System-Call-Level Dynamic Analysis for Code Translation Candidate Selection

Narumi Yoneda, Ryo Hatano and Hiroyuki Nishiyama
Department of Industrial and Systems Engineering, Graduate School of Science and Technology, Tokyo University of Science, 2641 Yamazaki, Noda, Chiba, Japan

Keywords: Dynamic Analysis, System-Call Sequence, Code Translation, Natural Language Processing.

Abstract: In this study, we propose a methodology that uses dynamic analysis (DA) data to select better code-translation candidates. For the DA data, we recorded the history of system-call invocations to understand the actions of the program during execution, providing insights independent of the programming language. We implemented and publicized a DA system, which enabled a fully automated analysis. In our method, we generated multiple translation candidates for programming languages using TransCoder. Subsequently, we performed DA on all the generated candidates and original code. For optimal selection, we compared the DA data of the original code with the generated data and calculated the similarity. To compare the DA data, we used natural language processing techniques on DA data to fix the sequence length. We also attempted to directly compare the variable-length system-call sequences. In this study, we demonstrated that the characteristics of system-call invocations vary even within the same code. For instance, the order of invocations and the number of times the same system-calls an invocation differ. We discuss the elimination of these uncertainties when comparing system-calls.

1 INTRODUCTION

Legacy migration has recently emerged as a global issue. This is the process of transitioning from old systems, known as legacy systems, which have been in place for an extended period and have decreased scalability and maintainability, to new systems. For instance, the case study of the Commonwealth Bank of Australia on legacy migration revealed that it spent $750 million in five years transitioning its banking system platform from COBOL to Java. This implies that legacy migration can be costly in terms of both financial and temporal resources.

Translating programming languages is an important task in legacy migration. Meta developed an artificial intelligence (AI) system called TransCoder (Roziere et al., 2020; Szafraniec et al., 2023), a self-supervised neural transcompiler system that automatically translates functions from one programming language to another to reduce the costs associated with legacy migration. However, various issues are associated with the code generated by these transcompilers. For example, the generated code may contain errors and thus cannot be executed. In addition, even if the code is executable, it may not perform the same operations or yield the same execution results as the original code. To address these issues, Meta suggested three strategies. First, they defined "computational accuracy" to evaluate the generated functions. This metric is defined as the ratio of the generated function to the reference function returning the same output when given the same arguments, because it is significant that the translated code should work properly. Second, they implemented a helpful function called "Beam Search Decoding" which generates multiple translation candidates in TransCoder. This function was implemented to increase the probability that the generated code contained no errors and that the operations and execution results were consistent before and after translation. Finally, they proposed using an intermediate representation (IR) in the learning process to teach source code semantics. The IR holds information about the content that the program executes and is independent of language and machine.

When selecting the best candidate(s) generated using "Beam Search Decoding", the selection process must be investigated and improved from several perspectives. In this study, in addition to executing unit tests, we conduct a system-call-level dynamic anal-

Yoneda, N., Hatano, R. and Nishiyama, H.
System-Call-Level Dynamic Analysis for Code Translation Candidate Selection.
DOI: 10.5220/0012372900003636
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 16th International Conference on Agents and Artificial Intelligence (ICAART 2024) - Volume 3*, pages 576-583
ISBN: 978-989-758-680-4; ISSN: 2184-433X
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

ysis (DA) on both the original code and generated translation candidates and use the resultant DA data to select better code translation candidates. We aimed to develop and validate a new approach for using DA data in programming language translation.

The remainder of this paper is organized as follows: Section 2 introduces relevant studies. The historical development of TransCoder, relevant works, the differences between their approach and ours are discussed. Section 3 introduces the concept of system-call-level DA, presents an overview of the proposed method, and explains how the methodology was evaluated using known metrics for sequence-to-sequence comparisons. Section 4 explains our experimental environment, and Section 5 presents the results of the experiment with a discussion, including the characteristics of the DA data and how they should be used.

2 RELATED WORK

2.1 Translation of Programming Languages

(Roziere et al., 2020) developed a self-supervised deep learning method for programming language translation known as TransCoder. TransCoder can translate functions between the Python, Java, and C++ programming languages. As part of TransCoder, they also implemented Beam Search Decoding, which generates multiple translation candidates by allowing small differences in the generated programs, such as changes in return types or variable types between Java and C++, or operator modifications in Python. In addition, they defined a new metric called "computational accuracy" for evaluating the generated programs. This metric was proposed to address the problem of traditional metrics for the machine translation of natural languages, such as the BLEU score (Papineni et al., 2002), which does not consider syntactic accuracy.

(Szafraniec et al., 2023) proposed a method to enhance code translation using the intermediate representation (IR). They employed a low-level virtual machine intermediate representation (LLVM IR). By leveraging LLVM IR, the probability of making mistakes in variable types and operators decreased.

2.2 Position of Present Work

First, (Roziere et al., 2020; Szafraniec et al., 2023) investigated models for programming language translation using the literal text of program datasets as training data. However, we not only investigated models for programming language translation but also new methods for using system-call-level DA data of translated programs using such models.

Second, (Szafraniec et al., 2023) adopted an approach similar to ours. The IR represents what the source code will do when executed, whereas the system-call sequence (DA data) reflects what the source code did during execution. Therefore, both IR and DA data contain information regarding the semantics of the source code. However, an approach involving compiler-level IR, such as LLVM IR , may restrict the scope of applicable programming languages; it can only be applied to programming languages that use a compiler, such as C++ and Java. Therefore, this approach cannot handle scripting languages that use interpreters to execute instructions. Furthermore, because the generated IR is dependent on the compiler's language-specific front end, there might exist some variations in the generated IR, such as dialects in natural languages. To overcome this issue, we propose an approach to use DA data, which is independent of the programming language or file type and may be helpful in improving the quality of code translations for further development.

3 METHODS

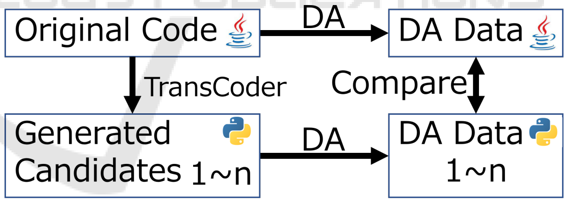

Figure 1: Overview of our Method.

Figure 1 shows an overview of the proposed approach. We used TransCoder to generate multiple translation candidates from Python to Java and vice versa. "Beam Search Decoding" was used to generate 100 translation candidates for each test. Next, system-call-level DA was performed on both the original program and generated translation candidates to obtain the DA data. Subsequently, DA data of the original program were compared with those of the generated translation candidates, and their similarities were calculated. Finally, the results were compared, and the characteristics of the DA data were investigated. In the following sections, we explain each of these processes in detail.

3.1 Multiple Translation Candidates Generation by TransCoder

In this study, we use the "original" TransCoder (Roziere et al., 2020), instead of the latest version (Szafraniec et al., 2023), because we want to stick to the original implementation for simplicity. We obtained a dataset of paired functions with the same functionality in Java and Python (717 entries for Java and 702 entries for Python) from the official GitHub repository of the original implementation provided by (Roziere et al., 2020). In this study, we used 615 entries from the dataset in which the function names in Java and Python matched and there were no errors in the tests.

3.2 System-Call-Level DA

DA is a method for analyzing the behavior of a program during its execution. In this study, we employed DA to capture data with respect to program behavior during execution, which should be independent of the programming language or file type. We captured the history (or time-series data) of the system-calls of a process and its child processes invoked by an executed program as DA data. The time-series data of system-calls invoked by relevant process IDs (PIDs) were serialized as single time-series DA data instead of being segmented by PID.

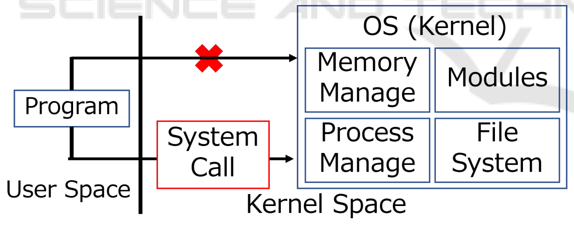

Figure 2: Overview of System-Call.

A system-call is the only application program interface (API) through which programs in the user space can use the functionalities provided by an operating system (OS) (Figure 2). User space is a memory region that is not occupied by the core processes of an OS. Numerous types of end-user programs can be executed in user spaces such as office applications, web browsers, scientific experimental programs written in Python, and network programs written in Java. Such programs in the user spaces execute instructions by leveraging the functionalities provided by the OS. In this study, we used Debian Linux, which has 388 system-calls available, such as "open" to open files and "write" to write files, as listed in `arch/x86/entry/syscalls/syscall_64.tbl` in the kernel source code. Thus, it can be stated that the history of system-call invocations provides information on the OS functionalities of the analysis program used during its execution.

We employed a Debian Linux OS with a modified Linux kernel and a Virtual Box-based virtualization environment to capture the history of system-call invocations. The modified kernel can transmit system-call IDs and process IDs (PIDs), which initiate system-calls using UDP communication in the `do_syscall_64` and `do_int80_syscall_32` functions responsible for handling system-calls. In addition, we revised the `kernel_clone` function, which generates processes for sending both the PID and parent PID information. This modification enables the tracking of all the PIDs related to the program under analysis through their parent-child relationships.

Our DA system can capture all initiated system-calls in real time, regardless of their PID, while maintaining temporal order. For DA, we executed all target programs in separate processes, filtering only those with an exit status of zero, which indicated that the program had been halted without error for data collection. Furthermore, we implemented a fully automated system for DA of multiple files. The system was implemented to perform DA using numerous programs. In this study, we analyzed more than 500 million files. Thus, we required a system that could operate the desired analysis tasks properly for one month, continuously and automatically. Our system is currently available at the GitHub repository (Yoneda, 2023).

3.3 Comparison of DA Data

Table 1: Comparison approaches for DA Data. Here, the "Fix method" is assigned "N/A" if the sequence length is not fixed and "Doc2Vec (Bow; Bag of Words)" if the sequence length is fixed by Doc2Vec (Bow).

Fix method	Metric	Abbreviation
N/A	Jaccard Coefficient	jaccard
N/A	Dice Coefficient	dice
N/A	Overlap Coefficient	overlap
N/A	Levenshtein Distance	levenshtein
Doc2Vec	Cosine Similarity	cossim_d2v
Doc2Vec	Euclidian Distance	euclid_d2v
BoW	Cosine Similarity	cossim_bow
BoW	Euclidian Distance	euclid_bow

We emphasize that DA data are merely a sequence (or history) of system-calls and can be compared using known similarity metrics in natural language processing (NLP). To compare the sequences appropriately, we used two approaches:

Table 2: Experimental Conditions: We used Google Colaboratory to generate translation with TransCoder. We used Local PC for every culculation task including DA. "Python (on Windows)" was used to build a fully automated DA system. It controlled a virtual environment, recieved UDP packets sent from the kernel, and saved them as a CSV file.

Name	Configuration
Google Colaboratory	T4GPU w/ High Memory
Local PC	Core i9-13900k CPU w/ RTX4090 GPU, Windows 11 Pro
Virtual OS	Debian Linux ver. 11.7 w/ modified Kernel ver. 5.10.0-25-amd64
Virtualization software	Virtual Box ver. 7.0.10 r158379
Python (on Windows)	Python ver. 3.11.5
Python (on Virtual OS)	Python ver. 3.9.2
Java (on Virtual OS)	Oracle JDK ver. 11.0.20

(1) The variable-length sequences were compared using designated coefficients or distance metrics (rows 1–4 in Table 1).

(2) The vector similarity or distance was calculated using fixed-length vectors converted from DA data by applying well-studied methods in NLP (rows 5–8 in Table 1).

The key idea behind using Approach (1) is that we can regard a system-call as a character in a letter. This enables us to consider a sequence of system-calls as a string or an ordered set over the alphabet. Hence, we can evaluate the distance or coefficient in the DA data. We also consider treating a system-call as a word in a sentence, which is the key idea in Approach (2). From this perspective, NLP techniques can be used to transform sequences into vectors. We can then use the similarity between the vectors to compare the DA data.

In this study, we used Word2Vec to transform the DA data into vectors. Therefore, a unique Word2Vec model (scsq2vec) that is trained on sequences of system-calls (DA data, sentences of system-calls) must be constructed. We extracted Python and Java source codes from the CodeNet dataset (Puri et al., 2021) and performed DA to obtain the DA data. We then trained the scsq2vec model using the obtained data.

CodeNet is a large-scale dataset of sample programs created for teaching programming to AI, obtained from Atcoder or the AIZU online judge. Most sample programs involve numerical computation and string manipulation. Using source code from CodeNet is suitable for our purposes because the functions used for our experiment also involve numerical computations and string manipulations.

4 EXPERIMENT

The experimental conditions are listed in Table 2. The programs used in our experiment are available in the GitHub repository (Yoneda, 2023).

The outline of our experiment follows the flow illustrated in Figure 1 in Section 3. To perform DA on a function of a certain program, we must provide appropriate arguments and invoke that function. In this study, we used the first element from the list of arguments (Roziere et al., 2020), which was used to calculate the computational accuracy of TransCoder.

Furthermore, to conduct the DA experiment, we additionally followed the following steps:

Python Programs. We added the declaration `import numpy as np` to the original and generated Python source codes. This is because TransCoder sometimes uses a `NumPy` module, instead of a `math` module for generated functions.

Java Programs. We removed the lines containing `import javafx.*` from the original and generated Java source codes. This is because it failed to set up a DA environment to use the `JavaFX` module. Fortunately, every testing program for this experiment was successfully executed without requiring declaration.

5 RESULTS AND DISCUSSION

5.1 Overview of Captured DA Data

Recall that the term "original source codes" refers to the pre-translation functions input into TransCoder (Roziere et al., 2020), while "generated source codes" refers to the translation candidates generated by TransCoder. In this section, we provide an overview of the DA data obtained from the execution of the original source code, generated source code, and source code from CodeNet. Because TransCoder translates a source code at a function level, we will merely refer to the source code of a function as "source code" in the context of the translation. Here, we show the number of DA data and the average sequence length. DA data are a sequence of system-calls, and the average sequence length is the average number of system-call invocations for executing each source code.

Table 3: DA Data of Original Source Codes (Roziere et al., 2020).

Language	# of Obtained Data	Average Length
Python	615	4,959.2
Java	615	2,469.5
Total	1,230	3,714.4

Table 4: DA Data of Generated Source Codes.

Language	# of Obtained Data	Average Length
Python	27,760	4,821.8
Java	20,080	2,457.6
Total	47,840	3,829.4

Table 5: DA Data of Source Codes from CodeNet (Puri et al., 2021).

Language	# of Obtained Data	Average Length
Python	1,925,718	685.9
Java	427,832	2,541.4
Total	2,353,550	1,023.2

Tables 3, 4 and 5 show the DA data of the original source codes, generated source codes, and source codes from CodeNet, where "# of Obtained Data" denotes the total number of source codes that were successfully executed in our system (in other words, the number of source codes whose exit status become zero). "# of Obtained Data" of original source codes (Table 3), for both Python and Java are marked as "615", which means all 615 functions obtained from official TransCoder repository (Roziere et al., 2020) were executed successfully. We generated 100 candidates from each of the 615 functions; thus, 61,500 functions were executed in each language. "# of Obtained Data" of generated source codes (Table 4) shows the number of functions that were executed successfully (those with an exit code of zero).

From the tables, we conclude that the average Java sequence lengths were almost identical among the three datasets. In addition, the average sequence lengths in Python were almost identical for both the original and generated source codes. However, there were considerable differences between these codes and the source code from CodeNet. The reason for this is whether to import a `NumPy` module into the Python code. Every original and generated source code imports it, whereas the source code from CodeNet rarely imports it. Although it is just a single line stating the "import numpy as np", it executes the initialization processes for `NumPy` modules, leading to the potential for a certain amount of computational activity (or invocations of system-calls). Therefore, we split the CodeNet source codes based on whether they imported `NumPy`.

Table 6: DA data for Python files in CodeNet, divided by whether or not importing NumPy.

Import NumPy?	# of Obtained Data	Average Length
YES	35,590	4151.1
NO	1,890,128	620.7
Total	1,925,718	685.9

Table 6 presents the differences in the DA data for importing `NumPy`. The average sequence length of the execution of the NumPy-imported codes in CodeNet was over 4,000, which was close to that of the original and generated codes (cf. Tables 3 and 4). However, the average sequence length of the non-imported code was approximately 620, which is much smaller than that of the NumPy-imported code. We also performed DA on the Python code with only one line of `import numpy as np` 200 times to determine the total number of system-calls invoked for the `NumPy` initialization process. Figure 3 presents a bar graph with the sequence length on the vertical axis and the order of the trial counts on the horizontal axis. The average, maximum, and minimum sequence lengths were 4845.3, 9378, and 742, respectively.

From these results, we obtained the following insights:

1) The DA data of the original and generated source codes, which appeared similar, showed significant differences in the average sequence length of the Python codes compared with that of CodeNet. One possible reason for this discrepancy is the importation of `NumPy` (By Tables3, 4, 5, and 6).
2) The system-calls invoked for the `NumPy` initialization process accounted for a large part of the DA data for both the original and generated Python codes. According to Table 6, approximately 85% of the data were related to the initialization process of `NumPy`.
3) We revealed that the DA data for the same code, such as a code with only one line of `import numpy as np` can have a significant variation in its sequence length. For example, Figure 3 shows that the maximum length was 9,378, whereas the minimum was 742, with a difference of more than 8,000. In addition, there is no correlation between the number of executions and the sequence length; that is, the length may be shortened or lengthened almost randomly.

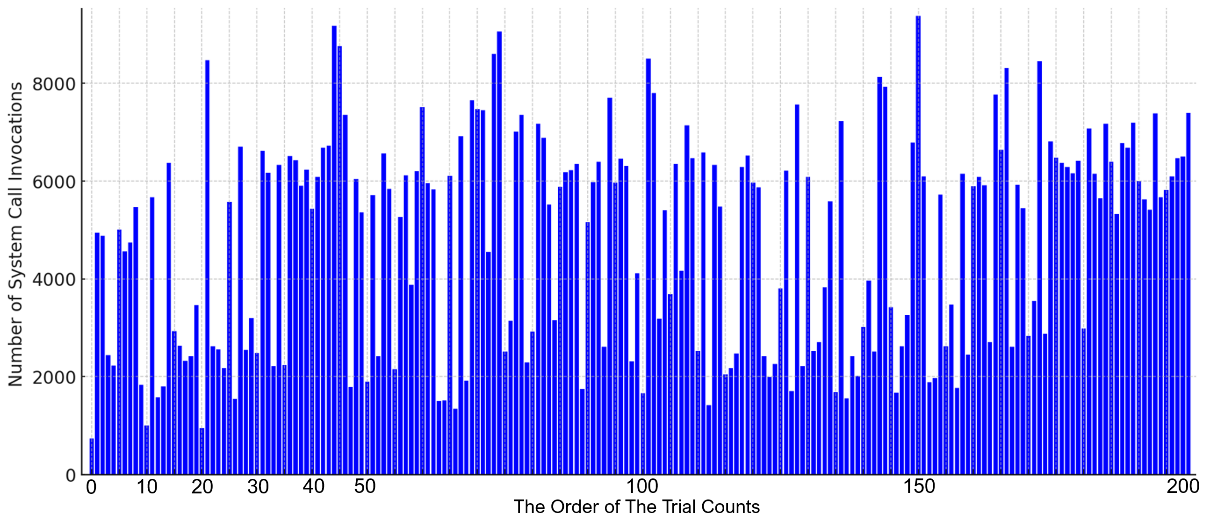

Figure 3: DA Data of `import numpy as np` in Python, 200 Times.

5.2 Similarity of DA Data

This section presents the results of evaluating the similarity among DA data from the executions of Python and Java source codes, and discusses why we obtained the results and how we can improve our approach.

Table 7: Average Similarity of DA Data.

Evaluation Metric	py2j	j2py
jaccard	0.538	0.538
dice	0.696	0.696
overlap	0.760	0.760
levenshtein	4818.8	4687.8
cossim_d2v	0.378	0.384
cossim_bow	0.083	0.089
euclid_d2v	3.851	3.836
euclid_bow	2801.9	2671.1

Table 7 presents the similarity between the DA data of the original code and generated translation candidates. Here, “py2j” (“j2py”) is the average of the similarity between the DA data of the original Python (Java) code and that of the generated Java (Python) translation candidate codes.

5.2.1 Evaluation of the scsq2vec Model

We created a Doc2Vec model that learns system-call sequences (scsq2vec) to relax differences in how different program languages invoke system-calls. This approach addresses the issue that the similarity between system-call sequences does not reflect the similarity between the contents of their original source codes. We attempted to reduce such differences by training a Doc2vec model on a large dataset of system-call sequences created by performing DA on sample source code from CodeNet. However, as is evident from the cossim_d2v metric, which is calculated by the cosine similarity with vectorized DA data using scsq2vec, scsq2vec may distinguish between the DA data of Java and Python. A possible reason for this is the significant difference in the sequence length between the DA data from Python and Java. The scsq2vec model may have learned to distinguish between Python and Java based on their sequence lengths, because we input these sequences without any preprocessing. In addition, most of the Python DA data were related to NumPy’s initialization process, as mentioned in the previous section. In other words, the ratio of system-call sequences that may be unrelated to `NumPy` and specific operations in the Python files is expected to be low. This factor accounts for the distinction of the scsq2vec model between Java and Python.

5.2.2 Possible Approach for Further Improvement

One issue that must be addressed is that the similarity between system-call sequences does not reflect the similarity between the contents during the execution of their source codes. A possible approach to resolve this issue is to preprocess the system-call sequence to remove unnecessary parts. For example, a system-call comes with arguments, and returns a value. Obtaining such data provides useful information for preprocessing. A system-call may fail and return an error code (typically a negative integer) as the return value. Therefore, we should exclude the failed invocations from our dataset.

Another issue is that the DA data for the same source code can have significant sequence length variations, as mentioned in Section 5.1 and as shown in Figure 3. In the example shown in Figure 3, the minimum and maximum sequence lengths were 742 and 9,378, respectively. Because we executed the same source code, it was theoretically possible to represent a sequence of 9,378 system-calls using a sequence consisting of only 742 system-calls. A maximum sequence often involves multiple consecutive invocations of the same system-call, whereas a minimum sequence does not exhibit this behavior. We can determine whether to compress multiple consecutive invocations into the same system-call if we have information about the arguments and return the value. For example, in a system-call `write`, the first argument is file descriptor `fd` (an abstract handle of a resource managed by an operating system, represented as a positive integer), the second argument is `buf` (the starting address of the memory area where the data are temporarily stored), and the third argument is `count` (the number of bytes to write). In other words, by invoking a `write`, the data specified by the second and third arguments are written to a file descriptor specified by the first argument. Therefore, if the `write` is invoked multiple times in succession and the file descriptor has the same value for each invocation, we can compress them into a single system-call invocation while maintaining temporal order by concatenating the writing data (Code 1). In the example code, the second argument is written as a Python string, which is a combined representation of `buf` and `count`. In this manner, we can compress a sequence of system-calls that are invoked multiple times.

Code 1: How we can compress `write`.

```
# current invocation
write(1,"I have a pen.")
write(1,"I have an apple.")

# possible compressing
write(1,"I have a pen.
         I have an apple.")
```

Furthermore, we can consider the relationship between a single line of code and its associated system-call subsequences. For example, Python has a built-in `open` function. This provides the capability to open files. The system-calls that enabled this functionality were `open` (#2) and `openat` (#257). In other words, when executing `open('a.txt', 'w')` in Python, the interpreter invokes either system-call 2 or 257 from the user space to perform the file-opening process. To confirm this, we performed DA on both files every 200 times: on a file containing only the line `open('a.txt', 'w')` and on an empty file. The average sequence length for invoking the `open` function was 617.2, whereas that of the empty one was 587.1. This implies that approximately 30 system-calls were invoked to handle the `open` functions. In other words, to invoke the most critical system-calls, numbers 2 or 257, 29 other system-calls were invoked. Thus, the system-call subsequences for one line in the code contain many less-important system-calls. The problems depend on the implementation of the interpreter and compiler because they decide what the system-calls to invoke to execute a given line in the code. Consequently, to accurately compare system-call sequences obtained through any language, it may be effective to consider the importance of each system-call and eliminate less important ones.

Therefore, we can apply NLP methods to calculate the importance of system-calls because system-call sequences and natural languages have similar characteristics; They are both sequences of words or system-calls arranged in lexical and temporal order. Elements derive their meaning from the context or sequence in which they appear; therefore, they have a context-dependent nature. Furthermore, system-call sequences have the advantage that the variety of words in a corpus is significantly smaller than in natural languages, making it easier to train the relationships between adjacent words. In this study, we used 388 types of system-calls, and only 126 system-calls were collected for the original source codes, generated source codes, and source codes from CodeNet, which are significantly smaller than the variety of words that exist in natural language. This suggests the possibility of constructing a powerful machine-learning model by extending our approach.

6 CONCLUSION

This study introduced a novel approach for using system-call-level DA data in programming language translations. We introduced multiple comparison methods for DA data. To relax the differences in how different program languages invoke system-calls, we developed the scsq2vec model, a Doc2Vec model adopted to handle system-call sequences. This approach addresses the issue that the similarity between system-call sequences does not reflect the similarity between the contents of their source code. The key insights obtained from our results are as follows:

1) We revealed that the DA data for the same code can exhibit a large variation in sequence length.

2) We revealed that the initialization process for modules, which might not be essential for their source code, can potentially dominate the DA data.
3) Preprocessing of the system-call sequence to remove unnecessary parts should be implemented. For example, we should exclude failed invocations or compress the sequence of the same system-call invocation multiple times. Furthermore, obtaining information about the arguments and return values of the system-calls provides useful information for implementing preprocessing.
4) The system-call subsequences for one line in the source code contain many less important system-calls. This observation should be reflected in developing the preprocessing.

In future work, we will discuss the use of DA data to select better codes, assuming that there are multiple proper translation candidates; that is, TransCoder generates multiple 100% computer accuracy functions. For instance, the smaller the time and space complexities, the better the candidate. Our current DA system cannot compute or predict the approximate empirical computational efficiency (the time required to execute the code) or memory consumption (the amount of memory allocated by the process). Hence, by obtaining the arguments of the system-calls, it may be feasible to improve our system to calculate the approximate space and time complexities.

Code 2 shows the generated and reference source codes for test `ADD_1_TO_A_GIVEN_NUMBER` (Roziere et al., 2020). In the reference source code, the condition part for the while statement performs a bitwise AND operation between `x` and `m`. However, the generated source code performs the same operation, an integer typecasting operation, and verifies whether the result is one or greater. Therefore, the generated source code performs additional operations and is more time consuming in terms of empirical computational efficiency, which should be captured by the improved DA approach.

Code 2: Differences in conditional statements.

```
# generated translation
while int(x & m) >= 1:

# reference source code
while x & m:
```

Another issue is the choice between simple and advanced modules during library selection. For example, when we use mathematical operations in Python, we can select `math` and `NumPy` modules. Code 3 shows the reference source code ("f_gold," using the `math` module) and the generated source code ("f_filled," using the `NumPy` module) for a test called `PROGRAM_FOR_SURFACE_AREA_OF_OCTAHEDRON`. The `math` module has minimal implementation, whereas the `NumPy` module implements advanced parallel processing. Therefore, although `NumPy` is effective for large datasets, the parallel processing overhead is relatively large for small datasets. DA data can address these issues as they influence the manner in which the interpreter or compiler triggers system-calls. Our DA system can capture a sequence of system-calls while maintaining their temporal order, regardless of the PID or threads. Therefore, we must use the DA data gathered by our system for better code selection.

Code 3: Differences Between math and NumPy.

```
def f_gold(side):
    return 2 * (math.sqrt(3))
             * (side * side)

def f_filled(side):
    return 2 * (np.sqrt(3))
             * (side * side)
```

REFERENCES

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.

Puri, R., Kung, D. S., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J., Chen, J., Choudhury, M., Decker, L., Thost, V., Buratti, L., Pujar, S., Ramji, S., Finkler, U., Malaika, S., and Reiss, F. (2021). CodeNet: A large-scale ai for code dataset for learning a diversity of coding tasks.

Roziere, B., Lachaux, M.-A., Chanussot, L., and Lample, G. (2020). Unsupervised translation of programming languages. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems*, volume 33, pages 20601–20611. Curran Associates, Inc.

Szafraniec, M., Roziere, B., Leather, H. J., Labatut, P., Charton, F., and Synnaeve, G. (2023). Code translation with compiler representations. In *The Eleventh International Conference on Learning Representations*.

Yoneda, N. (2023). langMorphDA. https://github.com/naru-99/langMorphDA.git.