METHOD FOR AUTOMATIC DETECTION OF THE REASONS OF UNUSUAL SYSTEM BEHAVIOR BY EVALUATION OF CORRELATION BETWEEN EVENTS IN MONITORING TOOL

Ekaterina A. Gorelkina[1], Sergey S. Grekhov[1], Jaehoon Jeong[2], Mikhail P. Levin[1], *Member IEEE*

Abstract: This article describes the method related to simplifying detection of the problem reason in applications by using automatic analyzing of monitoring data on embedded system. Usually, dependencies in monitoring data are not obvious and usually it is very complicated to find the relationship between error symptom and the reason of the problem. The suggested method finds dependencies in system behavior automatically that gets the chance to detect hidden reasons of the problem in huge amount of monitoring data.

In particular, the proposed method detects the unusual system behavior and tries to find the most probable reason of such behavior by using statistical data mining algorithms. It structures monitoring data by special way and most probable reason of the problem is colored by monitoring tool to help the developer to analyze the system behavior more comfortable and more precise. The experiments show that this method can detect a wide set of problems in system behavior that accelerates and improves the analysis of system behavior according to monitoring information.

***Index Terms*: Automatic performance Analysis, Data Mining, Embedded Computing System.**

I. INTRODUCTION

A classical performance analysis of the programs is divided on two phases: monitoring and visualization. During monitoring phase program is instrumented and necessary data is collected. During visualization phase collected data is visualized for user. User processes data and detects the performance bottlenecks by hand. However, usually, amount of monitoring data for real programs is very huge and processing of specific kinds of data requires great experience in investigation area. Thus, in general case, processing of data by hand is very complicated task. There are a lot of monitoring tools that implement classical performance analysis. However, the next step after classical performance analysis is automatic performance analysis that tries to analyze monitoring data automatically to help developer to detect the reason of the problem without processing a huge amount of data by hand.

Method that is described in this article relates to automatic performance analysis. This method tries to help the user to detect not only the problem in program performance but also the reason of the problem. In particular, this method investigates the system calls for definition of correlation between arguments and return values. If system call returns error, method tries to detect the arguments that can influence on this result. In the other words, method tries to detect the reason of the error in system call if the reason of error is specific system call arguments values.

Our method allows finding dependencies between events (arguments of events) according to specific criteria (similarity measurement) that allows grouping events with not only the same parameters, but with similar parameters. Events that have similar parameters describe similar behavior of the system with high probability and can contain reason of such behavior. Thus, our method can provide automatic performance analysis by performing the following stages automatically:

- Detecting similar behavior;
- Detecting erroneous behavior;
- Detecting the reason of the problem.

All these stages we can do automatically and these stages are unique for our method.

Time for analysis of monitoring data is reduced significantly by using of our method, because of user needs to look through several groups that contains problem only; user should not do the following time consuming tasks:

- Finding dependencies manually in huge amount of data;
- Detecting erroneous behavior manually (it requires great experience in investigated area)
- Detecting reason of the problem manually (it requires great experience in investigated area)

Our method performs all these phases automatically that reduces time for analysis.

[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

Results of our method can not be obtain by setting conditions for a monitoring data (sorting), because of we use fuzzy logic in our algorithms for emulating human mining process for analysis of the data. Our method does not require any initial settings from user to start processing: all dependencies are extracted from monitoring data automatically without any human help.

Without our method it is difficult to get automatic analyses because of existent solutions do not allow localization of the problem by separating data into set of groups. Additional, existent methods do not indicated the reason of the problem automatically. Thus, there are no existent methods; those provide the same result as proposed method.

Our approach uses data mining clustering method for separating all events arguments and return values into clusters. Each cluster contains similar events (events that have similar arguments and return values). During future processing, events arguments in each cluster are analyzed and most probable error reason arguments are indicated. We have performed experiments for events that describe system call events in the system. However, our approach can be extended for other types of events.

Data mining has been widely used in area of science and engineering, such as machine linguistic, bioinformatics, genetics, medicine, education, and electrical power engineering. Using data mining technique for monitoring of programs behavior is very restricted and this article demonstrates novel result of applying such technique in this area.

II. DESCRIPTION OF THE METHOD

The suggested method is divided into two phases: clustering of monitoring data and definition of problem reason in each cluster.

To provide clustering method for monitoring data our approach splits into the following key steps:

a) Determining the objects of analysis. We need to take into account main features of monitoring data that describe system behavior, but we need to ignore useless features to provide meaningful result.

b) Determining the similarity function for the objects. Similarity function is criteria of grouping several events together. Choosing of criteria determines a set of problem that we need to detect.

c) Clustering algorithm for the monitoring objects. This algorithm should have low time and memory costs because we need to run this algorithm on embedded system.

Events generated by SWAP profiling tool have been chosen as objects for analysis. Each event corresponds to entry into (or return from) a kernel function called by a process. Event consists of the following information: function ID that cause the event generation, process and thread IDs (context of invocation), arguments (or return value), which provides function call related information.

List that contains arguments and return value of each system call is used as a vector for clustering procedure because of arguments contain information about the system behavior and can indicate a problem reason. We decided to examine arguments of the following types: string, integers (flag container, memory address, counter).

Dice's Coefficient has been chosen as similarity metrics for all examined types:

$$d(A,B) = \frac{2 \cdot |A \cap B|}{|A| + |B|}, \qquad (1)$$

where A and B are sets that represent two comparing objects (strings, integer numbers).

For each type we suggested concrete variant of Dice coefficient metric to provide meaningful result according to our goal.

For integers containing several logical values (true, false) or memory addresses the bitwise form of the number can be used for calculating similarity. The specificity of similarity of two such values can be described as following: the values can be similar bitwise, but its algebraic difference can be very big. Thus, the following metrics can be proposed. Let A and B are the integer number containing "flags". Let's consider these numbers using the bitwise form. Let A∩B is the number containing bits, identical for both A and B. Then, the (1) can be used with following specifications: |A| - number of bits in bitwise form of A and | A∩B | is the number of identical bits in bitwise form of A and B. Since, A and B have the same length of bitwise form, the following obvious simplification of (1) can be done:

$$d(A,B) = \frac{|A \cap B|}{|A|}. \qquad (3)$$

Such metrics allows avoiding artifacts when calculating the distance. Simple algebraic difference could be inconvenient for detecting the similarity, thus, the bitwise for of the number is considered. For example, let's take two numbers with following hexadecimal form: 0xC0080000 and 0xC0090000. The algebraic difference between them equals to 0x10000 or 65535 in decimal form. This is a relatively big number, while the bitwise forms of the numbers are very close. Thus, in case of finding the difference of memory addresses or flag containers, the bitwise form is more useful.

For avoiding artifacts in calculating the distance between integers containing counters it is proposed to use metrics similar to string-distance and bitwise-distance. Let A and B are the integer number containing counters. Let's consider these numbers using the bitwise form. Let W is the weight vector for

calculating the distance. Then, the following form of Dice's Coefficient can be used:

$$d(A,B) = \frac{\sum(w \times A, w \times B)}{\sum(w, w)}, \qquad (2)$$

where $w=(w_1, \ldots, w_n)$ – vector of weights, $w \times A$ – vector, where i-th position is w(i), if and only if bit A(i) is 1 (0 otherwise), ($w \times A$, $w \times B$) is scalar product of two vectors.

For example, let's take two integer numbers: 0 and 4096. The algebraic difference of these numbers is relatively big, but the bitwise representation distinguishes by only one bit. Besides, using the algebraic difference makes the procedure of determining similarity between two events more difficult: defining the threshold value (for details see below the procedure or detecting of similarity of two events) becomes a very complex problem. However, the simple bitwise form of Dice's coefficient will consider these numbers as similar and, thus, can not be applied here. Considering all this, one can propose to use a weight vector, for example, w=(32, 31, …, 2, 1), allows avoiding unwanted artifacts.

For events of monitoring tools the string argument usually specifies the name of the file that corresponds to the occurred function call (for example, file name for system call sys_open). The distance for such type of arguments should reflects how similar the two paths to the files are.

Let A and B are the strings, describing some paths in file system. Assume that A and B can be split into sets of disjoint substrings (a(1), …, a(n)) and (b(1), …, b(m)) respectively, where the symbol '/' is separator. Let $A \cap B$ is the set of distinct substrings that belongs to both A and B. Then metric (1) can be specified with following definitions: |A| means the number of substrings in set, corresponding to A; $|A \cap B|$ - the number of substring in set $A \cap B$.

For example, let's take two strings which represent the path to a library file: '/usr/lib/libc.so.6' and '/usr/local/lib/libc.so.6'. Simple comparing of these paths will detect that they are different. But it can be easily seen that the paths are very similar. Thus, the Dice's coefficient is more appropriate metrics for such type of data. For the considered two strings, the sets of substrings will be A={usr, lib, libc.so.6} and B={usr, local, lib, libc.so.6}. Then $A \cap B$={usr, lib, libc.so.6}. $|A \cap B| = 3$, $|A| = 3$, $|B| = 4$, $d(A,B) = 2 \cdot 3/(3+4) = 6/7$ – the final result can be interpreted as a degree of similarity. Considering that 6/7 is quite close to 1 (this can be done by using the threshold values, see next paragraph for details), one can say that the mentioned strings are similar.

As clustering algorithm we use on-line clustering algorithm with predefined threshold. In details, clustering method looks like the following. Accordingly to previously defined metrics for event arguments (1) – (3), we suggested to measure the similarity between two events by method, which takes into account the nature of arguments. The result of comparing arguments of two events A and B is represented as a vector d = (d(1), …, d(n)), where $0 \le d(i) \le 1$ – the similarity between arguments of events. Then, our method assumes that event A is similar to event B if and only if all compounds of d satisfy the following condition: $d(i) \ge S(i)$, $\forall i \in \{1, \ldots, n\}$, where S(i) is a threshold value for the data type corresponding to the i-th position in event. In the other words, two vectors are put into one cluster if values of similarity metric functions for all arguments are greater than threshold (for each type of arguments specific threshold is defined). This algorithm requires low time and memory resources that allow using this method on embedded systems.

Fig. 1 shows hierarchy of clusters that is built as result of events clustering algorithm. All collected events are split into clusters according to event type and then each cluster is divided to several sub-clusters according to events' similarity.

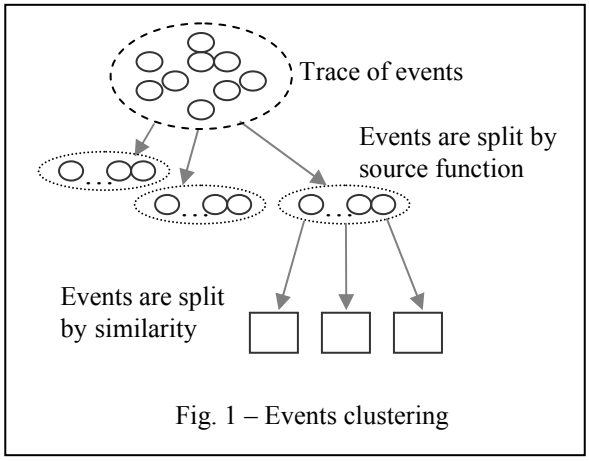

Fig. 1 – Events clustering

To provide definition of problem reason in each cluster, we provide the automatic search of stable argument in each cluster. Each stable argument we consider as possible problem reason. We detect stable argument by the following way. For i-th argument of each event, the average similarity M(i) with other events and its dispersion $\sigma(i)$ are calculated. We consider argument stable if distribution of similarity within the cluster is located in interval $[M(i) - \sigma(i), M(i) + \sigma(i)]$ (see Fig. 2).

Thus, all stable arguments are considered as the main reason of joining events in cluster and are considered as problem reason if all system calls in cluster contain erroneous return value.

III. EXPERIMENTAL RESULTS

We've performed a set of experiments for demonstration of our method results. These results are described in this section.

Table 1 describes the experiment environment:

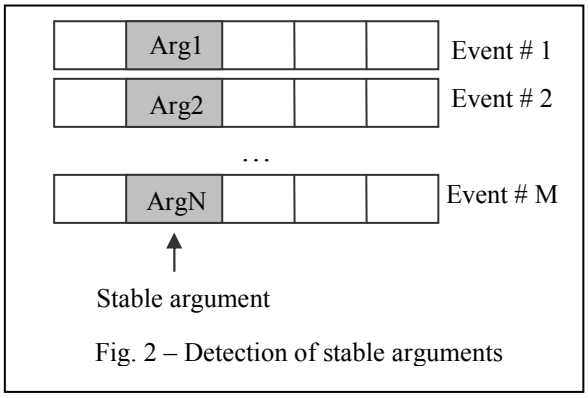

Fig. 2 – Detection of stable arguments

Table 1. The environment of the experiment

Component of experiment	Details
Target hardware	ARM CPU, OMAP5912 OSK
Considered application	MP3 codec MadPlay
Track duration	47 sec
Monitoring tool	SWAP (System-Wide Analyzer of Performance)
Monitored events in kernel	System call for manipulating files: open, close, read, write, ioctl

To perform our experiments we have done the following steps:

- Run production version of MP3 codec on embedded platform for track duration 47 seconds;
- Collect monitoring events from MP3 codec application by SWAP monitoring tool (system calls: open, close, read, write, ioctl);
- Run our method for automatic detection a performance bottleneck. Notice, that implementation of our method is built into SWAP monitoring tool. Thus, to run our method in this case, it is necessary to use special functionality of SWAP, which is called as "Event Clustering".

During method processing, all events were spited by affiliation with probing point (open, close, read, write and ioctl). After that, each group of events was divided into disjoint groups (clusters) with help of similarity metrics. Several clusters contain bottlenecks. Our method has detected a set of bottlenecks.

For example, our method has detected "Searching for invalid library path" performance bottleneck. This issue occurs when application tries to open invalid file path for specific library. Our method detected several big clusters of events (totally about 50 events) from 'open' probing point: MP3-player tried to open a library from several distinct destinations. Since the amount of data is large it is quite difficult to find this bottleneck by hand (see Fig. 3).

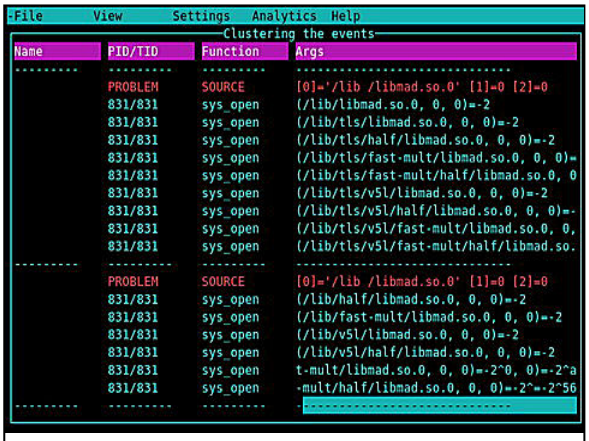

Fig. 3 – Cluster of events for sys_open containing a bottleneck

Additional, our method has detected "Closing invalid file" performance bottleneck. This issue occurs when close system call fails. Among about 30 calls of function 'close', there was singled out one call with mistaken input data – bad filename (as a result, function returned erroneous value). Our method detects this issue and indicates components of 'close' events that can be a reason of the problem in this case (see Fig. 4).

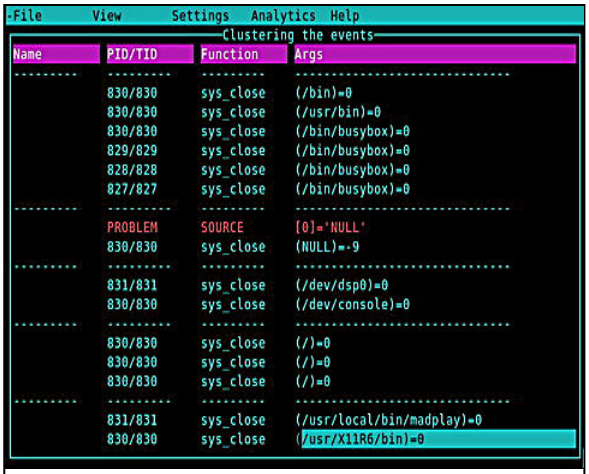

Fig. 4 – Cluster of events for close() system call containing an error

As last example, our method has detected "Un-read data" performance bottleneck. This performance bottleneck occurs while MP3 codec tries to read data from storage during processing the test MP3-file. In two cases, the 'read' system call returned less data than it was requested. Erroneous events were joined into one cluster that allowed to examine them

simultaneously and helped to determine the possible bottleneck easily (see Fig. 5).

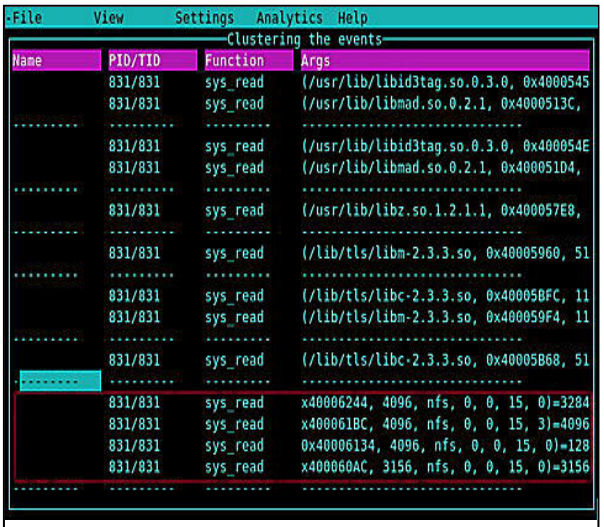

Fig. 5 – Cluster of events for read() system call containing a bottleneck (marked with red

Performance characteristics of our method for described above test environment are displayed in Table 2.

Table 2. Performance characteristics

Characteristic	Detailed description
Memory usage	1.2 Mb
Duration of data processing	5 sec

Functional characteristics of our method for described above test environment are displayed in Table 3.

Table 3. Functional characteristics

Characteristic	Detailed description
Number of detected bottlenecks	14
Events amount	4030
Number of clusters	51
Miss ratio	0
False alarm	0

Experiments show, that our method can indicate a wide set of different issues without any additional information from user. User needs run our method for collected data and looks through indicated clusters to analyze dependencies in monitoring data without finding dependencies in huge amount of data manually.

Additionally, our experiments show that performance characteristics of our method allow using it for data processing on embedded systems.

IV. CONCLUSION

The paper contains description of the method that finds the dependencies in system behavior by the measurement of similarity between two events generated by profiler for operating system (where each event could have specific arguments, such as: string, flags, memory address or counter). The experimental results have confirmed that this approach allows searching bottlenecks automatically, thus, the time of detecting the source of low performance of the system becomes much less than analyzing the trace of events by hand.

REFERENCES

[1]. 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)

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

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

[4]. 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).

[5]. 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).

[6]. 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.

[7]. Frank Ch. Eigler, RedHat, Problem Solving With Systemtap.(http://sourceware.org/systemtap/wiki/OLS2006Talks?action=AttachFile&do=get&target=problem-solving-with-stap.pdf).

[8]. 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.

[9]. 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.

[10]. 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.
[11]. Yiming Yang, Tom Ault, Thomas Pierce, Charles W. Lattimer. Improving text categorization methods for event tracking. Language Technologies Institute and Computer Science Department, Carnegie Mellon University, Pittsburgh;
[12]. Yi Zhang, Jamie Callan. Maximum Likelihood Estimation for Filtering Thresholds. School of Computer Science, Carnegie Mellon University;
[13]. Yiming Yang, Jaime Carbonell, Ralf Brown, Tom Pierce, Brian T. Archibald, Xin Liu. Learning approaches for Detecting and Tracking News Events. Language Technologies Institute and Computer Science Department, Carnegie Mellon University, Pittsburghames Allan, Ron Papka, Viktor Lavrenko. On-line New Event Detection and Tracking. Center for Intelligent Information Retrieval Department of Computer Science University of Massachusetts;
[14]. Charles L. Wayne. Topic Detection & Tracking (TDT). National Security Agency;
[15]. James Allan, Jaime Carbonelly, George Doddingtonz, Jonathan Yamronx, Yiming Yangy. Topic Detection and Tracking Pilot Study. UMass Amherst, CMU, DARPA, Dragon Systems;
[16]. Juha Makkonen, Helena Ahonen-Myka, Marko Salmenkivi. Topic Detection and Tracking with Spatio-Temporal Evidence. Department of Computer Science, University of Helsinki, Finland;
[17]. Khoo Khyou Bun, Mitsuru Ishizuka. Topic Extraction from News Archive Using TF*PDF Algorithm. Dept. of Information and Communication Engineering, The University of Tokyo;
[18]. K. Rajaraman, Ah-Hwee Tan. Topic Detection, Tracking and Trend Analysis Using Self-organizing Neural Network. Kent Ridge Digital Labs, Singapore;
[19]. Yi Zhang. Using Bayesian Priors to Combine Classifiers for Adaptive Filtering. Language Technology Institute, School of Computer Science, Carnegie Mellon University, Pittsburgh;

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.

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.

Ekaterina A. Gorelkina received MS degree in Computer Sciences from Moscow State Industrial 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.

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.