Proceedings of the International Multiconference on
Computer Science and Information Technology, pp. 703 – 709

ISBN 978-83-60810-14-9
ISSN 1896-7094

Task jitter measurement under RTLinux and RTX operating systems, comparison of RTLinux and RTX operating environments

Pavel Moryc
Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Measurement and Control
Centre for Applied Cybernetics
Ostrava, Czech Republic
Email:
pavel.moryc@mittalsteel.com

Jindřich Černohorský
Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Measurement and Control
Centre for Applied Cybernetics
Ostrava, Czech Republic
Email:
jindrich.cernohorsky@vsb.cz

Abstract — **This paper compares task jitter measurement performed under RTLinux and RTX hard real-time operating systems. Both the operating environments represent Hardware Abstraction Layer extensions to general-purpose operating systems, and make possible to create a real-time system according to the POSIX 1003.13 PSE 54 profile (multipurpose system). The paper is focused on discussion of experimental results, obtained on PC hardware, and their interpretation.**

I. Introduction: Applicability of general-purpose operating systems in control systems

REAL-TIME task is a task, which meets prescribed deadline. Control system is a computer system, which physically realizes real-time tasks. The physical realization of the real-time task is not ideal, but it shows latencies and jitters.

Operating system and hardware support the real-time tasks with standardized means, which have non-zero and variable overhead. Kernel latency is defined as a delay between scheduled and actual execution of a task (e.g. between scheduled and actual instance starting time) [5]. The kernel latency is not stable, but it shows a jitter. Jitter is defined as a variable deviation from ideal timing event. The causes of jitter encompass code branching into paths with different execution times, variable delays implied whenever variable amount of code or data is stored in or read from caches and buffers, as well as noise and electromagnetic interference.

The substance of jitter can be seen in a need for parallel processing, which in turn stems from inevitable conflict between predictability and unpredictability. The control system has to be predictable and deterministic as much as possible. Control systems are intended to communicate with technology, as well as with people managing the technology (at least an emergency stop pushbutton is provided). Seen from this viewpoint, control system shall be appropriately flexible. As the result, a control system design represents a compromise between stability and adaptability.

It is possible to design a control system on two computers, one designed to meet technology demands, and the other designed to meet the user interface demands, but there is also the possibility to realize such system on one computer only. Computer system of this scope is standardized in the POSIX 1003.13 standard, as the PSE 54 profile (multipurpose system). PSE 54 - sized solutions are often preferred for their challenging possibility to use a broad range of standardized and low-cost hardware components, primarily intended for general-purpose computers, but on the other hand, they can involve lack of predictability typical for general-purpose (non real-time) systems.

Traditional general-purpose kernel provides full range of API services specified in the POSIX 1003.1 standard, and because of that, it cannot guarantee appropriately deterministic behavior, required in most real-time and technology control applications. Basic approaches to make such architecture more real-time one include

- low-latency kernel,
- preemptible kernel,
- hardware abstraction layer (HAL).

Low-latency kernel represents a traditional approach. The low-latency kernel is monolithic (i.e. it cannot be preempted by task), but its design minimizes latencies and jitters of the kernel API services typically used in real-time applications.

Preemptible kernel can be preempted by task. Preemptivity by task means, that a task can preempt the kernel just servicing another task, and enter the kernel instead. This type of premptivity is called reentrancy. However, at least some parts of a kernel (scheduling, interrupt service mechanism) cannot be made reentrant. Moreover, the idea of a preemptible kernel itself does not imply, that the kernel is deterministically preemptible, i.e. its jitters are acceptably stable.

Last but not least, hardware abstraction layers can be applied. A hardware abstraction layer receives timer interrupt and sends a virtual (software) interrupt 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. In the free time, real-time tasks can be run. Obviously, latencies and jitters of the HAL layer must be kept low enough by design.

978-83-60810-14-9/08/$25.00 © 2008 IEEE

II. RTLinux and RTX basics

Both the RTLinux and RTX represent real-time Hardware Abstraction Layers. RTLinux is the product of the FSMLabs, Inc., designed for the Linux operating environment, while RTX is the product of the Citrix Systems Inc. (Ardence), and is designed to run under the Windows operating system.

A. RTLinux

RTLinux microkernel (fig. 1a) implements a Hardware Abstraction Layer inserted between hardware and Linux kernel. Both the RTLinux microkernel and the Linux kernel have to communicate mutually, thus it is necessary to make certain modifications into the Linux kernel. The modifications include

- modification of macros for disabling and enabling interrupts in order to use internal HAL signals (software interrupts) instead of using cli and sti assembler instructions,
- modification of interrupt handlers, in order to use signals instead of direct Interrupt Controller access,
- modification of device drivers in order to prevent them from using sti/cli assembler instructions directly as well.

Within a RTLinux thread, time is measured with resolution that depends on hardware. On the Pentium 4 platform, the time resolution is equal to 32 ns.

[1] and [4] discuss the overall architecture more deeply. Basic resources used for interprocess synchronization and IPC communication in the RTLinux/Linux environment are semaphores, mutexes, shared memory pools, and real-time data pipes, which are more complex structures combining buffers and mutexes. Moreover, many A/D cards are controlled by direct access to registers, thus direct I/O access is an important feature.

B. RTX

RTX microkernel is similar to RTLinux kernel in functionality, but different in realization (fig. 1b). It supports real-time tasks, which are called RTSS threads. The Windows kernel is a proprietary solution, and its source code is not freely available to the public. Fortunately, two possible access paths to its modification exist. The first customizable part is the Windows HAL, and the second one is a device driver [2]. Basically, it is necessary to modify the Windows HAL for three purposes:

- to add interrupt isolation between Windows kernel and RTSS threads,
- to implement high-speed clocks and timers,
- to implement shutdown handlers.

The two interconnection points between Windows kernel and RTX microkernel mentioned above make possible to realize connection between RTX microkernel and Windows kernel, providing the same functionality as the interface between RTLinux microkernel and the Linux kernel (fig. 1b), [2, fig. 1]. The communication interface between Windows and RTX kernels implements a low-latency client-server mechanism, which includes both buffers and Service Request Interrupts (SRI) [2].

Due to the communication interface, subset of Windows API services is callable from within a RTSS thread. It includes APIs for storing data to file, thus real-time pipes are not 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 Windows APIs available from 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 real-time precise timer realization. Within a RTSS thread, time is measured 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 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 the RTX (RTSS) real-time task,
- time is measured with 100 ns resolution,
- interrupts to the Windows kernel are masked while the RTX (RTSS) real-time task runs,
- a real-time interrupt routine has two mandatory parts, which can be used as upper and bottom ISR part,
- it is possible to implement a shutdown handler as the last resort resource.

III. Applied measurement method

The measurement method applied is described in [3] and more deeply in [4]. Based on RTLinux resource analysis, following important RTLinux characteristics have been identified:

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

The I/O access is also included, because it characterizes hardware, and presents the basic method of communication with both sensors and actuators.

A generalized application has been written, which uses the above-mentioned RTLinux key resources. The application is called RT-golem, and its design is described in [3] and [4].

As RTLinux and RTX operating environments are functionally similar and their key resources are merely the same, it can be supposed, that similar design can be used for jitter measurement under the RTX operating environment too.

The RT-golem is written in C language, thus it should be portable. But, both the environments contain non-portable

extensions, and as a result, the design had to be partially re-written. The RT-golem re-written for the RTX environment is called Win-golem.

The measurement method is realized in measurement architecture. The measurement architecture includes both software (measurement and workload tasks and operating system), which all is a mere design abstraction, and hardware, which presents its physical realization. Nonetheless, we should note, that the border between software and hardware design is rather fuzzy, and many non-trivial resources formerly realized in software design are recently applied in hardware design too.

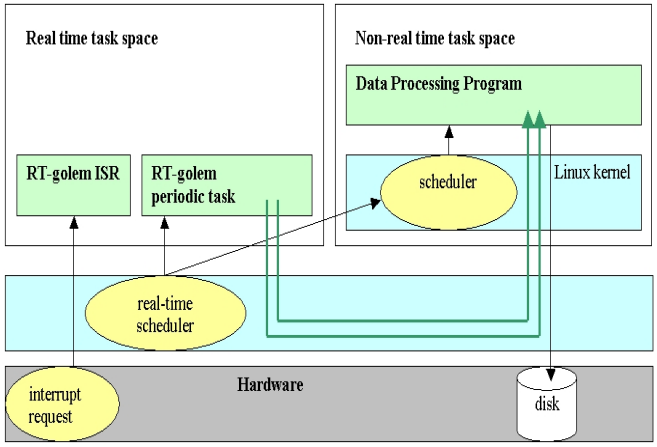


Fig. 1a RT-golem architecture

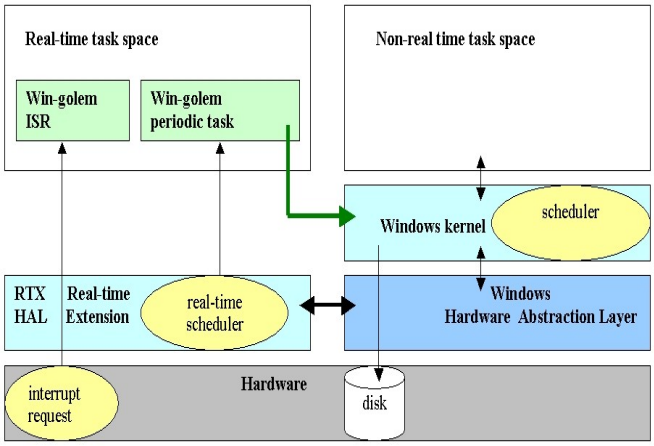


Fig. 1b Win-golem architecture

IV. Experimental setup

Series of measurements have been performed. It has been measured on different hardware (PC Dell, PC no name), under different operating environments (RTLinux Free v. 3.1, RTX v. 8) and under different workload (basic workload only, basic workload and additional workload). Experimental setup configuration chart is given in Figure 1, while hardware configurations are presented in Table 1.

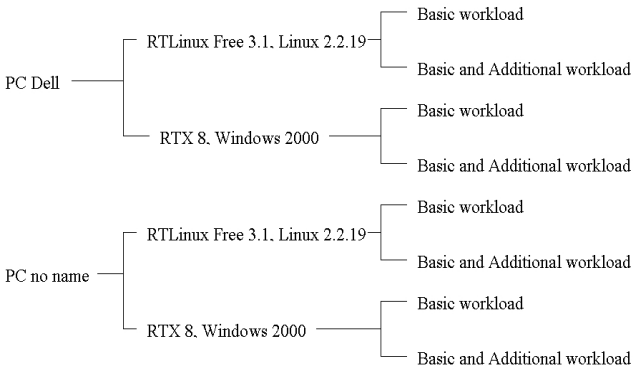


Fig. 2 Experimental setup configuration chart

Basic workload means a workload caused by the operating system kernel (kernel overhead), daemons normally needed and running, and the measurement task. Additional workload is presented with a shell script copying short files (bash or command.com) in a loop.

Table 1. Test system details

PC Dell GX 280

CPU	Intel P4 3.0 GHz, 1 MB L2 cache
RAM	1024 MB
HDD	SAMSUNG SV0842D, SATA, 75GB
	WDC WD800JD-75JNC0, 8 GB, ATA-66

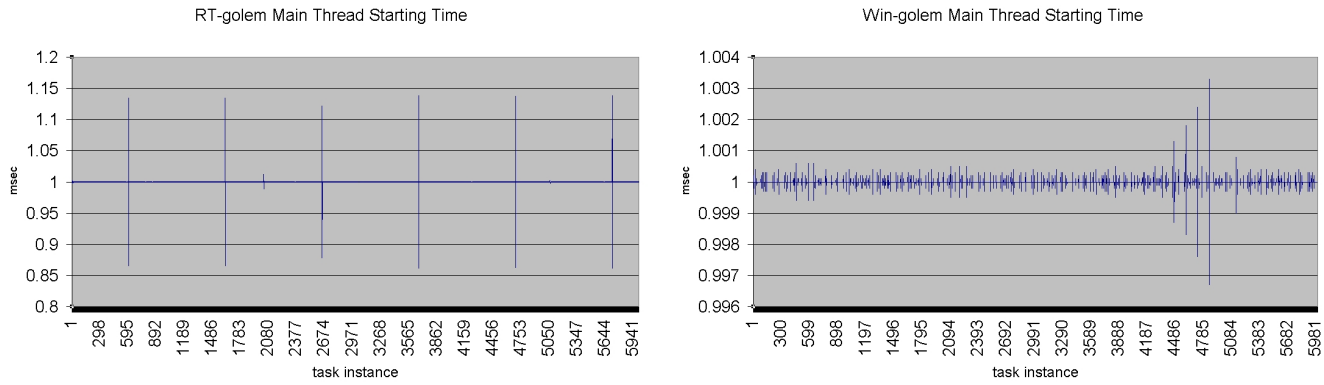


Fig. 3. RT-golem and Win-golem results comparison: Periodic Task Starting Time, PC Dell, basic workload only

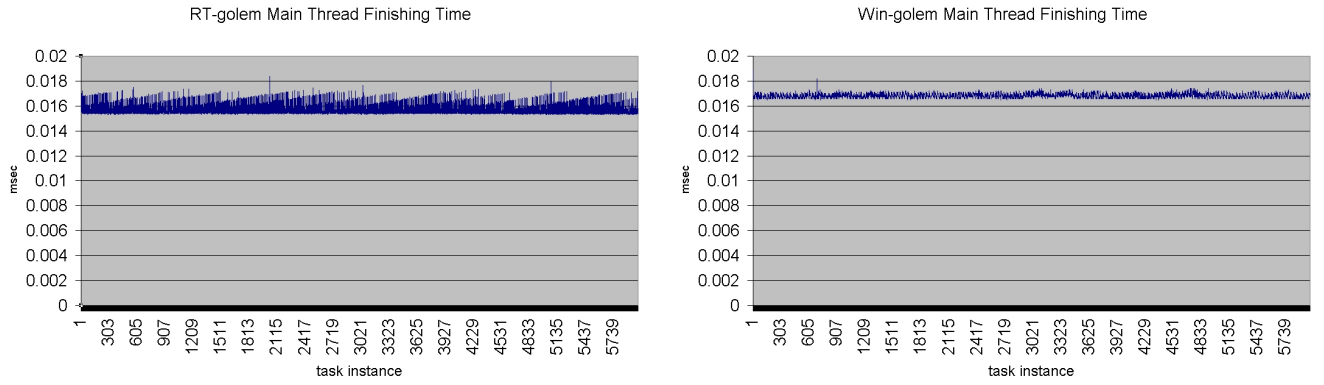


Fig. 4. RT-golem and Win-golem results comparison: Periodic Task Finishing Time, PC Dell, basic workload only

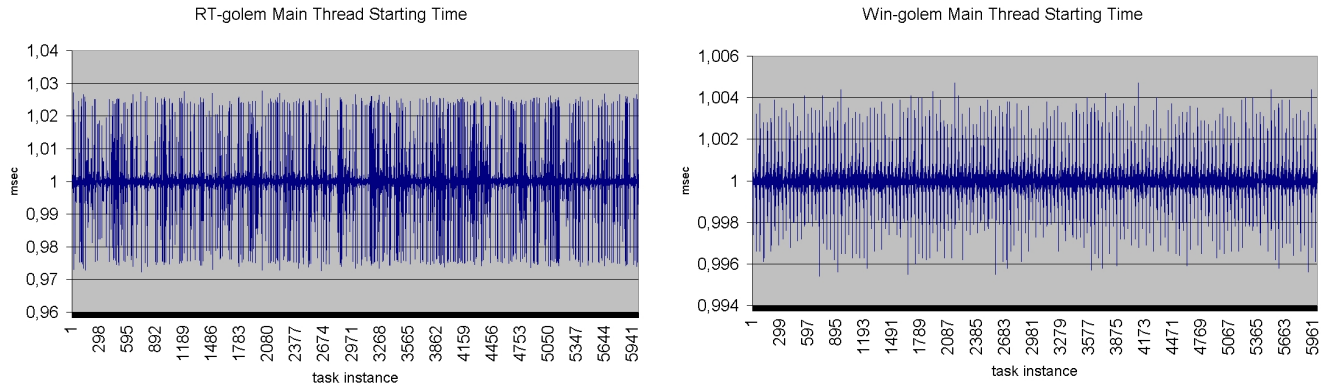


Fig. 5. RT-golem and Win-golem results comparison: Periodic Task Starting Time, PC Dell, basic and additional workload (copying files)

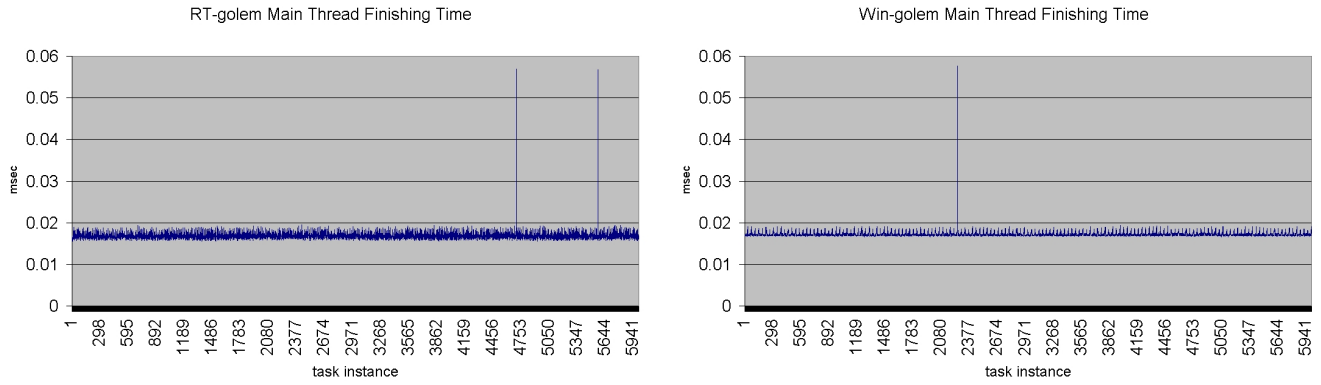
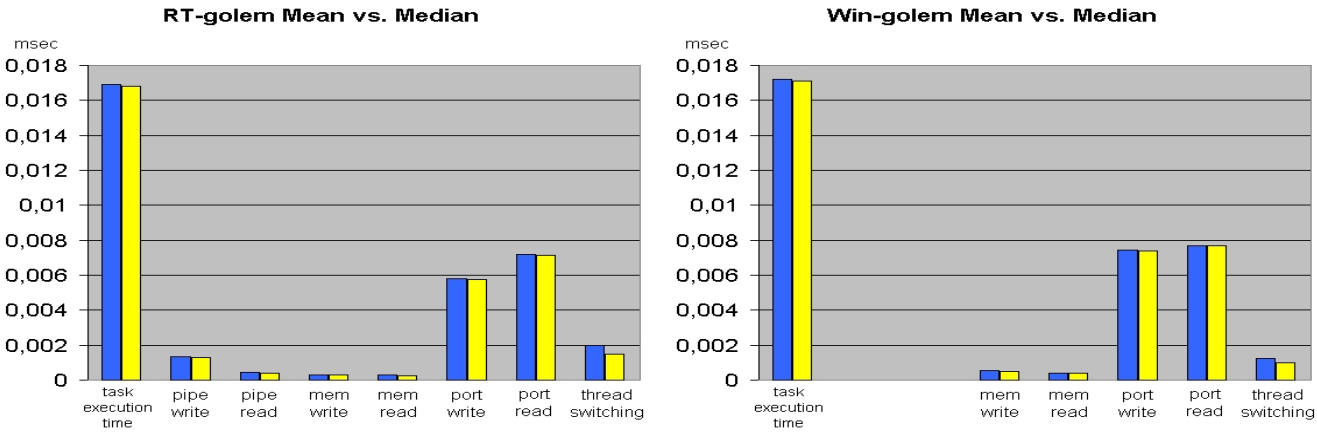


Fig. 6. RT-golem and Win-golem results comparison: Periodic Task Finishing Time, PC Dell, basic and additional workload (copying files)



blue: mean
yellow: median

Fig. 7. RT-golem and Win-golem results comparison: Execution Time Means vs. Medians, PC Dell, basic and additional workload (copying files)

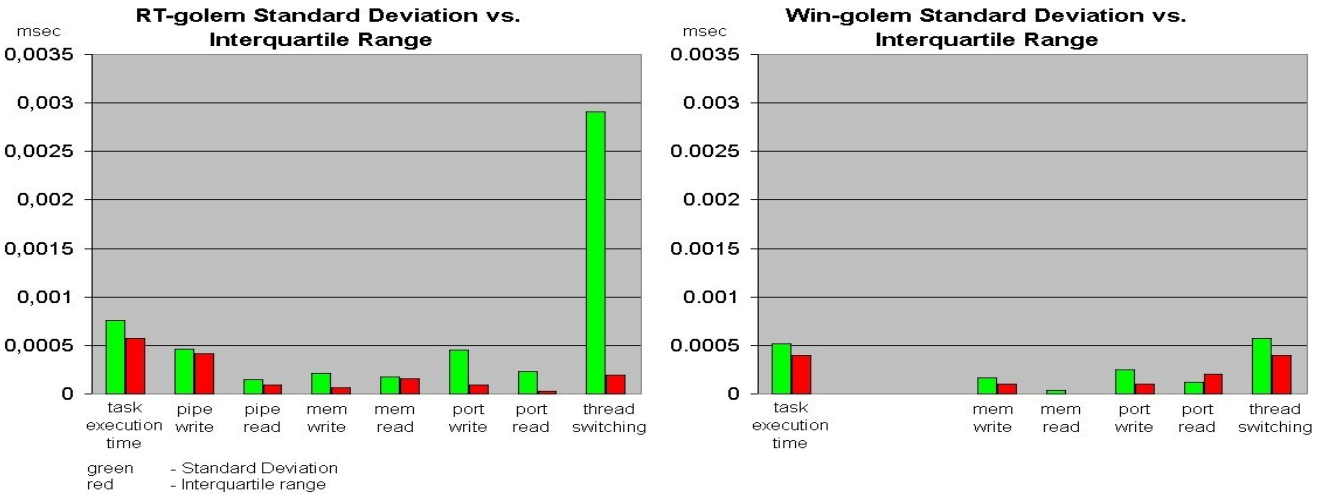


Fig. 8. RT-golem and Win-golem results comparison: Execution Time Standard Deviations vs. Interquartile Ranges, PC Dell, basic and additional workload (copying files)

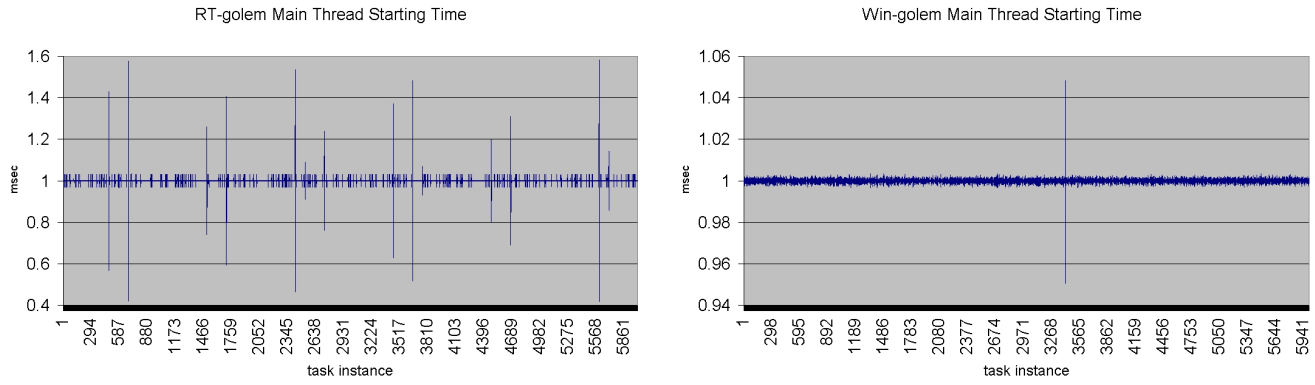


Fig. 9. RT-golem and Win-golem results comparison: Periodic Task Starting Time, PC no name, basic and additional workload (copying files)

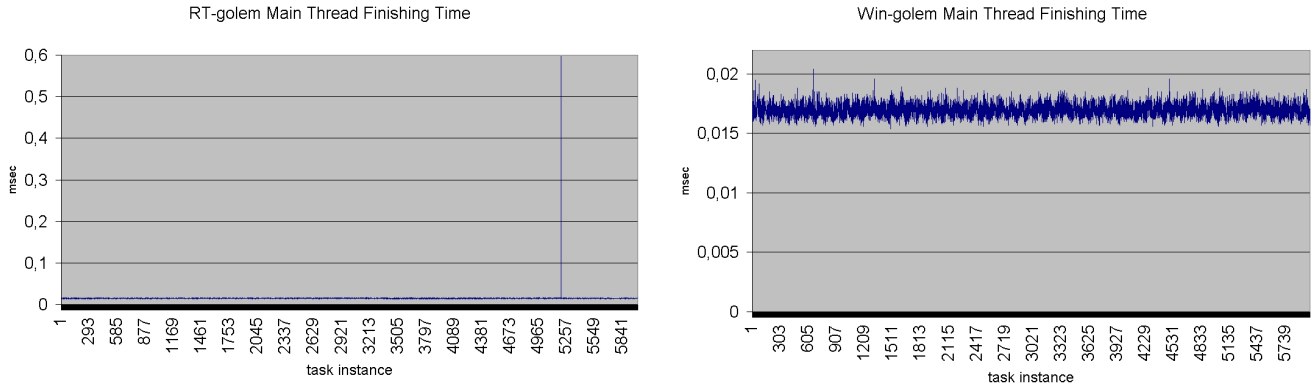


Fig. 10. RT-golem and Win-golem results comparison: Periodic Task Finishing Time, PC no name, basic and additional workload (copying files)

TABLE 1. (CONTINUED) TEST SYSTEM DETAILS

PC NO NAME

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

V. EXPERIMENTAL RESULTS

Representative selection of results is presented further. The series of graphs, presented in fig. 3 through 6, and in figures 9 and 10, show the task instance starting (or finishing) times comparison. These graphs are completed with statistical data evaluation graphs (fig. 7 and 8), which show mean vs. median comparison and standard deviation vs. interquartile range comparison on the PC Dell platform.

The task instance starting time is calculated from the previous task instance starting time (as in [3]). This means, the starting time delay impacts two adjacent values. First, the difference between the correct and delayed instance is longer, which causes the spike up on the graph, and then, the difference between the delayed and next correct instance is shorter, which causes the spike down. If both spikes are symmetrical, the second value is okay. Finishing time is calculated from task instance starting time [3]. Spikes on the relative starting time graphs below oscillate around 1 msec, because they show scheduling jitter, i.e. a difference of the actual relative starting time from the nominal value, which is 1 msec.

VI. CONCLUSION

It can be concluded from the presented experimental results, that the measured task starting time jitter (kernel jitter, [5]) is significantly lesser on the RTX-based measurement architectures, than on the RTLinux Free-based architectures (fig. 3, 5, 9). The median of 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. 7). As with the task starting time, task finishing time shows significantly lesser jitter on the RTX-based architectures (fig. 4, 6, 8, 10).

Using the RTLinux Free operating system, it has been observed (on graphs presented above and in [2], [3], and [4]), that most of the jitter instances are near the best-case values, but sometimes significantly higher spikes occur. These spikes can form typical patterns (measurement of task relative starting time, left graphs on fig. 3, 4, 9), or can be observed randomly (left graphs on fig. 6, 7, 10), but in any case their amplitude is typical for the underlying hardware. However, with the RTX and Windows system, the spikes are significantly less (fig. 3, right graph), or none at all.

It can be supposed, that the significant worst-case jitter spikes observed in experiments with RTLinux Free are caused by cache writing and flushing effects (in accordance with [2]). 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. However, the absence of this phenomenon on both test hardware with RTX operating system can imply, that the RTX microkernel prevents the hardware from flushing the cache freely. Moreover, some further tracks can be given. [2] notes video drivers as most cache demanding part of 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 task. Thus, the conflict between video and RTX microkernel can be seen as the conflict between two real-time cache-demanding tasks, which can lead to swapping the RTX code out of the cache.

Unfortunately, the RTX microkernel source code is not freely available, and it is not possible to verify the tracks given above with the code analysis. However, the measurement results as well as the tracks given above can suggest, that the mechanism of locking the real-time code in the hardware cache is worthy to be studied and implemented.

ACKNOWLEDGMENT

This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic under Project TM 0567.

REFERENCES

[1] FSM Labs Inc.*,"Getting Started with RT Linux"*, 2001.
[2] M. Cherepov et. al.: *"Hard Real-Time with Ardence RTX on Microsoft Windows XP and Windows XP Embedded"*, www.ardence.com, 2002.
[3] P. Moryc, J. Černohorský: *"Task jitter measurement under RTLinux operating system"*, in: Proceedings of the International Multiconference on Computer Science and Information Technology, ISSN 189-7094, pp. 849 to 858, 2007.
[4] P. Moryc: *"Měření procesů reálného času v operačním systému RTLinux"*, Doctoral Thesis, Technical University of Ostrava, Faculty of Electrical Engineering and Computer Science, 2007.
[5] I. Ripoll et al.: *"WP1: RTOS State of the Art Analysis: Deliverable D1.1: RTOS Analysis"*, 2002.