An Approach to Generating Recommendations for Improving the Performance of Software for Heterogeneous Computing Platforms

Artyom V. Gorchakov and Liliya A. Demidova

***Abstract*—The widespread use of heterogeneous computing platforms, as well as the incorporation of computationally expensive implementations of intelligent data analysis algorithms into modern software systems leads to the demand in moving software fragments to most suitable hardware accelerators that are available on a heterogeneous computing platform. In this research, we propose an approach to the generation of recommendations for improving the performance of software systems by finding candidate algorithm implementations for hardware acceleration, and by suggesting the most suitable hardware accelerator among the specialized processors that are available on a given heterogeneous computing platform. The proposed approach is based on a code-to-code search technique, which extracts code fragments from an abstract syntax tree (AST), converts them into vectors containing program features, and compares the vectors with the query program vector. The obtained results confirm that the use of automatically recommended hardware accelerators for the code fragments identified using the proposed approach indeed allows to increase the performance of software systems solving machine learning tasks.**

***Keywords*—program analysis, heterogeneous computing, abstract syntax tree, recommender systems, software systems**

I. Introduction

Innovations in computer architecture made it possible to create heterogeneous computing platforms for solving specialized computational tasks [1]. In a heterogeneous platform, a general-purpose central processing unit (CPU) is complemented by specialized accelerators, such as graphics processing units (GPUs) [2], application-specific integrated circuits (ASICs), field programmable gate arrays (FPGAs) [3].

Modern trends in software development are focused on moving implementations of computationally expensive algorithms to hardware accelerators that are available on a given heterogeneous computing platform with the aim to improve the performance of software by balancing the load on processors of different types [4].

The choice of a specialized processor for hardware acceleration of a code fragment depends on the algorithm implemented by the code fragment. Thus, [5] reports the performance of specialized and general-purpose processors in machine learning tasks, the reported results indicate that the TitanXp GPU allows to speed up the training of a convolutional artificial neural network (ANN) LeNet-5 by 8.8 times compared to the CPU E-1620. At the same time, according to [5], hardware implementation of the trained ANN on the Arria-10 FPGA accelerates the decision-making process by 44.4 times compared to the E-1620 CPU. According to estimates reported in [6], the Xilinx PYNQ-Z1 FPGA makes it possible to speed up the image classification process using the convolutional ANN AlexNet by 64 times when compared to a dual-core ARM Cortex-A9 CPU, and by 1.6 times when compared to a quad-core CPU Intel i5-6400. As it is shown in [5, 6], heterogeneous computing, including the use of a CPU for data preparation, a GPU for training an ANN, and an FPGA for making ANN predictions allows to achieve the best performance of software in intelligent data analysis tasks when using neural network-based algorithms.

During the development of software systems developers use general-purpose programming languages, such as Python, Java, C#, C, JavaScript, while specialized languages are used for programming hardware accelerators. For example, Open Computing Language (OpenCL) [7] or Compute Unified Device Architecture (CUDA) language [8] is used to program GPUs, and hardware description languages (HDL) such as Verilog or Very high-speed integrated circuits Hardware Description Language (VHDL) are used to configure FPGAs [9]. To simplify the transfer of algorithms to FPGAs, tools for high-level synthesis (HLS) of register transfer level (RTL) instructions exist [10]. Such tools allow to automatically convert algorithms implemented in high-level languages, for example, in C or OpenCL, into RTL code for subsequent FPGA configuration.

Moreover, domain-specific programming languages (DSLs) exist, which allow synthesizing high-performance HLS code [11-12]. The tool described in [11] allows to automatically translate a Python-like DSL into Vivado HLS or Intel OpenCL. During the code translation process, a static data flow graph is constructed, after that the graph is rewritten to apply performance optimizations, and then the low-level code is synthesized for the target platforms. In [12], methods and algorithms are described that allow synthesizing machine-dependent optimization rules for the use in DSL compilers for specialized processors.

In [13, 14], the hardware-software partitioning optimiza-

Manuscript received March 10, 2024.

Artyom V. Gorchakov, Postgraduate Student, Assistant of the Department of Corporate Information Systems, Institute of Information Technologies, MIREA – Russian Technological University, 78, Vernadsky pr., Moscow, 119454 (email: worldbeater-dev@yandex.ru).

Liliya A. Demidova, Dr. Sci. (Eng.), Professor, Professor of the Department of Corporate Information Systems, Institute of Information Technologies, MIREA – Russian Technological University, 78, Vernadsky pr., Moscow, 119454 (email: liliya.demidova@rambler.ru).

tion problem was considered, and discrete population-based optimization algorithms were used to solve the problem.

However, in order to formulate and solve the hardware-software partitioning problem, it is necessary to first select code fragments that have the potential for hardware acceleration, and then to translate the selected code fragments into representations suitable for running on specialized hardware. In order to improve the performance of a software system implemented using general-purpose programming languages and operating on a heterogeneous computing platform, software developers currently have to identify fragments of code that are suitable for hardware acceleration, also, the developers have to select the appropriate accelerator. Approaches to software acceleration include both the use of specialized processors, and the use of data parallelism in a general-purpose CPU [12, 13].

Hence, modern research in the field of static software analysis is devoted to the identification of code fragments for their subsequent refactoring with the aim to move computationally expensive algorithm implementations to specialized hardware accelerators that are available on a given heterogeneous computing platform [15, 16]. For example, the methodology proposed in [16] is based on call graph analysis of a program, the methodology is limited to acyclic call graphs and involves the construction of control flow and data flow graphs.

In this research, we propose an approach to the generation of recommendations for improving the performance of software systems by finding candidate algorithm implementations for hardware acceleration, and by suggesting the most suitable hardware accelerator among the specialized processors that are available on a given computing platform. The proposed approach is based on a code-to-code search technique, which extracts code fragments from an AST, converts them into vectors containing program features, and compares the vectors with the query program vector. We convert programs into vectors based on Markov chains [17], as according to the results reported in [17], the use of program vectors that are based on Markov chains constructed for ASTs allows to achieve the best classifier quality in multiclass program classification problems when compared to word2vec [18], code2vec [19], histograms of assembly language instruction opcodes [20] and other methods. Additionally, we propose a new method to program conversion into vectors based on Markov chains constructed for ASTs and for definition-use graphs simultaneously.

In the conducted experimental study, we aimed to find answers to the following research questions (RQs):

RQ1: How the quality of program vector-based representations changes when constructing Markov chains not only for ASTs [17], but also for DU-graphs?

RQ2: How the performance of software changes when applying the proposed approach to generating recommendations for improving the performance of software for a heterogeneous computing platform?

The rest of the paper is structured as follows. Section II describes the proposed approach to generating recommendations for improving the performance of software for heterogeneous computing platforms, as well as the related methods and algorithms. Section III reports the results of the conducted experimental study and provides answers to the RQs. Finally, the conclusion section highlights the direction for future research and presents the discussion regarding the results of the conducted experimental study.

II. Methods and Algorithms

Code-to-code search [21] can be used to identify program fragments that have similar properties as the example program which is used as a search query. Thus, the task of identifying program fragments and generating recommendations for increasing their performance can be reduced to:

- The creation of a database containing: sample programs; ported versions of the programs compatible with accelerators that are available on a given heterogeneous computing platform; acceleration coefficients for the sample programs computed using every available hardware accelerator.

- The code-to-code search execution for every sample program that is stored in the database.

The proposed approach to generating recommendations for improving the performance of software for heterogeneous computing platforms includes the following steps:

Step 1. The creation of a database containing software implementations of algorithms.

Step 2. Translation of the software implementations of algorithms into representations suitable for running on each of the hardware accelerators that are available on a given heterogeneous computing platform.

Step 3. Performance estimation of the software implementations of algorithms on each accelerator that is available on the heterogeneous computing platform.

Step 4. Population of the database with coefficients calculated based on the performance assessments of algorithms on each accelerator that is available on the heterogeneous computing platform.

Step 5. Code-to-code search execution using the software implementations of algorithms contained in the database as search queries.

Step 6. Creation of a report on the acceleration perspectives of the analyzed software system.

The code-to-code search problem, which is solved at the 5-th step of the proposed approach to generating recommendations for improving the performance of software, is given by a pair (G_t, G_q), where $G_t \in \mathbb{G}$ is the AST of the statically analyzed program, $G_q \in \mathbb{G}$ is the AST of the example program which is used as a search query, and $\mathbb{G}$ is the set of all possible ASTs. The solution to the code-to-code search problem is a mapping $g: \mathbb{G} \times \mathbb{G} \to \{\mathbb{G}\}$, which maps the analyzed AST G_t and the AST of the search query G_q into a set of recommended program fragments $P = \{p_1, p_2, \dots, p_k\}$ found in G_t by the search query G_q, where $P \in \{\mathbb{G}\}$, and $\{\mathbb{G}\}$ denotes the set of all AST sets $\mathbb{G}$, $\forall p_i \in P: p_i \in \mathbb{G}$.

If a finite set $A = \{a_1, a_2, \dots, a_n\}$ containing n correct recommendations is known (the set A can be obtained, for example, by manually extracting fragments from a program), then the code-to-code search problem is represented by a triplet (G_t, G_q, A), where $G_t \in \mathbb{G}$, $G_q \in \mathbb{G}$, and $\forall a_i \in A: a_i \in \mathbb{G}$, where $\mathbb{G}$ is the set of all possible ASTs. In this case, the quality of the mapping $g: \mathbb{G} \times \mathbb{G} \to \{\mathbb{G}\}$ can be assessed by comparing the k recommendations from the P set to the manually extracted fragments from the A set. The k recommended fragments $p_1, p_2, \dots, p_k$ can be either true positive (TP) or false positive (FP). Formally, $\text{TP} = |\{p \in P: p \in A\}|$, $\text{FP} = |\{p \in P: p \notin A\}|$. The expected answers that are not included into the P set containing k recommendations are considered false negatives (FN), formally, $\text{FN} = |\{a \in A: a \notin P\}|$. Hence, the quality of the mapping $g: \mathbb{G} \times \mathbb{G} \to \{\mathbb{G}\}$ can be assessed with such metrics as Precision@k, Recall@k, and F_1 Score [22].

The algorithms for searching program fragments that were proposed in [23] allowed to achieve ≈100% Recall@k value and ≈80% F_1 Score value in test code-to-code search problems, so in the current study we first describe the language-agnostic versions of the Python-specific code-to-code search algorithms proposed in [23] as part of the described approach to generating recommendations for improving the performance of software for heterogeneous computing platforms.

As it was shown in [17], program vectors based on first-order Markov chains constructed for ASTs are well-suited for practical applications, and allow achieving high classifier accuracy in code classification problems, so we also use Markov chain-based program vectors in our current study.

Below we describe the related algorithms and provide theoretical estimates of their computational complexity. Algorithm 1 describes the construction of a first-order Markov chain for a given graph-based representation of a program.

Algorithm 1 — Markov chain construction for a graph G.

```
Input: $G = (V, E)$ ▷A graph-based code representation.
 1.  **Define** $p: V \to T$, $p$ maps $v \in V$ to its type $t \in T$.
 2.  $M = \emptyset$.
 3.  $T = \{g(v): v \in V\}$.
 4.  **For each** vertex type $t \in T$ **do**:
 5.      $V_d = \{v_d: (v_s, v_d) \in E \land g(v_s) = t\}$.
 6.      $T_d = \{g(v_d): v_d \in V_d\}$.
 7.      **For each** vertex child type $t_d \in T_d$ **do**:
 8.          $\omega \leftarrow \frac{1}{|V_d|} |\{v_d: v_d \in V_d \land g(v_d) = t_d\}|$.
 9.          $M \leftarrow M \cup \{(t, t_d, \omega)\}$.
10.      **End loop**.
11.  **End loop**.
12.  **Return** $(T, M)$.
```

Algorithm 1 accepts a graph-based representation of code $G = (V, E)$, for example, an AST, which can be constructed with language-specific tools. For the Python language we used the “parse” function for AST construction [24]. An example of a first-order Markov chain (T, M) constructed using Algorithm 1 is shown in Figure 1.

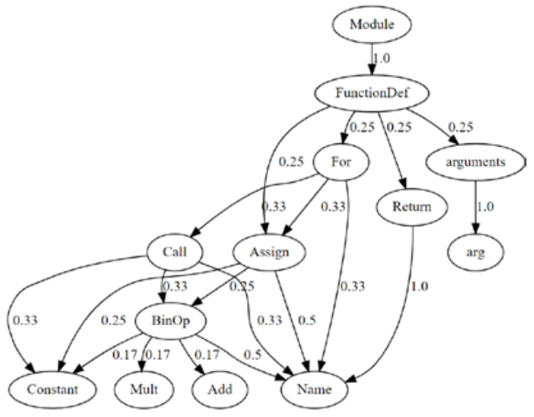


Fig. 1. A Markov chain constructed for an AST of a Python program.

The mapping $p: V \to T$ defined at line 1 of Algorithm 1 was implemented using the “type” function with $O(1)$ time complexity. Algorithm 1 at line 3 constructs the T set containing types of vertices, this takes $O(|V|)$. Next, for each vertex type $t \in T$ the set of child nodes V_d is constructed (see line 5), this takes $O(|E|)$. Then, the set of types of child nodes is constructed (see line 6), this takes $O(|V_d|)$. Finally, the computation of edge weights at line 8 takes $O(|T_d||V_d|)$.

Taking the above estimates into account, Algorithm 1 average time complexity is $O(|V| + |T||E| + |T|bc)$, where b is the average count of child nodes of nodes of the same type (see line 7), c is the average count of types of child nodes of nodes of the same type (see line 8).

Algorithm 2 converts an ordered set of graph-based program representations into an ordered set of vector-based program representations, the vectors are formed by concatenating rows of weighted adjacency matrices of Markov chain state transition graphs constructed using Algorithm 1.

Algorithm 2 — Conversion of graphs into vectors.

```
Input: $\mathbb{G} = (G^1, G^2, \dots, G^r)$. ▷Ordered set of $r$ graphs.
 1.  $H = \emptyset$.
 2.  $R = \emptyset$. ▷Ordered set of edge sets.
 3.  **For each** graph $G^i \in \mathbb{G}$ **do**:
 4.      $(T^i, M^i) \leftarrow$ **Algorithm 1**$(G^i)$. ▷Markov chain.
 5.      $H \leftarrow H \cup T^i$.
 6.      $R \leftarrow R \cup (M^i)$ ▷Add $M^i$ to the end of $R$.
 7.  **End loop**.
 8.  $B = \emptyset$. ▷Ordered set of vectors.
 9.  $h = |H|$.
10.  **For each** set of Markov chain edges $M^i \in R$ **do**:
11.      $G^i_* = (H, M^i)$. ▷Temporary graph.
12.      **Construct** adjacency matrix $\mathbf{p}_i \in \mathbb{R}^{h \times h}$ for $G^i_*$.
13.      **Convert** $\mathbf{p}_i \in \mathbb{R}^{h \times h}$ into $\vec{v}_i \in \mathbb{R}^m$, $m = h^2$.
14.      $B \leftarrow B \cup (\vec{v}_i)$. ▷Add $\vec{v}_i$ to the end of $B$.
15.  **End loop**.
16.  **Return** $B$.
```

Graph-based representations of different programs can contain nodes of different types, node types that are present in one graph might not exist in the other graph. Hence, adjacency matrices of Markov chain state transition graphs might belong to different spaces for different graphs. Algorithm 2 resolves this issue by maintaining the H set containing all node types that occur in graphs from $\mathbb{G}$ (see line 5), and by constructing adjacency matrices for every i-th intermediate graph $G_*^i = (H, M^i)$ containing edges from the i-th Markov chain and vertices from the H set (see line 4 and line 11). The complexity of Algorithm 2 is $O\big(|\mathbb{G}|(|V| + |T||E| + |T|bc + h^2)\big)$, where $\mathbb{G}$ is the input set containing graph-based representations of programs, and h is the count of node types in the H set (see line 9 in Algorithm 2).

Figure 1 shows an AST-based Markov chain constructed by Algorithm 1; the AST-based Markov chain can be converted into a vector by concatenating the rows of the weighted adjacency matrix of the Markov chain state transition graph according to Algorithm 2.

Algorithm 1 and Algorithm 2 are not limited to AST-based program representations. As it was shown in [23] for the Python programming language, Markov chains can be constructed for definition-use graphs (DU-graphs). A DU-graph is a graph-based program representation in which the places in ASTs where variables are defined are connected to the places in ASTs where the defined variables are used. The use of program vectors based on Markov chains constructed for DU-graphs allows a classification algorithm to capture information about data flow in a program [23].

Algorithm 3 describes the language-agnostic construction process of a DU-graph from an AST. Algorithm 3 recursively traverses the AST, starting with AST root node v.

Algorithm 3 — Construction of a DU-graph.

```
Input: $v$ ▷The analyzed AST node.
       $s$ ▷A dictionary linking variables with AST nodes.
 1.  $V = \emptyset$.
 2.  $E = \emptyset$.
 3.  If $v$ is a function definition, do:
 4.      $s \leftarrow \emptyset$. ▷Set $s$ to an empty dictionary.
 5.  End if.
 6.  For each name $\eta$ used by $v$, do:
 7.      $E \leftarrow E \cup \{(s[\eta], v)\}$.
 8.      $V \leftarrow V \cup \{s[\eta]\} \cup \{v\}$.
 9.  End loop.
10.  For each name $\eta$ defined by $v$, do:
11.      If $v$ is an assignment operator, do:
12.          $v \leftarrow v_{\text{rhs}}$, where $v_{\text{rhs}}$ is the assignment source.
13.      End if.
14.      $s[\eta] \leftarrow v$. ▷Add node $v$ into $s$ with key $\eta$.
15.  End loop.
16.  For each node $v_c$, which is a child of $v$, do:
17.      $E \leftarrow E \cup$ Algorithm 3$(v_c, s)$.
18.  End loop.
19.  Return $(V, E)$.
```

Algorithm 3 expects that every AST node v contains references to child nodes v_c, ASTs satisfying such property can be constructed using language-specific tools such as the "parse" function from the standard library [24] for the Python programming language, or the Roslyn compiler [25] for the C# programming language. Average time complexity of Algorithm 3 is $O(|V|\varepsilon)$, where $|V|$ denotes the count of nodes in an AST, and ε is the average count of names that are defined or used by every node in the AST.

An example of an AST augmented with edges belonging to a DU-graph built using Algorithm 3 is shown in Figure 2.

The blue dashed lines in Figure 2 denote DU-graph edges, the black solid lines denote AST edges. The blue vertices denote such vertices that belong to the AST and to the DU-graph at the same time, the black vertices denote such vertices that belong only to the AST.

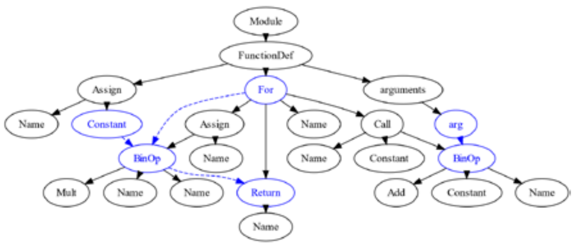


Fig. 2. A DU-graph constructed from an AST of a Python program [24].

An example of a Markov chain constructed using Algorithm 1 for the AST augmented with DU-graph edges (see Figure 2) is shown in Figure 3.

The blue dashed lines in Figure 3 denote the edges of a Markov chain that was constructed for the DU-graph (see blue vertices and edges in Figure 2). The black solid lines in Figure 3 denote the edges of a Markov chain that was constructed for the AST (see black edges, black and blue vertices in Figure 2). The blue vertices in Figure 3 denote such vertices that belong to both Markov chains, the black vertices denote such vertices that belong to a Markov chain constructed only for the AST.

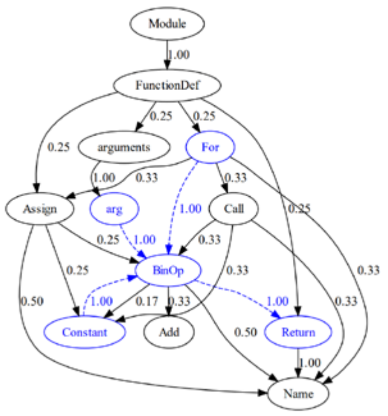


Fig. 3. A Markov chain constructed for an AST and a DU-graph.

Algorithm 2 accepts a set of graph-based program representations and converts the graphs into vector-based representations of programs by constructing Markov chains with Algorithm 1. Algorithm 2 supports arbitrary graphs, such as ASTs or DU-graphs constructed using Algorithm 3.

In a code-to-code search problem it is required to have a method for measuring program similarity. As recommended in [26] and [27], we use Jensen-Shannon divergence (JSD) for measuring the similarity of two Markov chain-based program vectors.

The JSD metric value for two vectors is computed as:

$$\text{JSD}(\vec{v}_i, \vec{v}_h) = \frac{1}{2}\sum_{k=1}^{m} v_{ik} \log_2 \frac{v_{ik}}{\frac{1}{2}(v_{ik}+v_{hk})} + \frac{1}{2}\sum_{k=1}^{m} v_{hk} \log_2 \frac{v_{hk}}{\frac{1}{2}(v_{ik}+v_{hk})}, \quad (1)$$

where $\vec{v}_i$ and $\vec{v}_h$ are the vector-based representations of the compared programs; v_{ik} is the k-th component of $\vec{v}_i$; v_{hk} is the k-th component of $\vec{v}_h$; m denotes the component count in the vector-based representations of programs.

Algorithm 4 defines how Algorithm 2, Algorithm 3, and the JSD metric (1) are used together for measuring similarity of two programs.

Algorithm 4 — Code similarity measurement.

```
Input: $G_1$ ▷AST of the first program.
       $G_2$ ▷AST of the second program.
       $\omega_1 \in [0,1]$ ▷Weight of AST-based distance.
       $\omega_2 \in [0,1]$ ▷Weight of DU-graph based distance.
1. $(\vec{v}_1^{\mathrm{AST}}, \vec{v}_2^{\mathrm{AST}}) \leftarrow$ **Algorithm 2**$(G_1, G_2)$.
2. $G_1^{\mathrm{DU}} \leftarrow$ **Algorithm 3**$(v_1^{\mathrm{root}}, \emptyset)$.
3. $G_2^{\mathrm{DU}} \leftarrow$ **Algorithm 3**$(v_2^{\mathrm{root}}, \emptyset)$.
4. $(\vec{v}_1^{\mathrm{DU}}, \vec{v}_2^{\mathrm{DU}}) \leftarrow$ **Algorithm 2**$(G_1^{\mathrm{DU}}, G_2^{\mathrm{DU}})$.
5. $\rho = \frac{\omega_1}{2}\mathrm{JSD}(\vec{v}_1^{\mathrm{AST}}, \vec{v}_2^{\mathrm{AST}}) + \frac{\omega_2}{2}\mathrm{JSD}(\vec{v}_1^{\mathrm{DU}}, \vec{v}_2^{\mathrm{DU}})$.
6. **Return** the distance between programs $\rho$.
```

First, Algorithm 4 builds program vectors that are based on Markov chains constructed for ASTs (see line 1). Then, DU-graphs are constructed for ASTs G_1 and G_2 (see line 2 and line 3). v_1^{root} on line 2 denotes the root of the AST G_1, and v_2^{root} on line 3 denotes the root of the AST G_2. After building program vectors that are based on DU-graphs (see line 4), Algorithm 4 computes JSD according to (1) for program vectors of different types (see line 5).

Average time complexity of Algorithm 4 depends on the complexities of Algorithm 2 and Algorithm 3, and is estimated as $O(|T||E| + |T|bc + h^2 + |V|\varepsilon)$, where $|T|$ denotes the average count of node types in ASTs G_1 and G_2, $|V|$ is the average count of nodes in ASTs, $|E|$ is the average count of edges in ASTs, h is the average count of node types in the H set (see line 9 in Algorithm 2), ε is the average count of names that are defined or used by every node in ASTs G_1 and G_2, b is the average count of child nodes of nodes of the same type, c is the average count of types of child nodes of nodes of the same type, Algorithm 4 is an auxiliary algorithm which is then used in Algorithm 5.

Algorithm 5 is designed to solve a code-to-code search problem represented by a pair (G_t, G_q), where G_t is the AST of the statically analyzed program, and G_q is the AST of the example program which is used as a search query.

Algorithm 5 — Code-to-code search in an AST.

```
Input: $G_t = (V_t, E_t)$ ▷AST of the analyzed program.
       $G_q$ ▷AST of the search query.
       $\omega_1$ ▷Weight of AST-based distance.
       $\omega_2$ ▷Weight of DU-graph based distance.
       $\vartheta$ ▷Count of statements in AST fragments.
       $\kappa$ ▷Count of recommendations.
1.  $R = \emptyset$.
2.  **For each** node $v \in V_t$ **do**:
3.      $\xi = v_{\mathrm{body}}$. ▷Set $\xi$ to a sequence of statements.
4.      **If** $\xi \neq \emptyset$ **do**:
5.          **For each** combination $(v_1, \dots, v_\vartheta) \in \binom{\xi}{\vartheta}$ **do**:
6.              **Create** AST $G$ with statements $(v_1, \dots, v_\vartheta)$.
7.              $\rho \leftarrow$ **Algorithm 4**$(G, G_q, \omega_1, \omega_2)$.
8.              $R \leftarrow R \cup \{(\rho, G)\}$.
9.          **End loop**.
10.     **End if**.
11. **End loop**.
12. **Return** $\kappa$ pairs with smallest $\rho$ values from $R$.
```

Algorithm 5 visits every node v of the analyzed AST G_t and processes tuples of sequentially executed statements ξ associated with the node v (see line 3). For example, if the node v represents the loop operator (see the “For” AST node in Figure 2), then the tuple of statements ξ contains instructions that are sequentially executed on every iteration of the loop. If v represents the function definition operator (see the “FunctionDef” AST node in Figure 2), then ξ contains instructions that are executed when the control flow of a program enters the function. If the ξ tuple is not empty, Algorithm 5 extracts all possible combinations of ϑ nodes $(v_1, \dots, v_\vartheta)$ from ξ (see line 5 in Algorithm 5).

The total count of combinations extracted from ξ is:

$$C_{|\xi|}^{\vartheta} = \binom{|\xi|}{\vartheta} = \frac{|\xi|!}{\vartheta!(|\xi|-\vartheta)!}, \qquad (2)$$

where ϑ is the count of nodes in the combinations extracted from ξ, and $|\xi|$ is the length of the ξ tuple.

The temporary AST G containing the extracted nodes $(v_1, \dots, v_\vartheta)$ (see line 6) is compared to the AST G_q used as a search query, Algorithm 4 is used for the comparison.

Average time complexity of Algorithm 5 is estimated as $O\left(|V_t|C_{|\xi|}^{\vartheta}(|T||E| + |T|bc + h^2 + |V|\varepsilon)\right)$, where $|V_t|$ is the count of nodes in the AST G_t, $C_{|\xi|}^{\vartheta}$ (2) is the average count of node combinations extracted from ξ, $|T|$, $|E|$, and $|V|$ denote the average count of node types, edges, and nodes in G_q and in the fragments extracted from the AST G_t (see line 7), h is the average count of node types in the H set (see line 9 in Algorithm 2), ε is the average count of names that are defined or used by every node in G_q and in the fragments extracted from the AST G_t, b is the average count of child nodes of nodes of the same type, c is the average count of types of child nodes of nodes of the same type.

III. EXPERIMENTAL EVALUATION

***RQ1**: How the quality of program vector-based representations changes when constructing Markov chains not only for ASTs, but also for DU-graphs?*

Aiming to compare the quality of program vectors based on Markov chains constructed for ASTs (see Algorithm 2), and the quality of program vectors based on Markov chains constructed for ASTs and DU-graphs simultaneously (see Algorithm 3 and Algorithm 2), we considered the publicly available dataset [27] containing 13 881 small Python programs solving unique programming exercises of 11 different types, the programming exercises were generated by the Digital Teaching Assistant (DTA) system used at RTU MIREA [28].

We considered the task classification problem which is a multiclass classification problem. A solution to this problem is a mapping $a: X \to Y$ which maps a program text $x \in X$ to the type of the task $y \in Y$ solved by the program. The set of known task types Y was finite, $Y = \{1,2,\dots,11\}$ [27]. The dataset $\{(x_i, y_i): x_i \in X, y_i \in Y\}$ used for classifier training contained pairs of programs and their task labels. We considered such classifiers as the k-nearest neighbor (KNN) classifier [29], support vector machine (SVM) [30], random forest (RF) [31], and multilayer perceptron (MLP) [32]. Implementations of the classifiers were borrowed from sklearn [33].

In order to compare the quality of program vectors based on Markov chains constructed for ASTs (see Figure 1) and for ASTs and DU-graphs (see Figure 3) we transformed the original training dataset into two different datasets containing program vectors and their labels. After that we trained KNN, SVM, RF and MLP, and assessed the quality of the classifiers using Accuracy, Precision, Recall, and F_1 Score [17]. Classification quality was accessed using 5-fold cross validation.

The obtained results are shown in Table I. Best metric value for every classifier is highlighted in bold.

TABLE I. CLASSIFICATION QUALITY ASSESSMENTS

Vectors	Alg.	Accuracy	Precision	Recall	F_1 Score
AST-based Markov chains	KNN	81.0	78.8	78.2	75.4
	SVM	84.0	78.6	80.0	77.0
	RF	92.0	88.2	89.4	87.8
	MLP	86.0	79.6	82.0	79.5
DU-graph- and AST-based Markov chains	KNN	**84.0**	**81.8**	**80.9**	**79.0**
	SVM	**85.0**	**80.9**	**81.8**	**79.4**
	RF	92.0	88.2	89.4	87.8
	MLP	**89.0**	**85.8**	**85.8**	**83.1**

According to Table 2, the use of program vectors that are based on Markov chains built for ASTs and DU-graphs simultaneously indeed improves the quality of KNN, SVM and MLP-based classifiers when compared to program vectors that are based on Markov chains constructed only for ASTs, and the quality of an RF-based classifier remains unchanged.

In [17], a specialized metric was proposed for measuring the sensitivity of program vectors to the used classifier. The quality of a classifier is usually assessed by using k-fold cross-validation and a specialized classification quality metric (for example, Accuracy, Precision, Recall or F_1 Score). k-fold cross-validation outputs a set $Q_t = \{q_1^t, q_2^t, \dots, q_k^t\}$ containing k quality assessments of the t-th classifier. In order to measure the sensitivity of program vectors to the used classifier [17] proposed the $\bar{\sigma}$ metric:

$$\bar{\sigma}(\mathbb{Q}) = \sqrt{\Sigma_{t=1}^{n}\big(\mu(Q_t) - \bar{\mu}(\mathbb{Q})\big)^2 \frac{\sigma^{-1}(Q_t)}{\Sigma_{i=1}^{n}\sigma^{-1}(Q_i)}}, \tag{3}$$

$$\bar{\mu}(\mathbb{Q}) = \Sigma_{t=1}^{n}\mu(Q_t)\frac{\sigma^{-1}(Q_t)}{\Sigma_{i=1}^{n}\sigma^{-1}(Q_i)}, \tag{4}$$

where $Q_t = \{q_1^t, q_2^t, \dots, q_k^t\}$ is the set containing k quality assessments of the t-th classifier; $\mathbb{Q} = \{Q_1, Q_2, \dots, Q_t, \dots, Q_n\}$; $\mu(Q_t)$ is the mean quality of the t-th classifier; $\sigma(Q_t)$ is the standard deviation; n is the count of classifiers evaluated during the sensitivity assessment of program vectors to the used classifier; $t = \overline{1,n}$, $i = \overline{1,n}$; the less the value of $\bar{\sigma}(\mathbb{Q})$ is, the better program vectors are.

The results of the sensitivity assessment of program vectors based on Markov chains constructed for ASTs (see Algorithm 2), and of program vectors based on Markov chains constructed for ASTs and DU-graphs simultaneously (see Algorithm 3 and Algorithm 2) are shown in Figure 4.

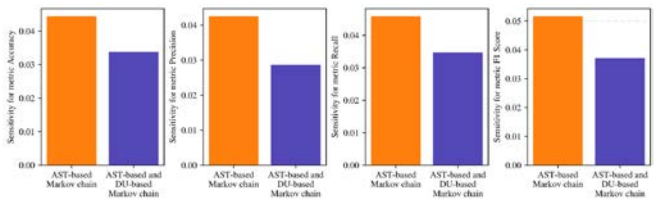


Fig. 4. A Markov chain constructed for an AST and a DU-graph.

As it is shown in Figure 4, program vectors that are based on Markov chains constructed for ASTs and DU-graphs simultaneously (highlighted in blue in Figure 4) are less sensitive to the used classifier in the sense of (3) when compared to program vectors that are based on Markov chains constructed only for ASTs that were studied in [17].

Overall, the results shown in Table 1 and Figure 4 indicate that the quality of program vectors improves when building Markov chains not only for ASTs, but also for DU-graphs.

***RQ2**: How the performance of software changes when applying the proposed approach to generating recommendations for improving the performance of software for a heterogeneous computing platform?*

The approach to generating recommendations for improving the performance of software for heterogeneous computing platforms, which is based on Algorithm 5 performing code-to-code search, was used to generate recommendations for accelerating Python programs implementing:

- Logistic regression training algorithm for binary classification problems;

- Extreme learning machine [34] prediction making algorithm for multiclass classification problems.

The analysis of programs using the proposed approach was carried out with the aim of accelerating them by moving code fragments to specialized accelerators that are available on a heterogeneous computing platform, the characteristics of the computing platform are listed in Table II.

TABLE II. CHARACTERISTICS OF THE COMPUTING PLATFORM

Parameter	Value
CPU	Intel® Core™ i7-4770, 3.40 GHz (4 cores)
GPU	NVIDIA GeForce GTX 1050TI, 1341 MHz, 4 GB GDDR5

At the first step of the proposed approach, a database of Python implementations of simple algorithms was formed, the database included an algorithm for calculating the scalar product of two vectors, an algorithm for transposing a matrix, an algorithm for applying mathematical operations to each element of a vector.

At the second step of the proposed approach, each of the Python implementations of the algorithms was translated into representations suitable for running on each of the specialized accelerators available on the considered heterogeneous computing platform (see Table II). In order to parallelize calculations on the CPU and GPU using the SIMD (Single Instruction, Multiple Data) principle, we used PyTorch DSL [35].

At the third step of the proposed approach, the operating time of algorithm implementations on each of the accelerators available on the computing platform was assessed. When assessing the performance, the algorithms were fed with random vectors belonging to $\mathbb{R}^{1000000}$, the calculations were repeated 2000 times, the average operating time was calculated based on the results of repeated measurements.

At the fourth step of the proposed approach, the database of algorithm implementations was supplemented with coefficients calculated based on performance assessments of algorithms on accelerators available on the heterogeneous computing platform (see Table II). The resulting database of algorithm implementations and acceleration coefficients is presented in Table III.

TABLE III. DATABASE WITH ALGORITHMS AND COEFFICIENTS

Python Implementations of Algorithms	CPU	GPU

out = 0
for i **in** range(len(a)):
    out = out + b[i] * a[i]
	0.43	**2.31**

transpose = []
for i **in** range(len(m[0])):
    t = []
    for j **in** range(len(m)):
        t.append(m[j][i])
        transpose.append(t)
	1.41	0.71

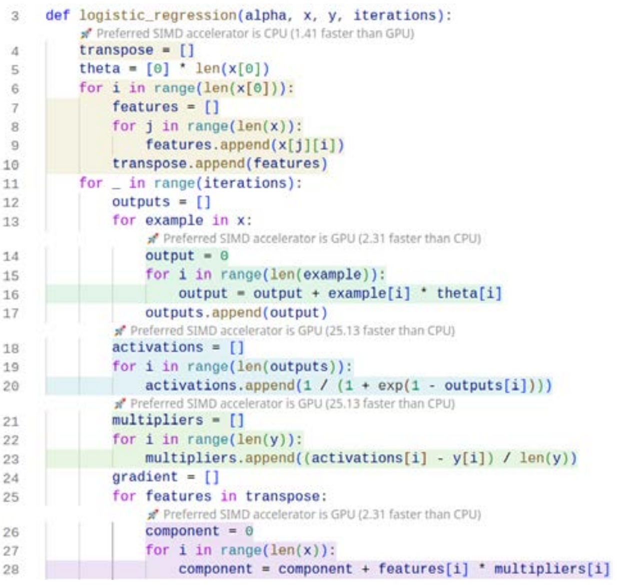
out = []
for i **in** range(len(a)):
    out.append((a[i] + a[i]) / a[i])
	0.04	**25.13**

We developed an extension for the Visual Studio Code® editor that supports the code-to-code search for fragments of programs that are potentially suitable for acceleration, the extension can display the preferred accelerator and show the performance gain coefficient when transferring the identified code fragment to a preferred hardware accelerator compared to other accelerators that are available on a given heterogeneous computing platform (see, for example, Table II).

At the fifth step of the proposed approach, we used the developed Visual Studio Code® extension for performing code-to-code search for each of the algorithm implementations stored in the database (see Table III). Each of the Python programs was used as a search query G_q in Algorithm 5.

At the sixth step of the proposed approach the developed Visual Studio Code® extension generated a report containing recommendations for improving the performance of software for the heterogeneous computing platform (see Table II).

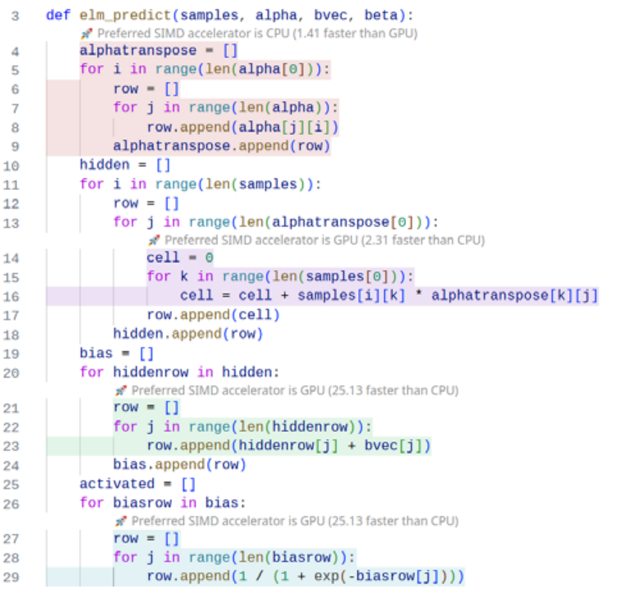
The generated report for the Python implementation of the logistic regression training algorithm for binary classification problems is shown in Figure 5.



```
def logistic_regression(alpha, x, y, iterations):
    Preferred SIMD accelerator is CPU (1.41 faster than GPU)
    transpose = []
    theta = [0] * len(x[0])
    for i in range(len(x[0])):
        features = []
        for j in range(len(x)):
            features.append(x[j][i])
        transpose.append(features)
    for _ in range(iterations):
        outputs = []
        for example in x:
            Preferred SIMD accelerator is GPU (2.31 faster than CPU)
            output = 0
            for i in range(len(example)):
                output = output + example[i] * theta[i]
            outputs.append(output)
        Preferred SIMD accelerator is GPU (25.13 faster than CPU)
        activations = []
        for i in range(len(outputs)):
            activations.append(1 / (1 + exp(1 - outputs[i])))
        Preferred SIMD accelerator is GPU (25.13 faster than CPU)
        multipliers = []
        for i in range(len(y)):
            multipliers.append((activations[i] - y[i]) / len(y))
        gradient = []
        for features in transpose:
            Preferred SIMD accelerator is GPU (2.31 faster than CPU)
            component = 0
            for i in range(len(x)):
                component = component + features[i] * multipliers[i]
```

Fig. 5. The generated report for the Python implementation of the logistic regression training algorithm for binary classification problems.

The generated report for the Python implementation of the Extreme Learning Machine prediction making algorithm for multiclass classification problems is shown in Figure 6.



```
def elm_predict(samples, alpha, bvec, beta):
    Preferred SIMD accelerator is CPU (1.41 faster than GPU)
    alphatranspose = []
    for i in range(len(alpha[0])):
        row = []
        for j in range(len(alpha)):
            row.append(alpha[j][i])
        alphatranspose.append(row)
    hidden = []
    for i in range(len(samples)):
        row = []
        for j in range(len(alphatranspose[0])):
            Preferred SIMD accelerator is GPU (2.31 faster than CPU)
            cell = 0
            for k in range(len(samples[0])):
                cell = cell + samples[i][k] * alphatranspose[k][j]
            row.append(cell)
        hidden.append(row)
    bias = []
    for hiddenrow in hidden:
        Preferred SIMD accelerator is GPU (25.13 faster than CPU)
        row = []
        for j in range(len(hiddenrow)):
            row.append(hiddenrow[j] + bvec[j])
        bias.append(row)
    activated = []
    for biasrow in bias:
        Preferred SIMD accelerator is GPU (25.13 faster than CPU)
        row = []
        for j in range(len(biasrow)):
            row.append(1 / (1 + exp(-biasrow[j])))
```

Fig. 6. The generated report for the Python implementation of the Extreme Learning Machine prediction making algorithm for multiclass classification problems.

The program implementing the logistic regression training algorithm for binary classification problems was supplied with vectors belonging to $\mathbb{R}^{1000000}$, the vectors were taken from a synthetic set of vectors generated using sklearn [33], 200 iterations of the training algorithm were evaluated. The program implementing the algorithm for making predictions by an extreme learning machine was supplied with a set of 10000 vectors belonging to the space $\mathbb{R}^{10000}$, the number of neurons in the hidden layer was set to 1000. Performance measurements were repeated 50 times for each of the 3 parallelization options, the first option was based on homogeneous SIMD using only CPU, the second option was based on homogeneous SIMD using only GPU, the third option was based on heterogeneous computations, where the preferred accelerator was selected according to the recommendations (see Figure 5 and Figure 6). Code parts where the preferred accelerator is GPU were merged and translated into PyTorch DSL [35] with CUDA [8] support, code parts where the preferred accelerator is CPU were translated into PyTorch DSL with support for CPU SIMD intrinsics.

The obtained results are shown in Table IV.

TABLE IV. PERFORMANCE OF DIFFERENT PARALLELIZATION OPTIONS

Algorithm Implementation	**CPU Only**	**GPU Only**	**CPU+GPU**
Logistic regression training algorithm for binary classification	20.91	4.17	**4.00**
Extreme learning machine prediction making algorithm for multiclass classification	6.72	1.58	**1.34**

According to Table IV, following the recommendations for code refactoring that were generated according to the proposed approach (see Figure 5) allowed to achieve the best performance of the Python implementation of the logistic regression training algorithm for binary classification. Program performance improves by 5.23 times when compared to the CPU-only SIMD parallelization, and improves by 1.04 times when compared to the GPU-only SIMD parallelization on the considered heterogeneous computing platform (see Table II).

Moreover, according to Table IV, following the recommendations for code refactoring that were generated according to the proposed approach (see Figure 6) allows to achieve the best performance of the Python implementation of the extreme learning machine prediction making algorithm for multiclass classification. Program performance improves by 5.01 times when compared to the CPU-only SIMD parallelization, and by 1.18 times when compared to the GPU-only SIMD parallelization.

IV. CONCLUSION

In the presented research we proposed an approach to generating recommendations for improving the performance of software for heterogeneous computing platforms, which is based on a code-to-code search algorithm. The obtained results indicate that reworking programs implementing intelligent data analysis algorithms according to the generated recommendations allows to improve the performance of software on a given heterogeneous computing platform.

In addition, we described language-agnostic algorithms for constructing program vectors that are based on Markov chains (see Algorithm 1 and Algorithm 2). The Markov chains were constructed for ASTs and DU-graphs (see Algorithm 3). We also presented Algorithm 4 which allows comparing programs based on ASTs and DU-graphs.

Experimental results show that program representations based on Markov chains constructed for ASTs and DU-graphs are less sensitive to the used classifier when compared to AST-based Markov chains proposed in [17] (see Figure 4), and allow to increase the quality of program classifiers in the considered task classification problem (see Table I) [27].

Future work could cover the incorporation of language-agnostic ASTs into the proposed algorithms. Currently, adding support for a new language requires the re-implementation of the listed algorithms to use the application programming interfaces of the compiler of the new language, and language-agnostic ASTs such as [36] could simplify the implementation of source code analyzers for different languages.

ACKNOWLEDGMENT

The research was supported by the Ministry of Science and Higher Education of the Russian Federation (Project No. FSFZ-2024-0023).

REFERENCES

[1] V.N. Glinskikh, A.R. Dudaev, O.V. Nechaev, "High-performance CPU – GPU heterogeneous computing in resistivity logging of oil and gas wells," in *Vychislitelnye Tekhnologii*, 2017, Vol. 22, No. 3, pp. 16–31.

[2] D.I. Mirzoyan, "Osnovnye aspekty primeneniya GPGPU sistem," in *Modern Information Technologies and IT-Education*, 2011, No. 7, pp. 988–994.

[3] E. Andrianova, P. Sovietov, I. Tarasov, "Hardware acceleration of statistical data processing based on FPGAs in corporate information systems," in Proceedings of the 2020 2nd International Conference on Control Systems, Mathematical Modeling, Automation and Energy Efficiency (SUMMA), IEEE 2020, pp. 669–671.

[4] S. Mittal, J.S. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," in *ACM Computing Surveys* (CSUR), 2015, Vol. 47, No. 4, pp. 1–35.

[5] X. Liu, H.A. Ounifi, A. Gherbi, Y. Lemieux, W. Li, "A Hybrid GPU-FPGA-based Computing Platform for Machine Learning," in *Procedia Computer Science*, 2018, Vol. 141, pp. 104–111.

[6] F. Al-Ali, T.D. Gamage, H.W. Nanayakkara, F. Mehdipour, S.K. Ray, "Novel casestudy and benchmarking of AlexNet for edge AI: From CPU and GPU to FPGA," in Proceedings of the 2020 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), IEEE, 2020, pp. 1–4.

[7] J.E. Stone, D. Gohara, G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," in *Computing in science & engineering*, 2010, Vol. 12, № 3, p. 66.

[8] S. Cook, CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[9] G. Gannot, M. Ligthart, "Verilog HDL based FPGA design," in Proceedings of the International Verilog HDL Conference, IEEE, 1994, pp. 86–92.

[10] W. Meeus, K. Van Beeck, T. Goedeme, J. Meel, D. Stroobandt, "An overview of today's high-level synthesis tools," in *Design Automation for Embedded Systems*, 2012, Vol. 16, pp. 31–51.

[11] J.F. Licht, T.D. Matteis, T. Ben-Nun, A. Kuster, O. Rausch, M. Burger, C.J. Johnsen, T. Hoefler, "Python FPGA Programming with Data-Centric Multi-Level Design," in *arXiv*, 2022, arXiv:2212.13768.

[12] P.N. Sovetov, "Development of DSL Compilers for Specialized Processors," in *Programming and Computer Software*, 2021, Vol. 47, No. 7, pp. 541–554.

[13] D. Saha, R.S. Mitra, A. Basu, "Hardware software partitioning using genetic algorithm," in Proceedings of the Tenth International Conference on VLSI Design, IEEE, 1997, pp. 155–160.

[14] P. Arató, S. Juhász, Z.A. Mann, A. Orban, D. Papp, "Hardware-software partitioning in embedded system design," in Proceedings of the IEEE International Symposium on Intelligent Signal Processing, IEEE, 2003, pp. 197–202.

[15] P. Mokri, M. Hempstead, "Early-stage automated accelerator identification tool for embedded systems with limited area," in Proceedings of the 39th International Conference on Computer-Aided Design, 2020, p. 115.

[16] G. Zacharopoulos, L. Ferretti, G. Ansaloni, G. Di Guglielmo, L. Carloni and L. Pozzi, "Compiler-assisted selection of hardware acceleration candidates from application source code," in Proceedings of the 2019 IEEE 37th International Conference on Computer Design (ICCD), IEEE, 2019, pp. 129–137.

[17] A.V. Gorchakov, L.A. Demidova, P.N. Sovietov, "Analysis of Program Representations Based on Abstract Syntax Trees and Higher-Order Markov Chains for Source Code Classification Task," in *Future Internet*, 2023, Vol. 15, № 9, p. 314.

[18] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, J. Dean, “Distributed Representations of Words and Phrases and their Compositionality,” in *Advances in Neural Information Processing Systems*, 2013, T. 26, pp. 3111–3119.

[19] U. Alon, M. Zilberstein, O. Levy, E. Yahav, “code2vec: Learning distributed representations of code,” in Proceedings of the ACM on Programming Languages, Association for Computing Machinery, 2019, Vol. 3, No. 40, pp. 1–29.

[20] A.F. Da Silva, E. Borin, F.M.Q. Pereira, N.L. Queiroz, O.O. Napoli, “Program Representations for Predictive Compilation: State of Affairs in the Early 20’s,” in *Journal of Computer Languages*, 2022, Vol. 73, p. 101171.

[21] Y. Yu, Z. Huang, G. Shen, W. Li, Y. Shao, “ASTENS-BWA: Searching partial syntactic similar regions between source code fragments via AST-based encoded sequence alignment,” in *Science of Computer Programming*, 2022, No. 222, p. 102839.

[22] T. Silveira, M. Zhang, X. Lin, Y. Liu, S. Ma, “How good your recommender system is? A survey on evaluations in recommendation,” in *International Journal of Machine Learning and Cybernetics*, 2019, Vol. 10, pp. 813–831.

[23] A.V. Gorchakov, “Methods and Algorithms for Identifying Program Fragments for Making Recommendations with the Aim to Increase the Speed of Software Systems,” in *Vestnik of Ryazan State Radio Engineering University*, 2023, Vol. 86, pp. 96–109.

[24] Python Software Foundation. AST — Abstract Syntax Trees, Python Documentation. URL: https://docs.python.org/3/library/ast.html (accessed at 20.02.2024)

[25] S.M. Staroletov, A.V. Dubko, “A method to verify parallel and distributed software in C# by doing Roslyn AST transformation to a Promela model,” in *System Informatics*, 2019, No. 15, pp. 13–44.

[26] A.V. Gorchakov, L.A. Demidova, “Intelligent Accounting of Educational Achievements in the Digital Teaching Assistant System,” in *International Journal of Open Information Technologies*, 2023, Vol. 11, No. 4, pp. 106–115.

[27] L.A. Demidova, E.G. Andrianova, P.N. Sovietov, A.V. Gorchakov, “Dataset of Program Source Codes Solving Unique Programming Exercises Generated by Digital Teaching Assistant,” in *Data*, 2023, Vol. 8, No. 6, p. 109.

[28] P.N. Sovietov, A.V. Gorchakov, “Digital Teaching Assistant for the Python Programming Course,” in Proceedings of the 2022 2nd International Conference on Technology Enhanced Learning in Higher Education (TELE), IEEE, 2022, pp. 272–276.

[29] E. Fix, J. Hodges, “Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties,” in *International Statistical Review*, 1989, 57, pp. 238–247.

[30] C. Cortes, V. Vapnik, “Support-vector networks,” in *Machine Learning*, 1995, Vol. 20, pp. 273–297.

[31] T.K. Ho, “Random Decision Forests,” in Proceedings of the 3rd International Conference on Document Analysis and Recognition, Montreal, QC, Canada, 14–16 August 1995; IEEE: Piscataway, NJ, USA, 1995; pp. 278–282.

[32] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain,” in *Psychological Review*, 1958, Vol. 65, pp. 386–408.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, “Scikit-learn: Machine learning in Python,” in *Journal of Machine Learning Research*, 2011, Vol. 12, pp. 2825–2830.

[34] G.B. Huang, Q.Y. Zhu, C.K. Siew, “Extreme learning machine: theory and applications,” in *Neurocomputing*, 2006, Vol. 70, No. 1–3, pp. 489–501.

[35] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimenshein et al. “PyTorch: An Imperative Style, High-performance Deep Learning Library,” in *Advances in Neural Information Processing Systems*, 2019, Vol. 32.

[36] J. Curtis, “On language-agnostic abstract-syntax trees: student research abstract,” in Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, 2022, pp. 1619–1625.

Artyom V. Gorchakov, Postgraduate Student, Department of Corporate Information Systems, Institute of Information Technologies, MIREA – Russian Technological University (78, Vernadsky pr., Moscow, 119454). worldbeater-dev@yandex.ru. https://orcid.org/0000-0003-1977-8165

Liliya A. Demidova, Dr. Sci. (Eng.), Professor, Professor of the Department of Corporate Information Systems, Institute of Information Technologies, MIREA – Russian Technological University (78, Vernadsky pr., Moscow, 119454). liliya.demidova@rambler.ru. https://orcid.org/0000-0003-4516-3746