# A Node Access Frequency based Graph Partitioning Technique for Efficient Dynamic Dependency Analysis

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 (simply "traces" for short) contain data/control dependency information, and are indispensable to novel kinds of debugging such as back-in-time debugging. However, traces easily become large and complicated. For a practical use, maintainers need to be able to interactively invoke an analysis process when required and obtain rapid feedback. To this end, the authors develop an approach for efficient macroscopic analysis of traces of large sizes with complex data structures. We propose an approach that involves storing graphs in a database that reduces the number of attributes in the main memory during dependency analysis. We also introduce a criterion for the application of this approach that can maximize its effectiveness. Finally, we conduct experiments to assess its effectiveness for efficient dependency analysis.**

***Keywords–Dynamic 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 [1] [2]. 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 [3].

In the last decade, the so-called *Back-in-time debuggers* have emerged as a new kind of debugging supporting tools. These debuggers use traces containing dependency information [4]–[6]. Such debuggers analyze dependencies to determine the operation that assigns value to a referenced variable [4], to examine the reasons for why a given statement is or is not executed [5], and what happens during the execution of a method that has already been successfully invoked [6]. 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 [3]. 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 [7], we demonstrated two kinds of dynamic dependency analysis (simply called dependency analysis in this paper) that detect symptoms of an infection caused by defects in the application of the Java framework application [8].

Although our previous study has raised the prospect of a solution to the scalability problem, yet implementation of our dependency analysis remains inefficient. The main cause of the inefficiency is the richness of data in the model of our traces. The design of our trace proposed here aims not only at the requirements of symptom detection [7], 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. [9].

In addition to currently studied Back-In-Time Debuggers [4]–[6], which aim at a microscopic perspective for the dependency analysis of a specific statement, our previous study [7] 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 [10]. 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.

This study implements an efficient dependency analysis environment for macroscopic dependency analysis similar to that in [7]. Hence, the bottleneck in our dependency analysis environment needs to be resolved. In our previous study [11], we had clarified a factor affecting efficiency in our dependency analysis environment and had proposed a trace-partitioning approach for it. However, our approach did not enhance the efficiency of dependency analysis. In this study, we assess the effectiveness of our proposed approach for efficient dependency analysis.

We will introduce related to dependency analysis, and describe the demands of for dynamic analysis environment in Section II. Then, in Section III, we illustrate our implementation of dynamic analysis environment that consists of a trace generation part and a trace processing part using the graph database. In Section IV, we propose a trace-partitioning approach based on graph database for efficient trace analysis. We will conduct an experiment of dynamic analysis performance for evaluating our proposed approach.

 ISBN: 978-1-61208-548-7 

## 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. It is a serious problem because defects and infections are often found in methods that have been completed before the program fails [6]. 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 [1].

Maintainers using a debugger must repeat a task to specify a breakpoint (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 [3].

Using traces for debugging support is a natural idea to overcome the above limitation in existing debuggers [4] [5] [12]. An omniscient debugger [4] examines assignment operations with set values referenced from variables. If a maintainer wants to determine why a statement has or has not been executed, Whyline [5] analyzes related dependencies and generates the results of the analysis using sophisticated Graphical User Interfaces (GUI).

Dynamic Object Flow Analysis [12] aims to understand program execution from the aspect of object references. Its area of application ranges from dependency analysis of methods for software testing [13] to performance engineering for a back-in-time debugger [6].

To the best of our knowledge, no existing dependency analysis approaches to debugging support aims at macroscopic dependency analysis except for our previous proposal [7]. An omniscient debugger deals with only the correspondence between the value of a variable and the assignment operation that has sets this value. Whyline navigates a maintainer along 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.

The above approaches to microscopic dependency analysis provide useful debugging aids. However, understanding a program from a macroscopic viewpoint is necessary for debugging [14]; 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 [7]. 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 kinds of dependency analysis of statements. Therefore, we developed two kinds of techniques for the analysis of the relevant symptom in our previous study [7]. The proposed trace was designed to execute these symptom analyses. For this reason, our trace tended to be large and complex, and usually led to inefficient processing of analysis. In order to conduct an efficient dependency analysis, an analysis environment is needed that can handle our trace.

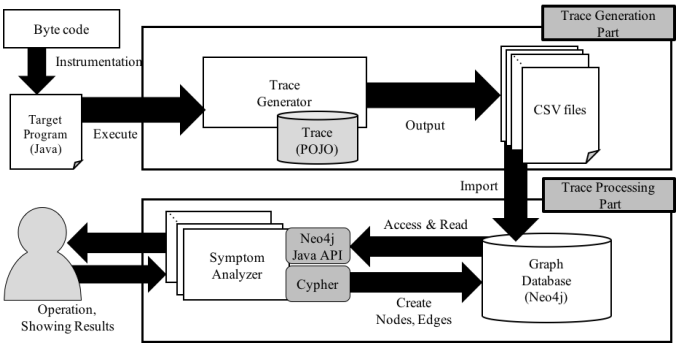



Figure 1. Dependency analysis environment.

Figure 1 illustrates the entire process, which involves the execution of a Java program under instrumentation and several sub-processes of symptom analysis in a dependency analysis environment. In trace generation, our system generates a trace using Java byte code instrumentation technology. In trace processing, on the other hand, it stores the generated trace in a graph database system (GDBS) and supports 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 [7].

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

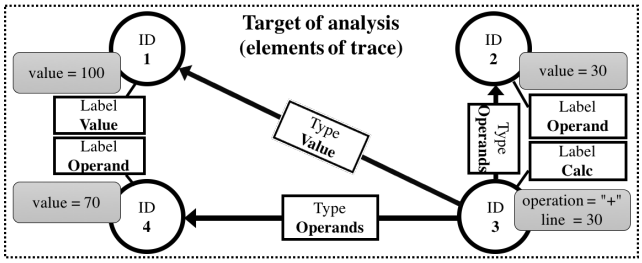
In order to meet the above requirements, our trace model defines the following basic elements of program executions:

- 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 executed instruction in an execution, its trace records the control instruction under which it is executed. If

 ISBN: 978-1-61208-548-7 




**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.

the instruction references a value, the trace records from the instruction form 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 the proposed approach can first be represented using the property graph model shown in Figure 2. This is a data model defined in the TinkerPop project in Apache [15]. This data model features good description capability, and hence can represent various kinds of data.

The proposed data 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, which is 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. The proposed 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 in applying dependency analysis to the execution of a software system. 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 GDBS to improve analysis performance. This paper adopts a GDBS called Neo4j following the property graph model [16] because it is suitable for storing traces with complex data structures. Moreover, Neo4j is considered the best for handling graph data for all GDBSs [17] [18].

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

## IV. A Trace-partitioning Approach and a Rule for Applying the Proposed Approach

The loading nodes, the edges, and their attributes used for dependency analysis are very important for the efficient use of the main memory. Our environment loads only use nodes and edges. When the nodes and edges are loaded, so are all their attributes. However, not all of the loaded attributes are used for all analyses of dependency. Therefore, this paper focuses on the selection of loading attributes.

In the previous study [11], we proposed an approach for partitioning our trace that can load attributes as needed. However, this did not help improve dependency analysis performance. Therefore, we formulate a rule in this section to determine whether a given attribute should be loaded for a given trace.

### A. Trace-partitioning Method for Memory Reduction

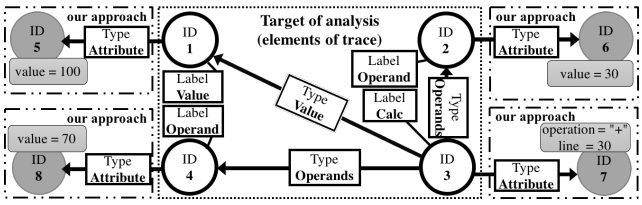
In order to cope with the problem described above, nodes in GDBS are divided into two categories in order to sort them. One category includes those nodes that are analysis targets, while another includes nodes whose attributes are analysis targets.

In this way, it is possible to load only nodes and attributes that are targets of the dependency analysis and eliminate unnecessary ones. We believe that this is the best approach, as kinds of nodes need to be distinguished more frequently than attributes of nodes in dynamic analysis.

The proposed approach is shown in Figure 3, where a node and an edge are first created. This node stores attributes (the node IDs are 5, 6, 7 and 8 in Figure 3.), which are generated for convenience of an analysis (the node IDs are 1, 2, 3 and 4 in shown Figure 3.). The edge distinguishes the nodes that are used to store attributes. The node is described as one used to store attributes and the edge as one used to access the attributes of the nodes in the trace (this edge is called an attribute relationship in this paper). Therefore, deviations from the property graph model obtain: 1) The number of nodes stored doubles in a GDBS. 2) The number of edges connecting nodes of the trace increases.

### B. Inefficient Processing in Proposed 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, 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 attributes of a node that are unnecessary for analysis, an attribute-relationship is loaded with sorting 2). The fixed-length data size of edge on the Neo4j is larger than node's one. However, we can assume that the data




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.

**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**
    $before \leftarrow S_{load}(\boldsymbol{f}, N_{attr}, N_{node})$
    $\boldsymbol{f}[l] \leftarrow$ **true** {Applying our approach to $l$.}
    $after \leftarrow S_{load}(\boldsymbol{f}, N_{attr}, N_{node})$
    $traversal \leftarrow S_{trav}(\boldsymbol{f}, N_{trav}, N_{node})$
    **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 4. Optimization algorithm for the proposed approach.

size of edges loaded in the memory is small because the size of an attribute of edges, such as references and dependencies, is less than that of a nodes. Moreover, the time needed to confirm the edges needed to traverse the graph traversal by sorting 2) increases in all nodes, and we predict that leads to inefficient graph traversal performance.

Furthermore, if it is necessary to access an attribute, the attribute-relationship is traversed during dependency analysis. Since traversing attribute-relationship is not necessary in the case of an original trace, as the number of processes increases, efficiency worsens.

### C. Optimization Algorithm for our Previous Approach

The purpose of this approach is to reduce the memory size used by attributes of nodes to improve the efficiency of graph traversal. However, our previous approach [11] has been unable to improve the effectiveness of traversing the proposed trace because we had not considered the situation where the attributes of each node are loaded into the main memory. As a result, the previous approach made additional traversals to analyze attribute relationships. The traversal of attribute 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 attributes over and above those that are required. If a minimum number of such attributes can be loaded into the main memory, the effectiveness of the proposed approach will improve.

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

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

We developed an algorithm for the automatic application of the proposed, as stated above. This algorithm is shown in Figure 4. Given a set of labels of nodes as $L$, every node is labelled $l \in L$ as $N_{node}(l)$ in Figure 4, and every attribute is labelled as $N_{attr}$. We also represent the frequency of the attributes 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. $S_{load}$ is the sum of the number of loading attributes while conducting dependency analysis, and $S_{trav}$ is the sum of the number of traversing attribute 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 (1), (2):

$$S_{attr}(L) = \sum_{l \in L} s_{attr}(l, \boldsymbol{f}[\boldsymbol{l}]) \quad (1)$$

$$where:$$

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

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

$$where:$$

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

In (1), $s_{load}(l, \boldsymbol{f}[l])$ is calculated to multiply the number of loading attributes of nodes labeled $l$ by the number of nodes labeled $l$ in GDBS. In (2), we also calculate $s_{trav}(l, \boldsymbol{f})$

to multiply the number of traversing attribute 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 an attribute 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 attribute relationships and minimizes the sum of loading attributes $S_{attr}$.

## V. Experiment

As described in Section IV-A, 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: This trace was non-transformational.
- ALL: We employed the approach for all nodes in the trace.
- OPT: We employed the approach for a few nodes selected by the rule in Section IV-C.

The experiments in this section were conducted on a kernel-based virtual machine with 64 GB RAM and the Cent OS 7 operating system.

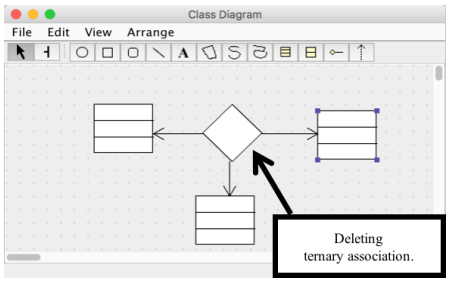
### A. Unified Modeling Language Editor "GEFDemo"

We used trace for the execution of the demonstration program on the Graph Editing Framework (GEFDemo) [8] for dependency analysis in Section V-B. GEFDemo is a simple Unified Modeling Language (UML) editor program that used the application framework as shown in Figure 5(a). A flaw, such as in Figure 5(b), is known to occur during the delete operation, a ternary association, which is a defect in implementation of the GEFDemo.

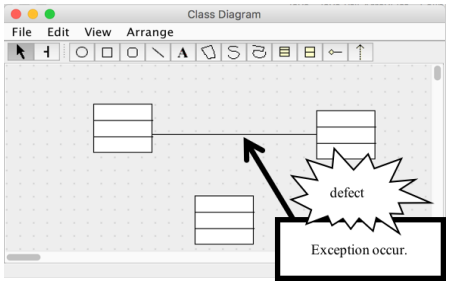
Accurate inspection of the analysis program was possible because the cause of the defect shown in Figure 5 was manually confirmed. The trace used in this experiment recorded the execution process of GEFDemo that intentionally produced an exception, as shown in Figure 5 in the following procedure:

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 46 kinds of labels for nodes and 44 kinds of relationships. Furthermore, the size of the trace was 63.8 MB as text. Hence, our trace contained a large amount of




(a) Creating three Classes and a Ternary Association.




(b) Deleting a Ternary Association.

Figure 5. Operating the GEFDemo Program

information about the runtime state of the program. However, it can easily become large and complex.

### B. Outdated-state Analysis

As described in Section V-A, a defect of the GEFDemo is 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. [7]. 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:

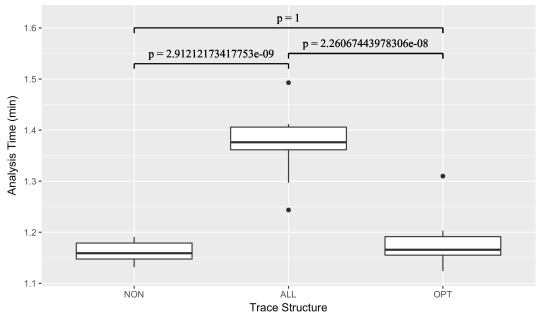
1) Investigating method called in execution order one by one.
2) Investigating dependencies with state of objects with many instructions occurring in each method.
3) When analyzing an instrument concerning the change in the state of the object, a node was created to record the frequency of change of the object for a GDBS.
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 the database in Procedure 1). Finally, it analyzes data generated using Procedure 3).

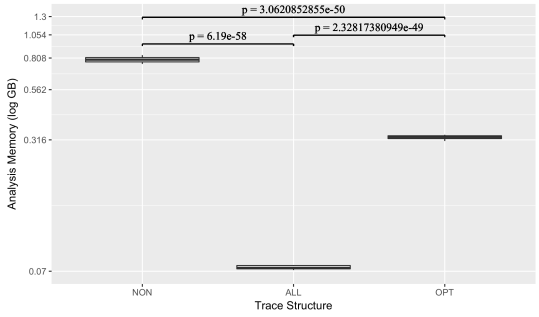
### C. Measurement of Effects on Entire Dependency Analysis

In order to evaluate the effectiveness of the approach to dependency analysis proposed in this paper, we measured

 ISBN: 978-1-61208-548-7 




(a) Traversal time.




(b) Memory consumption.

Figure 6. Dependency analysis performance.

its processing time and memory consumption. 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.

Figure 6 shows the results of three approaches. Figures 6(a) and Figure 6(b) show the average value of 10 traversals and instances of memory consumption in the dependency analysis, respectively. In these figures, NON represents our previously approach proposed in [7]. ALL refers to the naïve approach proposed in [11], and OPT represents the approach in this paper.

The six p-values in Figure 6 indicated that OPT could reduce processing time 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 as NON but consumes less memory using Figure 4. 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 4 can help considerably improve memory consumption for dependency analysis with the same efficiency as NON.

## VI. Conclusion

This paper 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, we introduced a policy to restrict the number of loading operations on node's attributes to the main memory in order to prevent it from being occupied by unnecessary data.

We applied this approach to a trace dealing with dependency across macroscopic program execution. In this experiment, the proposed approach yielded good performance in terms of analysis time and memory consumption during dependency analysis.

## Acknowledgment

This work was partially supported by a grant-in-aid from the Science and Engineering Research Institute (the Harris Science Research Institute) of Doshisha University, MEXT/JSPS KAKENHI [Grant-in-Aid for Challenging Exploratory Research (No. 15K12009), and Scientific Research (B) (No. 26280115)], the Artificial Intelligence Research Promotion Foundation, and the Kayamori Foundation of Informational Science Advancement.

## References

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

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

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

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

[5] 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.

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

[7] 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.

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

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

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

[11] K. Kusu, I. Kume, and K. Hatano, "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.

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

[13] 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.

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

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

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

[17] 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.

[18] 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.

 ISBN: 978-1-61208-548-7 