SYSTEM-WIDE ANALYZER OF PERFORMANCE: PERFORMANCE ANALYSIS OF MULTI-CORE COMPUTING SYSTEMS WITH LIMITED RESOURCES

Alexey A. Gerenkov[1], Ekaterina A. Gorelkina[1], Sergey S. Grekhov[1], Sergey Yu. Dianov[1], Jaehoon Jeong[2], Olexiy Kokachev[2], Leonid V. Komkov[1], Sang Bae Lee[2], Mikhail P. Levin[1], *Member IEEE*

Abstract: **In this paper a new dynamic performance analyzer intended for uni- and multi-core computing systems with limited resources is presented and described. This analyzer is called SWAP (System-Wide Analyzer of performance). Initial version of SWAP tool, called as ELP1 (Embedded Linux Probe 1), was developed in Samsung Research Center (SRC) in 2006. This tool uses functional interface of Kprobes to provide the dynamic instrumentation of Linux kernel for ARM and MIPS architectures. ELP1 functionality allowed collecting raw data information (such as register values, memory dump, etc) for any user predefined function of Linux kernel. Next revision of ELP1 tool, ELP2, was developed in SRC in 2007. ELP2 functionality allowed collecting traces from predefined functions in Linux kernel that contains general information of system characterization. Traces in ELP2 contained information from main Linux kernel functions such as arguments, return values and content of global kernel structures. ELP2 had pseudo-graphical interface on target that allows stand-alone monitoring of the system without any network connection. SWAP can monitor both kernel and application levels of the Linux system. Additional to ELP2 functionality, SWAP provides evaluation of the set of important system characteristics for main Linux subsystems (such as Memory Management, Process Management, File System and Network). Also SWAP has some automatic performance analysis features such as trace comparison, automatic bottleneck region localization, etc.**

***Index Terms*: Monitoring Tool, Embedded Systems, Debuggers, Multi-core, Dynamic Instrumentation, Tracing, System Behavior, Run-time Statistics.**

I. INTRODUCTION

Operating system kernel monitoring and debugging is one of the most challenging and least developed areas of software engineering. Debugging activities include queries on many aspects of kernel behavior: sequences of kernel functions performed, histories of global kernel structures values, checking of pre- and post-conditions at specific points, and validating other assertions about kernel execution. Performance testing and debugging involves a variety of profiles and time measurements. In case of multi-core processors, performance analyzers allow debugging real-time systems in real environment. It becomes possible due to separate execution of target real-time software and the performance analyzer on different cores of the system.

Operating system kernels are complex entities whose internals are often difficult to understand. Various special programs can be used for performance analysis of kernels and software systems. At this time there are various static and dynamic analyzers of performance. Static analyzers require recompilation of the kernel and the target software. The dynamic analyzers do not require any recompilation. Besides, dynamic analyzers allow measuring performance in real-time mode and find the bottlenecks more precisely.

The existent kernel debugging tools collect raw data only. The collected data are not structured and are shown to user as the sequences of low level information (registers, call stack, system calls etc.). But such data are too large to be analyzed by hand. Some tools (DTrace, KerInst etc.) have features to allow user to aggregate the data by a high-level language. But the automatic semantic analysis of the collected data is very restricted in almost all existent debugging tools. LTTng (Linux Trace Toolkit) [3] is a tracing tool that permits to get all the possible execution information from the Linux Kernel. It includes Linux Trace Toolkit Viewer (LTTV) that is a second generation of visualization tool. LTTng is statically instrumented into the kernel of operation system and, thus, require its recompilation. SystemTap [4] is a tool that allows developers and administrators to write and reuse simple scripts to deeply examine the activities of a live Linux system. SystemTap is based on Kprobe – the technology developed by joint efforts of Red Hat, IBM, Intel, and Hitachi. The disadvantage of this tool is relatively barren set of automatic analysis features. OProfile [6] is a low-overhead, transparent profiler for Linux, capable to provide an instruction-grain profiling of all processes, shared libraries, the kernel and device drivers, via the hardware performance counters. The disadvantages of this tool are: limited set of collected data and absence of the measured history.

In other side, monitoring of user-space applications is also very important task during both development and production stages. However, existent application monitoring tools have a set of disadvantages and, usually, require a great amount of resources. As result, these tools cannot be used on embedded platforms.

For example, post-compiler instrumentation tools for user-space applications, such as EEL, ATOM, or Etch that provide code insertion into a binary before it starts to execute. However, data those are collected during application execution may not be known until runtime.

[1] Advanced Software Group, Samsung Research Center in Moscow

[2] OS Group, Samsung Advanced Institute of Technologies

 978-1-4244-3861-7/09/$25.00 ©2009 IEEE

Thus, run-time instrumentation tools are more comfortable for the user.

KProbe has no ability to instrument the user-space functions in application. Thus, it is impossible to monitor the user-space application behavior by using original KProbe functionality.

DynInst [1] is also well-known tool that provides a run-time instrumentation of user-space applications. However, DynInst instrumentation method has a set of disadvantages because of it uses ptrace() system calls for monitoring of application behavior:

- Instrumentation of all functions in application requires a loading of whole application into a memory (that essentially changes application behavior and requires great amount of memory resource);
- This method produces the big overhead because of great amount of kernel-user space copy operations.

Thus, DynInst disadvantages do not allow using this method on embedded system.

To use run-time instrumentation method for monitoring of user-space application on embedded systems it was be necessary to design the method that satisfy the following requirements:

- Using of paging mechanism for providing little changes in system behaviour during method usage;
- Low resource usage and low overhead that allows using this method in conditions of limited resources.

II. BASIC SWAP FEATURES

A. Dynamic Instrumentation

To date there are two types of instrumentation methods: namely static and dynamic. Dynamic instrumentation method provides monitoring of binary code without any source code file. In contrast static instrumentation requires the source code file. In static instrumentation some special functions measuring the performance of investigating code are to be inserted into the source code and after that the source code should be recompiled. Thus, it is not possible to provide performance analysis without source code.

The principal difference between the static and dynamic instrumentation implementation consists in the following. To implement the static instrumentation methods, it needs

- to insert the static instrumentation code into kernel source code;
- to recompile the kernel;
- to run the kernel with instrumented code;
- to use monitoring approaches for collected data.

In contrast with implementation of the dynamic instrumentation, it needs to do only two following steps

- to insert the dynamic instrumentation code into the kernel **binary** code;
- to use the monitoring features to collect the data.

The dynamic instrumentation method is more comfortable approach for developers because it reduces the monitoring time and does not need recompilation of investigated code. SWAP provides dynamic instrumentation of investigated code and kernel of OS also. In this case the kernel or application being modified is able to continue its running and does not need to be re-compiled, re-linked or restarted.

When the kernel or application has been dynamically instrumented, this instrumentation collects data for measurement of performance and for the tracing. For example, using these data it is possible to calculate how many times a function has been called. For this purpose the monitoring call must be inserted at the beginning of this function. Then, during the application execution, the monitoring code is invoked and the internal counter is incremented.

SWAP is built on capabilities of Kprobe kernel debugging infrastructure. Kprobe utility was developed by joint efforts of Red Hat, Inc. and IBM. It provides a simple and lightweight mechanism in Linux for insertion of breakpoints into a running kernel. To use Kprobes with MIPS and ARM processors, it needs to create a loadable kernel module with calls into the Kprobes interface. These calls specify a kernel instruction address, the probe point, and an analysis routine or probe handler. Kprobes arranges for control flow to be intercepted by patching the probe point in memory, with control passed to the probe handler. Kprobes has been carefully designed to allow safe insertion and removal of probes and to allow instrumentation of almost any kernel routine. It lets developers add debugging code into a running kernel. Because the type of Kprobe instrumentation is dynamic, there is no performance penalty when probes are not used.

The basic control flow interception facility of Kprobes has been enhanced with a number of additional facilities of two special probes Jprobes (this probe can be inserted at the beginning of the function) and Kretprobe (this probe can be inserted at the end of the function). Jprobes makes it easy to trace function calls and examine function call parameters. Kretprobes is used to intercept function returns and examine return values. Although it is a powerful system for dynamic instrumentation, a number of limitations prevent Kprobes from broader use:

- Kprobes does very little safety checking of its probe parameters, making it easy to crash a system through accidental misuse.
- safe using of Kprobes often requires detailed knowledge of the code path to be instrumented. This limits the group of developers who will use Kprobes.
- due to references to kernel addresses and specific kernel symbols, the portability of the instrumentation code using the Kprobes interface is poor. This lack of portability also limits re-usability of Kprobes-based instrumentation.
- Kprobes does not provide a convenient mechanism to access function's local variables, except for a Jprobe's access to the arguments passed into the function.
- although using Kprobes doesn't require a kernel build-install-reboot, it does require knowledge to build a kernel module (for ARM and MIPS processors) and lacks the support library routines for common tasks. This is a significant barrier for potential users. A script-based system that provides the support for common operations and hides the details of building and loading a kernel

module will serve a much larger community.

Additional, KProbe has not ability to instrument user-space functions in application because of there are the following problems:

- the kernel-space memory can be instrumented directly in contrast to user-space applications that use virtual memory addresses and it is necessary to find correspondence between virtual and physical memory addresses;
- the kernel is the part of the operating system that loads first and remains in main memory. In contrast to user-space binaries, those are not loaded completely into the memory at the start. Operating system loads pages with binary code of application by request during application execution via page fault exception mechanism. Thus, the precise instrumentation requires tracking of loading application into memory during whole application execution.
- Kprobe needs to gain control of the central processing unit when the probed subroutine returns. This control is accomplished by replacing the return address of the probed subroutine with the address of a piece of code, also known as a trampoline. Kprobe trampoline is place in kernel space. A user application cannot access kernel space directly. This fact requires to implementation new method for monitoring returns from subroutines.

SWAP extends KProbe [7] utility functionality for dynamic instrumentation of kernel and user-space applications. SWAP uses loadable kernel module to provide more flexible usage scenario. Using SWAP functionally does not require kernel patching or recompilation.

SWAP maintains a set of probes by using Kprobe utility. The probe collection encompasses the most vital and important domains of operating system when functioning. Every probe has its own ordinary number, a name which repeats a function where a corresponding probing point is set. To run ahead it should be mentioned that the program provides key accelerators to activate and de-activate a particular probe selectively. A status of each probe is displayed in the last column of the entire table of SWAP pseudo-graphical interface.

B. Tracing and Profiling

Let us consider tracing and profiling features of SWAP in details. SWAP supports these two monitoring features to make monitoring process more convenient for user.

At first, let us consider how the tracing feature is implemented. When the control reaches any current function in the kernel, the event object pointing on the call of current function is created. This event object is stored in the trace buffer for the future analysis. The event object contains the following information on current event, namely Event ID, Event Type, Event Name, Timestamp, and Parameters of Event. Each event is some significant activity and is an encoded instance of the action and its attributes. Each record on tracing event has a set attributes, namely **What, When, Where** and **Parameters**. Attribute **What** describes action that has occurred (or event identifier or/and function name). Attribute **When** describes the time when an event has occurred (timestamp). Attribute **Where** describes the location where an event has occurred (host, task, source file, module name, etc.). **Parameters** describe event-type depended details (e.g. execution details, such as CPU identifier, process identifier, function parameters and circumstantial parameters).

Event tracing provides a good base for performance analysis. This technique is the most invasive technique, but it is the most general and the most flexible. The main advantage is a big quantity of information about the kernel behavior. A generated global trace (for all processes) simultaneously is representative of what really happened in the system; hence it allows for a reconstruction of the system behavior and can be a source for performance analysis. The disadvantage of tracing is its potential intrusion, the implementation complexity and large amount of produced data. The big information quantity causes the storage problem especially for embedded systems. A trace contains events from the kernel that has been run for many hours (days) may occupy huge amount of disk space or memory. Therefore, some precautions are taken into account by SWAP users for monitoring SWAP tracing method. To reduce the amount of generated data, SWAP provides, for example, selective instrumentation or binary compressed trace format.

Profiling allows collect reduced set of performance data. In general case these should give information regarding

- the total CPU usage;
- the total memory usage;
- the total number of process, etc.

To deliver this information the profiling components contain the implementation of functions that can evaluate important characteristics of computer systems. Profiling gives to user various kinds of reports on Linux real-time execution. It is not a way to improve the performance directly, but it gives the important information allowing improve the kernel behavior in various cases. For example, it is possible to indicate the process that gets a dominating percentage of the system execution.

Now let us consider how profiling feature is operating in SWAP. In general case there are three different ways to collect the profiling data. On the first way the profiling data periodically by each ***N*** seconds are stored. For each profiling thread its all data are stored and available for further visualization. On the second way some special internal counters are counted and further visualization of data is provided simultaneously by the specified time interval. Profiling data include total number of running processes, total context switches, and all data on each process. In the third way profiling data is evaluated according to tracing data. Tracing data is processed and necessary characteristics are extracted from it.

SWAP supports the following mechanism for obtaining profiling data:

- some types of profiling data are stored in context of events that can influence on profiling data content. For example, information about current process time slice is changed by scheduler_tick() kernel function. SWAP instruments scheduler_tick() kernel function and stores information about time slice for current process in handler for this function. Thus, SWAP does not

use periodic collecting of profiling information. It collects profiling information in kernel points that can change this information. As result of this method, SWAP collects information with high precision and low overhead to the system.

- some types of profiling data are obtained during processing of trace data (post-processing). As result of this method, there is no overhead for obtaining profiling characteristics during data collecting. This method has the best precision in comparison to other profiling methods.

Thus, SWAP supports low overhead methods for profile information obtaining that have a high precision.

Now let us provide comparison the following well known profiling tools such as **Top** in Linux and **Windows Task Manager** with SWAP profiling facilities.

These above mentioned tools display overall summary data on system running. However **Top**-tool does not reflect the history of running processes and does not reflect the complete list of the process in the system on the screen. **Window Task Manager** has complete list of processes running in the system and reflects the history of running processes. But, this history is strictly limited. In contrast with these SWAP contains complete list of processes and reflects the full history of running processes. Also, SWAP supports more profiling features those are not supported by above mentioned well-known tools. SWAP stores all profiling data in one trace and this allows provide fast and various analysis of stored date, namely it is possible to evaluate the maximal, minimal, average values of any characteristic in profiling data. Also it is possible to make several back steps evaluating these characteristics.

SWAP collects extended set of the profiling characteristics, such as:

- Characteristics relative to memory management:
  - Monitoring process VMA (heap, code, stack, data, etc);
  - Monitoring of "Live" memory ("Live" memory size indicates how much of the resident memory size of the process have been really accessed during predefined period of time);
  - Memory Fragmentation evaluation.
- Characteristics relative to process management:
  - CPU load;
  - Monitoring of priorities of processes.
- Characteristics relative to file system:
  - Monitoring "Live" files (accessed files are called as "Live" files);
  - Monitoring of Read-a-head buffer operations.
- Characteristics relative to network:
  - Monitoring of receive/send queues for TCP/IP.

Result of comparison SWAP and well-known tools is illustrated by Table 1.

Table 1. Total comparison of SWAP profiling features with **Top** and **Windows Task Manager** profiling features

Feature \ Tool	ELP 2	top	Task Manager
Measuring General Summary Characteristics in Operating System	Implemented	Implemented	Implemented
Reviewing History of Measurements	Implemented	Not	Limited
Reviewing Full List of Processes in Operating System	Implemented	Not	Implemented

Now let us provide comparison the following well known tracing tools such as **SystemTap** and **LTT** Linux utilities with SWAP kernel tracing facilities.

At first it should be mentioned that **LTT** uses the static mechanism of instrumentation, but **SystemTap** uses the dynamic mechanism of instrumentation at the stage of measurements. In contrast, SWAP always uses the dynamic mechanism of instrumentation.

At second **LTT** collects the data at predefined points of the kernel. This fact releases the user from necessity to know the details of the kernel internals to start the collection. In contrast, by using **SystemTap**, user should know the kernel internals for setting the probe on current point in the kernel. SWAP also as **LTT** uses predefined set of watch points in the kernel. User should start the measurement and events will be stored in the trace. But also, SWAP has some additional features that are not supported by above mentioned well-known tools. The powerful interface on target is the unique SWAP feature, which distinguishes it from other similar tools. This comparison is illustrated by Table 2.

It is important to underline that in contrast with all above mentioned tools, SWAP has some unique tracing and profiling features not available in all above mentioned well-known tracing and profiling tools. Namely they are:

- reviewing history of all events in the kernel between profiling measurements;
- ability to process profiling data for obtaining different characteristics for collected parameters (average, min, max etc.);
- enhanced views of measured characteristics.

Table 2. Total comparison of SWAP tracing features with **SystemTap** and **LTT** tracing features

Feature \ Tool	ELP 2	SystemTap	LTT
Collecting trace of events from system kernel	Implemented	Implemented	Implemented
Dynamic instrumentation avoiding recompilation of system kernel	Implemented	Implemented	Not
Ability to specify purposeful predefined set of probing points in operating system	Implemented	Not	Implemented
Powerful User Interface on Target (Standalone)	Implemented	Not	Not
Automatic Analysis (Trace Comparison)	Implemented	Not	Not

All properties listed above makes SWAP more attractive in comparison with the above mentioned tools.

C. Visualization

Once the considering application has been instrumented and its performance data are available, the second step for "measure and modify" approach can be performed. This step visualizes a generated trace. Similar the tools that support this "measure and modify" method of classical analysis, SWAP can display post-mortem (after stopping data collection) trace. Also, SWAP supports a real-time system monitoring (like **Top** utility and **Windows Task Manager**). SWAP does this via different perspectives, such as tables, pseudo-bar charts, Gantt graph etc.

As the next step, user must perform the following steps:

1. Analyze generated views;
2. Select the most problematic regions;
3. Change the application source code.

This process repeats again until an adequate performance is achieved or a bug is fixed.

D. *SWAP User Benefits*

Before consideration SWAP in details let us shortly summarize its user benefits.

- At first, it needs to note that SWAP does not need a lot of resources. It uses less than 5% of CPU resources and less than 2 megabytes of memory. Therefore it is able to run immediately on embedded platform.
- At second, it needs to underline again that SWAP provides dynamic instrumentation of OS kernel and user-space applications in run time by using extended Kprobes functionality. This allows provide investigations and optimization without recompiling of programs and without source code files.
- At third, SWAP stores a lot of data regarding running software. Also it provides various facilities to manipulate collected data and allows provide various types of statistical analysis based on these data. In the other words, **SWAP** supports evaluation and visualization of wide set of system characteristics for memory management, process management, file system and network Linux kernel subsystems.
- **SWAP** does not need debug information and can monitor production version of Linux OS and applications.
- User interface of **SWAP** is developed with help of pseudo-graphics. Such design of interface allowed using of this tool in comfortable way with little amount of required resources. **SWAP** user interface supports menu, tables, bar graphs, Gantt graphs and other views that makes monitoring data more understandable for user.
- **SWAP** does not require network interface for it processing. Thus, **SWAP** can monitor Linux OS and applications on devices that do not support network interface at all.
- **SWAP** supports automatic analysis, such as trace comparison, smart system call analysis and performance bottleneck localization features.

C. *Platform requirements*

To date SWAP is designed and is available for the following platforms enumerated in Table 3.

Table 3.List of available platforms and Linux kernel versions for SWAP

№	Platform	Linux kernel version
1.	DTV ATI Xilleon X260 target board (MIPS)	2.6.10
2.	DMB2_AMD_Ver0.3 Au1200 evaluation board (MIPS)	2.6.11
3.	OSK OMAP5912 evaluation board (ARM)	2.6.10
4.	COSMOS (PXA 320) (ARM)	2.6.22

III. ACKNOWLEDGEMENT

Authors would like to thank Mrs. Jamee Lee, Leader of OS Group, Samsung Advanced Institute of Technologies for fruitful and extensive discussion on the proposed approach and help.

REFERENCES

[1]. Daniel Ariel Tamches, Barton P. Miller, Using Dynamic Kernel Instrumentation for Kernel and Application Tuning, International Journal of High Performance Computing Applications, Volume 13 , Issue 3, August 1999, pp. 263 – 276.

[2]. Richard McDougall, Jim Mauro, and Brendan Gregg, *Solaris Performance and Tools: Dtrace and Mdb Techniques for Solaris 10 and Opensolaris*, Prentice Hall, 2006.

[3]. Mathieu Desnoyers, and Michel R. Dagenais, *The LTTng tracer: A Low Impact Performance and Behavior Monitor for GNU/Linux,* Linux Symposium Proceedings, Volume 1, Ottawa, Ontario, 2006. (https://ols2006.108.redhat.com/reprints/desnoyers-reprint.pdf)

[4]. William E. Cohen, *Instrumenting the Linux Kernel with SystemTap,* RedHat Magazine, Issue#11, September 2005. (http://www.redhat.com/ magazine/011sep05/features/systemtap/)

[5]. http://www.lynuxworks.com/corporate/news/press/2003/042303c.php3

[6]. William E. Cohen, *Tuning Programs with OProfile*, Wide Open Magazine, Premiere Issue, 2004, pp. 53-62.

[7]. William E. Cohen, Gaining insight into the Linux® kernel with Kprobes, RedHat Magazine, Issue#5, March 2005. (http:// www.redhat.com/magazine/ 005 mar05/features/kprobes/)

[8]. Giridhar Ravipati, Andrew R. Bernat, Nate Rosenblum, Barton P. Miller and Jeffrey K. Hollingsworth, Toward the Deconstruction of Dyninst, Technical Report, Computer Sciences Department, University of Wisconsin, Madison (ftp://ftp.cs.wisc.edu/paradyn/papers/Ravipati07SymtabAPI.pdf)

[9]. Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston, Anil Keshavamurthy, Masami Hiramatsu, Probing the guts of Kprobes, Ottawa Linux Symposium, pp.101-114, July 2006.

[10]. Amitabh Srivastava, Alan Eustace, ATOM: A System for Building Customized Program Analysis Tools.

[11]. James R. Larus and Eric Schnarr, Computer Sciences Department, University of Wisconsin, Madison, EEL: Machine-Independent Executable Editing. (ftp://ftp.cs.wisc.edu/wwt/pldi95_eel.ps).

[12]. Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, and Brian Bershad, University of Washington, Brad Chen, Harvard University, Instrumentation and Optimization of Win32/Intel Executables Using Etch. (http://etch.cs.washington.edu/etch-usenixnt/etch-usenixnt.html).

[13]. David J. Pearce, Paul H.J. Kelly, Tony Field, Uli Harder, Imperial College of Science, Technology and Medicine, London, GILK: A dynamic instrumentation tool for the Linux Kernel.

[14]. Frank Ch. Eigler, RedHat, Problem Solving With Systemtap.(http://sourceware.org/systemtap/wiki/OLS2006Talks?action=AttachFile&do=get&target=problem-solving-with-stap.pdf).
[15]. Bryan M. Cantrill, Michael W. Shapiro and Adam H. Leventhal, Dymanic Instrumentation of Production System, Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference, p.2-2, June 27-July 02, 2004, Boston, MA.
[16]. Tamches, Ariel and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99), pp. 117-130. New Orleans, LA, February 1999. USENIX.
[17]. V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In Proceedings of the Linux Symposium, volume 2, pages 49-64, July 2005.
[18]. Daniel P. Bovet, Marco Cesati, *Understanding the Linux Kernel*, 3rd Edition, O'Reilly, 2005.
[19]. Richard M. Stallman, Roland H. Pesch and Stan Shebs, *Debugging with GDB: The GNU Source-Level Debugger*, 9th Edition, Free Software Foundation, 2002.
[20]. *ARM Architecture Reference Manual*, David Seal, ed., Addison-Wesley, 2000.
[21]. Andrew Sloss, Dominic Symes and Chris Wright, *ARM System Developer's Guide: Designing and Optimizing System Software*, Morgan Kaufmann, 2004.
[22]. Eric Youngdale, *The ELF Object File Format: Introduction*, Linux Journal, Issue 12, April 1995. (http://www.linuxjournal.com/article/ 1059)
[23]. Eric Youngdale, *The ELF Object File Format by Dissection*, Linux Journal, Issue 13, April 1995. (http://www.linuxjournal.com/article/ 1060)
[24]. *ARM ELF File Format*, ARM DUI 00101-A, ARM Limited, 1998. (http://www.arm.com/pdfs/DUI0101A_Elf.pdf)

Alexey A. Gerenkov received MS degree in Computer Sciences from Moscow Technical University of Telecommunication and Informatics in 2001. From 2001 till 2008 he jointed with EPSCom AG. He was involved in development of software for digital media devices, for telecommunication devices and for IP phones. He joined to Samsung Research Center in Moscow in 2008. His research interests include design and analysis of efficient algorithms, embedded systems, Linux Kernel.

Ekaterina A. Gorelkina received MS degree in Computer Sciences from Moscow Industrial Technical University in 2003 with the first class honors. She joined to Samsung Research Center in Moscow in 2006. She was involved in implementation of various projects related with compilers design and optimization, design of analyzers and optimizers of programs, operating system, embedded systems, real-time systems, artificial intelligence, parallel calculations, web-applications, computer graphics. Her research interests include design and analysis of efficient algorithms, embedded systems, Linux Kernel.

Sergey S. Grekhov received MS degree in Computer Sciences from Novosibirsk State University in 2004. He joined to Samsung Research Center in Moscow in 2005. His research interests include design and analysis of efficient algorithms, embedded systems, Linux Kernel development.

Sergey Yu. Dianov received MS degree in Computer Sciences from Moscow State Physics Engineering Institute (Technical University) in 2002. From 2002 till 2007 he jointed with Russian Federal Nuclear Center. He joined to Samsung Research Center in Moscow in 2007. His research interests include design and analysis of efficient algorithms, embedded systems, Linux Kernel development.

Jaehoon Jeong received MS degree in Electronic engineering from Hanyang University, Seoul Korea, in 2001. He joined to Samsung Software Lab. in 2001. His research interests include System Software design and development for embedded systems, Linux Kernel BSP and new kernel feature development for CE Product.

Oleksiy Kokachev received MS degree in Physics from Dniepropetrovsk National University, Ukraine, in 2002. He was involved in developments of Linux based embedded network management system, DVB products/systems, TV/Radio transmitter control devices, etc. He joined Samsung Software Lab. in 2005. His research interests include System Software design and development for embedded systems, Linux Kernel.

Leonid V. Komkov received MS degree in Computer Sciences from Moscow Aviation Institute (State Technical University) in 1998 and received MS degree in Computer Science from Western-Kennedy University, Wyoming, USA in 2001. He joined to Samsung Research Center in Moscow in 2002. His principal interests include distributed artificial intelligence, software agents, design and development software for embedded systems.

Sang Bae Lee received BS degree in Electronics Engineering from Korea University, Seoul Korea in 1997. He has 10 years experiences of Embedded System, especially, Real Time Operating System and Linux for Telecommunication System and Consumer Electronics Devices. Now his research interest is focusing on Embedded System Performance analysis.

Mikhail P. Levin received MS degree in Mechanics with Honor from Moscow Power-Engineering Institute (State Technical University) in 1978, PhD degree in Physics and Mathematics from Computing Centre of USSR Academy of Science in 1984, and Senior Research Scientist (Associate Professor) degree from Computing Centre of Russian Academy of Sciences in 1993. From 1980 till 1994 he jointed to Computing Centre of Russian (USSR) Academy of Sciences as a research staff. In 1994-2000 he was a principal software developer at German-Russian Joint Venture EuroSoft GmbH. He jointed to Korea Advanced Institute of Science and Technology (KAIST) from 2000 till 2001 as a professor at the Departments of Mathematics and Aerospace Engineering. In 2001-2003 he jointed to deCODE Genetics, Inc., Reykjavik, Iceland as a research staff. In 2003-2005 he jointed to General Energy Technologies Ltd. as a principal researcher in mathematical modeling. In 2005-2006 he jointed to Kraftway Corporation PLC as a leading staff member in High Performance Computing. He jointed to Samsung Research Center in Moscow in 2006. His research interests include design and analysis of efficient algorithms, embedded systems, Linux Kernel development, numerical methods and optimization of systems with distributed parameters, parallel algorithms and high performance computing. Dr. Levin has published more than 80 research papers and books on computer sciences, applied mathematics and numerical methods.