International Journal of Control and Automation
Vol. 12, No. 8 (2019), pp.111-124
http//dx.doi.org/10.33832/ijca.2019.12.8.10

A STUDY ON THE PREDICTION OF PROGRAM COMPLEXITY SECTION FOR OFFLOADING EXECUTION DECISION

Jaehyun Kim and Yangsun Lee*

Dept. of Computer Engineering, Seokyeong University
16-1 Jungneung-Dong, Sungbuk-Ku, Seoul 02713, Korea
yslee@skuniv.ac.kr

***Abstract*— The IoT-Cloud fusion virtual machine system is a platform for low-performance IoT(Internet of Things) devices, and uses offloading technology, one of the cloud computing technologies that delegate tasks requiring high-performance computing power to the cloud server. Although the offloading technique has solved the problem that the performance of the virtual machine is dependent on the performance of the IoT device being operated, there is a network cost due to the communication between the cloud and the local device. Therefore, depending on the program complexity, the offloading performance may be lower than the performance of the local device.**

In this paper, we predicted a program area complexity based on deep learning to solve this problem. The program complexity area prediction predicts the execution complexity of the program by learning the program complexity estimate and actual execution complexity analyzed by the static profiler. Since the complexity area can be used as an index for determining the offloading execution, the offloading can be performed only when the offloading performance is advantageous, and efficient offloading can be performed.

***Keywords*— Internet of Things (IoT), Offloading, Cloud Computing, Program Complexity, Deep Learning**

1. INTRODUCTION

The IoT-Cloud fusion virtual machine system is a platform for low-performance IoT(Internet of Things) devices. It uses offloading technology, which is one of the cloud computing technologies that delegate tasks requiring high-performance computing power to the cloud server. Although the offloading technique solves the problem that the performance of the virtual machine is dependent on the performance of the IoT device being operated, there is a network cost due to the communication between the cloud and the local device. Therefore, depending on the program complexity, the offloading performance may be lower than the performance of the local device [1-5].

In this paper, we have developed an index for determining offloading execution by predicting program complexity area based on deep learning. The program complexity area prediction predicts the execution complexity of the program by learning the program complexity estimate and actual execution complexity analyzed by the static profiler. Since the offloading execution can be determined only when the offloading execution gains, the generated program complexity section can be efficiently offloaded.

Received: May 7, 2019
Reviewed: July 22, 2019
Accepted: August 9, 2019
* Corresponding Author

ISSN: 2207-6387 IJCA

2. RELATED STUDIES

2.1. IOT-CLOUD FUSION VIRTUAL MACHINE SYSTEM

The IoT-Cloud fusion virtual machine system is a virtual machine system that provides high-performance cloud computing power to the low-performance IoT device. This system can provide the computing power of a high-performance cloud server when running computationally intensive tasks, which can compensate for the drawback of virtual machine systems, and accommodate applications written in various programming languages by maintaining the platform-independent execution environment, which is advantage of existing smart virtual machine system [6-9]. Figure 1 shows the overall structure of the IoT-Cloud fusion virtual machine system.

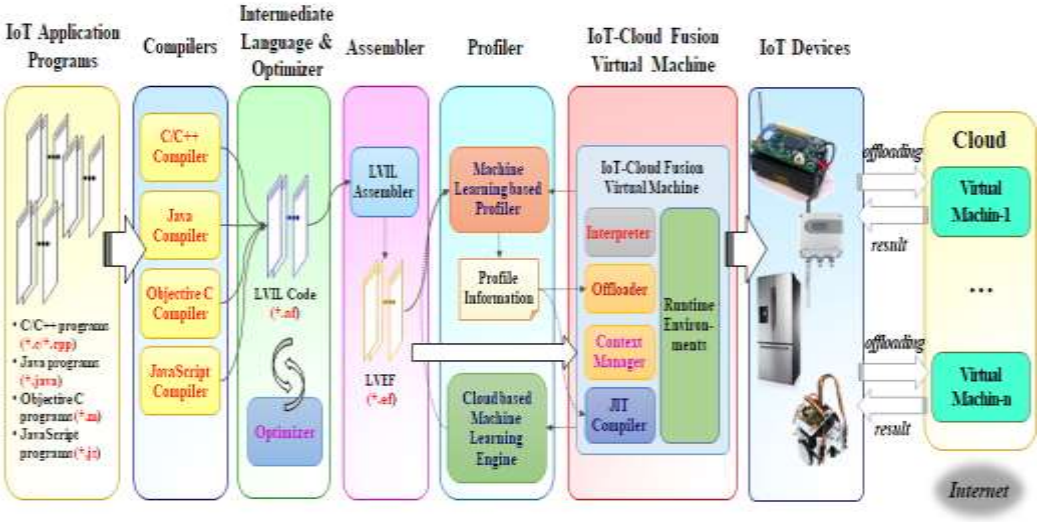

Fig. 1 IoT-Cloud Fusion Virtual Machine System

The configuration of the IoT-Cloud fusion virtual machine system consists largely of a compiler, an optimizer, an assembler, a profiler, and a lightweight virtual machine. The compiler generates assembly files by compiling applications written in various languages, and the optimizer optimizes the intermediate code of the assembly file generated by the compiler to improve execution efficiency.

The assembler translates the assembly file into an executable file, which is a binary code type executable in the virtual machine, and the profiler analyzes the complexity of the function by traversing the executable file generated by the assembler on a function by function. Function complexity is used to determine offloading when an executable file is executed on a virtual machine by delegating highly computationally intensive tasks to a high-performance cloud server.

A lightweight virtual machine is a virtual machine for a low-performance IoT device that can be ported to a configuration suitable for IoT devices through a structured design, and consumes less memory when executing an application while maintaining performance because it uses less memory when loading instructions.

2.2. OFFLOADING

Offloading [10-14] is a cloud computing method that delegates tasks requiring high computation performance to servers and receives the results of the operations performed on the servers [10,11]. Figure 2 shows a cloud computing module.

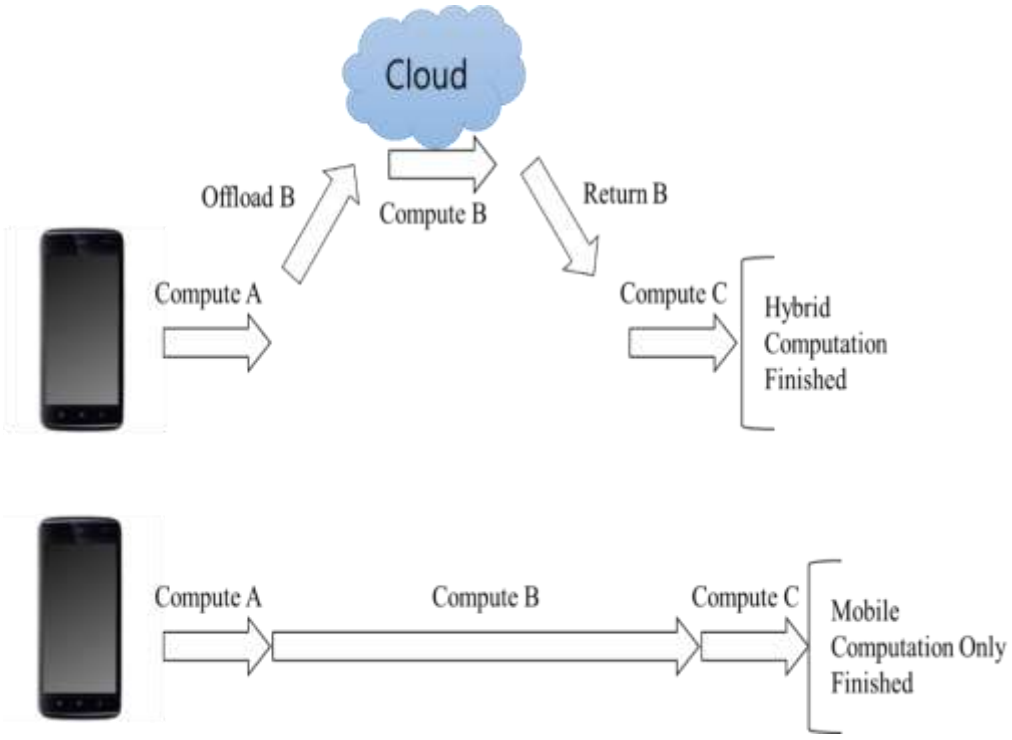

Fig. 2 Cloud Computing Module

Cloud computing technology [15-19] provides high computing performance regardless of the performance of the local device, and enables resources such as memory to be provided as services through the network. Offloading can improve performance, but since network communication overhead can occur, unconditional offloading may result in decreased performance. Therefore, consideration should be given to factors affecting performance to decide offloading execution.

2.3. RECURRENT NEURAL NETWORKS

Recurrent neural networks [4,20] have recently shown useful results in various applications, especially when there is a sequential dependency on the data. One of the recurrent neural networks, LSTM (Long Short Term Memory) [6], is popularly used because of its ability to learn hidden sequential dependencies. However, recurrent neural networks have a scaling issue that is not easy to train when there are a large number of neuron units or a large number of input units.

3. COMPLEXITY AREA PREDICTION

The complexity area prediction model is a learning model for generating an index for efficient offloading decision, and amis at obtaining an index to be used in the offload execution decision by predicting the complexity area of the offloading target program, which is usable as an offloading decision index before the offloading is executed. Figure 3 shows the system diagram of the complexity area prediction model generator.

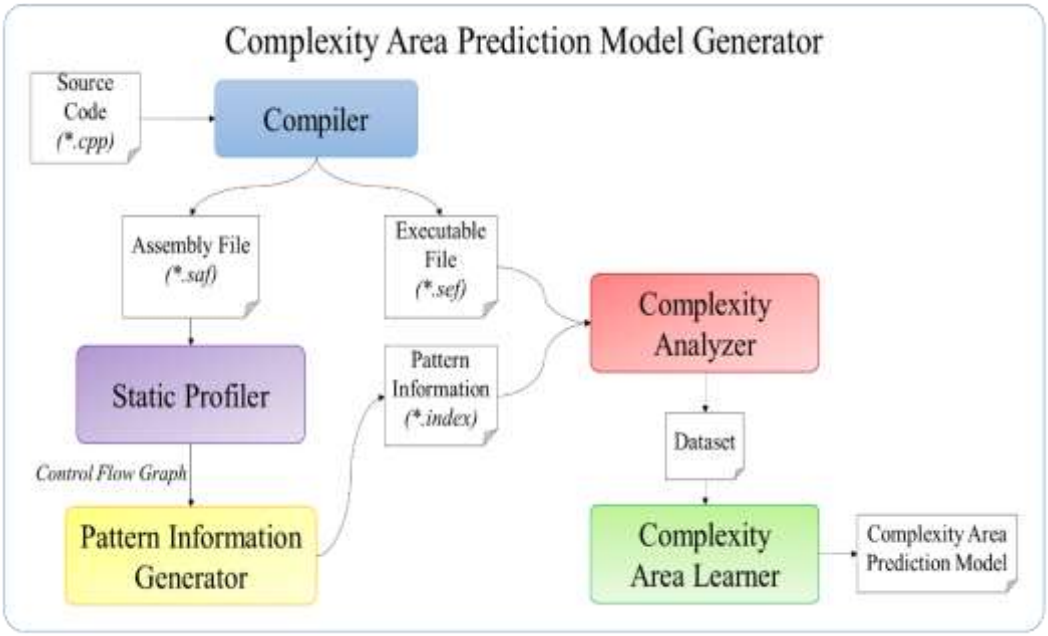

Fig. 3 System Diagram of the Complexity Area Prediction Model Generator

The complexity area prediction model generator includes a pattern information generator for generating pattern information by analyzing an assembly file generated by a compiler for generating learning data, a complexity analyzer for measuring actual execution complexity based on the generated pattern information, and a complexity area learning module for generating a model for predicting a complexity area.

3.1. PATTERN INFORMATION GENERATOR

This section deals with the process of generating the block pattern information by analyzing the control flow graph generated by the existing static profiler to generate the learning data set for the complexity section prediction. Figure 4 shows the control flow graph generated by the static profiler.

In this control flow graph, there are three patterns of Loop, Branch, and Sequential and overlapping patterns in which three or more patterns are superimposed. Depending on this pattern, the complexity values will vary. For example, when an application program is composed of a single Loop pattern and when it is composed of an overlapping Loop pattern, if the repetition times of the patterns are the same, the program complexity is large when the program is composed of overlapping patterns.

In this paper, we classify the control flow graph by patterns, and generate pattern information by analyzing the start point and end point of each pattern, the expected complexity value and the number of iterations of each pattern. Figure 5 shows the structure of the pattern information generator.

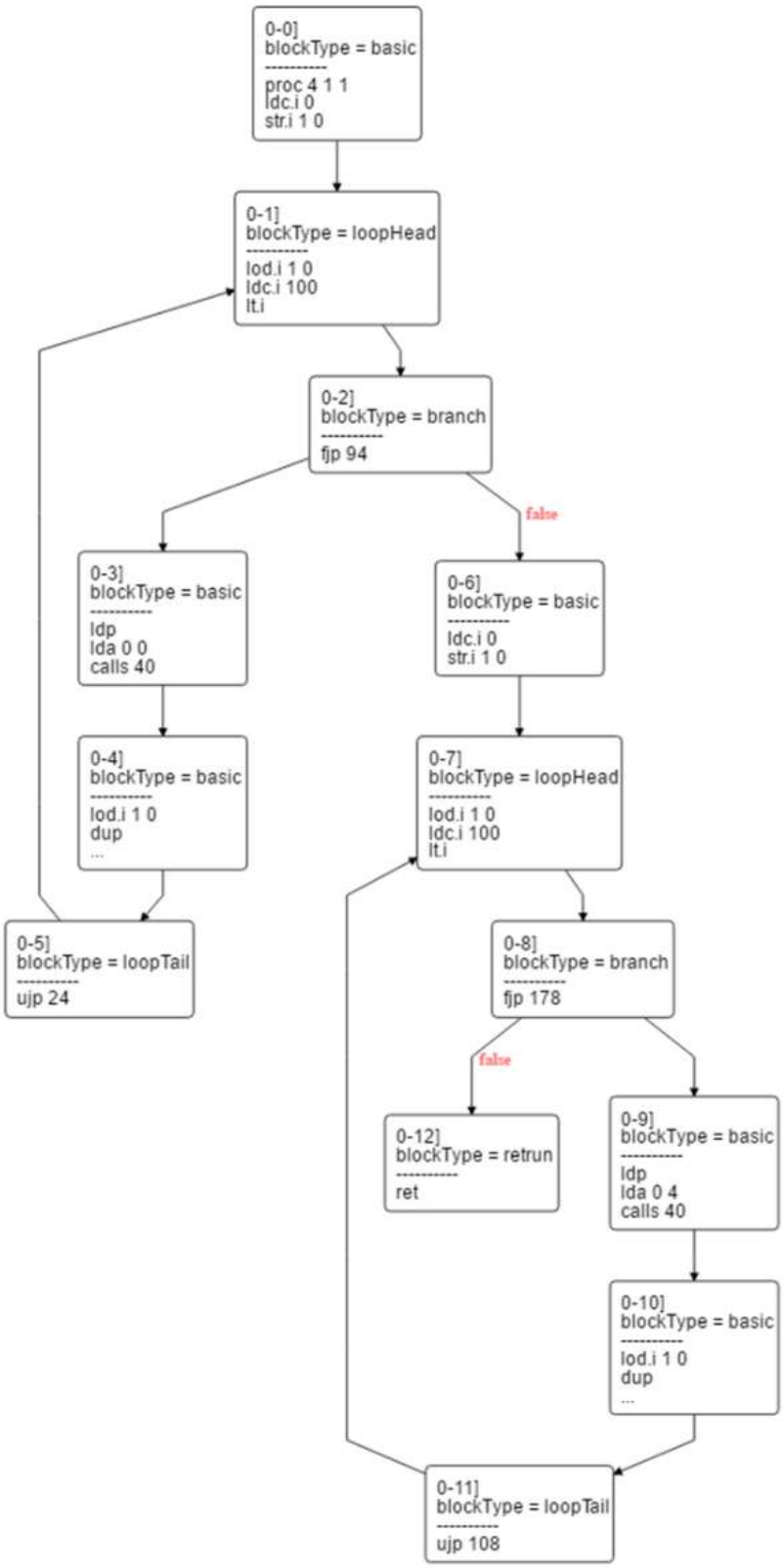

Fig. 4 Control Flow Graph

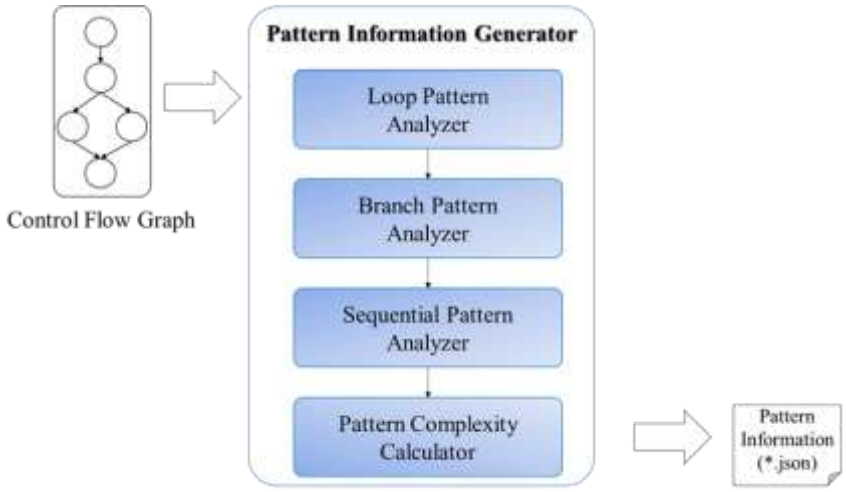

Fig. 5 Structure of the Pattern Information Generator

3.1.1 REPEATING PATTERN INFORMATION

When the control flow graph is generated in the static profiler, the iteration pattern consists of Loop Head, Loop Exit Branch, Loop Body, and Loop Tail blocks. The Loop Head block is a block in which a repetition pattern starts, and s Loop Tail block is a block for branching to a Loop Head block after execution of a repetition body, and a Loop Exit Branch block is a branch block to be used when the iteration condition is not satisfied, that is, to exit the loop structure. Figure 6 shows the repetitive pattern structure.

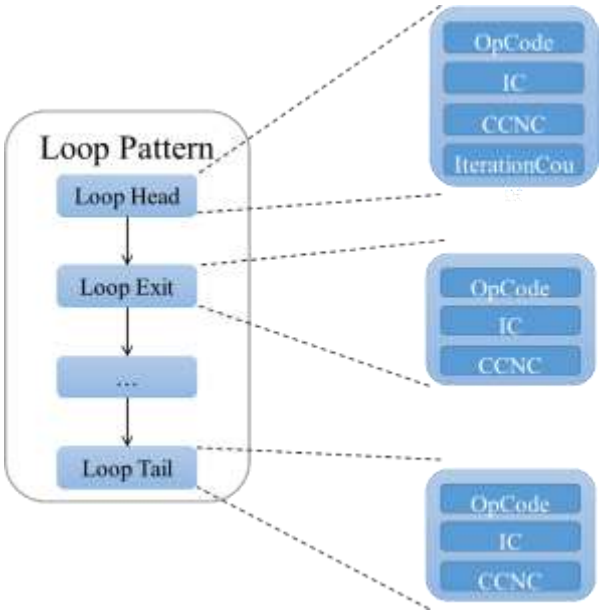

Fig. 6 Repetitive Pattern Structure

Therefore, the iterative pattern analyzer traces the control flow graph, records the position of the start instruction when it reaches the loop head block, and records the position of the end instruction of the loop pattern when it reaches the loop exit block. When the recording of the start and end points of the pattern is completed, all the blocks included in the pattern are traced, and the pattern information is completed by accumulating the instruction cycle and the pattern repetition count per block analyzed by the static profiler. Figure 7 shows the repetitive pattern information generated by the pattern classifier.

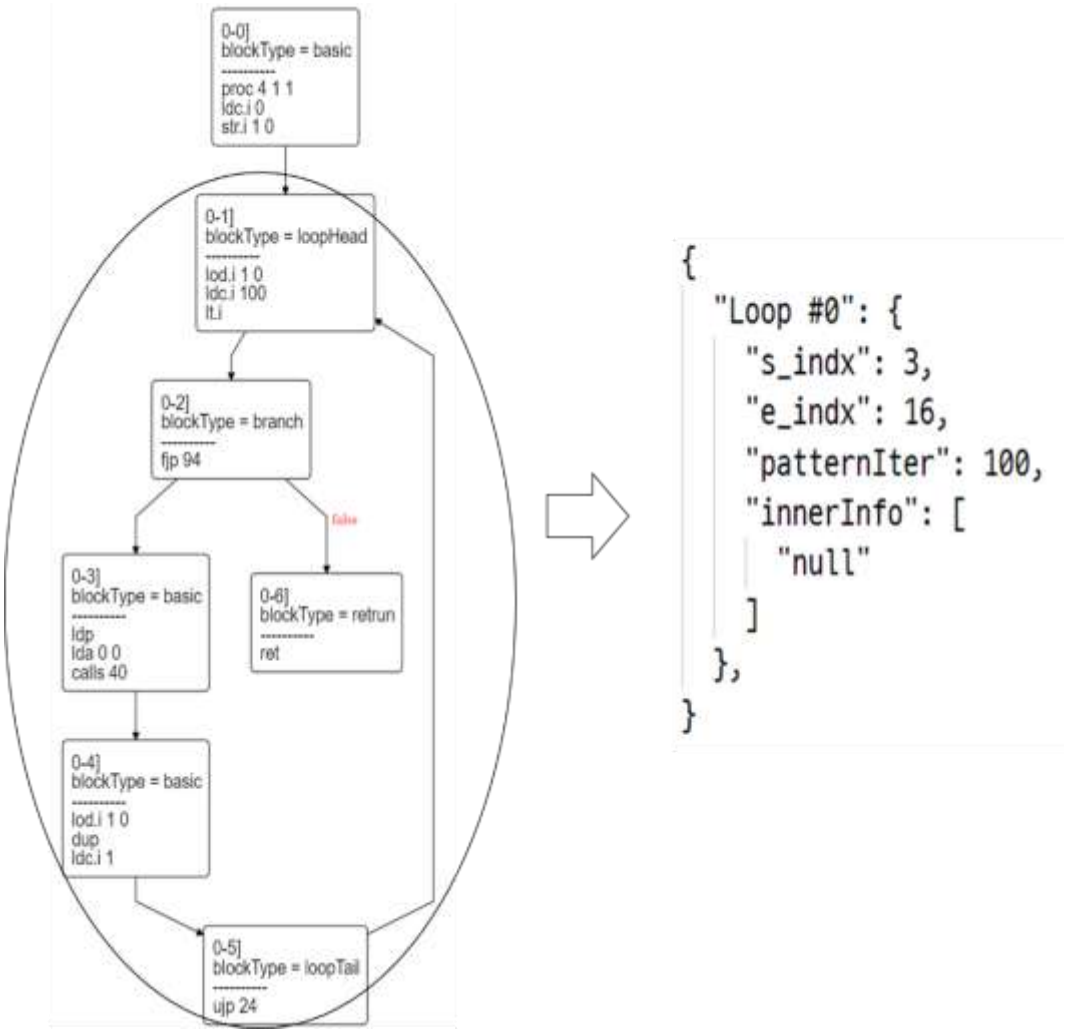

Fig. 7 Repetitive Pattern Information

3.1.2. BRANCH PATTERN INFORMATION

A branch pattern is a pattern in which a branch is divided according to a condition in a branch statement such as an if-else. Figure 8 shows the structure of the branch pattern.

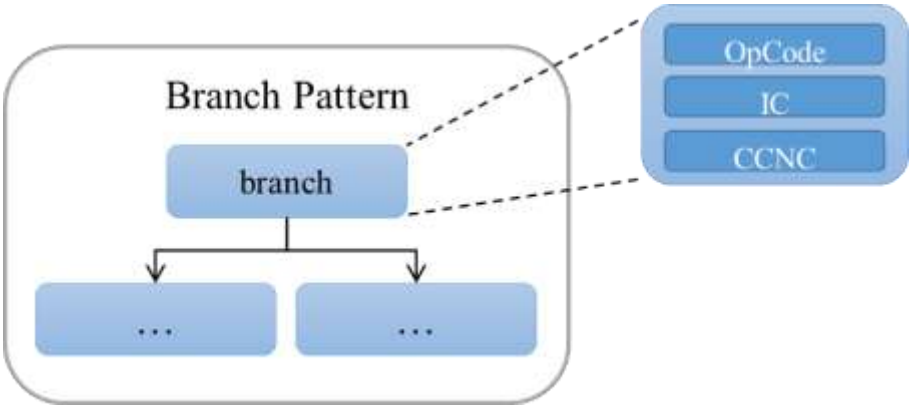

Fig. 8 Branch Pattern Structure

In the control flow graph generated by the static profiler, the branch block consists of branch instructions in assembly instructions. The branch block determines the branch

according to the true and false of the condition, so it is the start of the branch pattern. Therefore, the branch pattern analyzer traces the control flow graph, records the position of the start instruction when it reaches the branch block, and records the end instruction position of the pattern when it reaches the join block where the divided branches are added. When the start point and the end point of the pattern are completed as in the repeat pattern, all the blocks included in the pattern are traced, and the pattern information is completed by accumulating the instruction cycle and the pattern repeat count analyzed for each block analyzed by the static profiler. Figure 9 shows the branch pattern information generated by the pattern classifier.

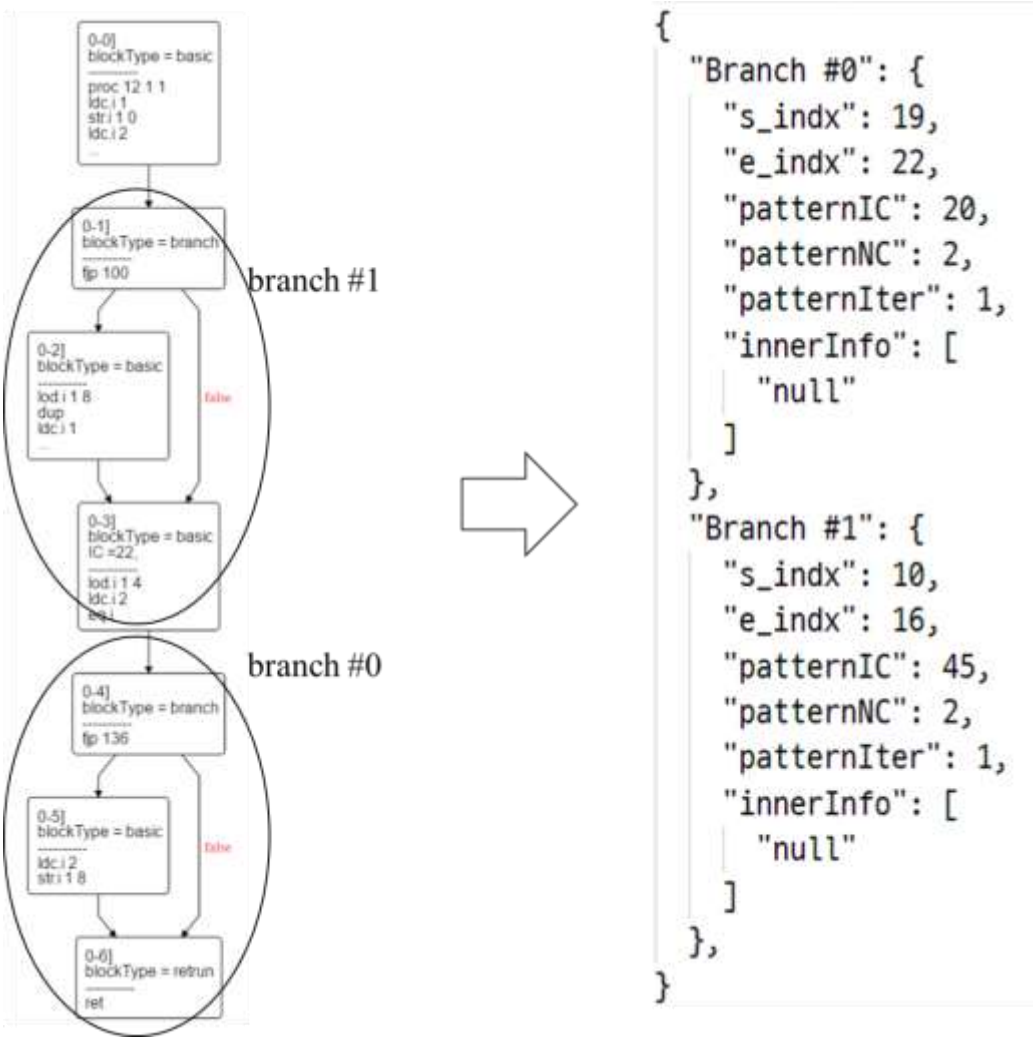

Fig. 9 Branch Pattern Information

3.1.3. OVERLAPPING PATTERN INFORMATION: The control flow graph may have a overlapping pattern in addition to the branching and repeating patterns. In the control flow graph, in the control flow graph, the structure of the overlapping pattern exists as an internal pattern toward the bottom of the graph. In order to analyze this structure, we analyzed the superposition pattern structure by using the method of back tracing the graph and propagating the inner pattern upward. Figure 10 shows the pattern information including the overlapping pattern structure.

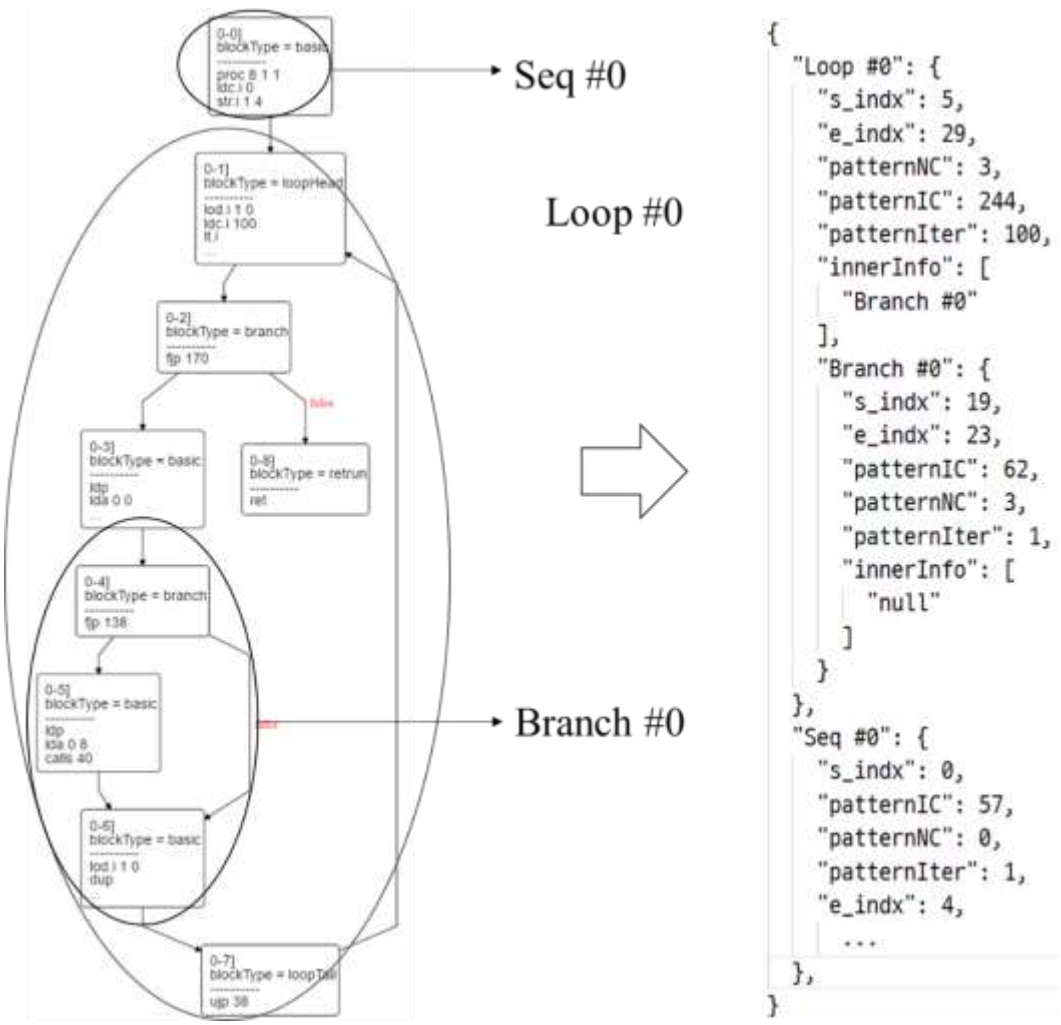

Fig. 10 Pattern Information including the Overlapping Pattern Structure

3.2. BLOCK PATTERN COMPLEXITY MEASUREMENT

This section measures the execution complexity by pattern based on the generated pattern information. Execution complexity by pattern is measured by actually executing in the virtual machine based on the CPU execution cycle of the predefined assembly instruction. Figure 11 shows the structure of the pattern complexity predictor.

The pattern information loader parses the pattern information generated by the pattern information generator, and stores the pattern information in the memory. The parsed pattern information is used for complexity analysis. The instruction cycle calculator calculates execution cycles for each pattern while actually executing the program based on a predefined instruction cycle. If there is an overlapped pattern in the execution cycle calculation, the execution cycle of the inner pattern is calculated first and then propagated to the upper pattern to accumulate the execution cycle of the upper pattern. Like the instruction cycle calculator, the repetition count calculator propagates the repetition count of the lower pattern to the upper pattern and accumulates it. The pattern repetition count and the pattern execution cycle thus calculated are processed into a data set in the pattern complexity data set generator.

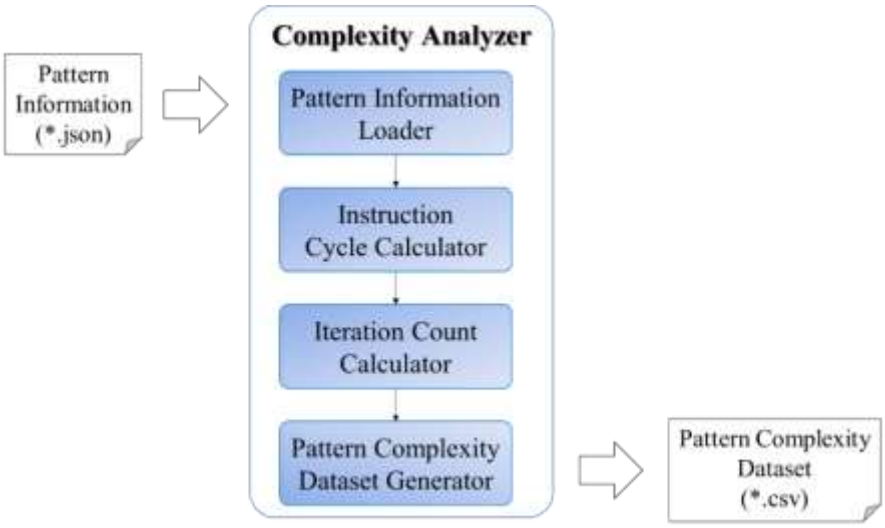

Fig. 11 Structure of the Pattern Complexity Predictor

3.3. PROGRAM COMPLEXITY SECTION LEARNING

In this section, complexity interval learning is performed based on the generated data set. Since the program has execution order, this paper adopts LSTM neural network as a deep learning neural network. In order to learn LSTM neural network, it is necessary to convert the data set configuration into a time-ordered configuration. Therefore, in this paper, LSTM neural network learning is performed using processed data after processing raw data set to enable neural network learning.

The data processor processes raw data sets into a form suitable for neural network learning, and consists of a data set loader, a historical data generator, a complexity interval generator, and a data normalizer. Figure 12 shows the structure of the data processor.

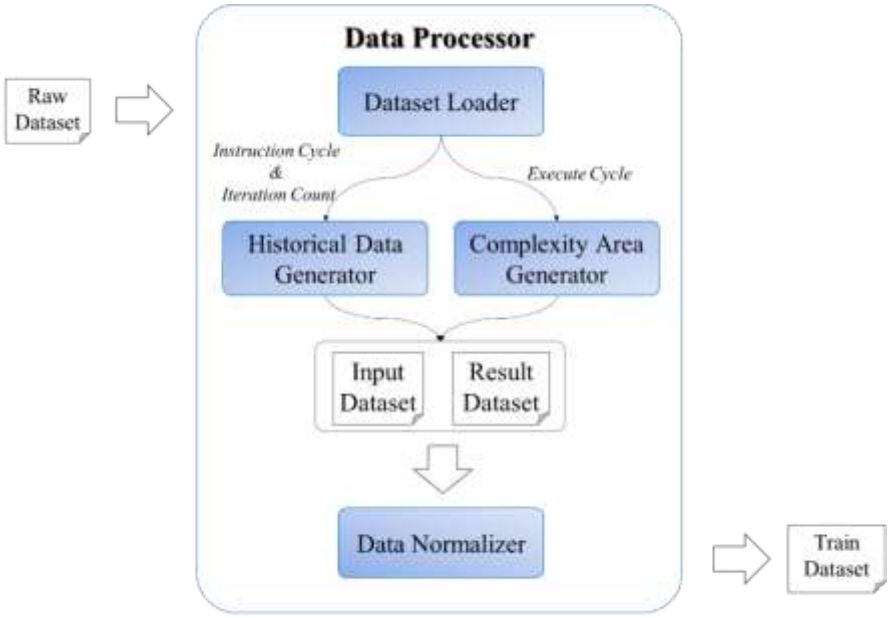

Fig. 12 Structure of the Data Processor

The history information generator receives the command cycle and pattern repetition count data from the data set loader and generates time series data suitable for LSTM neural

network learning. The input data of this paper generates data for LSTM neural network to learn from the present time to past 5 data, and padding to 0 when there is no past data. The complexity section generator takes an execution cycle as input in the data set loader and generates a result for the current input, *i.e.*, complexity section data. The complexity section scale was divided into 4 scale. Table I shows the complexity section scale.

Table I. Complexity Section Scale

Execution Cycle	under 10000	10000 ~ 100000	100000 ~ 3000000	over 3000000
Complexity Section	very low	low	middle	high

The complexity section generator generates a result dataset according to the criteria in Table 1. The generated data is processed so that it can be used for learning through original hot encoding. The data normalizer normalizes the data sets generated by the history information generator and the complexity section generator, and processes the data sets to a value between 0 and 1. This normalization process is an essential process because non-normalized data increases the loss rate in neural network learning. Normalization was normalized using the MinMax normalization method. Equation 1 is a MinMax regular expression.

$$z = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{1}$$

70% of the data processed in the data processor is used for LSTM neural network learning and 30% is used for testing the learned neural network.

4. EXPERIMENTAL RESULTS AND ANALYSIS

This section deals with the results of LSTM neural network learning and measures the accuracy of the results predicted by the neural network. The following definition is used for precision calculation.

- True Positive (TP) : the number of times the learned neural network model is True, with the actual answer being True

- False Positive (FP) : the number of times the learned neural network model is True, with the actual answer being False

- True Negative (TN) : the number of times the learned neural network model is False, with the actual answer being False

- True Negative (FN) : the number of times the learned neural network model is False, with the actual answer being True

The above definitions are used to calculate precision, such as Sensitivity, Specificity, Precision and Accuracy. Formula 2, 3, 4, and 5 represent a precision equation.

$$Sensitivity_t = \frac{TP_t}{TP_t + FN_t} \tag{2}$$

$$Specificity_t = \frac{TN_t}{TN_t + FP_t} \tag{3}$$

$$Precision_t = \frac{TP_t}{TP_t + FP_t} \tag{4}$$

$$Accuracy_i = \frac{(TP_i + TN_i)}{(TP_i + TN_i + FP_i + FN_i)} \quad (5)$$

Sensitivity is a statistic of how much the learned model answers the actual answers. For example, shows the probability that the actual answer is 0, but the learned model is zero. Specificity is a statistic of how much a learned model calls an incorrect answer. For example, shows the probability that a learned model is not zero for things that are not actually correct. Precision is the probability that the results presented by the learned model are the actual answers. That is, if the learned model classifies as zero, the probability that the actual answer is zero. Accuracy is a statistic about how much the correct answer is. That is to say, how accurate the models learned are.

Table II. Precision of Dense Neural Network Learning Model

Dense_Only				
category	sensitivity	specificity	accuracy	precision
0	0.96666667	0.9189189	0.93269231	0.82857143
1	0.60714286	0.9078947	0.82692308	0.70833333
2	0.6	0.9240506	0.84615385	0.71428571
3	0.85	0.8915663	0.88349515	0.65384615
avg	0.75595238	0.9106076	0.87231609	0.72625916

Table III. Precision of LSTM+Dense Neural Network Learning Model

LSTM+Dense				
category	sensitivity	specificity	accuracy	precision
0	0.96666667	0.9189189	0.93269231	0.82857143
1	0.57142857	0.92	0.82524272	0.72727273
2	0.64	0.9102564	0.84466019	0.69565217
3	0.80952381	0.9156627	0.89423077	0.70833333
avg	0.74690476	0.9162095	0.8742065	0.73995742

Table IV. Precision of LSTM Neural Network Learning Model

LSTM_Only				
category	sensitivity	specificity	accuracy	precision
0	0.96666667	0.8831169	0.90654206	0.76315789
1	0.64285714	0.9066667	0.83495146	0.72
2	0.72	0.9615385	0.90291262	0.85714286
3	0.85	0.9277108	0.91262136	0.73913043
avg	0.79488095	0.9197582	0.88925687	0.7698578

Table V. Precision of GRU Neural Network Learning Model

GRU_Only				
category	sensitivity	specificity	accuracy	precision
0	0.96666667	0.9452055	0.95145631	0.87878788
1	0.57142857	0.8933333	0.80582524	0.66666667
2	0.56	0.9102564	0.82524272	0.66666667
3	0.75	0.8795181	0.85436893	0.6
avg	0.71202381	0.9070783	0.8592233	0.7030303

Table II, III, IV, V are the results of measuring the precision of using a combination of LSTM and Dense neural networks, and with LSTM, GRU neural networks. The experimental results show that the overall Accuracy is similar, but the learning model using LSTM only has a higher answer rate than the probability of correct answers, Sensitivity and Precision.

5. CONCLUSIONS AND FURTHER RESEARCHES

The offloading technique of IoT-cloud fusion virtual machine systems has solved the problem that virtual machine performance depends on the performance of the IoT operating device, but the complexity of the task results in the offloading performance being lower than that of the local device because of the network cost arising from communication between the cloud and local device.

This paper has designed and implemented a complexity section predictor that predicts the actual complexity section of the program by creating data set, learning the generated data set, and predicting the actual complexity section of the program, in order to predict the complexity section, an indicator for determining efficient offloading execution. These complexity section predictors allow more efficient offloading execution because offloading runs only when the program's complexity is greater than the overhead that occurs when offloading runs.

Currently, the offloading system uses only the workload of the cloud server as an indicator to determine the offloading run. It is planned to study offloadings only if the performance of the offloading increases when offloading a task that is subject to offload by adding the complexity section that was created in the future to the offloading.

ACKNOWLEDGEMENTS

This research was supported by Seokyeong University in 2019.

REFERENCES

[1] C. Michael, L. Markus, R. Roger, "The Internet of Things", McKinsey Quarterly, (2014).
[2] Y. Son, J. Jung, Y. Lee, "An Adaptive Offloading Method for an IoT-Cloud Converged Virtual Machine System Using a Hybrid Deep Neural Network", Sustainability, Vol.10, No.11, (2018): 1-15.
[3] J. Kim, Y. Lee, "A Study on Context Synchronization Method for Offloading on IoT-Cloud Fusion Virtual Machine," International Journal of Grid and Distributed Computing, Vol.10, No.2, (2017): 1493-160.
[4] Y. Son, S. Oh, Y. Lee, "Hybrid Deep Neural Network based Performance Estimation Method for Efficient Offloading on IoT-Cloud Environments," International Journal of Grid and Distributed Computing, Vol.11, No.7, (2018): 23-30.
[5] Y. Son, Y. Lee, "Offloading Method for Efficient Use of Local Computational Resources in Mobile Location-Based Services Using Clouds," Mobile Information Systems, Netherlands Hindawi Publishing Corp., Vol. 2017, (2017): 1-9.
[6] Y. Lee, Y. Son, "A Study on the Smart Virtual Machine for Executing Virtual Machine Codes on Smart Platforms", International Journal of Smart Home, SERSC, Vol. 6, No. 4, (2012): 93-105.
[7] Y. Lee, Y. Son, "A Study on the Smart Virtual Machine for Smart Devices", Information-an International Interdisciplinary Journal, International Information Institute, Vol. 16, No. 2, (2013): 1465-1472.
[8] Y. Son, J. Kim, Y. Lee, "Design and Implementation of HTML5 based SVM for Integrating Runtime of Smart Devices and Web Environments", International Journal of Smart Home, Vol. 8, No. 3, (2014): 223-234.
[9] Y. Son, J. Jung, Y. Lee, "Design and Implementation of the Secure Compiler and Virtual Machine for Developing Secure IoT Services", Future Generation Computer Systems, Vol. 76, (2014): 350-357.
[10] K. Kumar, "A Survey of Computation Offloading for Mobile Systems", Mobile Networks and Applications, Vol. 18, No. 1, (2013): 129-140.
[11] K. Yang, S. Ou, and H.H. Chen, "On Effective Offloading Services for Resource-Constrained Mobile Devices Running Heavier Mobile Internet Applications", IEEE Comm. Magazine, Vol. 46, No. 1, (2008): 56-63.
[12] D. Kovachev, T. Yu, R. Klamma, "An Offloading Decision Scheme Considering the Scheduling Latency of the Cloud in Real-time Applications", Proceedings of IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, (2013): 784-791.

[13] H. Chen, Y. Lin, C. Chen, "COCA: Computation Offload to Clouds using AOP", Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, (2012): 466-473.
[14] C. Wang, Z. Li, "A Computation Offloading Scheme on Handheld Devices", Journal of Parallel and Distributed Computing, Vol. 64, No. 6, (2004): 740-746.
[15] J. Park, S. Kim, "A Prediction-based Dynamic Component Offloading Framework for Mobile Cloud Computing", Journal of KIISE, Vol. 45, No. 2, (2018): 141-149.
[16] H. Min, J. Jung, B. Kim, J. Heo, "Context-Aware Decision Engine for Mobile Cloud Offloading", KIISE Transactions on Computing Practices, Vol. 23, No. 6, (2017): 392-396.
[17] H. La, S. Kim, "A Taxonomy of Offloading in Mobile Cloud Computing", Proceedings of the 7th IEEE International Conference on Service-Oriented Computing and Applications, (2014): 147-153.
[18] H. T. Dinh, C. Lee, D. Niyato, P. Wang, "A Survey of Mobile Cloud Computing: Architecture, Applications, and Approaches", Wireless Communications and Mobile Computing, vol. 13, no. 18, (2013): 1587-1611.
[19] T. Lin, C. Hsu, C. King, "Context-Aware Decision Engine for Mobile Cloud Offloading", Proceedings of IEEE Wireless Communications and Networking Conference Workshops, (2013): 111-116.
[20] S Hochreiter, J Schmidhuber, "Long Short-term Memory," Neural Computation, Vol. 9, (1997): 1735-1780.

