LTP-BASED VALIDATION OF A MODIFIED LINUX REAL-TIME ENVIRONMENT

Arsim Susuri[1], Mentor Hamiti[2], Marika Apostolova[2], Elissa Mollakuqe[1]

[1] University "Ukshin Hoti" Prizren; [2] South East European University, Tetovo
e-mails: arsim.susuri@uni-prizren.com, m.hamiti@seeu.edu.mk,
m.apostolova@seeu.edu.mk, elissamollakuqe@hotmail.com
[1] Kosovo, [2] North Macedonia

Abstract: The purpose of this paper is to validate the real-time system in the Linux-RTAI/LXRT environment, in the presence of hardware and software faults. Due to the simplicity of creating and injecting faults, the method of SoftWare Implemented Fault Injection (SWIFI) is selected. With this mode of fault injection, a tested system can emulate real conditions, in which it is required to function. The Linux operating system has been modified with an RTAI/LXRT patch in order to gain real-time system functionality. The purpose of using this test framework has been to carry out a comprehensive verification of the system tested under stress conditions and the presence of errors (hardware or software). Linux Test Project (LTP) testing framework was used. Subsequently, modification of the respective parts of the operating system was made to enable the injection of faults. The overall results have confirmed the robustness of the RTAI/LXRT system, despite the presence of faults.

Key words: LTP, Real-time, Fault injection, RTAI/LXRT.

1. INTRODUCTION

Emulating hardware and software faults is one way of testing fault-tolerant systems, applications and the power of non-fault-tolerant systems.

Assessing the reliability of various computer systems involves the study of failures and errors. Because of the features of the faults and the duration of the errors, it is very difficult to identify the cause of computer system failures during their operating time. For this reason, the deliberate introduction of faults occurs in different computer systems. This deliberate introduction of faults is called fault injection. Injecting faults is important because through this method we do the testing of the reliability of computer systems.

2. RTAI

Linux is a time-sharing operating system that provides good performance and very perfect services.

Like other operating systems, Linux offers at least these services: a hardware management layer that deals with interrupts of the processor/peripheral devices and communication tools between applications. Linux suffers from lack of real-time support. To get correct behavior in time, it is necessary to make some changes to kernel resources, e.g. interrupt management and time division mode. In this way, a real-time platform, with little delay and with great predictability, can be gained, all within the standard (non-real-time) Linux environment with access to TCP / IP, files and databases, graphics, etc.

RTAI [1, 2] offers the same services that the Linux kernel provides, adding to the features of the real-time operating system used in industry [3]. It consists of an interrupt distributor: RTAI "captures" peripheral device interrupts and whether to redirect them to Linux. This does not mean that any intrusive modification in the kernel has been made. In this case RTAI uses the HAL (Hardware Abstraction Layer) concept to gain information from Linux and capture some of the basic functions [4].

2.1. RTAI Architecture

RTAI treats the common Linux kernel as a low-priority real-time task that can its normal operations when high-priority real-time tasks are not executed. The following figure shows the basic architecture of RTAI [5].

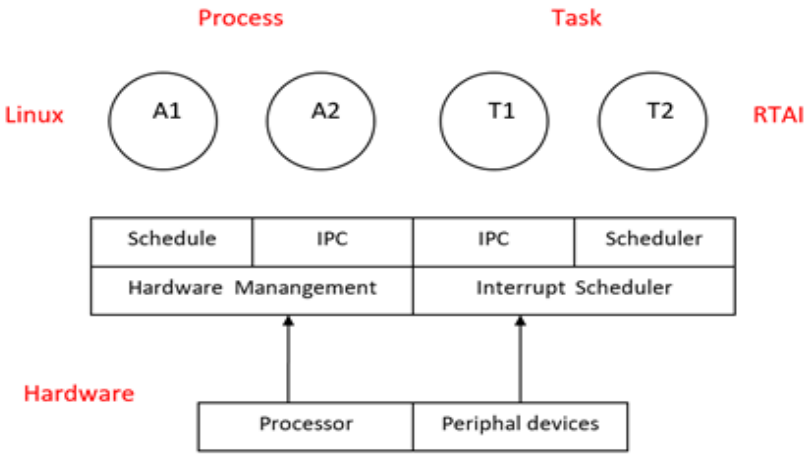

Fig. 1. RTAI architecture.

We notice the interruptions emanating from the processor and peripheral devices. Processor originated interrupts are handled by the standard Linux kernel, while peripheral devices interrupt (e.g. time interrupts) are handled by the interrupt distributor. RTAI drives these interruptions to standard Linux kernel handlers (Hardware Management) when there is no real-time active task. The instructions for activating and deactivating the interrupts (*cli ()* and *sti ()*) found in the kernel are replaced with macros that follow the instructions in the RTAI.

2.2. LXRT

LXRT provides user interface to enable the implementation of RTAI features. It provides a symmetric API that can be used both from real-time RTAI pocket as well as Linux processes [6].

LXRT is one of the most important features of Linux. LXRT allows users to develop a real-time application using the RTAI interface in user space [7]. The main advantage of this approach is that the pocket can be developed as a real-time soft task under standard Linux memory protection, while kernel-run tasks have access to unprotected memory and can overwrite critical parts of the kernel memory. An additional feature of LXRT is that it enables the passage of real-time tasks from hard to soft state within user application [7]. This enables the creation of new real-time tasks from the actual ones.

Developing real-time applications completely in user-space has many advantages. First, kernel-space memory is not protected by access to any buggy program. Inadvertent modification of any location in memory may cause data damage and cause Linux kernel malfunction. Secondly, if the corresponding Linux kernel is modified (update), the kernel modules must also be reconfigured, which makes it impossible to relocate real-time between different kernel versions.

When developing real-time applications in user-space (as real-time soft processes), the programmer can use standard debug tools as long as the application is considered error-free. Once the task is debugged, the task can be converted to the kernel space module as a real-time hard task. However, Linux system calls cannot be offered in real-time hard task services, so this task should use the services provided by RTAI.

The application transition from the real-time task user-space process is easy because LXRT provides a symmetric API for communication between processes as well as other services. This means that the same API can also be used from the kernel-space task and from user-space processes. The same interface can be used when two user-space processes or two real-time tasks communicate with one another. This means that the LXRT's various messaging and timer systems can be used by a user-space application even when this application does not have real-time requests. LXRT allows applications to dynamically switch from real-time hard-to-hard to real-time using a single user-space function call.

When an application is in real-time soft mode, it uses the Linux scheduler, but requires that the process change the scheduling in the FIFO deadline, which is used when critical time applications are to be executed on Linux. The FIFO deadline provides static priorities with preventive deadlines i.e. better scheduler control than the dynamic priority Linux scheduling algorithm.

To enable any application to pass on real-time hard disk, LXRT creates a real-time kernel-space agent for every user-space process with real-time requirements. When the process enters the real-time hard mode, the agent created disrupts the interrupts, removes the process from the order of executions in the Linux scheduler and sends the process to the RTAI scheduler's order. Now this process cannot be hampered by interruptions or other Linux processes. To switch to real-time hard mode, the process also needs to ensure memory stays in RAM and therefore deactivates memory paging using the *mlockall* call [2].

Linux system calls cannot be used in real-time hard mode, but this restriction can be alleviated if using real-time soft processes, which uses Linux services on behalf of real-time hard processes. These two processes communicate with each other through the IPC (Inter Process Communication) methods provided by RTAI.

2.3. Using LXRT

Based on [7] we can see an example of using RTAI and LXRT APIs. This example shows the creation of real-time hard tasks in user space. From the following figure we can observe the Linux host's memory space, which divides the user-space and kernel-space part. A real-time application is executed in user-space, creating two threads, the beginning of the hard-real-time controller (*pthread_create ()*), and a file of the Linux server (*pthread_create (linuxser)*), which operates in real-time soft mode and can use Linux services. Threads communicate using a common memory space. Initially, the core process provides memory for shared data used from different sources. This is done through the RTAI memory management module (*rt_malloc ()*). Then the main module calls rt_task_init () to create a real-time agent in the kernel-space. This is needed to make real-time hard-task execution in user-space and to utilize RTAI services. The mainstream gives priority to the pocket which priority is used if the pocket turns into real-time hard.

Then, the main module, using RTAI implementation for Posix threads, creates the controller start-up and the start-up of the Linux server. These two arrays will do the most work in this example.

After creating the queues, the main thread can be stopped to wait for the start of the controller and the Linux server. When the root gets permission to complete it, it needs to delete the internal data and delete the real-time agent through *rt_task_delete ()*. And finally the main thread releases the reserved memory. The Linux server task specifies the deadline in FIFO (through *sched_complete_scheduler ()*), based on recommendations for critical Linux processes at the time. Even the Linux server calls the *rt_task_init ()* because it needs RTAI services. Finally, before waiting for the request from the controller start-up, the Linux server initiated the call *mlockall ()* call to block the RAM memory page of the source code and the startup file of the Linux server.

The controller thread is the only task with real-time hard requirements. At first, this task also sets the FIFO deadline policy in Linux planners to timely serve real-time soft operations. Then, the controller's initiator initiates information for the RTAI pocket and creates a real-time agent through *rt_task_init ()*. Before entering into hard real-time mode, the controller's start blocks the page pages and data in RAM through *mlockall ()*. The master controller enters my real-time hard drive by calling *rt_make_hard_real_time ()*. This causes suspension of Linux startup and activation of the controller agent pocket in the RTAI scheduler. When reactivated the next time, it is in my real-time hard drive and comes back from the *rt_make_hard_real_time ()* call to the program that first made the call. At this moment, Linux interrupts are deactivated and none of the Linux processes can prevent the controller from running. The first call in the hard-real-time mode for the controller's start is *rt_task_make_periodic ()*, which makes the task set at periodic intervals.

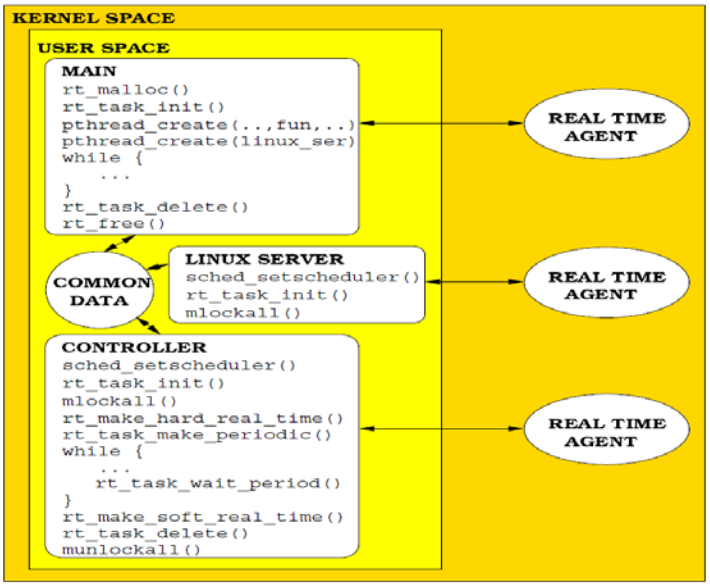

Fig. 2. Simple real-time application architecture [7].

After planning the scheduling period, the hard real-time controller enters the main ring. After each ring repeat, the controller waits until the next scheduled period with the *rt_task_make_periodic* call. When the controller's main ring is completed, the controller startup invokes *rt_make_soft_real_time ()* which suspends the start in the RTAI scheduler. Linux interrupts are reactivated and the start is enabled in the Linux scheduler. Finally, the controller's start releases its real-time agent and relevant data in the RTAI and debugs the blocked memory.

Inter-process communication mechanisms are implemented separately for Linux and RTAI. Even the scheduler is special for the Linux and RTAI side.

3. RELATED WORK

3.1. Injecting errors in user space and kernel

FIAT [8] is an automated error-based testing that is capable of emulating different distributed system architectures and monitors the response of the tested system.

FIAT is used to validate the dependency of the system. Experiments carried out through FIAT include the comprehensive injection of three types of errors into different system locations, including part of code and buffer memory. Types of errors are variations of memory bit errors. Through this testing system it is possible to predict the system level errors. This is very important during the design phase of the system.

FINE [9] implements injecting software-related errors caused by the hardware, on UNIX, and following the execution flow and the core variables of the kernels. FINE consists of a bug injector, an experiment monitoring software, a load generator for the tested system, a controller, and some other auxiliary tools.

Experimentations with the SUNOS 4.1.2 operating system through the FINE system were implemented to investigate the propagation of failures and assess the impact of these failures of different types [9]. Hardware and software failure spreads have been built. Loss of performance as a result of error injection was analyzed through Markov's transient analysis. After these experiments, authors have noted that memory and software failures usually have a very long delay, while failures in the processor cause an immediate drop in the system.

Xception [10, 11] presents a system of error injection which enables accurate and flexible verification and validation of critical computer systems. This Xception validation reaches through the injection of errors. This testing system has a special emphasis on injecting hardware and software errors. It is characterized by a high degree of automation that enables users to plan and implement easier error injection experiments. It features a relatively advanced GUI (graphical user interface) through which errors can be injected, experimentation and analysis of results. Data to be placed on a SQL database is used to manage and automate experiments. Through Xception, errors can be injected into the kernel and user code. Within a distributed system it is possible to inject errors into different processors. Xception is used in different systems, ranging from Online Transaction Processing Systems (OLTP) [12] to Spatial Application Systems [13].

PROPANE [14] is a tool for experimenting error injection into personal computer software (desktop). PROPANE allows software errors to be injected (making changes to the source code) and data errors (making manipulations with variables and memory contents). Supports different types of errors and enables injecting errors defined by the user of this test tool.

DEPEND [15] is a simulation system through which computer architectures are evaluated. This is accomplished by simulating a tolerant system based on triple modular redundancy (TMR) tandem based on Unix operating system. Through this simulation, system response testing is performed on almost random errors caused by latent and correlative failures. In [16], a comparison was made between Linux kernels 2.4, 2.4 kernels with RTAI and RTAI / LXRT installed, and 2.6 modified kernels with precaution patch, using three real-time applications in two cases when the operating system is a bit charged and when it is too busy.

The results show that in the 100 Hz controller system, Linux 2.4 kernel with RTAI and RTAI / LXRT meets the requirements of delays in 100 percent of the time. If we apply this application to a 600 Hz control system, 2.6 modified patch kernel for prevention would supplement them. The experiment carried out in [17] has enabled the application of both patches (prevention and low latency) in the same kernel. It was noted that the obtained results were better than applying patches individually.

UMLinux [18] represents the User Mode of Linux that can be used for realistic experiments with regard to error injection. UMLinux simulates a Linux system. The simulated hardware can be forced to fail through the injection of errors and then become a reaction to the Linux operating system and the applications running on those machines. To be as close to reality, UMLinux is implemented by protecting kernel memory and the entire virtual machine (operating system with all processes) is executed as a single process in the computer system. The whole system works so that first the virtual server

system is activated and then the injecting of the errors is done through the error injector which for this purpose is configured immediately after the virtual server system is upgraded. Error injecting is done on virtual hardware. In this experiment, authors [18] have made bit flip in processor registers and in main memory.

VERIFY [19] presents a technique for evaluating digital systems based on the VERIFY (VHDL-based Evaluation of Reliability by Injecting Faults efficientlY) injector. Virtually implies a software tool designed to describe the response of hard-ware components in the case of errors. This is accomplished by expanding the VHDL language () with error injection signals along with the frequency of their use.

Verification of embedded real-time systems was implemented in [20, 21] with a framework for the verification of safety and timing properties of digital embedded real-time systems, based on models built with *SystemC*, using timed automata and the UPPAAL model checker.

SWIFI was implemented for validating of the real-time system in the Linux-RTAI/LXRT environment, in the presence of hardware and software faults [22].

4. EXPERIMENTS

As mentioned earlier, the main purpose of the testing was to validate the real-time operating system so as to confirm the assertion of its creators that indeed, regardless of the number and type of processes that can be executed at one and the same time real-time system, this system will not be blocked and will continue to function normally.

To do this, preparations have been made to create a suitable test system. This system should include at least one computer (preferably PC architecture), a real-time Linux operating system. Validating the real-time system is implemented in a PC architecture. Red Hat Linux v9.0 and Suse Linux 10.0 were used as testing platforms.

This operating system as such was not ready to apply the validation process. The main reason is that it is not a real-time operating system. For this reason, a patch has been applied and with this procedure this operating system has gained additional functionalities in terms of real time.

Kernel version 2.6.9 was used (kernel version that was used with operating system was 2.4.20-18). This kernel version was chosen as the most appropriate because of the RTAI version (RTAI 3.1r3) which was more stable and matched to the kernel version used. After installing the new kernel, the RTAI patch was installed and finally, after the system reset, the real-time operating system was tested.

4.1. Fault Injection and System Testing

The purpose of using this test framework has been to carry out a comprehensive verification of the system tested under stress conditions and the presence of errors (hardware or software).

LTP testing frame was used. But before activating tests from this framework, some modifications to some Linux files were made. This modification is done in order to create the prerequisites for making errors in disk I/O capabilities.

In the following sections we will clarify the steps taken to enable injecting errors in the RTAI/LXRT system.

4.2. Modifications of RTAI/LXRT System

In order to enable the injection of errors through which the system malfunctions are caused, some modifications should be made to the RTAI / LXRT system itself. There are changes to these files (see appendix A) within */home/education/Arsim/linux-2.6.20/:* block/genhd.c, file mlock/ll_rw_blk.c, fs/partitions/check.c, include/linux/genhd.h, include/linux/should_fail.h, lib/Kconfig.debug.

The functions that are affected by this error injection are: malloc, realloc, memalign.

The period of disorder of the *malloc* is regulated by the variables:

- FAILMALLOC_PROBABILITY
- FAILMALLOC_INTERVAL
- FAILMALLOC_TIMES

In the first case, the memory loss failure probability is applied to the 80% scale through the command: *Env LD_PRELOAD = libfailmalloc.so FAILMALLOC_PROBABILITY = 0.8 1s* Through *FAILMALLOC_SPACE* is adjusted the size of the free space where memory can be safely reserved. These commands have made possible injecting errors: *fail_make_request = 100.10, -1.0 echo 1> / sys / mlock / hda / hda8 / make-it-fail* Number 100 in *fail_make_request = 100.10, -1, 0* determines in percentage how often failure should occur. Number 10 specifies the range of failures.

The number -1 determines how many times the failures can occur most. Number 0 represents the size of the free space where I/O (Input Output) of the disc can be set. This causes the failure of *generic_make_request ()* in */dev/hda8* (partition with the current installed kernel) every tenth time. *Generic_make_request ()* is used to make I/O block device requirements. The device block may contain addressable, reusable data.

The file system can be a device block. Following this state of affairs, it continued with the execution of tests within LTP. At the same time, activation of the module for monitoring of the system is done.

4.3. Testing Framework LTP

Linux Test Project (LTP) is a joint project of IBM, SGI, OSDL, Bull and Wipro Technologies with a view to creating an open Source Community to enable Linux sustainability, stability and stability testing in conditions Extreme work.

LTP started as a collection of 100 test programs, developed by SGI. Later, in collaboration with the aforementioned companies, came to the number 2500 of test programs and some automated testing tools. Supports a number of computer architectures including: x86, IA32 / 64, PPC32 / 64 and s / 390 32 and 64 bits. Tests are executed through the test driver that is used for executing and completing test programs.

The *runalltests* script calls pan to enable the execution of a test community. Saving earned results is done in log type files. This is accomplished through the command:

./runltp -p -l rezultati.log

(-p i.e. invokes pan, -l file log type)

The activation of the tests is performed through the *runltp* module. We perform the system state monitoring through the ball module and the results are saved through the pan module. This applies to the first test phase (realized through LTP-20040108). In the second and third stage of testing (realized through LTP-20080229) system status

monitoring is through the *sar* module (within the *sysstat-8.1.2* program). If the above command is not used, saving results is done in the outputs folder where only failed tests are stored.

4.4. Testing method

Stability and durability of the tested system were measured during the time of the LTP test execution.

The testing started with basic executions of the test framework by executing a total of 795 tests with a total duration of 50 minutes, with emphasis on kernel testing, memory management and I/O.

During the testing of the system, the following versions were used:

- LTP-20040108 on RedHat Linux 9 with 2.6.9 kernel
- RTAI-3.1r2 LTP-20080229 on Suse Linux 10.0 with 2.6.20 kernel and RTAI-3.6.

4.5. Testing strategy

The testing procedure involved several steps: Test Frame Installation (LTP), Compilation, and Execution.

Testing is divided into 2 phases: Initial Testing and "Stress" Sustainability Testing.

The first test phase included the consecutive execution of the tests included in the LTP within the runalltests.sh script to make the kernel verification. The script performs a series of successive tests. At the end of the test run, a test results report (result1.log file) was created. This script has executed these tests: File system stress tests, I/O disk tests, IPC tests, Scheduling tests, Memory management tests, Verification tests for system call functionality.

The second stage of testing ("stress" sustainability testing) enables verification of system stability and hardness during high system use (maximum utilization of processor and other computer resources). To create a working job status for the tested system, activation of the ltpstress.sh script has been designed, which is designed to simultaneously activate some kernel components together with the memory management.

This script also performs similar tests in parallel and various tests in sequence. The purpose of this test execution is to prevent the interference of the tests between themselves. This script performs these tests: NFS stress tests, Hard drive I/O tests, IPC tests, Memory management stress tests, Verification tests for system calls functionalities, Pthread stress tests, Mathematical tests.

After the completion of the second test phase, the report generation was done.

4.6. The obtained results

The execution of 795 tests (the total number of tests included in the test framework) was performed in the first phase of the test.

In the first phase when the free system was tested, after *runalltest* script execution, out of 795 tests performed, 6 of them failed (value 1).

Tests that failed are within system calls (*syscalls*): getrlimit02, gettimeofday02, ioperm02, nanosleep02, setegid01, setgroups02.

Load verification is done through the ball monitoring tool. In the second and third phase, a total of 860 tests (the newest version of the LTP test framework had a greater number of tests compared to the first) were made in a total of 60 minutes through the LTP-full-20080229 testing framework.

This test was performed on the Suse Linux 10.0 operating system with 2.6.20 kernel installed and patch RTAI-3.6. In the second phase, the same method is applied only now in the presence of load (stress testing). After reading the log file we came to the conclusion that the number of failed tests was 17.

In addition to the above-mentioned 6 tests, the other 11 failed tests were: mem01, aio01, getpid02, getrusage01, flock01, flock03, HTinterrupt, nanosleep03, socketcall02, socketcall04, accept01.

5. CONCLUSION

The tests that were made in this work indicate that RTAI is likely to meet hard-real-time requirements, depending on the nature of the timing requirements are.

In this paper work various methods for validating the RTAI/LXRT system were implemented by previously inserting faults into the system and then the testing was conducted through the LTP test framework. With this, a validation of RTAI/LXRT was achieved. In conclusion to this, we can say that the RTAI/LXRT system accomplishes the goals set out for a hard- real-time system.

From the results obtained we can notice that despite the injection of errors, a very high percentage of tests (over 98%) have been successful. What can help improve and increase the variety of testing is the automated procedure for changing the RTAI / LXRT system that could be achieved through the scripts. This could improve and accelerate the procedure for creating the RTAI/LXRT system with the possibility to inject errors through the software.

Based on the test made the RTAI/LXRT system, performance was not degraded during the long duration of the test. Every run generated a high success rate (over 98%). The very small number of failures that occurred was as a result of concurrent executions of tests. These concurrent tests are designed to overload system resources.

In all, the Linux kernel, with RTAI/LXRT patches, and other core components (device drivers, file systems, IPC, memory management, etc.) operated consistently and competed all the expected durations of runs with zero critical failures.

For a better validation of real-time systems, a continuous development of software fault injection (hardware or software faults), especially in situations where the CPU load exceeds 95%, is recommended. This way of using the testing framework with these fault injectors can be a start for developing other fault injectors and to continuously validate real-time systems.

REFERENCES

[1] Cloutier, et. al.: DIAPM-RTAI Position paper. *Workshop on Real Time Operating Systems and Applications and 2nd Real Time Linux Workshop*, 2000.

[2] Lineo, Inc.: *DIAPM RTAI programming guide*. 2000.

[3] Dozio L., Mantegazza P. *Linux Real Time Application Interface (RTAI) in low cost high performance motion control*. Dipartimento di Ingegneria Aerospaziale,Politecnico di Milano, 2003.

[4] Mantegazza P. *Dissecting DIAPM RTHAL-RTAI*. Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano, 2002.

[5] Sarolahti P. Real-Time Application Interface. *Research seminar on Real-Time Linux and Java*, University of Helsinki, Department of Computer Science, 2001.

[6] Leucht A. *Hard real-time capability under Linux development within the micro-kernel real-time architecture*, 2002.

[7] Bianchi E. and Dozio L. Some experiences in fast hard real-time control in user space with RTAI-LXRT. *Workshop on RealTime Operating Systems and Applications and 2nd Real Time Linux Workshop*, 2000.

[8] Barton J., Czek E., Segall Z., and Siewoirek D. Fault Injection Experiments using FIAT. *IEEE Transactions on Computers*, **4** (vol. 39), 1990.

[9] Kao W. I., Iyer R. K., and Tang T. FINE: A fault injection and monitoring Environment for tracing the UNIX system behavior under faults. *IEEE Transactions on Software Engineering*, **11** (vol. 19), 1993, pp. 1105-1118.

[10] Carreira J., Madeira H., and J. G. Silva J. G. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Transactions on Software Engineering*. **2** (vol. 24), 1998, pp. 125-136.

[11] Maia R., Enriques L., Costa D., Madeira H. Exception – Enhanced Automated Fault-Injection Environment. *Fast Abstract Session*, 2002.

[12] Costa D., Rilho T., and Madeira H. Joint Evaluation of Performance and Robustness of a COTS DBMS Through Fault-Injection. *IEEE/IFIP Dependable Systems and Networks Conference – DSN*, New York, 2000.

[13] Madeira H., Some R. R., Moreira F., Costa D., and Rennels D. Evaluation of a COTS System for Space Applications. In *International Conference on Dependable Systems and Networks*, Washington D.C., 2002, pp. 325-330.

[14] Hiller M., Jhumka A., Suri N. PROPANE: An Environment for Examinig the Propagation of Errors in Software. *International Symposium on Software Testing and Analysis*, Rome (2002).

[15] Goswami K.K. DEPEND: a simulation-based environment for system level dependability analysis. Computers, *IEEE Transactions on. Computer*, **1** (vol. 46), 1997, pp. 60-74.

[16] Laurich P. A Comparison of hard real-time Linux alternatives. *Linux Journal*, 2004.

[17] C. Williams C. Linux Scheduler Latency. *Linux Devices*, 2002.

[18] Höxer H. J., Buchacker K., and Sieh V. *UMLinux - A Tool for Testing a Linux System's Fault Tolerance*. Institut für Informatik 3, Erlangen, 2002.

[19] Sieh V., Tschäche O., Balbach F. VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions. *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, Washington, 1997.

[20] Herber P. and Glesner S.: Verification of Embedded Real-time Systems. Formal Modeling and Verification of Cyber-Physical Systems. *1st International Summer School on Methods and Tools for the Design of Digital Systems*, Bremen (2015, pp. 1-25.

[21] Cimatti A, Narasamdya I., Roveri M. Software model checking systemc. *IEEE Transactions on CAD of Integrated Circuits and Systems*, **5** (vol. 32), 2013, pp.774–787.

[22] Susuri A., Hamiti M., Selimi B., and Luma A. Emulating Software and Hardware Faults In A Modified Linux Real-Time Environment. International *Journal on Information Technologies and Security*, **4** (vol. 9), 2017, pp. 87-96.

Information about the authors:

Arsim Susuri, PhD, is Assistant Professor at the Faculty of Computer Science at the University "Ukshin Hoti" Prizren. His research focuses on real-time systems and machine learning. He teaches courses in Artificial Intelligence, Operating Systems, Machine Learning, Cloud Computing, etc.

Mentor Hamiti, PhD, received a PhD degree in Computer Sciences from South East European University, Tetovo, Macedonia in 2010. He is currently associate professor in Computer Sciences at the South East European University, Faculty of Contemporary Sciences and Technologies. His research focuses on Programming Languages and Technologies.

Marika Apostolova, PhD, is Assistant Professor at the Faculty of Contemporary Sciences and Technologies at the South East European University in Tetovo. Her research focuses on software development and applications.

Elissa Mollakuqe, PhD candidate, is an Assistant at the Faculty of Computer Science at the University "Ukshin Hoti" Prizren. Her research focuses on software development and algorithms evaluation.

Manuscript received on 14 January 2019