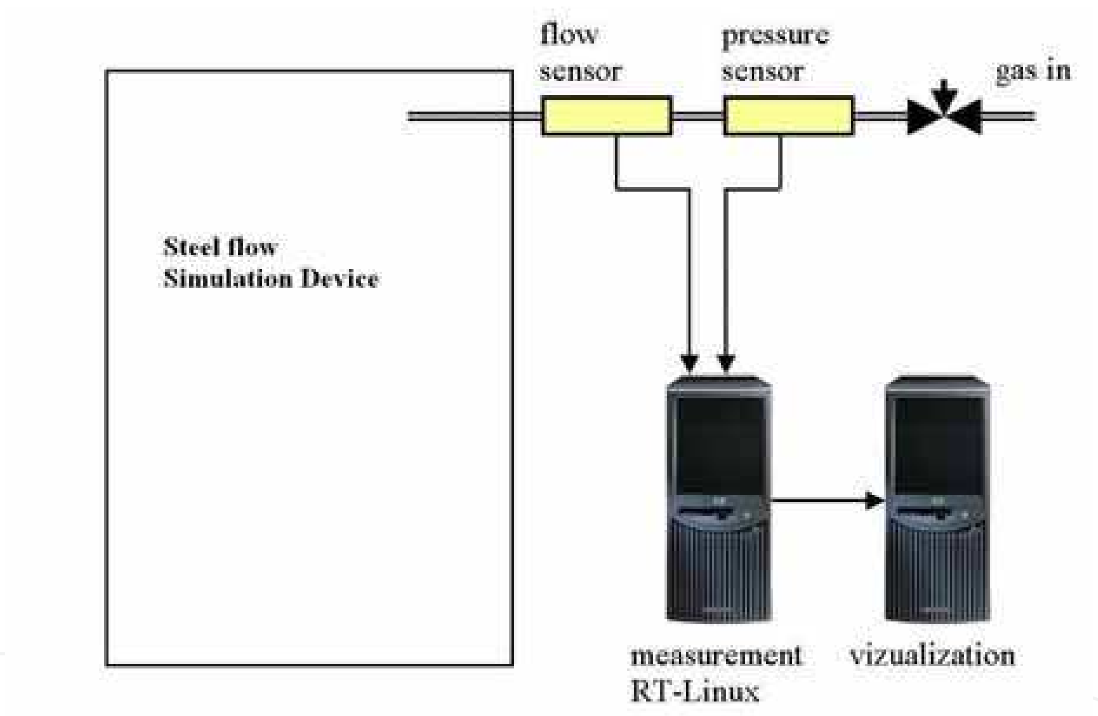


Fig. 16. Steel flow model

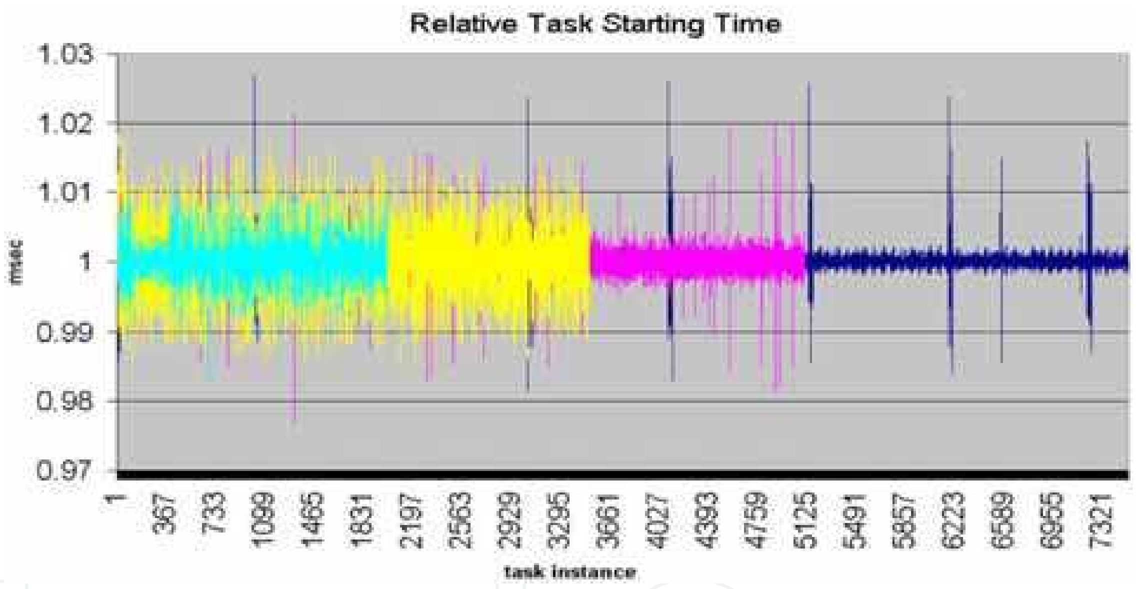
In the model, the steel flow is simulated by a gas flow in a liquid environment. The data acquisition system consists of

- pressure and flow rate sensors,
- measurement subsystem,
- control and visualization subsystem.

It is required by the end-user, that both the gas flow and the gas pressure are acquired once per two microseconds, with maximum tolerance of +/- 100 µsecs. The control and visualization subsystem shall provide:

- control of the measurement subsystem via rcmd utility,
- graphical presentation of currently measured values,
- graphical presentation of previously acquired values stored in database.

At the beginning of the design process, it has been intended, that the measurement subsystem will be built using industrial PC (Pentium P2 Celeron 333 MHz), and ICP-DAS 823 A/D card, which were available.
Most of the design and verification effort has been concentrated on the measurement subsystem, as the subsystem contains the core of the design, and has to meet relatively strict deadlines. At early design stages, it has been concentrated on the question, whether the intended hardware will fit the deadline specifications. Series of simulations have been performed using the RT-golem architecture ported on the target hardware. The task has been configured as close to the intended target design as possible, and various workload conditions have been used to get the possibly most adverse results (see fig. 17, the blue, red and yellow curves). It is commonly supposed, that the worst jitter is produced when the system is heavily loaded, but due to cache loading and flushing effects, the system behavior under lower workloads needs to be examined too (Proctor, 2001).



blue - RT-golem, basic workload only
red - RT-golem, basic and real-time workload
yellow - RT-golem, basic and non-real-time workload
cyan - Steel flow model realization, real workload

Fig. 17. Steel flow model: measurement task relative starting time jitter, simulation vs. realization

The control of the measurement system is realized using the rcmd protocol, as both the computers are connected with a cross-wired Ethernet cable in secure environment. The measured data messages delivery is realized using the Microsoft Server Message Block (SMB) protocol.
The control and visualization subsystem ensures experiment control and supplies results visualization. It's typical latency has been stipulated to meet a common human reaction time (which is not less than 200 msecs). The visualization subsystem displays flow vs. time and

pressure vs. time online graphs. Its design consists of two periodic threads, the first one timed at sample rate, and the second one timed at display rate. It is necessary to adjust both the sample and display rates on the given hardware to achieve optimum performance. On the used Dell notebook (Pentium 4 Celeron at 2.4 GHz), the sample rate has been set to 10 msecs, while the display rate has been set to 100 msecs.
During the commissioning process, ca. 100 measurements were made under various possible working conditions specified, and the measurement and visualization system has been found meeting the requirements (see fig. 17, the cyan curve).

9. Conclusion. Future work

9.1 Results discussion

It can be supposed, that the significant worst-case jitter spikes observed in the experiments with RTLinux Free are caused by cache writing and flushing effects. (Proctor, 2001) suggested the same conclusion, what can be seen as another argument supporting this opinion. But, as the measurements are performed on the top of the hardware and software stack, and the virtual resources presented to the measurement task by the operating system API services are quite distant to the resources presented to the operating system by the hardware (more precisely, by the part of the operating system realized in hardware), it is not possible to validate such hypothesis by methods described here. The mere absence of this phenomenon on both test architectures using the RTX operating system can imply, that the RTX microkernel prevents the hardware from flushing the cache arbitrarily. Moreover, some further tracks can be given. (Cherepov, 2004) notes that video drivers are the most cache demanding part of the Windows operating system, and in the RTX platform evaluation kit, video is used as a workload. Video is a real-time task as well as a RTX microkernel (RTSS) task. Thus, the conflict between a video and a real-time RTSS task can be seen as the conflict between two real-time cache-demanding tasks, which can lead to swapping the RTSS task code out of the cache. Unfortunately, the RTX microkernel source code is not freely available, and therefore it is not possible to verify the tracks given above with the code analysis. However, the aforementioned discussion can suggest, that the mechanism of locking the real-time code in the hardware cache is worthy to be studied and implemented.

9.2 Applicability in engineering practice

In most today applications, it is used a PLC for technology control or data acquisition, and a PC with the Microsoft Windows for data visualization. However, a fully PC-based technology control and visualization system can be considered as well. In the present time, RTLinux and RTX-based control system solutions are yet recognized by business vendors (e.g. Siemens WinAC platform). The published method of jitter measurement in multipurpose real-time systems has been successfully applied, and it is ready to be used in similar designs.

9.3 Vision. Future work

The outcomes of the published work are going to be used in Experimental real-time database testing system (see Acknowledgement below). The proposed benchmark methods yet have been used to evaluate possible platforms for the real-time database realization, and two more applications are intended:

- RT-golem port on QNX,
- design and realization of a communication architecture, which makes possible to measure time needed for various TCP/IP communication steps and phases.

It can be suggested, that real-time databases will present an interesting field of study. Applications in flight supporting devices, mobile communication devices, speech recognition systems, and many other areas can be foreseen.

10. Acknowledgement

This research has been performed at the Technical University of Ostrava, Faculty of Electrical Engineering and Computer Science, Department of Measurement and Control, Centre for Applied Cybernetics, and supported by the Ministry of Education, Youth and Sports of the Czech Republic under Project 1M0567.

11. References

Cherepov, M. et. al. (2002). Hard Real-Time with Ardence RTX on Microsoft Windows XP and Windows XP Embedded, source: www. ardence.com

Dougan, C. & Mwaikambo, Z. (2004). Lies, Misdirection and Real-time Measurements, source: www.rtl.com

Krol V., Pokorny J., & Cernohorsky, J. (2006). Architecture of experimental real-time databases for embedded systems, *Proceedings of IFAC Workshop on Programmable Devices and Embedded Systems PDeS 2006*, pp. 384-388, ISBN 80-214-3130-X, Brno, Czech Republic, Feb. 2006

Krol V., Pokorny J. (2006). *Design of V4DB - Experimental Real-Time Database System, The 32-nd Anunual Conference of the IEEE Industrial Electronics Socienty,* Paris, France, Nov. 2006

Moryc, P. & Cernohorsky, J. (2007). Task jitter measurement under RTLinux operating system, *Proceedings of the International Multiconference on Computer Science and Information Technology,* pp. 849-852, ISSN 1896-7094, Wisla, Poland, 15.-17. Oct. 2007

Moryc, P. & Cernohorsky, J. (2008). Task jitter measurement under RTLinux and RTX operating systems, comparison of RTLinux and RTX operating environments, *Proceedings of the International Multiconference on Computer Science and Information Technology,* pp. 703-709, ISBN 978-60810-14-9, ISSN 1896-7094, Wisla, Poland, Oct. 2008

Moryc, P. & Cernohorsky, J. (2009). Task jitter measurement under RTLinux operating system, *Journal of Automation, Mobile Robotics and Intelligent Systems,* Vol. 4, 01/2009, pp. 62-65, ISSN 1897-8649

Pokorny J., Franek Z. (2008). *Databases in Real-Time: Experimental Research.* IT&T 8-th International Conference on Information Technology and Telecommunications, ISSN 1649-1246, Galway, Ireland, 2008.

Proctor, F.M. (2001). Measuring Performance in Real-Time Linux, *Proceedings of the Third Real-Time Linux Workshop,* Milan, Italy, Oct. 2001

Ripoll I. et al. (2001). WP1: RTOS State of the Art Analysis: Deliverable D1.1: RTOS Analysis, OCERA, source: www.ocera.org