Russian Technological Journal. 2021;9(6):7–15

ISSN 2500-316X (Online)

UDC 004.051
https://doi.org/10.32362/2500-316X-2021-9-6-7-15

REVIEW ARTICLE

Comparative analysis of software optimization methods in context of branch predication on GPUs

Igor Yu. Sesin @**,**
Roman G. Bolbakov

MIREA – Russian Technological University, Moscow, 119454 Russia
@ *Corresponding author, e-mail: isesin@protonmail.com*

Abstract. General Purpose computing for Graphical Processing Units (GPGPU) technology is a powerful tool for offloading parallel data processing tasks to Graphical Processing Units (GPUs). This technology finds its use in variety of domains—from science and commerce to hobbyists. GPU-run general-purpose programs will inevitably run into performance issues stemming from code branch predication. Code predication is a GPU feature that makes both conditional branches execute, masking the results of incorrect branch. This leads to considerable performance losses for GPU programs that have large amounts of code hidden away behind conditional operators. This paper focuses on the analysis of existing approaches to improving software performance in the context of relieving the aforementioned performance loss. Description of said approaches is provided, along with their upsides, downsides and extents of their applicability and whether they address the outlined problem. Covered approaches include: optimizing compilers, JIT-compilation, branch predictor, speculative execution, adaptive optimization, run-time algorithm specialization, profile-guided optimization. It is shown that the aforementioned methods are mostly catered to CPU-specific issues and are generally not applicable, as far as branch-predication performance loss is concerned. Lastly, we outline the need for a separate performance improving approach, addressing specifics of branch predication and GPGPU workflow.

Keywords: general-purpose computing for graphical processing units, optimizing compilers, predication

• **Submitted:** 22.03.2021 • **Revised:** 26.05.2021 • **Accepted:** 25.07.2021

For citation: Sesin I.Yu., Bolbakov R.G. Comparative analysis of software optimization methods in context of branch predication on GPUs. *Russ. Technol. J*. 2021;9(6):7–15. https://doi.org/10.32362/2500-316X-2021-9-6-7-15

Financial disclosure: The authors have no a financial or property interest in any material or method mentioned.

The authors declare no conflicts of interest.

ОБЗОР

Сравнительный анализ методов оптимизации программного обеспечения для борьбы с предикацией ветвлений на графических процессорах

И.Ю. Сесин [@], **Р.Г. Болбаков**

МИРЭА – Российский технологический университет, Москва, 119454 Россия
[@] *Автор для переписки, e-mail: isesin@protonmail.com*

Резюме. Технология GPGPU (General Purpose computing for Graphical Processing Units – расчеты общего назначения на графических процессорах) является мощным инструментом для переноса задач параллельной обработки информации на GPU (Graphical Processing Unit – графический процессор). Эта технология находит применение практически в любой области, требующей проведения массы параллельных расчетов, и применяется как в научной и коммерческой, так и в любительской среде. Разработчики программ общего назначения, запускаемых на GPU, неизбежно сталкиваются с падением производительности ввиду предикации ветвления кода. В условиях предикации ветвления исполняются обе ветви условного оператора вне зависимости от истинности условия, но посредством маскирования выполняемых инструкций программа учитывает только результат работы верной ветви. Из-за этого программы общего назначения, имеющие большие участки кода, скрытые за условными операторами, становятся существенно менее производительными на графических процессорах. В статье рассматриваются существующие в предметной области методы и подходы к увеличению производительности программного обеспечения в рамках их применимости к решению проблемы падения производительности при предикации. Приводится описание методов, их сильных и слабых сторон, а также рамок их применимости, на базе чего делается заключение о возможности их использования на GPU. В число рассмотренных методов и подходов вошли следующие: оптимизирующие компиляторы, JIT-компиляция, предсказатель переходов, спекулятивное исполнение, адаптивная оптимизация, специализация алгоритма во время исполнения, оптимизация на основе профилирования. Показано, что указанные аппаратные и программные подходы к увеличению производительности программного обеспечения преимущественно ориентированы на решение проблем специфичных для CPU (Central Processing Unit – центральный процессор) и в целом неприменимы для разрешения потерь производительности при предикации на GPU. Указывается на необходимость создания отдельного подхода, ориентированного именно на решение проблемы предикации ветвления на GPU.

Ключевые слова: расчеты общего назначения на графических процессорах, оптимизирующие компиляторы, предикация

• **Поступила:** 22.03.2021 • **Доработана:** 26.05.2021 • **Принята к опубликованию:** 25.07.2021

Для цитирования: Сесин И.Ю., Болбаков Р.Г. Сравнительный анализ методов оптимизации программного обеспечения для борьбы с предикацией ветвлений на графических процессорах. *Russ. Technol. J*. 2021;9(6):7–15. https://doi.org/10.32362/2500-316X-2021-9-6-7-15

Прозрачность финансовой деятельности: Никто из авторов не имеет финансовой заинтересованности в представленных материалах или методах.

Авторы заявляют об отсутствии конфликта интересов.

INTRODUCTION

Graphics processors, or Graphics Processing Unit (GPU), are specialized hardware that performs the processing of graphical information. Unlike a central processing unit (CPU), graphics processors are specialized for parallel processing of large amounts of data, which causes certain differences in their architectural design:

- CPUs have a small number of physical cores (from 1 to 32), GPUs have orders of magnitude more physical cores (can be hundreds or thousands[1, 2], depending on the compromise between the number of cores and their power chosen by the hardware manufacturer);
- cores in the CPU have additional hardware functionality, such as their own caches, instruction pipelines, and branch prediction modules. GPU kernels are very simple arithmetic kernels focused on fast processing of floating-point numbers. A number of modern GPUs also include specialized cores for ray tracing and tensor computing [1, 2];
- CPU is separated from random access memory; GPUs have direct access to video memory located on the same board;
- GPU cores cannot write data in the memory area allocated for the executable code;
- CPU falls into the Multiple Instruction, Multiple Data (MIMD) classification, GPU falls into the Single Instruction, Multiple Data (SIMD) classification according to Flynn [3].

Using General Purpose Computing on Graphics Processing Units (GPGPU) technology, it is possible to run programs on GPUs other than highly specialized shader programs. This technology finds application in many areas—from mining crypto currencies to calculating the protein folding[3].

Applying the GPGPU technology, one has to deal with certain peculiarities of the construction of programs and their behavior in the conditions of execution on a graphics processor. For example, in the course of writing a GPGPU program, the authors of this article noticed a downward trend in its performance as more functionality was added. At the same time, the performance losses were much more significant than could be assumed based on the complexity of the algorithm of the added functionality.

When looking into the above problem, it was found that this happens due to the specifics of the SIMD architecture of graphics processors: when the program executes a conditional operator, both branches will be executed, but the operations of the wrong branch will not be applied. This feature is called predication [4, p. 168] of branches of execution, and it is needed, first of all, to replace the dependence on the flow of execution with the dependence on data. The very need for such measures is justified by the fact that most of the hardware part of the GPU architecturally falls into the SIMD class of Flynn's taxonomy [3], and individual modules of the system cannot have their own threads of execution and, accordingly, cannot follow along various branches of the conditional operator.

The degree of influence of the above feature on the program differs from program to program and correlates with the number of possible settings. So, if the program is initially designed to solve one maximally specific task without the possibility of customization (for example, calculating a certain hash function), then the influence of predication will be minimized.

Programs that perform more general tasks, and, therefore, have a large list of plug-in or optional functional that will be used in the process of the program only with certain input data, are more susceptible to performance degradation.

Let us take a 3D scene renderer as an example of such a program. Rendering in the field of computer graphics is a term used to describe the process of obtaining an image from certain data characterizing the objects of the displayed scene. A program that implements such a process is called a renderer. Professional renderers can include thousands of different options that change the behavior of the program—from adjusting the angle of the camera's field of view to the detailed configuration of the bidirectional reflectance distribution function for each surface.

Taking a renderer program as an example, imagine a situation in which a certain scene is rendered with a set of primitives that can only scatter or emit light, but not reflect it. In this case, the code responsible for calculating the reflections should not be called, but its very presence will slow down the program due to predication.

To enable or disable the desired program behavior, in the vast majority of cases, one would use conditional statements that depend on some input data of the program, regardless of whether it just data or configuration settings.

As program's functionality grows, the number of conditional statements will increase, and accordingly,

[1] Advanced Micro Devices, Inc, Graphics Specifications, 2021. URL: https://www.amd.com/en/products/specifications/graphics. Accessed March 1, 2021.

[2] NVIDIA. Comparison of specifications of RTX 30 video cards. URL: https://www.nvidia.com/ru-ru/geforce/graphics-cards/30-series/compare/?section=compare-specs. Accessed March 1, 2021.

[3] Houston M. General Purpose Computation on Graphics Processors (GPGPU). ATI HD 2000 Series. Launch, Tunis, Tunisia; 2007. URL: https://graphics.stanford.edu/~mhouston/public_talks/R520-mhouston.pdf. Accessed March 1, 2021.

the costs spent by the GPU on processing conditional program branches will also increase, which, in turn, leads to a certain ceiling of program complexity—after a certain critical mass of branches, the program will slow down so much that will trigger the internal protection of the driver (if any) and abort the execution.

It should be noted that from a programmer's point of view, the mere presence of additional code slows down the program, despite the fact that the code should not be executed. This behavior is highly uncommon for programs running on the CPU; Moreover, the correct use of conditional jumps is often the key to writing more efficient programs, thanks to the branch predicator built into the cores of modern CPUs.

RESEARCH OF EXISTING SOFTWARE AND HARDWARE FOR INCREASING PERFORMANCE

Let us consider the existing technologies for optimizing programs and increasing performance, assessing their applicability for solving the described problem.

It should be noted that within this article there will be no attempts to numerically compare the methods under consideration. This is due to the fact that almost all methods of increasing software performance are based on a certain characteristic of the optimized program, hardware platform or programming language. Techniques that work for some programs may be useless or even harmful to others. The quality of implementation of a particular method also has a direct impact on the result obtained, and the same approach, implemented in different ways, can give strikingly different results. The specific metric of the success of the application of methods depends on many factors, both quantitative and qualitative, which cannot be excluded from the study without jeopardizing the reproducibility of the study itself. On the other hand, the inclusion of these factors will narrow the study down to comparing particular cases, namely, comparing specific programs on specific hardware in a specific configuration, which is not representative for describing the overall picture.

In view of the above, the authors analyze mainly the qualitative characteristics to determine applicability of methods in lieu of software performance improvement.

Optimizing compilers continue to play a leading role in improving software performance. In general, an optimizing compiler is any compiler that performs special operations on compiled code to improve its performance.

Initially, this meant replacing certain operations with equivalent, but more efficient ones, such as replacing multiplication or integer division by numbers that are powers of two, with bit shift operations, but with the development of the scientific field, software development methodologies, and hardware capabilities, compilers have acquired an extensive arsenal of optimization techniques.

They are usually divided into low-level and high-level optimizations.

Low-level optimizations involve changing the generated machine code to make the best use of hardware platform features. This includes the use of more efficient machine code constructs, including the use of special commands available in the target architecture, vectorization of operations, function inlining, etc.

High-level optimizations operate at the level of abstract algorithmic elements that make up a program—loops, branches, and basic blocks. They use data about the structure of the program to transform the intermediate representation of the program into a more efficient form.

In practice, the line between these groups is blurred, since many methods involve elements of both high-level code analysis and low-level control over the generation of machine instructions.

Optimizations widely used by compilers include:

- constant folding—if a certain expression consists only of constants, then it is calculated at the compilation stage and its result is substituted instead of the original expression;
- eliminating the "dead code" [5], i.e., code sections that cannot be reached by the program;
- eliminating the dead stores [6], i.e., removal of operations storing value in the variable that goes unread further in the code;
- optimizing the register allocation [7] reorganizes the code in such a way as to minimize the number of memory accesses during the program operation by keeping the most frequently used variables in certain general-purpose registers of the processor;
- operation parallelization—changing the order of operations so that they can be run in parallel at the level of threads, memory or instructions;
- strength reduction—replacing slow operations with equivalent, but faster ones on the target architecture;
- loop optimization—a wide group of methods focused on working with loops, including such approaches as moving invariants out of the loop, inversion of loops, loop unrolling, dividing and merging loop bodies, removing conditional statements from the loop, etc.;
- instruction selection [8] allows the compiler to pick the most efficient machine code combination for the target processor;

- Instruction Scheduling [9, 10]—reorganization of instructions so as not to cause downtime of the central processor pipeline as a result of long memory accesses, exhaustion of processor resources, or branching.

Most of the traditional methods of program optimization used in optimizing compilers can be excluded from consideration, since they are focused on CPU features that are not inherent in arithmetic cores used in graphics processors. The methods that can be applied are either already implemented by compilers of programs for GPUs, or do not solve the presented problem.

Let us consider more dynamic methods for increasing the performance.

JIT-compilation technology [11] (just-in-time) improves the performance of programs in languages that are compiled into the bytecode. The bytecode is called an intermediate representation of the program code [12–14], which is executed by a virtual machine, which is its key difference from machine code executed directly by the processor.

In the framework of the JIT-compilation technology, the bytecode of programs is compiled into machine code as needed, right at the time of the program execution[4]. This allows you to speed up the work of programs in several ways at once:

- the startup delay, which without this technology is caused by lengthy processing of the source code, is significantly reduced; compilation from source code to bytecode is significantly slower than compilation from the bytecode to machine code;
- using the features of specific hardware to improve the performance;
- optimizing the program using data obtained during program operation;
- the ability to dynamically link libraries without the overhead inherent in compiled languages.

However, the use of the JIT compilation within programs running on a GPU currently does not appear to be feasible due to architectural restrictions on dynamically changing the executable machine code on GPUs.

In addition, embedding compilation into the operation cycle of the GPU itself will require a lot of both hardware and software changes, and, in general, will not give the same performance gain as on the central processor; it is because there is no problem of connecting libraries on the GPU (due to the static compilation of programs), and the intermediate representation of the program will be fully compiled into machine code in the process of transferring the program to the GPU. In general, these problems make the use of JIT impractical on the GPU.

One of the key elements ensuring the performance of modern CPUs is the branch predictor[5]. A branch predictor allows a pipelined processor to begin loading instructions from one of the branches of a branching statement into the pipeline before a condition is determined to be true.

This plays an important role in improving performance due to the parallel execution of instructions operating on mutually independent data. In the general case, without a predictor, the processor pipeline will not know which instructions from which branch should be used, and will be able to start loading them only after evaluating the truth of the condition in the conditional statement, which would entail pipeline stalling for each of the conditions. This can be avoided by predicting the result of the condition and starting loading instructions from the corresponding branch into the pipeline in advance. If the predictor fails, the pipeline will idle until the correct instructions are loaded.

The branch prediction itself is carried out heuristically [15−17], commonly, based on the statistics of the execution of a given section of the code. The time cost of a predictor error is inevitable due to imperfect prediction mechanisms, but with a sufficiently accurate prediction mechanism, the time loss from incorrect predictions becomes insignificant compared to the gain from correct predictions.

The branch prediction approach was developed in the form of speculative execution[6]—in addition to loading instructions onto the pipeline; the predicted branch is also executed before the condition is established. In case of an error, time losses increase, because you need to flush the entire pipeline and load the correct branch instructions into it.

The branch prediction and speculative execution approaches are implemented in hardware and are completely transparent to the programmer. Nevertheless, with certain program constructions, it is possible to get performance degradation, which is most noticeable in the case when the program consistently forces the module to mispredict the branch. The existing practice of optimizing low-level programs for

[4] Croce L. Just in Time Compilation. Columbia University. URL: http://www.cs.columbia.edu/~aho/cs6998/Lectures/14-09-22_Croce_JIT.pdf. Accessed March 1, 2021.

[5] Fog A. The microarchitecture of Intel, AMD and VIA CPUs. An optimization guide for assembly programmers and compiler makers. Technical University of Denmark. URL: https://www.agner.org/optimize/microarchitecture.pdf. Accessed March 1, 2021.

[6] Gabbay F. Speculative execution based on value prediction. Research Proposal towards the Degree of Doctor of Sciences. Technion-Israel Institute of Technology (IIT), Department of Electrical Engineering. 1996. 65 p. URL: http://citeseerx.ist.psu.edu/viewdoc/download?-doi=10.1.1.29.5397&rep=rep1&type=pdf. Accessed March 1, 2021.

better interaction with the branch predictor is a direct consequence of the imperfection of the used prediction methods.

The very presence of a branch prediction and speculative execution mechanism has also opened the way for a whole class of hardware vulnerabilities that allow a program to gain unauthorized access to information. A specially designed program can determine the contents of memory cells using side channels (typically access latency) arising from erroneous branch predictions [18–20].

The authors of this article believe that predicting transitions to the GPU does not make sense due to the absence of a pipeline on its cores, as well as the absence of branches in the executable machine code in their typical understanding due to predication. Instead, the code is structured linearly and no conditional statements are executed when executing conditional statements' "jumps" of the instruction counter register.

Even if we assume that this obstacle will be overcome, implementing the prediction module into the GPU would be extremely difficult due to the complexity of such a module. A typical GPU includes a large number of weak arithmetic cores, orders of magnitude larger than the number of cores on a typical market CPU. Adding a predictor to each of them will increase their size and power consumption to unacceptable levels.

The adaptive optimization method [21] relies on a special toolkit, similar to JIT compilers, to recompile the program code while it is running. This method applies different optimization techniques to a program depending on the productivity of the code. For example, adaptive optimization will apply more aggressive optimization techniques in bottlenecks where the code is spending the most CPU time. Such aggressive optimization techniques are time-consuming and wasteful to apply to the entire program. Therefore, sections of the program that are rarely called will be optimized with more primitive, but faster methods, allowing you to save time in general, albeit at the expense of the fact that this code will run slower.

The application of the above approach to GPUs is impossible for a reason similar to JIT; for it to work, the program requires the ability to rewrite its code, which is not supported by modern GPUs.

Let us consider the run-time algorithm specialization, an approach that comes from the field of automatic theorem proving [22]. This approach implies the creation of specific implementations of complex functions for certain inputs, for which it is possible to represent the original function in a simpler form. This can sometimes translate into creating table values for the function and caching individual results for frequently repeated values. In more complex cases, several code variants can be generated corresponding to particular variants of the function, where, for example, part of the calculations is excluded due to specific values of the input data.

The aforementioned approach and methods similar to it [23] require not only writing to the memory area of the executable code, but also a large amount of additional memory for the code of the obtained specializations, which makes its application within the GPGPU technology impossible.

Let us consider now the profile-guided optimization technology. It is an approach to program optimization in which the optimization process is controlled by the performance results of the program [24, 25]. This approach often requires the use of a special compiler, which takes on the task of instrumentation and taking measurements of the program execution time. Such a special compiler translates, builds, and runs the program many times, analyzing during these test runs the frequency of use of various sections of the program and the speed of their invocation. Using this data, the compiler applies various optimization strategies to code sections that showed poor results during test runs. These strategies include selecting the optimal register allocation, function embedding, as well as a lot of techniques tied to the successful grouping of executable code in memory pages to speed up the work of caches at several levels at once—from the mechanism for managing virtual memory at the operating system level to the CPU instruction cache.

Due to the fact that the statistics collected by such a compiler is representative only of those actions that were performed by the program during the testing process, a typical set of tests should include the most common scenarios for working with the program. It could be much easier implemented for GPU programs, due to their lack of interactivity.

The authors of the article believe that the use of this method for GPU programs is impractical, since in the conditions of working with a GPU it is extremely difficult or even impossible to instrument the code and, accordingly, to obtain measurements of the performance of individual sections of the code necessary for the method to work.

From the analysis of existing technologies for increasing performance, it is clear that the problem of loss of performance with increasing functionality, which is typical for graphics processors, is atypical and non-trivial.

None of the approaches discussed solve the problem of branch predication on GPUs, but they are an important starting point in the process of finding a solution.

On the one hand, to solve the problem, the use of dynamic methods is required to maintain the flexibility of the program functionality for the GPU, but on the other hand, the considered existing solutions applied on the CPU cannot be used in the context of GPU in practice due to the particular features of the architecture focused on massively parallel calculations. Changes to the established GPU architecture are obviously impractical due to a number of factors, including a multiple increase in production costs and an inevitable decrease in performance.

By virtue of the stated provisions, it can be established that there is a need to find an approach to optimizing programs for GPU that is compatible with the limitations of graphic processors, but at the same time flexible enough to ensure the preservation of the program's functionality.

CONCLUSIONS

The article discussed the problem of decreasing the performance of programs for general-purpose calculations on GPU, which arises in the course of increasing their functionality. The connection of this problem with branching predication—an essential feature of the organization of the hardware platform of graphics processors—was established.

A number of existing approaches and technologies for increasing the performance of programs on the CPU were considered and the low degree of their applicability to programs using the GPGPU technology was shown.

It was noted that the existing methods, even though their direct application is impossible, are an important starting point for further research.

Further development of the problems outlined in the article implies the development of a specialized method for optimizing programs for GPU.

Authors' contribution. All authors equally contributed to the research work.

REFERENCES

1. Markidis S., Chien S.W.D., Laure E., Peng I.B., Vetter J.S. NVIDIA Tensor Core Programmability, Performance & Precision. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Vancouver, BC, Canada; 2018, p. 522–531. https://doi.org/10.1109/IPDPSW.2018.00091
2. Sanzharov V.V., Gorbonosov A.I., Frolov V.A., Voloboy A.G. Examination of the Nvidia RTX. *CEUR Workshop Proceedings*. 2019;2485:7–12. http://dx.doi.org/10.30987/graphicon-2019-2-7-12
3. Flynn M.J. Very high speed computing systems. *Proceedings of the IEEE*. 1966;54(12):1901–1909. https://doi.org/10.1109/PROC.1966.5273
4. Fisher J.A., Faraboschi P., Young C. *Embedded computing: A VLIW approach to architecture, compilers, and tools*. Elsevier; 2004. ISBN: 978-1-55860-766-8. URL: https://www.researchgate.net/publication/220690439_Embedded_computing_a_VLIW_approach_to_architecture_compilers_and_tools
5. Knoop J., Rüthing O., Steffen B. Partial dead code elimination. In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI '94)*. 1994, p. 147–158. https://doi.org/10.1145/178243.178256
6. Fink S., Knobe K., Sarkar V. Unified analysis of array and object references in strongly typed languages. In: Palsberg J. (Ed.). *Static Analysis. SAS 2000. Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer. 2000. V. 1824. P. 155–174. https://doi.org/10.1007/978-3-540-45099-3_9
7. Runeson J., Nyström S.-O. Retargetable graph-coloring register allocation for irregular architectures. In: Krall A. (Ed.). *Software and Compilers for Embedded Systems (SCOPES 2003). Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer. 2003. V. 2826. P. 240–254. https://doi.org/10.1007/978-3-540-39920-9_17
8. Blindell G.H. *Instruction Selection: Principles, Methods, & Applications*. Springer; 2016. 171 p. ISBN 978-3-319-34017-3. http://dx.doi.org/10.1007/978-3-319-34019-7
9. Gibbons P.B., Muchnick S.S. Efficient instruction scheduling for a pipelined architecture. *ACM SIGPLAN Notices*. 1986;21(7):11–16. https://doi.org/10.1145/13310.13312
10. Su Ch.-L., Tsui Ch.-Y., Despain A.M. Low power architecture design and compilation techniques for high-performance processors. In: *Proceedings of COMPCON '94*. 1994, p. 489–498. https://doi.org/10.1109/CMPCON.1994.282878
11. Aycock J. A brief history of just-in-time. *ACM Comput. Surv.* 2003;35(2):97–113. https://doi.org/10.1145/857076.857077
12. Ogihara M. *Fundamentals of Java Programming*. Springer; 2018. 532 p.
13. Sage K. The Origins of Programming. In: *Concise Guide to Object-Oriented Programming. Undergraduate Topics in Computer Science*. Springer, Cham.; 2019, p. 1–9. https://doi.org/10.1007/978-3-030-13304-7_1
14. Saabith A.S., Fareez M.M.M., Vinothraj T. Python current trend applications-an overview. *IJAERD*. 2019;6(10):6–12. URL: http://ijaerd.com/papers/finished_papers/IJAERDV06I1085481.pdf
15. McFarling S. *Combining Branch Predictors*. Digital Western Research Lab (WRL). Technical Report, TN-36. 1993. 29 p. URL: https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf

16. Skadron K., Martonosi M., Clark D.W. A Taxonomy of branch mispredictions, and alloyed prediction as a robust solution to wrong-history mispredictions. In: *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*. Philadelphia. 2000. https://doi.org/10.1109/PACT.2000.888344
17. Vintan L.N., Iridon M. Towards a high performance neural branch predictor. In: *IJCNN'99. International Joint Conference on Neural Networks Proceedings*. 1999. https://doi.org/10.1109/IJCNN.1999.831066
18. Kocher P., Horn J., Fogh A., Genkin D., et al. Spectre attacks: Exploiting speculative execution. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. 19 p. https://doi.org/10.1109/SP.2019.00002
19. Bhattacharyya A., Sandulescu A., Neugschwandtner M., Sorniotti A., et al. SMoTherSpectre: exploiting speculative execution through port contention. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. 2019, p. 785–800. https://doi.org/10.1145/3319535.3363194
20. Chen G., Chen S., Xiao Y., Zhang Y., Lin Z., Lai T.H. SgxPectre: Stealing intel secrets from SGX enclaves via speculative execution. In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, p. 142–157. https://doi.org/10.1109/EuroSP.2019.00020
21. Arnold M., Fink S., Grove D., Hind M., Sweeney P.F. Adaptive optimization in the Jalapeno JVM. In: *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2000, p. 47–65. https://doi.org/10.1145/353171.353175
22. Riazanov A. *Implementing an Efficient Theorem Prover*. PhD thesis. The University of Manchester; 2003. 210 p. URL: https://www.researchgate.net/publication/2906405_Implementing_an_Efficient_Theorem_Prover
23. Grant B., Mock M., Philipose M., Chambers C., Eggers S.J. Annotation-directed run-time specialization in C. *ACM SIGPLAN Not.* 1997;32(12):163–178. https://doi.org/10.1145/258994.259016
24. Pettis K., Hansen R.C. Profile guided code positioning. In: *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation (PLDI '90)*. 1990, p. 16–27. https://doi.org/10.1145/93542.93550
25. Wicht B., Vitillo R.A, Chen D., Levinthal D. *Hardware Counted Profile-Guided Optimization*. 24 November 2014. URL: https://arxiv.org/pdf/1411.6361.pdf

About the authors

Igor Yu. Sesin, Postgraduate Student, Department of the Tool and Applied Software, Institute of Information Technologies, MIREA – Russian Technological University (78, Vernadskogo pr., Moscow, 119454 Russia). E-mail: isesin@protonmail.com. https://orcid.org/0000-0002-7323-9595

Roman G. Bolbakov, Cand. Sci. (Eng.), Associate Professor, Head of the Department of the Tool and Applied Software, Institute of Information Technologies, MIREA – Russian Technological University (78, Vernadskogo pr., Moscow, 119454 Russia). E-mail: bolbakov@mirea.ru. https://orcid.org/0000-0002-4922-7260

Об авторах

Сесин Игорь Юрьевич, аспирант, кафедра инструментального и прикладного программного обеспечения Института информационных технологий ФГБОУ ВО «МИРЭА – Российский технологический университет» (119454, Россия, Москва, пр-т Вернадского, д. 78). E-mail: isesin@protonmail.com. https://orcid.org/0000-0002-7323-9595

Болбаков Роман Геннадьевич, к.т.н., доцент, заведующий кафедрой инструментального и прикладного программного обеспечения Института информационных технологий ФГБОУ ВО «МИРЭА – Российский технологический университет» (119454, Россия, Москва, пр-т Вернадского, д. 78). E-mail: bolbakov@mirea.ru. https://orcid.org/0000-0002-4922-7260

Translated by E. Shklovskii