A Graph Partitioning Approach for Efficient Dependency Analysis using a Graph Database System

Kazuma Kusu
Graduate School of Culture and Information Science,
Doshisha University
1-3 Tatara-Miyakodani, Kyotanabe,
Kyoto 610-0394, Japan
Email: kusu@ilab.doshisha.ac.jp

Izuru Kume
Graduate School of Information Science,
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma,
Nara 630-0192, Japan
Email: kume@is.naist.jp

Kenji Hatano
Faculty of Culture and Information Science,
Doshisha University
1-3 Tatara-Miyakodani, Kyotanabe,
Kyoto 610-0394, Japan
Email: khatano@mail.doshisha.ac.jp

***Abstract*—Program execution traces, which include data/control dependency information, are indispensable for new types of debugging such as back-in-time techniques. In this study, we implement a dependency environment for the Java programming language focusing on tracing the relationships in dependency analyses, using the graph database (Neo4j) optimized for tracing graph edges. In the dependency analysis environment, we propose an efficient approach for handling the traces on a graph database system by evaluating memory usage and analysis time. Traces of practical programs are prone to have vast complex data, making it difficult to develop practical back-in-time debuggers. To address this challenge, our dependency environment enables an efficient analysis of the traces. The trace in our dependency analysis environment has a graph structure whose nodes denote executed Java bytecode instructions, and edge that represent data/control dependencies between the nodes. By a simple implementation of our dependency analysis environment, we confirm the existence of bottlenecks through evaluation experiments, which are then remedied in order to improve the performance of the technique's memory usage and analysis time. As a result, our environment enabled efficient process dependency analysis, reducing memory usage by 43.1% and analysis time by 4.3%.**

Keywords–Dependency Analysis; Back-in-time Debugger; Debugging Support; Graph Database; Graph Search; Java.

I. INTRODUCTION

The examination of runtime states and their dependencies are indispensable to program debugging [2] [3]. Debuggers that are currently in use allow maintainers to suspend program execution at specified break points and examine the runtime states at these points. However, such debuggers do not have a provision for maintainers to examine states prior to the designated points for the suspension of execution. Therefore, they cannot trace backwards to detect causes of erroneous states by following the dependency of statements [4].

In the last decade, so-called *back-in-time debuggers* have emerged as a new kind of debugging support tools. These debuggers use traces containing dependency information [5] [6] [7]. Such debuggers analyze dependencies to determine the operation that assigns value to a referenced variable [5], to examine the reasons why a given statement is or is not executed [6], and what happens during the execution of a method that has already been successfully invoked [7]. This kind of dependency analysis is useful for the examination of a particular instruction.

The scalability of process traces containing dependency information has been discussed in the literature [4]. We believe that the recent, rapid developments in hardware and software technologies have made it possible to process the traces of a certain scale of software products. In previous work [8], we demonstrated two kinds of dependency analysis that detect symptoms of a malfunction caused by defects in the application of the Java framework application [9].

Although our previous study raised the prospect of a solution to the scalability problem, the implementation of our dependency analysis remained inefficient. The main cause of this was the richness of the data in the model of our traces. The design of our trace proposed here aims not only at the requirements of symptom detection [8], but also at the analysis of other aspects of program execution. Therefore, our trace design incorporates the richness of data to enable various kinds of dependency analysis instead of reducing the amount of data, such as in the approach proposed by Wang et al. [10].

In addition to back-in-time Debuggers [5] [6] [7], which aim at a microscopic perspective for the dependency analysis of a specific statement, our previous study [8] dealt with all-state updates via *persistent variables* and their value dependency across the entire trace. A persistent variable is either a class variable, an instance variable, or an array component. It implements a state that persists after the invocation of a method is completed [11]. This macroscopic nature of our dependency analysis renders it inefficient, although the algorithm works in practice. In order to solve this problem, an approach is needed to support the efficient analysis of dependency in a large trace.

In this paper, we implement an efficient dependency analysis environment to perform out the trace studies as done previously [8]. Moreover, we clarify the inefficiency factor in our dependency analysis environment, and suggest an approach to address this factor. Furthermore, we evaluate our approach for improving the processing of analysis results after applying our approach. According to this, we clarify bottlenecks in our dependency analysis environment that need to be resolved. Previously, we identified the factors affecting efficiency in our dependency analysis environment and expanded a trace-partitioning approach for use in our study [1]. Moreover, we proposed an extension of a previous approach [12] for partitioning trace. We reduced memory consumption in a dependency analysis, but did not reduce the processing time.

Therefore, we propose an approach for efficiently traversing our trace in this paper. Finally, we conduct an experiment characterizing the effectiveness of our approach.

We introduce concepts related to dependency analysis, and describe demands for dependency analysis environments in Section II. Then, in Section III, we illustrate our implementation of a dependency analysis environment that consists of trace generation and a trace processing parts using a graph database system (GDB). In Section IV, we propose a naïve trace-partitioning approach based on the characteristics of the GDB for efficient dependency analysis. We conduct a preliminary experiment for evaluating dependency analysis performance on our environment in Section V. In Section VI, we reconsider a trace-partitioning approach guided by an analysis of the bottleneck in our dependency analysis environment clarified during the preliminary experiment. In Section VII, we conduct an evaluation of the efficiency and scalability of our expanded approach. Finally, in Section VIII, we consider our contribution for efficient dependency analysis as indicated by the experiment.

II. Related Work

Debuggers widely used in software development projects support a common feature to suspend program execution at a specified *break point* and show the runtime state at that point. They do not record the execution and, thus, have the common drawback that there is no way to examine the execution of a method whose invocation has been already completed. This is a serious problem because defects and infections are often found in methods that have been completed before the program fails [7]. A defect is an error in program code while an infection, in software engineering, is a runtime error caused by the execution of a defect [2].

Maintainers using a debugger must repeat a task to specify a breakpoint, as it is usually very difficult to find a suitable breakpoint in the program code, and re-execute the program to examine the executions of methods that have been completed. Such a debugging style, forced by the common limitation in current of existing debuggers, leads to inefficient debugging [4].

Using traces for debugging support is a natural idea to overcome the above limitation in existing debuggers [5] [6] [13]. An omniscient debugger [5] examined assignment operations with set values referenced from variables. If a maintainer wanted to determine why a statement has or has not been executed, Whyline [6] analyzed related dependencies and generated the results of the analysis using sophisticated Graphical User Interfaces (GUI).

Dynamic Object Flow Analysis [13] aims to understand program execution from the aspect of object references. Its area of application ranges from dependency analysis of methods for software testing [14] to performance engineering for a back-in-time debugger [7].

To the best of our knowledge, no existing dependency analysis approaches to debugging support are aimed at macroscopic dependency analysis except for our previous proposal [8]. An omniscient debugger deals with only the correspondence between the value of a variable and the assignment operation that has set this value. Whyline navigated a maintainer to the dependencies among statements to the extent of his/her manual examination. Dynamic object flow analysis performs macroscopic analysis but only deals with object references.

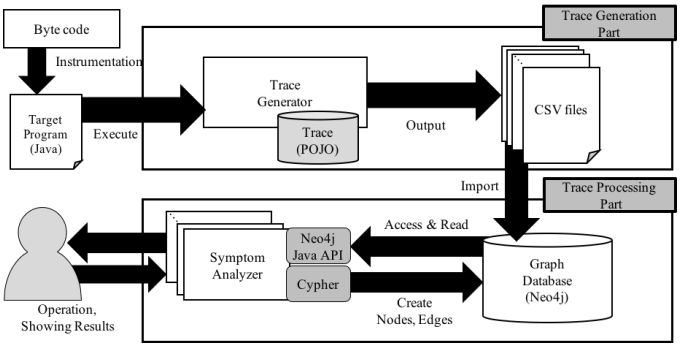


Figure 1. Our dependency analysis environment

The above approaches to microscopic dependency analysis provide useful debugging aids. However, understanding a program from a macroscopic viewpoint is necessary for debugging [15]; therefore, maintainers have to spend time and effort to obtain this perspective through manual dependency analysis.

We studied several kinds of macroscopic dependency analysis in this context in our last study [8]. Of these, *outdated-state* analysis aims to identify symptoms to suggest possible infections incurred by the accidental use of an old value of a field or array component along with its updated value.

III. Implementation of Dependency Analysis Environment

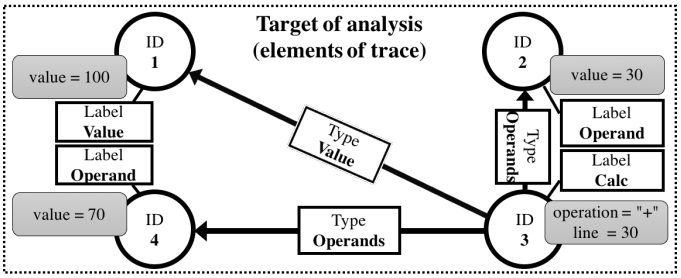
Debugging a program requires various analyses of statement dependencies. Therefore, we developed two kinds of techniques for analyzing the relevant symptoms in our previous study [8]. The proposed trace was designed to execute such dependency analyses. For this reason, our trace tended to be large and complex, and to be usually led to inefficient processing of dependency analysis. In order to conduct an efficient dependency analysis, an analysis environment is needed that enable to handle our trace efficiently.

Figure 1 illustrates the entire process, which involves the execution of a Java program under instrumentation and several sub-processes of symptom analysis in our dependency analysis environment. In the trace generation portion, our system generates a trace using Java byte-code instrumentation technologies. The trace processing portion, on the other hand, stores the generated trace in a GDB and supports its efficient processing of various kinds of dependency analysis.

A. Trace Data Model

Dependency analysis approaches from various aspects of execution are necessary for practical debugging support. In previous work, we developed two kinds of dependency analysis algorithms to detect symptoms that indicate infections in a failed execution [8].

Both of the proposed algorithms process control data dependency across the entire extent of an execution. One



Nodes

Node ID	Attribute Key	Attribute Value
1	value	100
2	value	30
3	operation	"+"
	line	30
4	value	70

Relationships

Start Node ID	End Node ID	Attribute Key	Attribute Value
3	1	-	-
3	2	-	-
3	4	-	-

Figure 2. Property graph model

algorithm checks a complex condition that specifies data flow to associate operations in a class instance caused by the invocation of a certain kind of method. The other algorithm keeps track of side effects via fields and array components. We propose a new kind of dependency analysis that aims to abstract the effects of methods and operations on objects based on inputs by the debugger users.

In order to satisfy the above requirements, we defined our trace model as the following basic elements of program executions:

- Method execution
- Execution of abstracted byte code instructions to represent statements.
- Creation and reference of values by instructions.
- Values to be created or referenced.

Some abstracted instructions represent "control statements," such as conditional statements, method invocations, and throw and catch. Abstracted instructions contain assignment operations on local variables, fields, and array components. The instruction set also contains constants, instance creations, and array creations, as well as various calculation operations. Values created, calculated, and assigned are referenced by the instructions that use them.

For each instruction in an execution, its trace records the control instruction under which it is executed. If the instruction references a value, the trace records from the instruction from which the value originates. In this way, we can obtain control and data dependency information among instructions, including a method invocation structure.

A trace generated by our approach enables to first be represented using the property graph model shown in Figure 2. This is a data model defined in the TinkerPop project in Apache [16]. This data model features good descriptive capability, and hence can represent various kinds of data.

Our trace model allows programs to check data/control dependency for a large number of instructions in order to examine state changes on some objects or to find the cause of an infection. Algorithms to check such dependencies, represented by links among graph nodes, should be efficient.

B. Trace Processing

The requirements stated in Section III-A make it difficult to reduce trace size. Traces are needed not for a particular dependency analysis, but for various kinds of analysis dealing with the conditions of such program elements as classes, fields, and methods related to the four elements described in Section III-A. Therefore, rich data is required for the proposed trace model for such additional information.

For dependency analysis purposes, the instructions between, which the analysis is performed cannot be predicted. Therefore, for a failed execution, the trace of the entire extent of execution is first needed. Our algorithms then search for instructions that are the targets of dependency analysis.

Dependency analysis usually requires checking of complex conditions for the above four kinds of elements one by one along with their dependency relationships. Furthermore, the results of past condition checks must be stored for reference.

A situation sometimes arises where the Java virtual machine is quite inefficient, or even runs out of memory when applying dependency analysis to the execution of a software. Hence, data engineering approaches are needed to build a framework that enables efficient access to and processing of massive traces.

In this study, we develop a dependency analysis environment on the GDB to improve analysis performance. This paper uses a GDB called Neo4j following the property graph model [17] because it is suitable for storing traces with complex data structures. Moreover, Neo4j have considered the best for handling graph data for existing GDBs [18] [19].

In order to handle our trace, our dependency analysis environment was implemented using the native Java Application Program Interface(API) of Neo4j and its query language named Cypher.

IV. A Trace-partitioning Approach for Efficient Dependency Analysis

In Section III, we described how to store and process our trace on our dependency analysis environment using Neo4j. Our trace is expressed as graph data, which consists of nodes and edges. Therefore, the nodes and the edges unrelated to the graph data trace are not loaded into main memory when the dependency analysis is conducted. In short, memory efficiency of our dependency analysis environment is high. However, the size of the properties of a node or an edge that is loaded in main memory is large, this may likely to become a bottleneck of our dependency analysis environment. Especially, it goes double for becoming the bottleneck if the attribute is not related to dependency analysis. Therefore, we propose an approach for fixing this bottleneck.

A. Characteristics of Graph Database System

Initially, Neo4j manages graph data on hard disk drives until a query is issued. Once, a query is issued, Neo4j accesses the hard disk drive to load nodes and edges related to the query,

into the main memory. Usually, nodes and edges not related to the query are not loaded.

If the nodes and the edges are loaded into main memory, their properties are also automatically loaded. Therefore, loading the properties into main memory is not efficient if the property is not related to an issued query. Moreover, this problem becomes a factor to produce useless disk access.

B. A Naïve Trace-partitioning Approach for Memory Reduction

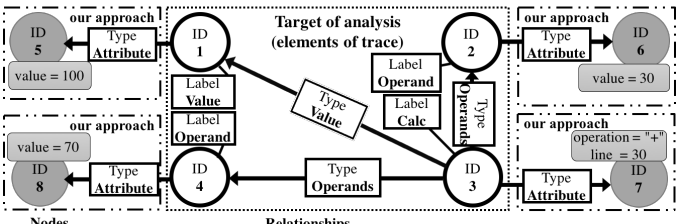
As we described in Section IV-A, loading properties unrelated to an issued query leads to the potential for memory inefficiency of GDB. Nodes and properties, which are not the target of dependency analysis, will not be loaded in the main memory while conducting the analysis. This is the characteristic of GDB, which is supported by the native graph engine.

We deal with this bottleneck by simply storing an extra node in GDB. The extra node is to store properties of node, which is the target of the dependency analysis. In this way, it is possible to load only nodes and its properties, which are targets of dependency analysis, and eliminate the unnecessary ones from the dependency analysis. We believe that this is the best way as it is more frequent to distinguish the kind of a node than to acquire properties of nodes.

We illustrate our proposed approach in Figure 3. At first, it is necessary to create a node and an edge. A node plays the role of storing properties of node (node ID is 5, 6, 7, 8 in Figure 3.), which is trace elements (node ID is 1, 2, 3, 4 in shown Figure 3.) An edge plays the role of distinguishing certain node, which is an extra node for storing properties. We describe this node as a property-node, and this edge as a property-edge in this paper. Therefore, a change of the following graph structure occurs.

1) The number of nodes stored in a graph database system doubles.
2) One edge connecting with each nodes of the trace increases.

We assume that the time required for import processing of our trace increases by 1. However, graph traversal performance is influenced by the increase of the nodes, such as 1), intended only for the node where the graph traversal is connected to a certain node on the native GDB such as Neo4j [20] [21]. Then, instead of being able to reduce the loading of the properties of a node that is unnecessary for dependency analysis, one property-edge comes to is loaded with change 2). The specifications of Neo4j have a bigger fixed-length data size of the edge than the node on the disk [20] [21]. However, we assume that a data size of edges loaded in the memory is low, because the number of edges such as references and dependencies is less than that of nodes. In addition, the time for confirming edges with the need to follow in the graph traversal by 2) increases once in all nodes and we predict that it makes the performance of graph traversal inefficient. Since our approach has a factor that can promote and not promote efficiency of dependency analysis as described above.



Nodes

Node ID	Attribut Key	Attribute Value
1	-	-
2	-	-
3	-	-
4	-	-
5	value	100
6	value	30
7	operation	"+"
	line	30
8	value	70

Relationships

Start Node ID	End Node ID	Attribute Key	Attribute Value
1	5	-	-
2	6	-	-
3	1	-	-
3	2	-	-
3	4	-	-
3	7	-	-
4	8	-	-

Figure 3. Graph partitioning approach for proposed trace.

V. PRELIMINARY EXPERIMENT

We propose a trace-partitioning approach for efficient dependency analysis in Section IV. In this section, we conduct a preliminary experiment for confirming the change of the analysis performance for an approximate application of our approach. We conduct this preliminary experiment on a kernel-based virtual machine with 64 GB RAM and the Cent OS 7 operating system.

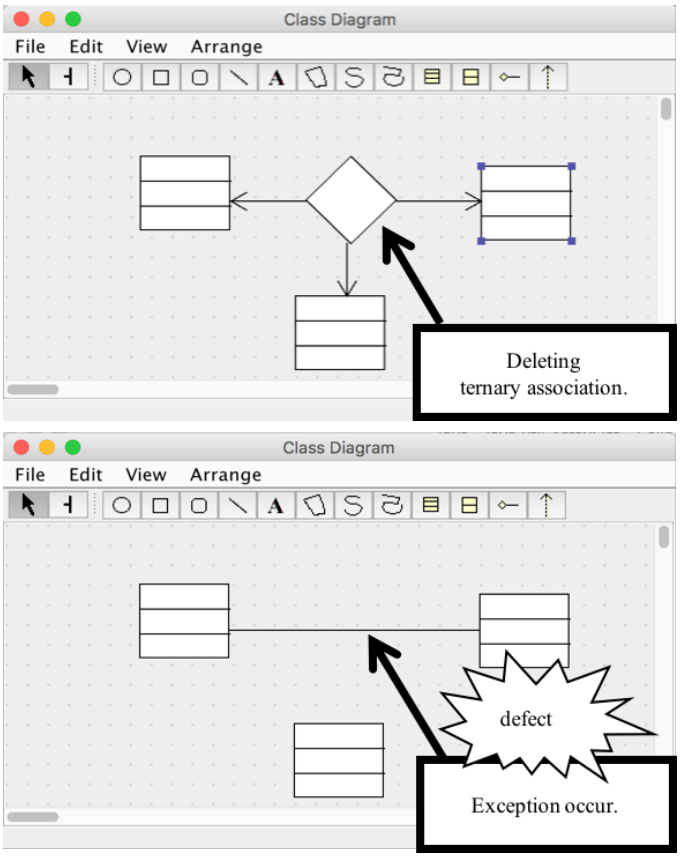
A. Unified Modeling Language Editor "GEFDemo"

We used our trace for the execution of the demonstration program on the Graph Editing Framework (GEFDemo) [9] for dependency analysis in Section V-B. GEFDemo is a simple Unified Modeling Language (UML) editor program that uses the application framework as shown in Figure 4(a). A flaw, such as in Figure 4(b), is known to occur during the delete operation, a ternary association, which is a defect in implementation of the GEFDemo.

Accurate inspection of the analysis program was possible because the cause of the defect shown in Figure 4 was manually confirmed. The trace used in this experiment recorded the execution process of GEFDemo that intentionally produced an exception, as shown in Figure 4 in the following procedure:

1) Creating three classes on the editor.
2) Creating an association for other classes from one class.
3) Creating an association for another association from the class that does not create an association.
4) A diamond object expressing the occurrence of a ternary connection occurs.
5) Deleting the diamond object.

The number of nodes in this trace was 510,370 and the number of relationships 4,437,367. Moreover, the trace into the GEFDemo contained 45 kinds of labels for nodes and 22 kinds of relationships. Furthermore, the amount of this trace was 292.63 MB.



(b) Deleting a Ternary Association.

Figure 4. Operating the GEFDemo Program

B. Outdated-state Analysis

As described in Section V-A, a defect of the GEFDemo was caused by changes in the process of execution of the program during the collection state, which is an object of Java. We used an outdated-state analysis, which is the approach of dependency analysis proposed by Kume et al. [8]. It can detect instructions that use different states of a specified object.

We executed the outdated-state analysis in a dependency analysis environment as described below:

1) Investigating method called in execution order one by one.
2) Investigating dependencies with state of objects with many instructions occurring in each method.
3) When analyzing an instrument concerning the change in the state of the object, a node was created to record the frequency of change of the object for a GDB.
4) Investigating instructions dependence on the combination of a new state and old states of the same object from nodes that we created by Procedure 3).

In Procedure 1), the outdated-state analysis consumed a large amount of memory because it was necessary to analyze instruments and values in a trace. Moreover, outdated-state analysis is a two-step process: (1) analyzing the trace, (2) creating the nodes and edges to record the status of objects (data generated during dependency analysis) on GDB in Procedure 1). Finally, it analyzes data generated in Procedure 3).

C. Measurement of Effects on Entire Dependency Analysis

In this section, we measured the method's time and memory consumption in order to evaluate the effectiveness of our approach for efficient dependency analysis in Section IV. Memory consumptions per second were recorded using vmstat, which is a UNIX command that can report information related to memory, paging, CPU activity, and so on, and can calculate the basic statistics of memory consumption. We conducted dependency analysis ten times as we described above.

Figure 5 shows the results of two trace formats as the following:

NON: a non-transformational trace
ALL: a trace partitioned properties of each node in our trace using IV

Figures 5(a) and Figure 8(b) show the average of memory consumption in the dependency analysis. In these figures, NON represents our previously approach proposed in literature [8]. ALL refers to the naïve approach proposed in Section IV and literature [12].

We conduct an independent t-test in order to confirm whether there are significant differences between average of the time consumption and average of the memory consumption. We calculate a t-value by Formula (1):

$$t = \frac{\bar{x}_{NON} - \bar{x}_{ALL}}{s\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \quad (1)$$

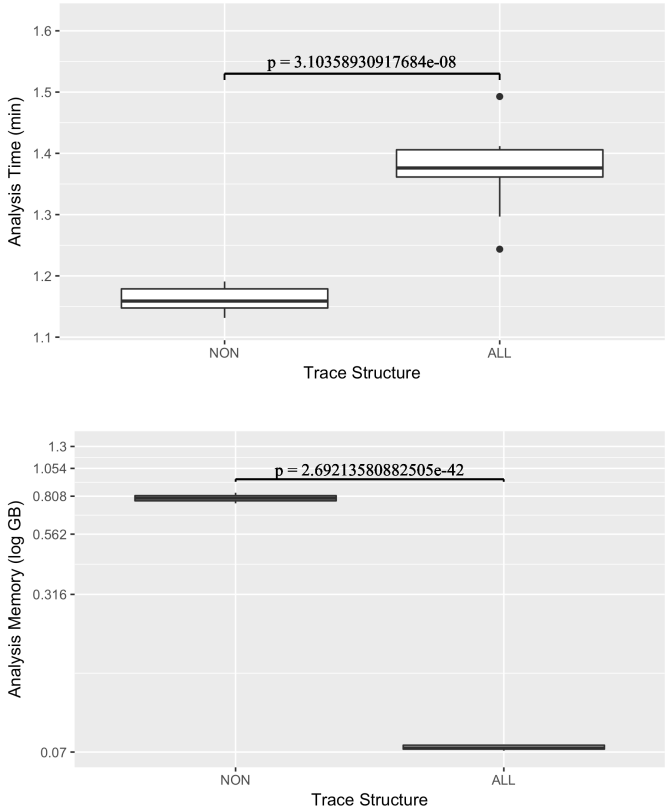
where x_{NON} denotes ten values of time consumption or memory consumption in a case of NON, $\bar{x}_{NON}$ denotes the mean of x_{NON}, x_{ALL} denotes ten values of time consumption or memory consumption in a case of ALL, $\bar{x}_{ALL}$ denotes the mean of x_{ALL}, n_{NON} and n_{ALL} are sample size of x_{NON} and x_{ALL}, and s is pooled variance with x_{NON} and x_{ALL}.

The two p-values in Figure 5 indicated that ALL could not reduce time consumption and memory consumption for dependency analysis compared with those of NON. On the contrary, our naïve approach worsen this performance.

D. Inefficient Processing in our Approach

We assume that the time required for importing a trace increases due to the above sorting 1). However, graph traversal performance is not influenced by the increase in the number of nodes; it is intended only for the node where graph traversal is connected to a certain node in Neo4j. On the other hand, instead of preventing the loading of property of a node that is unnecessary for analysis, a property-relationship is loaded with sorting 2). The fixed-length data size of edge on the Neo4j is larger than that of the node. However, we can assume that the data size of edges loaded in the memory is small because the size of a property of the edges, such as references and dependencies, is less than that of the nodes. The time needed to confirm the edges needed to traverse the graph traversal by sorting 2) increases in all nodes, and we predict that it leads to inefficient graph traversal performance.

Moreover, if it is necessary to access a property, the property-relationship is traversed during dependency analysis. Since traversing property-relationship is not necessary in the



(b) Memory consumption.

Figure 5. Dependency analysis performance.

case of an original trace, as the number of processes increases, efficiency worsens.

Furthermore, there are difference instructions referred same object, same variable, and same element of array in runtime program. In this case, the trace has many pair of nodes; one is different two node implied instructions, other is same node implied object. Then, it is inefficient to traverse same node many times during dependency analysis.

VI. Improving and Extending our Approach

The loading nodes, the edges, and their properties used for dependency analysis are very important for the efficient use of the main memory. Neo4j supported on-disk-database, which loads the properties of node and edge when the nodes and edges are loaded. However, not all of the loaded properties are used for all dependency analyses. Therefore, we focused on the selection of loading properties.

In the previous study [12], we proposed an approach for partitioning our trace that can load properties as needed. However, this did not help improve the dependency analysis performance. Therefore, we formulate a rule in this section to determine whether a given property should be loaded for a given trace in literature [1].

Moreover, we extend our approach to enable rapid access to a node, which is accessed once during dependency analysis.

Require: N_{node}, N_{attr}, N_{trav}
  for each $l \in L$ **do**
    {Not applying proposed approach to all labels of the node.}
    {Initializing $\boldsymbol{f}$ of the dictionary type.}
    {The key of $\boldsymbol{f}$ is $l \in L$, and let the value be **false**.}
    $\boldsymbol{f}[l] \leftarrow$ **false**
  end for
  for each $l \in L$ **do**
    $before \leftarrow S_{attr}(\boldsymbol{f}, N_{attr}, N_{node})$
    $\boldsymbol{f}[l] \leftarrow$ **true** {Applying our approach to l.}
    $after \leftarrow S_{attr}(\boldsymbol{f}, N_{attr}, N_{node})$
    $traversal \leftarrow S_{trav}(\boldsymbol{f}, N_{trav}, N_{node})$
    if $before > after$ **and** $traversal = 0$ **then**
      continue
    else
      $\boldsymbol{f}[l] \leftarrow$ **false** {Not applying our approach to l.}
    end if
  end for
  return $\boldsymbol{f}$

Figure 6. Optimization algorithm for the proposed approach.

A. Optimization Algorithm for our Approach

The purpose of this approach is to reduce the memory consumed by the properties of the nodes to improve the efficiency of graph traversal. However, our previous approach [12] has been unable to improve the effectiveness of traversing the proposed trace because we had not considered the situation where the properties of each node are loaded into the main memory. As a result, the previous approach made additional traversals to analyze property-relationships. The traversal of property-relationships does not occur in the original structure of the trace; hence, we propose an algorithm to automatically determine the node needed for the approach in order to avoid creating properties over and above those that are required. If a minimum number of such properties can be loaded into the main memory, the effectiveness of the proposed approach will improve.

To automatically determine the node in this approach, the analytical algorithm of our dependency analysis environment needs to be recognized. That is to say, one needs to understand that the algorithm traverses nodes and loads their properties in the trace using our approach. In this case, our approach requires knowing the number of properties loaded from all nodes, with each node labeled as N_{trav}. At the same time, it also requires knowing the number of properties denoted by N_{attr}.

However, we cannot correctly estimate N_{trav} because the dependency analysis is dynamically executed depending on the value of the property in the trace. Hence, we assume that all nodes of the trace can be traversed, and the maximum number of loading properties of nodes is N_{trav}. In short, we decide to partition the properties of node into extra node when a loading property has the potential to obtain the property of node.

We developed an algorithm for the automatic application of our approach, as stated above. This algorithm is shown in Figure 6. Given a set of labels of nodes as L, every node is labelled $l \in L$ as $N_{node}(l)$ in Figure 6, and every property

is labelled as N_{attr}. We also represent the frequency of the properties of loading nodes with label $m \in L$ when reaching label $l \in L$ of a node. Note that we take into account the identification of these labels ($l = m$).

We now introduce criteria for applying the proposed approach. S_{attr} is the sum of the number of loading properties while conducting dependency analysis, and S_{trav} is the sum of the number of traversing property-relationships. We can estimate these criteria using N_{node}, N_{attr} and N_{trav}, respectively. $S_{attr}(L)$ and $S_{trav}(L)$ can be calculated as Formula (2), Formula (3):

$$S_{attr}(L) = \sum_{l \in L} s_{attr}(l, \boldsymbol{f}[\boldsymbol{l}]) \quad (2)$$

$$where:$$

$$s_{attr}(l, \boldsymbol{f}[\boldsymbol{l}]) = \begin{cases} N_{attr}(l) \cdot N_{node}(l) & if\ \boldsymbol{f}[l] = false \\ 0 & otherwise \end{cases}$$

$$S_{trav}(L) = \sum_{l \in L} s_{trav}(l, \boldsymbol{f}) \quad (3)$$

$$where:$$

$$s_{trav}(l, \boldsymbol{f}) = \begin{cases} \sum_{m \in L} N_{trav}(l, m) \cdot N_{node}(m) & if\ \boldsymbol{f}[m] = true \\ 0 & otherwise \end{cases}$$

In Formula (2), $s_{attr}(l, \boldsymbol{f}[l])$ is calculated to multiply the number of loading properties of nodes labeled l by the number of nodes labeled l in GDB. In Formula (3), we also calculate $s_{trav}(l, \boldsymbol{f})$ to multiply the number of traversing property-relationships connected with nodes labeled m when reaching nodes labeled l. Note that the value of $s_{attr}(l, \boldsymbol{f}[l])$ is zero if the label l is applied because it does not obtain the traversal of a property-relationship.

Finally, our algorithm produces $\boldsymbol{f}$, which is a combination of whether the proposed approach is applied. This $\boldsymbol{f}$ allows for dependency analysis without traversing property-relationships and minimizes the sum of loading properties S_{attr}.

B. A Method for Reducing to Traverse Same Node

In a trace, there are many instructions, which refer to the same variables and objects. As described above, a trace have many-to-one relationships between instructions and object. For example, instructions 1 and 2 refer and use same object in Figure 7. Our dependency analysis environment traverses nodes and relationships one by one as we described in Section III-B. Thus, our environment loads the same node many times during dependency analysis when traversing many-to-one relationships. We have to solve this inefficiency of our environment as a way to keep loading nodes, which have many-to-one relationships when the node are loading to main memory in the first time.

In this paper, therefore, we solve the problem as described above by loading a pair of nodes, which have many-to-one relationship if traversed in the algorithm of dependency analysis. For example, our environment keep loading combination of instruction and object such as instruction 1 to object a, 2 to a, 3 to d, 4 to d, and 5 to d, to main memory in the case of Figure 7. However, if many-to-one relationships are not traversed in algorithm of dependency analysis, our environment does not load it to the main memory.

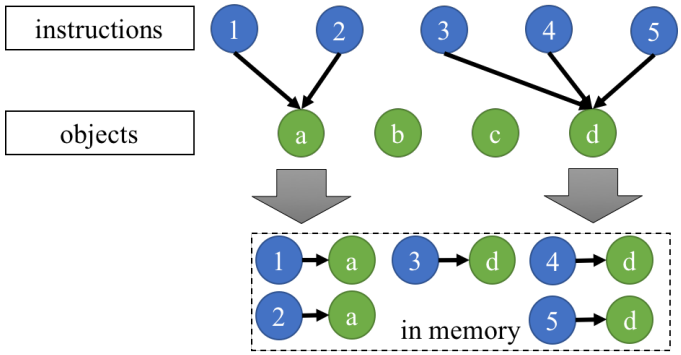


Figure 7. Target combinations of instruction and object in this approach

VII. Experimental Evaluations

As described in Section IV-B, we proposed an approach for solving the bottleneck in memory consumption in dependency analysis environments. In this section, we report an experiment to verify the effectiveness of our approach. For the assessment of macroscopic dependency analysis, not only is it necessary that memory consumption be evaluated, the time consumed for it is also a crucial factor to bear in mind. We assessed the improvement in analysis performance using the proposed approach by measuring the memory consumption and analysis time needed for dependency analysis.

We compared the experimental results with the following trace conditions:

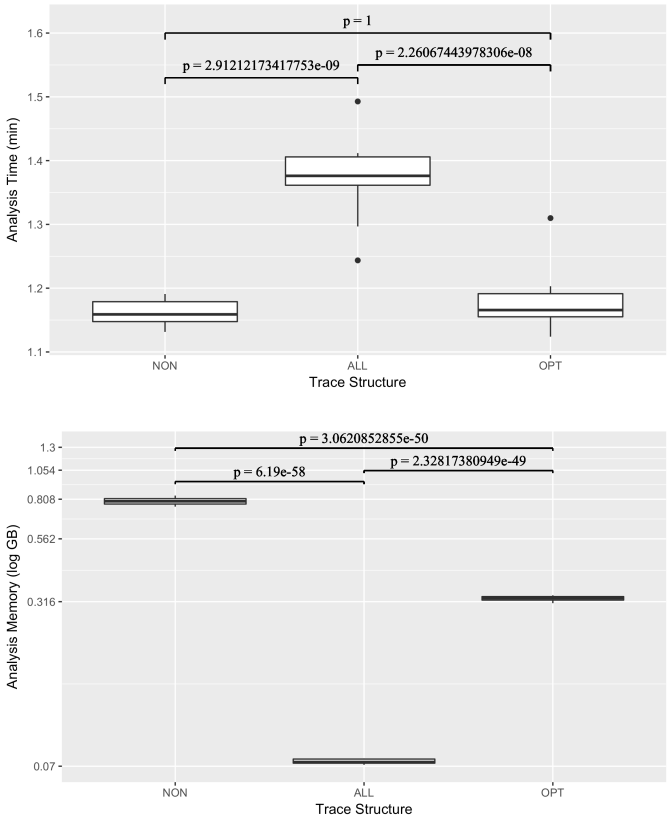
- NON: a non-transformational trace
- ALL: a trace partitioned properties of each node in our trace using our previous approach as we described in Section IV
- OPT: a trace employed partitioning approach for a few nodes selected by the rule as we described in Section VI-A.

We conducted experimental evaluations on same machine in Section V.

A. Comparing our Approach with our Previous Approach

We conducted same experimental evaluation with NON, ALL, and OPT in Section VII-A.

As a result, our approach employed proposed rules in Section VI-A labeled OPT worsen the memory efficiency compared with naïve approach for partitioning property of all nodes labeled ALL as suspected. However, OPT enables to massively reduce 43.1% of memory consumption compared with original trace format as NON. The six p-values in Figure 8 indicated that OPT could reduce time consumption and memory consumption of dependency analysis compared with those of ALL; however, we could not find any difference in traversal times for dependency analysis. In short, OPT can conduct dependency analysis with the same efficiency



(b) Memory consumption.

Figure 8. Dependency analysis performance compared with NON, ALL, and OPT.

as NON but consumes less memory using Figure 6. On the other hand, ALL could not conduct dependency analysis with the same efficiency and memory consumption as NON and OPT. Therefore, it can be concluded that Figure 6 can help considerably to improve memory consumption for dependency analysis with the same efficiency as NON.

B. Evaluation for Scalability and Analysis Speed

In this experiment, we evaluate the scalability and analysis speed of our approach as described in Section VI. We prepare four programs of Ashes2 [22] as the following list:

- BiSort: a program to conduct Bitonic Sort [23]
- Em3d: an integrated software application designed to facilitate the analysis and visualization of electron microscope tomography data [24]
- MST: a program to find Minimum Spanning Tree
- TreeAdd: a program to recursively traverse a tree by depth-first

These programs enable us to set a few options for adjusting the amount of calculation. We set up options of each program as shown TABLE I in order to prepare traces in various data amount. In this experiment, we conduct dependency analysis with six situations; this is combination of three types trace format and two cases whether or not our approach is employed. We label six situations as "NON non-approach", "NON approach", "ALL non-approach", "ALL approach", "OPT non-approach", and "OPT approach". We conducted dependency analysis 20 times with these traces in TABLE I.

TABLE I. A LIST OF PROGRAMS IN ASHES2

program	options	data amount [MB]
BiSort	-s 0025	2.0
	-s 0100	12.0
	-s 0250	31.0
	-s 0400	68.0
	-s 0550	141.0
	-s 0700	150.0
	-s 0850	163.0
Em3d	-n 0050 -d 005	23.0
	-n 0100 -d 005	46.0
	-n 0150 -d 005	69.0
	-n 0200 -d 005	93.0
	-n 0250 -d 005	117.0
	-n 0300 -d 005	141.0
	-n 0350 -d 005	166.0
Mst	-v 0016	7.0
	-v 0024	15.0
	-v 0032	28.0
	-v 0040	44.0
	-v 0048	63.0
	-v 0064	113.0
TreeAdd	-l 05	0.3
	-l 10	7.6
	-l 11	16.0
	-l 12	33.0
	-l 13	69.0
	-l 14	142.0

We show the result of this experiment TABLE II, TABLE III, and TABLE IV. TABLE II shows the averages of time consumption in the case of a trace formatted NON, TABLE III shows one in the case of a trace formatted ALL, and TABLE IV shows one in the case of a trace formatted OPT.

Moreover, we conducted a paired t-test with the result of experimental evaluation as shown TABLE II, TABLE III, and TABLE IV. We calculated a t-value by Formula (4):

$$t = \frac{\bar{d} - \mu}{\frac{s}{\sqrt{n}}} \tag{4}$$

where $\bar{d}$ denotes the mean of differences among two samples, μ denotes the population mean value, s denotes variance of d, and n denotes a sample size, in short, $n = 20$.

As a result, our approach labeled "OPT approach" enables reduce nine seconds on average compared with the situation labeled "NON non-approach". The sum of time consumptions in the situation labeled "OPT approach" was 4.3% lower than the situation labeled "NON non-approach". Moreover, the result of paired t-test shows there is significant difference of time consumption between "NON non-approach" and "OPT approach" as shown TABLE V. Furthermore, the situation labeled "OPT approach" has the best performance in all situations.

Then, we draw two line graphs as shown Figure 9(a) and Figure 9(b). Figure 9(a) shows the result in the situation labeled "NON non-approach", and Figure 9(b) shows the result in the situation labeled "OPT approach". As shown Figure 9(a) and Figure 9(b), we can reduce the time consumed for processing dependency analysis. However, in every situation, there is big difference of the time consumption for processing of dependency analysis even if there is a the difference in the trace.

TABLE II. THE AVERAGE OF ANALYSIS TIME WITH NON

trace	NON non-approach mean (± SD) [msec.]	NON approach mean (± SD) [msec.]
bisort-s0025	0.17 (± 0.03)	0.17 (± 0.03)
bisort-s0250	0.96 (±0.04)	0.93 (± 0.03)
bisort-s0550	3.85 (±0.12)	3.81 (± 0.09)
bisort-s0700	4.03 (±0.12)	3.93 (± 0.11)
bisort-s0850	4.15 (± 0.09)	4.02 (± 0.10)
em3d-n0050d005	0.96 (± 0.03)	0.93 (± 0.04)
em3d-n0100d005	2.17 (± 0.16)	2.05 (± 0.13)
em3d-n0150d005	3.78 (± 0.34)	3.69 (± 0.22)
em3d-n0200d005	6.39 (± 0.55)	5.99 (± 0.47)
em3d-n0250d005	9.59 (± 0.40)	9.22 (± 0.45)
em3d-n0300d005	12.46 (± 0.67)	12.44 (± 0.61)
em3d-n0350d005	16.72 (± 0.59)	16.74 (± 0.80)
mst-v0016	0.48 (± 0.02)	0.45 (± 0.05)
mst-v0024	0.86 (± 0.03)	0.85 (± 0.03)
mst-v0032	1.59 (± 0.09)	1.51 (± 0.05)
mst-v0040	2.65 (± 0.08)	2.56 (± 0.12)
mst-v0048	4.21 (± 0.09)	4.04 (± 0.09)
mst-v0064	9.80 (± 0.37)	9.89 (± 0.26)
treeadd-l05	0.12 (± 0.01)	0.11 (± 0.01)
treeadd-l10	0.47 (± 0.03)	0.39 (± 0.04)
treeadd-l11	0.70 (± 0.02)	0.68 (± 0.03)
treeadd-l12	1.14 (± 0.03)	1.14 (± 0.06)
treeadd-l13	2.14 (± 0.08)	2.15 (± 0.09)
treeadd-l14	4.33 (± 0.21)	4.27 (± 0.12)

SD: standard deviation

TABLE III. THE RESULT OF ANALYSIS TIME WITH ALL

trace	ALL non-approach mean (± SD) [msec.]	ALL approach mean (± SD) [msec.]
bisort-s0025	0.08 (± 0.01)	0.09 (± 0.01)
bisort-s0250	0.97 (± 0.03)	0.95 (± 0.05)
bisort-s0550	4.11 (± 0.17)	4.10 (± 0.10)
bisort-s0700	4.51 (± 0.17)	4.37 (± 0.15)
bisort-s0850	4.69 (± 0.14)	4.68 (± 0.23)
em3d-n0050d005	0.90 (± 0.06)	0.99 (± 0.05)
em3d-n0100d005	2.19 (± 0.13)	2.18 (± 0.15)
em3d-n0150d005	3.90 (± 0.23)	3.84 (± 0.24)
em3d-n0200d005	6.73 (± 0.63)	6.56 (± 0.47)
em3d-n0250d005	9.71 (± 0.60)	9.73 (± 0.52)
em3d-n0300d005	13.41 (± 0.63)	12.91 (± 0.68)
em3d-n0350d005	17.49 (± 0.71)	16.69 (± 0.95)
mst-v0016	0.32 (± 0.01)	0.32 (± 0.01)
mst-v0024	0.76 (± 0.04)	0.74 (± 0.04)
mst-v0032	1.59 (± 0.07)	1.56 (± 0.05)
mst-v0040	2.84 (± 0.14)	2.78 (± 0.17)
mst-v0048	4.53 (± 0.15)	4.39 (± 0.13)
mst-v0064	10.39 (± 0.22)	10.37 (± 0.22)
treeadd-l05	0.02 (± 0.00)	0.02 (± 0.00)
treeadd-l10	0.30 (± 0.01)	0.30 (± 0.02)
treeadd-l11	0.63 (± 0.04)	0.62 (± 0.04)
treeadd-l12	1.23 (± 0.06)	1.17 (± 0.05)
treeadd-l13	2.33 (± 0.09)	2.30 (± 0.08)
treeadd-l14	4.87 (± 0.09)	4.80 (± 0.13)

VIII. CONSIDERATION

In this section, we consider the performance and the scalability of our approach. Our approach labeled "OPT approach" is the best performance in six situations from the aspect of the memory consumption and the time consumption for processing dependency analysis. Our approach in Section VI-A avoids loading properties of the accessed node, leading to reduced memory consumption for processing dependency analysis. However, we cannot reduce time consumption for processing dependency analysis because it is less time-consuming to access a property of node. On the other hand, our expanded approach as described in Section VI-B enables to reduce the time consumption for processing dependency analysis. The reason is that accessing to main memory is more efficient than accessing to GDB after all.

However, our approach was better but by no means great. Our approach enabled to improve the efficiency of graph traverse, but our approach did not reduce the number of graph traversal during dependency analysis. Moreover, our approach did not account for the difference of characteristic of program. Therefore, it led to a major difference in time consumed even if traces are same amount as described in Section . In order to close the difference among different traces, we have to take in to account the structure of programs.

TABLE IV. THE RESULT OF ANALYSIS TIME WITH OPT

trace	OPT non-approach mean (± SD) [msec.]	OPT approach mean (± SD) [msec.]
bisort-s0025	0.07 (± 0.00)	0.07 (± 0.01)
bisort-s0250	0.79 (± 0.05)	0.76 (± 0.05)
bisort-s0550	3.88 (± 0.12)	3.81 (± 0.12)
bisort-s0700	4.14 (± 0.13)	3.90 (± 0.18)
bisort-s0850	4.14 (± 0.13)	4.09 (± 0.17)
em3d-n0050d005	0.73 (± 0.06)	0.71 (± 0.06)
em3d-n0100d005	2.06 (± 0.16)	2.02 (± 0.15)
em3d-n0150d005	3.86 (± 0.25)	3.63 (± 0.33)
em3d-n0200d005	6.18 (± 0.51)	6.11 (± 0.45)
em3d-n0250d005	9.85 (± 0.53)	9.48 (± 0.54)
em3d-n0300d005	12.38 (± 0.90)	12.40 (± 0.90)
em3d-n0350d005	16.49 (± 0.81)	16.19 (± 0.64)
mst-v0016	0.29 (± 0.01)	0.28 (± 0.01)
mst-v0024	0.65 (± 0.04)	0.63 (± 0.03)
mst-v0032	1.36 (± 0.05)	1.32 (± 0.05)
mst-v0040	2.52 (± 0.07)	2.53 (± 0.09)
mst-v0048	4.09 (± 0.10)	4.09 (± 0.22)
mst-v0064	9.79 (± 0.22)	9.72 (± 0.23)
treeadd-l05	0.02 (± 0.00)	0.02 (± 0.00)
treeadd-l10	0.27 (± 0.02)	0.26 (± 0.01)
treeadd-l11	0.50 (± 0.01)	0.49 (± 0.03)
treeadd-l12	0.99 (± 0.03)	0.95 (± 0.04)
treeadd-l13	2.11 (± 0.07)	2.09 (± 0.10)
treeadd-l14	4.47 (± 0.10)	4.37 (± 0.09)

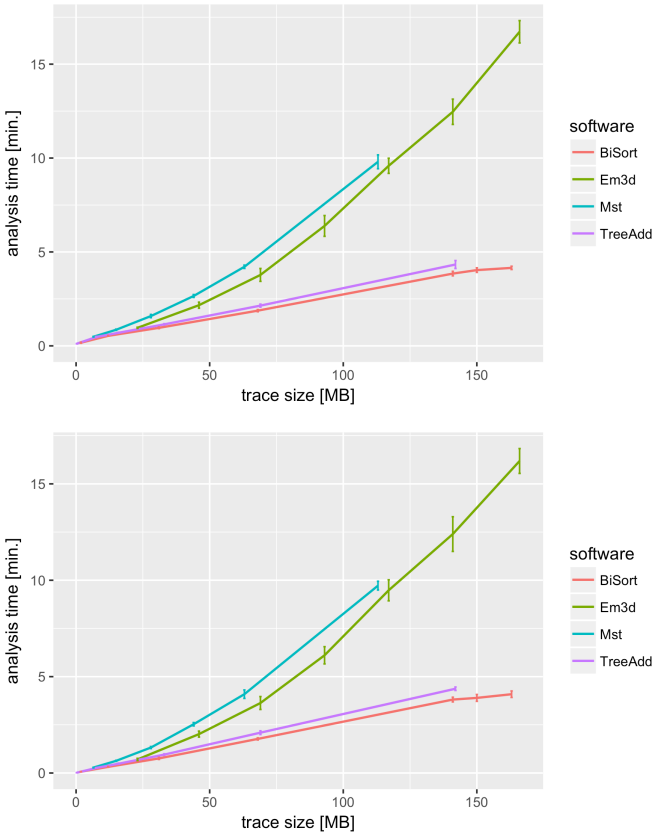
TABLE V. THE MEAN OF DIFFERENCE CALCULATED BY PAIRED T-TEST [SEC.]

	2)	3)	4)	5)	6)
1) NON non-approach	** 4.41	** -11.33	* 5.66	-0.74	** 9.55
2) NON approach	–	** -15.73	-5.15	1.25	** 5.14
3) ALL non-approach	–	–	* 10.59	** 16.99	** 20.88
4) ALL approach	–	–	–	6.40	10.29
5) OPT non-approach	–	–	–	–	** 3.89
6) OPT approach	–	–	–	–	–

*: five percent level of significance
**: one percent level of significance

IX. CONCLUSION

In this paper, we developed a prototype dependency analysis environment for efficient dependency analysis of large traces using complex graph structures. Our analysis environment is built on a graph database system that can efficiently traverse large and complex graph data. For efficient dependency analysis, moreover, we proposed trace partitioning based on the graph structure, and introduced a policy to restrict the number of loading operations on a node's properties to the main memory in order to improve the effect of our approach.



(b) Our approach with trace formatted OPT

Figure 9. Dependency analysis performance.

Furthermore, we proposed an approach to reduce loading the same node to main memory during dependency analysis.

In an experimental evaluation, our approach performed best in all situations. Our approach reduced time consumption 4.3% for processing dependency analysis compared to techniques not employing our approach.

In future work, we will account for the difference of program characteristics such as the number of each type of instruction, the number of each type of objects, and programming patterns.

ACKNOWLEDGMENT

This work was partially supported by a MEXT-Supported Program for the Strategic Research Foundation at Private Universities (#S1411030), JSPS KAKENHI [Grant-in-Aid for Challenging Exploratory Research (#15K12009), and Scientific Research (B) (#26280115)], the Artificial Intelligence Research Promotion Foundation, and the Kayamori Foundation of Informational Science Advancement.

REFERENCES

[1] K. Kusu, I. Kume, and K. Hatano, "A node access frequency based graph partitioning technique for efficient dynamic dependency analysis," in Proceedings of The Ninth International Conferences on Advances in Multimedia, 2017, pp. 73 – 78.

[2] A. Zeller, Why Programs Fail, Second Edition: A Guide to Systematic Debugging. Morgan Kaufmann, 2009.

[3] M. Weiser, "Program slicing," in International Conference on Software Engineering. IEEE, 1981, pp. 439–449.

[4] J. Ressia, A. Bergel, and O. Nierstrasz, "Object-centric debugging," in International Conference on Software Engineering. IEEE, 2012, pp. 485–495.

[5] B. Lewis, "Debugging backwards in time," in Proceedings of the Fifth International Workshop on Automated Debugging, 2003, pp. 225–235.

[6] A. J. Ko and B. A. Myers, "Designing the whyline: a debugging interface for asking questions about program behavior," in SIGCHI Conference on Human Factors in Computing Systems. ACM, 2004, pp. 151–158.

[7] A. Lienhard, T. Gîrba, and O. Nierstrasz, Practical Object-Oriented Back-in-Time Debugging. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 592–615.

[8] I. Kume, M. Nakamura, N. Nitta, and E. Shibayama, "A Case Study of Dynamic Analysis to Locate Unexpected Side Effects Inside of Frameworks," International Journal of Software Innovation, vol. 3, no. 3, 2015, pp. 26–40.

[9] "gefdemo project," http://gefdemo.tigris.org/, [retrieved: 1 Mar. 2017].

[10] T. Wang and A. Roychoudhury, "Using compressed bytecode traces for slicing java programs," in International Conference on Software Engineering. IEEE, 2004, pp. 512–521.

[11] J. Hogg, "Islands: Aliasing protection in object-oriented languages," in OOPSLA, 1991, pp. 271–285.

[12] ——, "A trace partitioning approach for efficient trace analysis," in Proceedings of the 4th International Conference on Applied Computing & Information Technology, 2016 4th Intl Conf on Applied Computing and Information Technology / 3rd Intl Conf on Computational Science/Intelligence and Applied Informatics / 1st Intl Conf on Big Data, Cloud Computing, Data Science & Engineering, 2016, pp. 133 – 140.

[13] A. Lienhard, Dynamic Object Flow Analysis. Lulu.com, 2008.

[14] A. Lienhard, T. Gîrba, O. Greevy, and O. Nierstrasz, "Exposing side effects in execution traces," in International Workshop on Program Comprehension through Dynamic Analysis, 2007, pp. 11–17.

[15] D. J. Agans, Debugging: the 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems. AMACOM, 2002.

[16] "The property graph model," http://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model, [retrieved: March 2017].

[17] "Graph database neo4j," http://neo4j.com/, [retrieved: 1 Mar. 2017].

[18] V. Kolomičenko, M. Svoboda, and I. H. Mlýnková, "Experimental comparison of graph databases," in Proceedings of International Conference on Information Integration and Web-based Applications & Services, ser. IIWAS '13. ACM, 2013, pp. 115:115–115:124.

[19] S. Jouili and V. Vansteenberghe, "An empirical comparison of graph databases," in Proceedings of the 2013 International Conference on Social Computing, ser. SOCIALCOM '13. IEEE Computer Society, 2013, pp. 708–715.

[20] I. Robinson, J. Webber, and E. Eifrem, Graph Databases. O'Reilly Media, Inc., 2015.

[21] S. Raj, Neo4J High Performance. Packt Publishing, 2015.

[22] "Benchmark programs named ashes2," http://www.sable.mcgill.ca/~bdufou1/ashes2/, [retrieved: September 2017].

[23] K. E. Batcher, "Sorting networks and their applications," in Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, ser. AFIPS'68. ACM, 1968, pp. 307–314.

[24] "Em3d," http://em3d.stanford.edu/about.html, [retrieved: September 2017].