PAPER

Comparison of Backward Slicing Techniques for Java

Yu KASHIMA[†a)], ***Nonmember***, **Takashi ISHIO**[†b)], ***Member***, ***and*** **Katsuro INOUE**[†c)], ***Fellow***

SUMMARY Program slicing is an important approach for debugging, program comprehension, impact analysis, etc. There are various program slicing techniques ranging from the lightweight to the more accurate but heavyweight. Comparative analyses are important for selecting the most appropriate technique. This paper presents a comparative study of four backward program slicing techniques for Java. The results show the scalability and precision of these techniques. We develop guidelines that indicate which slicing techniques are appropriate for different situations, based on the results.

key words: *Program slicing, Static execution before, Improved slicing*

1. Introduction

Program Slicing [1] is an analysis technique developers use to extract statements related to a specific behavior of interest. In particular, program slicing extracts a set of statements that may affect the value of a variable in a developer-specified statement. The set of statements extracted from a program is called a program slice. Kusumoto et al. [2] report that program slicing is effective for debugging tasks. If a variable has an incorrect value, developers can use the respective program slice to investigate program statements that are likely to have output the incorrect value. In addition to debugging, program slicing has been adopted by several advanced analysis techniques, e.g., information-flow analysis [3] and change impact analysis [4].

Program slices should be accurate, to ensure that developers can concentrate on the smallest number of statements during their tasks. The System Dependence Graph (SDG) [5] has been proposed to represent how statements interact with one another in a program to compute a program slice. A program slice is obtained by backward traversal on an SDG from a vertex corresponding to a variable in a specified statement. Although SDG cannot determine the minimal program slice [6], several techniques have been proposed to approach it. For example, Allen et al. propose representing exception handling in SDG [7]. SDG has also been extended to represent Java language constructs [8] and data-flow via fields of objects [9]–[11]. The accuracy of SDG data-flow information depends on the underlying points-to analysis accuracy, e.g., set-based [12], [13], object sensitive [14], and hybrid context sensitive analysis [15].

Today, not only accuracy, but also scalability is important. Acharya et al. [4] combine a lightweight program slicing technique with an accurate one for change impact analysis, because the accurate technique is very time consuming. Studies [16], [17] propose Static Execute Before/After analysis to alternate program slicing. The analysis depends on only a control-flow graph, instead of a traditional SDG. Consequently, Static Execute Before/After analysis is lightweight and scalable, though less accurate than a SDG-based technique. Beszedes et al. [17] report that the technique is useful for impact analysis.

Tailoring program slicing for developers' and researchers' needs is an important issue. Java is the most popular programming language, in both open source [18] and industrial software development [19]; however, accuracy and scalability of Java program slicing techniques have not yet been investigated. Binkley et al. [20] evaluate program slicing for C/C++. They compare slices obtained with various configurations of CodeSurfer [21]. Jasz et al. [16] compare static execute before analysis and program slicing for C/C++. Beszedes et al. [17] compare static execute after analysis with forward program slicing in C/C++ and Java. They did not include a simple backward program slicing technique and static execute before analysis for Java, in the comparison. Moreover, improved slicing [11], an advanced slicing technique for Java, has not been evaluated with practical applications.

One of the key differences of Java and C/C++ is method parameters. Java only supports call-by-value parameter, while C/C++ has both call-by-value and call-by-reference parameter. As a result, a parameter cannot be used as output directly in Java. Instead of call-by-reference parameters, fields of objects in a parameter are often used. Therefore, in Java analysis, treatment of a field is more important than that in C/C++. The other key difference is a treatment of virtual method call. A method call in Java is implicitly treated as a virtual method call, while a virtual method call in C/C++ is used only if the called method is declared as virtual method. Consequently, conservative analysis to resolve frequent virtual method calls may increase the size of program slices. Furthermore, Java includes several dynamic features such as reflection. These language differences may cause the difference between the slicing results of Java and C/C++.

We compared slicing techniques through Java program

Manuscript received September 17, 2014.
Manuscript publicized October 3, 2014.

[†]The authors are with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.
a) E-mail: y-kasima@ist.osaka-u.ac.jp
b) E-mail: ishio@ist.osaka-u.ac.jp
c) E-mail: inoue@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.2014EDP7310

analysis, to answer the following research questions.

RQ1. How accurate and scalable are slicing techniques compared to each other?

RQ2. Which slicing technique is most appropriate in specific situations?

Our analysis compared four slicing techniques as follows.

Static Execute Before (SEB) (See Sect. 2.1 for details): A lightweight technique based on control-flow analysis. Given a program statement, SEB extracts statements that may be executed before an execution of the statement which is completed. This approach is proposed as a replacement of program slicing [16]. Although SEB is context sensitive and never lost dependences, this technique has potential incorrectness caused by ignoring data dependences.

Context-insensitive Slicing (CIS) (Sect. 2.2): A simple program slicing technique [20], based on control-dependence and data-dependence analysis with a SDG which data dependence edges directly connects vertices representing field read/write statements instead of making trees of fields. CIS extracts statements that may affect the execution of a given statement. CIS covers all possible data-flow paths, and therefore may include infeasible control-flow paths.

Intersection of SEB and CIS (HYB) (Sect. 2.3): An improvement of the two aforementioned techniques by combining these techniques. HYB extracts an intersection of the statements extracted by both SEB and CIS, and excludes infeasible paths from CIS. This is natural improvement but is not evaluated. Therefore, we introduced this technique for comparison.

Improved Slicing (IMP) (Sect. 2.4): A sophisticated program slicing technique for Java [11]. This is an extended version of traditional program slicing technique [5] in order to precisely analyze Java. This technique analyzes the data structure of objects and how objects are manipulated in feasible control-flow paths. It is expected to extract a more accurate program slice than the other techniques. However, it is also expected that the analysis cost is higher than the others. The actual precision and scalability of IMP in Java is unclear.

Note that our comparison focuses on strategies to represent a program as a graph, rather than the accuracy of the underlying analysis techniques, such as points-to analysis, because all four methods can use the same analysis techniques as infrastructure.

We apply the four program slicing techniques to six applications in DaCapo Benchmarks [22], in the comparison. We investigated scalability with two configurations: *Application separate from the library*, and *the whole system including the library*. We analyzed only an application with the former configuration, by approximating control-flow and data-flow information in the library. We analyzed all the control-flow and data-flow paths in both the application and the library with the latter configuration.

In addition, we investigated the scalability of IMP in detail since IMP could not analyze the whole system in the above experiment. We prepared various analysis configurations using the six applications and sub packages in java and javax packages. We measured the analysis time and the analyzability by performing SDG construction of the configurations.

The rest of the paper is organized as follows. Section 2 presents the concepts of the four program slicing techniques. Section 3 explains the implementation details of our slicing tool. Section 4 describes the experiment using the tool. Sections 5 and 6 discuss the results and the threats to validity, respectively. Section 7 explains related work. Finally, Sect. 8 describes conclusions and future work.

2. Slicing Techniques under Evaluation

This section introduces the basic ideas of the four slicing techniques compared in this study. All four techniques extract a program slice in two steps: *Graph Construction*, and *Graph Traversal*. Each technique constructs a graph representation of a target program, in the graph construction step. A slice for a given program element is extracted by graph traversal, in the graph traversal step. After a graph is constructed once, all slices can be extracted from that graph.

Table 1 shows an example used in the following subsections to explain the differences among the techniques. Its source code column shows an example program. The main method of the program is located on line 2. It calls four methods `init`, `pass`, `sum`, and `mult` in sequential order. Two of the methods, `sum` and `mult`, call the same method `foo`. Note that the `init` method creates values for the `sum` and `mult` methods. The `pass` method does nothing for the other methods.

The right columns of Table 1 shows the slicing results by each slicing technique for comparison in the following subsections. The top row of the columns shows the slicing technique. The second row shows the criteria: `y` and `z` are variables in line 5 and 6, respectively. Each cell shows whether the slice includes the corresponding line or not. If a cell includes a check mark, the slice includes the corresponding line. For example, the slice of SEB with respect to the slicing criterion `y` includes lines 3, 4, and 5 in `main`, and all lines in `init`, `pass`, `sum`, and `foo`.

2.1 SEB: Static Execute Before

SEB [16] extracts statements that may be executed before the statement of interest execution is completed. SEB uses a control-flow graph for each method and a call graph for a target application. Given a program statement in a method, SEB identifies call sites that may directly or transitively invoke the method that includes the statement. Then, SEB identifies other statements, using graph traversal on control-flow graphs. SEB also identifies statements in methods that may be invoked before the given method. SEB extracts

Table 1 Example of source code and its slicing results.

		SEB	**CIS**	**HYB**	**IMP**	**SEB**	**CIS**	**HYB**	**IMP**
Line	Source code	**Criteria : y at line 5**				**Criteria : z at line 6**			
1	`class Main{`								
2	`public static void main(String[] args) {`								
3	`A a = init();`	✓	✓	✓	✓	✓	✓	✓	✓
4	`pass();`	✓				✓			
5	`int y = sum(a);`	✓	✓	✓	✓	✓	✓	✓	
6	`int z = mult(a); }`		✓			✓	✓	✓	✓
7	`static A init() {`								
8	`A a = new A();`	✓	✓	✓	✓	✓	✓	✓	✓
9	`a.x = 1;`	✓	✓	✓	✓	✓	✓	✓	✓
10	`return a; }`	✓	✓	✓	✓	✓	✓	✓	✓
11	`static void sum(A a) {`								
12	`l = a.foo()`	✓	✓	✓	✓	✓	✓	✓	
13	`return l + l; }`	✓	✓	✓	✓	✓	✓	✓	
14	`static int mult(A a) {`								
15	`l = a.foo()`		✓			✓	✓	✓	✓
16	`return 10 * l; }`		✓			✓	✓	✓	✓
17	`static void pass() { return ; }`	✓				✓			
18	`}`								
19	`class A {`								
20	`int x;`								
21	`int foo() { return this.x } }`	✓	✓	✓	✓	✓	✓	✓	✓

statements that may affect a given statement. Hence, it can be seen as the most lightweight variant of program slicing.

For example, if `y` in the line 5 is selected to compute a slice, the result includes lines 3, 4, 5 and also lines with the methods `init`, `pass`, `sum`, and `foo`, as shown in Table 1. The methods `sum` and `foo` are included because the variable `y` on line 5 is returned from `sum`, and the method calls `foo`. While `foo` is called from both `sum` and `mult`, `mult` is not included in the result, because the call site is not executed before line 5.

2.2 CIS: Context-Insensitive Slicing

CIS extracts statements that may affect the execution of a given statement. CIS is based on the context insensitive slicing technique implemented in CodeSurfer [21] which is evaluated in the study of Binkely et.al.[20]. Similar to traditional program slicing, CIS makes SDG. The vertices represent statements in the program, and edges represent control dependence and data dependence in the program. CIS extracts a program slice with backward traversal from vertices that correspond to a selected statement.

Control dependence represents the effect of control statements (e.g., `if` statements), while control-flow represents only the order of execution sequence. Data dependence represents the def-use relationship of variable. SDG represents data flow relationships with vertices that represent formal parameters of the method and actual arguments of the method call. Parameter-in/out edges connect vertices for parameters, according to method call relationships. If a method call instruction is a virtual method call, firstly, a possibly callable methods from the call instruction are extracted using with a points-to analysis result. Next, call edges and parameter-in/out edges are drawn between the call instruction and all callable methods.

We extend SDG of the context insensitive slicing technique [21], to represent data dependence of object fields and class variables, to compute a program slice for Java. An SDG has a data dependence edge for a field access between statements *s* and *t*, if *s* writes a field of an object and *t* may read the field of the object. Points-to analysis is used to check whether the two statements may have accessed the same object or not. Similarly, a data dependence edge exists between statements *s* and *t* if *s* writes a class variable and *t* reads the class variable.

Note that these data dependence edges for fields and class variables are potentially context-insensitive because these edges ignore method call relationship. As a result, performing context-insensitive slice rather than context-sensitive slice is required for keeping soundness of the slicing result.

Figure 1 is an excerpt of an SDG representing the program in Table 1. A vertex with a label that is a method name represents a method entry vertex. A vertex with a label that is a digit represents a statement. A rectangle vertex, with a label that is a parameter name, represents a formal parameter. A data dependence edge, from line 9 to line 21, shows data dependence through field `x`. If a developer selects variable `y` on line 5, CIS would extract lines 3, 5, and 6 in the `main` method, and lines in the `sum`, `foo`, `init`, and `mult` methods. This is because vertices corresponding to those lines are reachable from the vertex corresponding to line 5. As shown in Table 1, compared to the SEB of `y`, CIS excludes the line 4 and `pass` which are executed before line 5 but do not have data/control dependence on line 5.

One of the shortcomings of CIS is that it may include infeasible control-flow paths. An inter-procedural path is feasible if a method call in the path corresponds to a method return in the path. A control-flow path through lines 12, 21,

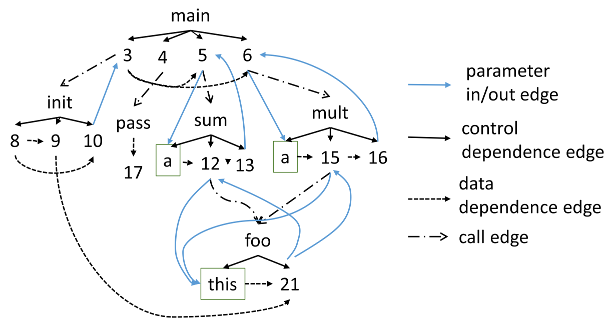

Fig. 1 SDG for CIS.

and 12 is feasible, because it starts from a method call and correctly returns to the call site. On the other hand, a path through lines 15, 21, and 12 is infeasible, because it goes to another call site. Including `mult` in the slice of CIS for `y` would be caused by traversing this infeasible path.

2.3 HYB: Context-Insensitive Slicing with Static Execution Before

HYB extracts an intersection of statements obtained by SEB and CIS. As mentioned above, CIS for `y` includes `mult` as a false positive, even though `mult` is never called before line 5. HYB combines SEB and CIS to remove such infeasible statements from the result of CIS, by computing an intersection of the SEB and CIS results.

As shown in Table 1, a slice for `y` in line 5 includes lines of `sum`, `foo`, and `init`, while it excludes pass and `mult`. The result is just an intersection of the result of SEB for `y` and that of CIS. As a result, the result of HYB is improved by either of the two techniques.

2.4 IMP: Improved Slicing

IMP [11] extracts statements that may affect the execution of a given statement, similar to CIS. For C/C++, Liang et al.[9] extended traditional program slicing [5] to handle fields of objects. IMP is a further extension of field handling for precise analysis of Java. IMP regards all fields accessed by a method as arguments of the method, instead of edges directly connecting field access between methods. This representation reflects how fields are manipulated through control-flow paths. Similar to CIS, if a method call instruction is a virtual method call, call edges and parameter-in/out edges are drawn between call instruction and all callable methods.

IMP represents fields of a parameter as a tree. Let `r.f` is a receiver object `r` and a field `f`, `p(r)` is points-to set of `r`. A vertex representing field `f` is added as a child of a vertex representing `r` if a vertex representing `r'` which points-to set `p(r')` is equal to `p(r)` does not exist in the pass from root to `r`. As a result, a tree made by IMP's approach has at least one vertex representing accessed field. Additionally, IMP's approach can handle recursive structure distinguishing their points-to sets. Whereas the existing technique [9] stops expansion of a field tree at a specific level given by a parameter in order to avoid infinite expansion of a recursive data structure, IMP uses points-to information to stop infinite expansion.

A program slice is extracted by backward two-phase slicing [5], which avoids inter-procedural infeasible paths. For two-phase slicing, summary edges are prepared. A summary edge connects an actual-in vertex to an actual-out vertex if the actual-out vertex depends on the actual-in vertex. After building summary edges, two-phase slicing is performed. The first phase of two-phase slicing perform backward traversal from a given criteria along data dependence edges, control dependence edges, call edges, summary edges, and parameter-in edges, but not along parameter-out edges. The second phase slicing perform backward traversal from all actual-out vertices visited in first phase along data dependence edges, control dependence edges, summary edges, and parameter-out edges, but not along call edges and parameter-in edges.

Figure 2 shows SDG for IMP, for the source code in Table 1, with omitted vertices that represent actual arguments and summary edges [23]. Vertices that represent parameters not only have argument/parameter variable vertices, but also have field vertices. For example, vertex 10, which is the statement vertex and also formal-out vertex of `init`, has field vertex `x`, because the variable a returned in line 10 has a field `x`. Similarly, formal-in vertices of `sum`, `mult`, and `foo` have `x` as field vertices.

Figure 3 shows the subgraph of SDG for line 5 and `sum` including omitted vertices in Fig. 2. Line 5 has the call instruction of `sum` which has argument `a` and returned value assigning to `y`. SDG in Fig. 3 has vertices of `a` and `$Actual-out` which correspond to the argument and the returned value respectively. Additionally, the SDG has a vertex `x` which represents a field `x` of the argument `a`.

A red dashed line in Fig. 3 shows a summary edge. In an execution of `sum`, parameter `a` is used as receiver object of call `foo` in line 12, and then a field `x` of `a` is used for return value of `foo`. The returned value of `foo` in line 12 is used for return value of `sum` in line 13. As a result, argument `a` and `x` has transitive dependences for return value of `sum`. Therefore, summary edges connect a vertex `a` and a vertex `x` to a `$Actual-out` vertex. Similar to the subgraph for line 5, subgraphs for lines 3, 12, and 15 have actual-in/out vertices and summary edges.

If the variable `y` on line 5 is selected, IMP performs two-phase slicing as follows: Firstly, first phase visits actual-out vertex of call `sum` via data dependence edge, and then actual-in vertices of call `sum` via summary edges, and more. As a result, first phase visits vertices of line 5 and line 3 including actual-in/out vertices. Second phase starts from actual-out vertices of line 5 and 3, and visits `sum`, `foo`, `init` via any edges except call and parameter-in edges. As a result, IMP extracts line 3, line 5, `sum`, `foo`, and `init`. Note that this result is the same as the result of HYB as shown in Table 1. HYB is effective in the case that SEB can remove infeasible path from the result of CIS.

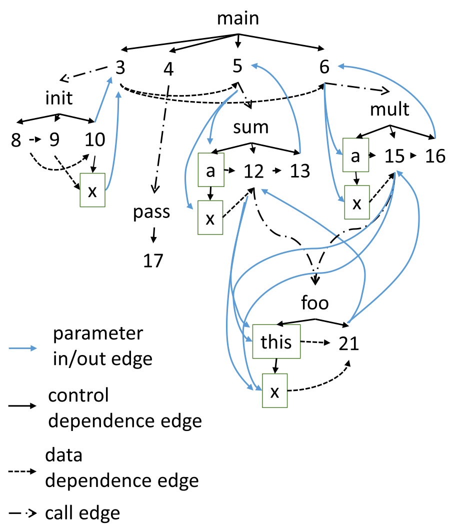

Fig. 2 SDG for IMP.

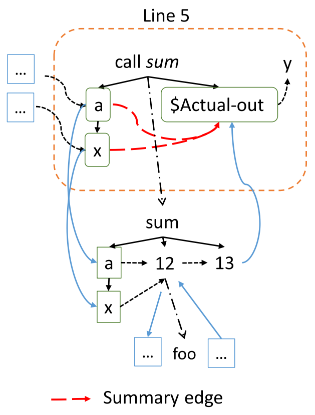

Fig. 3 Subgraph of SDG for line 5.

On the contrary, if the variable `z` on line 6 is selected, IMP extracts line 3, line 6, and `mult`, `foo`, and `init`. Table 1 shows that this is more precise than HYB, because HYB extracts `sum` in addition to the three other methods, because `sum` is executed before `mult`, and `sum` is reachable from `mult`, via an infeasible path in SDG.

2.5 Comparison of Graph Construction Process

Those four slicing techniques call both graph construction and control-flow analysis. SEB computation requires only the results of these analyses. CIS, HYB, and IMP all require additional processes, which include points-to and control/data dependence analyses. After that, SDG for CIS and HYB is constructed by making a traditional SDG and the connecting field/class variable. On the other hand, SDG for IMP is constructed by building a field tree of each argument and parameter, making the vertices and edges like in traditional SDG, and then computing the summary edges [23].

3. Implementation

We implemented four slicing techniques that target Java bytecode, because the bytecode is easier to analyze than the source code. Moreover, there are tools for points-to analysis and handling reflection targeting bytecode. We use the same points-to analysis and reflection handling process with all four slicing techniques. Therefore, all of the techniques under comparison use the same call graph and points-to information.

3.1 Points-to Analysis and Call Graph Construction

We used the Spark pointer analysis toolkit [24] with the Soot framework [25] for points-to analysis and call graph construction. Spark is implemented based on Andersen's points-to analysis algorithm [13], which is flow-insensitive, context-insensitive analysis. This context insensitive algorithm is good enough [26], because the slicing techniques under comparison do not require explicit context sensitivity, such as object sensitivity [14]. We used the default Spark configuration, the so called "on-fly-cg" and "field-sensitive", though Spark has many other options.

3.2 Handling Reflection

Reflection, which is a dynamic feature of Java, is the cause of imprecision in static program analysis. We used TamiFlex [27] to get the reflection result from a program execution. This tool can replace reflection method invocation with concrete method invocation, observed during a target program execution.

3.3 Approximation of Library

A slicing tool should analyze all methods reachable from the `main` method of the program, including the library used by the program, when computing a program slice. On the other hand, libraries may consume time and memory space for analysis, because class libraries, such as the JDK Platform API, include a large number of classes, even though most of them are not used by target programs. Additionally, some library methods are unanalyzable, e.g., native methods and methods protected by software license. We used the following approximations, to handle such unanalyzable code.

First, we assumed that for each library the method calls, the return value depends on the arguments. We connected edges from vertices representing arguments to their corresponding vertex representing the return value.

Secondly, we assumed that for collection classes (e.g., `List` and `Map`), method calls that modify a collection affect method calls that refer to the collection content. We manually listed methods such as `add` and `put` to modify a collection. We regarded other methods as those that refer to content. We translated the former method calls into statements, writing an artificial field representing collection, and the latter method calls into statements reading the field.

Finally, we excluded the `hashCode` and `equals` methods of all classes from analysis, if collection classes were not included in the analysis. This is because those methods are usually called back from collection classes.

4. Experiment

4.1 Design and Analysis Target

The goal of this experiment was to evaluate and compare scalability and accuracy of four slicing techniques. We measured the time required to construct an SDG and the size of the SDG, to analyze scalability. We measured the ratio of instructions included in a slice against the instructions in a target program, to analyze accuracy.

We analyzed six applications in DaCapo Benchmarks (version 9.12) [22]. DaCapo Benchmarks is a collection of real Java applications, with their execution scenarios. The six applications we used are avrora, batik, h2, luindex, pmd, and sunflow. Table 2 shows the size of the applications based on the number of methods that are reachable from their `main` methods. The table shows the number of classes, which includes reachable methods, the number of reachable methods, and the number of bytecode instructions in reachable methods. The columns "APP" and "LIB" show the numbers of classes/methods/instructions for application classes and library classes, respectively. We regard classes in the following packages as library classes: *java.**, *javax.**, *sun.**, *com.sun.**, *com.ibm.**, *org.xml.**, *apple.awt.**, and *com.apple.**. We obtained this list of packages from a TamiFlex document[†].

First, we executed the *default* application execution scenarios with TamiFlex, to record invoked methods by reflection. Next, Soot performed points-to analysis and call graph construction, with the output of TamiFlex. We constructed an SDG based on this information. A slicing criterion is a vertex in the SDG that corresponds to a bytecode instruction in APP. We performed backward slicing with each slicing criteria, and then measured the size of each slice.

We compared the relative slice sizes against an application, because the four techniques define graphs differently.

Table 2 Size of analysis targets.

	#Classes		#Methods		#Instructions	
	APP	LIB	APP	LIB	APP	LIB
avrora	49	1,701	290	10,697	9,690	321,666
batik	146	4,133	674	26,459	26,336	769,825
h2	130	1,741	670	11,118	18,875	331,844
luindex	74	1,705	380	10,710	11,678	322,652
pmd	139	1,712	480	10,762	13,838	322,857
sunflow	58	3,751	331	24,144	11,786	684,263

[†]`https://code.google.com/p/tamiflex/wiki/DaCapoAndSoot`

We computed slice size, as the ratio of the number of application method instructions in the slice to the number of application method instructions in the program.

We used two different configurations. The first was *Application separated from library*, which analyzes application classes with library approximation. The second was *the whole system including library* that analyzes the whole application and its libraries without approximation.

We used a machine with Windows 7 64bit, Intel Xeon E5-2620 2.00GHz 2 processor CPU, and 64GB of RAM.

4.2 Result

4.2.1 Configuration 1: Application Separated from Library

Figure 4 shows the bar charts of the time required to construct an SDG for each application. Note that HYB used the same SDG as CIS, so that those time are the same. Table 3 shows the time of each steps. The columns "Points-to Analysis (all)" and "Control-flow Analysis (all)" show the time required for points-to analysis and control-flow analysis. These analyses are common for all techniques. The column "Dependence Analysis (CIS/HYB)" shows the time required to analyze dependences for CIS and HYB, using the information obtained by the points-to and control-flow analyses. The column "Dependence Analysis (IMP)" shows the time required to analyze dependences for IMP, using the same information.

The time required for SDG construction was dependent

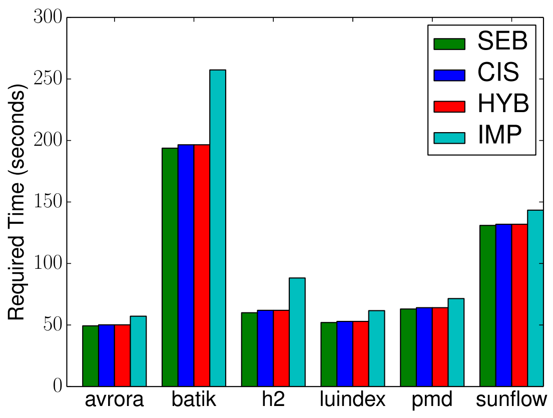

Fig. 4 Time to construct SDG in configuration 1.

Table 3 Detail of time to construct SDG in configuration 1.

	Points-to Analysis (all)	Control-Flow Analysis (all)	Dependence Analysis (CIS/HYB)	Dependence Analysis (IMP)
avrora	49.16 sec.	0.18 sec.	0.82 sec.	7.84 sec.
batik	193.49 sec.	0.25 sec.	2.75 sec.	63.59 sec.
h2	59.76 sec.	0.23 sec.	1.97 sec.	28.25 sec.
luindex	51.82 sec.	0.22 sec.	0.91 sec.	9.66 sec.
pmd	62.81 sec.	0.22 sec.	1.07 sec.	8.48 sec.
sunflow	130.83 sec.	0.13 sec.	0.94 sec.	12.40 sec.

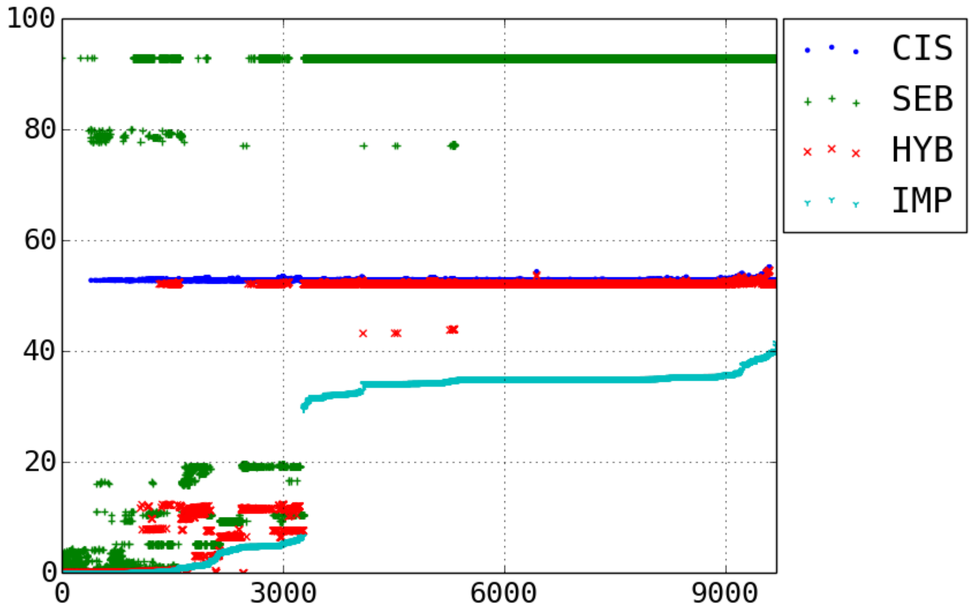

Fig. 5 Scatter plots of relative slice size (percentage) on configuration 1 of avrora.

Table 4 SDG size in configuration 1.

	CIS		IMP	
	#Vertices	#Edges	#Vertices	#Edges
avrora	14,948	32,916	43,767	178,110
batik	38,439	86,394	84,052	276,873
h2	30,291	64,538	151,738	1,054,257
luindex	18,388	39,701	47,957	181,496
pmd	22,782	48,096	50,322	175,634
sunflow	17,887	38,755	44,047	185,103

on the time required for points-to and control-flow analyses, in this configuration. CIS took only a few additional seconds, while dependence analysis of IMP took ten times longer than that of CIS. The time required was still shorter than that required for the underlying analyses. After SDG construction, all four slicing techniques under comparison were able to compute a program slice in less than one second (a few milliseconds in most cases).

Table 4 shows the number of vertices and the number of edges for each SDG. IMP usually constructs a larger graph than CIS, to more precisely represent data-flow. CIS had 44 percent of the number of vertices that IMP had and 29 percent of the number of edges that IMP had.

Figure 5 shows the distribution of slice sizes of avrora. Since the plot of the other applications show similar trends, we have picked up the one. The plots of the other applications are shown in our website†. The X-axis in Fig. 5 shows the index of slice criteria. It means that slices corresponding to the points in the same X-axis had the same criteria. The Y-axis shows the relative size of a program slice. The slices are sorted in ascending order of the relative IMP slice. The legend in Fig. 5 shows that CIS, SEB, HYB, and IMP are represented by blue, green, red, and light blue points, respectively.

IMP extracted smaller slices than the other three slicing techniques. For example, with avrora, IMP extracted less than 40 percent of the program instructions in 99 percent of the cases. IMP extracted 9.6 percent of the instructions on average. On the contrary, SEB extracted more than 80 percent of the instructions in all cases except for batik. One half of the instructions SEB extracted were likely false positives, according to IMP slice. An HYB slice was smaller than both SEB and CIS slices.

Figure 5 shows that several clusters of slices that had similar slice sizes are visible. A similar trend is reported in [28]. The phenomenon was caused by a dependence cluster [29], which is a strongly connected component in a graph. If traversal of program slicing reached an element in a dependence cluster, the traversal reached all elements in the dependence cluster.

4.2.2 Configuration 2: The Whole System Including Library

Figure 6 shows the bar charts of the time required to construct an SDG for each application. Table 5 shows the time required to construct an SDG for an application including the library. The table does not include IMP, because 64GB of memory was insufficient for IMP to construct an SDG. Points-to analysis took several minutes, and SDG construction for CIS also took several minutes. SEB was much faster than CIS for a large-scale program, because the time for SDG construction increased significantly compared with points-to analysis. Table 6 shows the number of SDG vertices and edges for CIS. The numbers of vertices and edges were, on average, 28 and 40 times larger than SDG without the library, respectively.

The maximum time to compute a slice with SEB, CIS,

†http://sel.ist.osaka-u.ac.jp/~y-kasima/compare_slice_experiment/

and HYB was 4.27, 5.13, and 8.55 seconds, respectively. The results show that a program slice can be computed for a large program, within a practical time period. HYB computation took only a few seconds longer than CIS computation, because CIS had to construct a control-flow graph.

Figure 7 shows the slice size distributions. The X-axis is the index of slices sorted by the ascending order of HYB slice size. The Y-axis shows the relative slice size. The relative slice size does not include instructions in library, because library instructions are not visible to developers. Similar to Fig. 5, we have picked up the result of avrora. The others are shown in our website.

The resultant distributions are similar to Fig. 5. Although SEB extracted a small slice in some cases, it often extracted more than 80 percent of the instructions. Slices extracted by CIS were all almost the same size due to SDG dependence clusters, while the maximum CIS slice size was lower than that of SEB. HYB extracted a significantly smaller slice in some cases, and a slightly smaller slice than CIS for most cases, because HYB was an intersection of SEB and CIS.

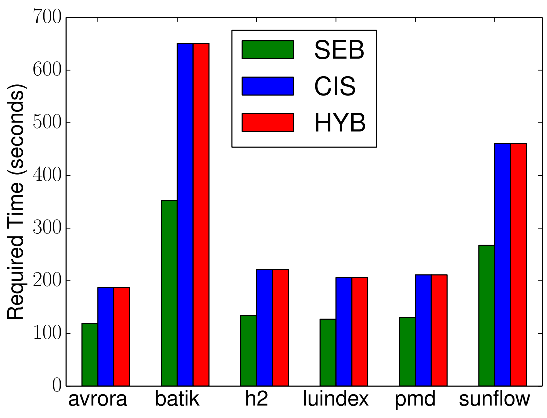

Fig. 6 Time to construct SDG in configuration 2.

4.3 Scalability Analysis of Improved Slicer

4.3.1 Design and Analysis Targets

While IMP cannot analyze an entire system as mentioned previously, it is important to know the scalability for practical use. To measure the scalability of IMP, we measured required time to construct SDG for various configurations. If the construction takes longer than a predetermined threshold (3,600 seconds), the configuration is classified as *unanalyzable*. The threshold is determined by the length of de-

Table 5 Detail of time to construct SDG in configuration 2.

	Points-to Analysis (all)	Control-Flow Analysis (all)	Dependence Analysis (CIS/HYB)
avrora	117.75 sec.	1.40 sec.	67.97 sec.
batik	349.31 sec.	3.10 sec.	298.52 sec.
h2	132.21 sec.	2.24 sec.	86.96 sec.
luindex	125.85 sec.	1.24 sec.	78.94 sec.
pmd	128.67 sec.	1.40 sec.	81.19 sec.
sunflow	264.57 sec.	2.89 sec.	193.19 sec.

Table 6 Size of SDG for CIS in configuration 2.

	#Vertices	#Edges
avrora	449,186	1,335,862
batik	1,124,286	3,927,998
h2	477,478	1,423,244
luindex	494,814	1,480,095
pmd	499,047	1,490,075
sunflow	1,033,777	3,207,852

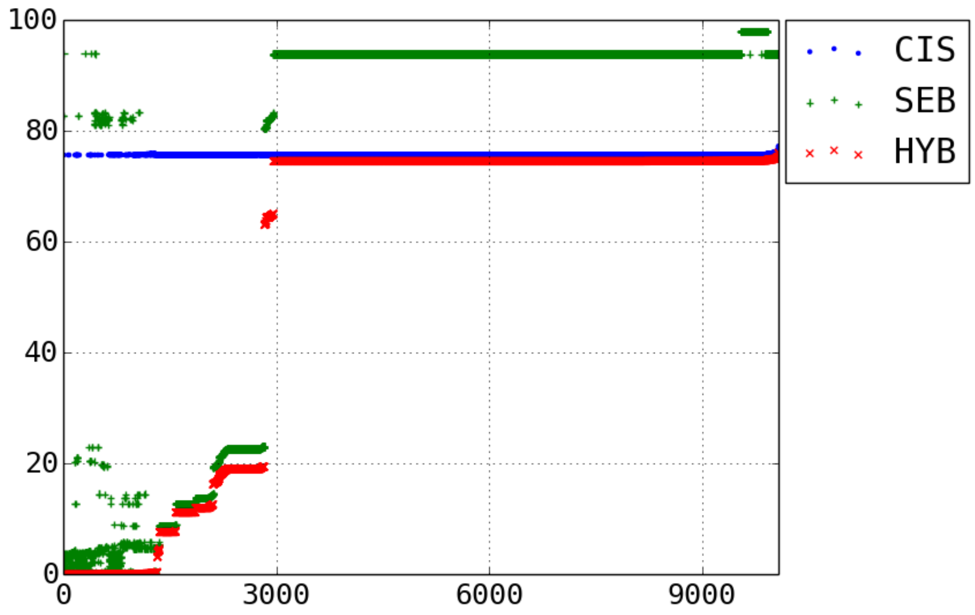

Fig. 7 Scatter plots of relative slice size (percentage) on configuration 2 of avrora.

velopers' typical daily sessions [30].

We have used the same six applications as the first experiment. We selected 14 major sub-packages in LIB which are shown in Table 7. Table 7 shows the numbers of classes, methods, and instructions reachable from a `main` method of an application. Note that the numbers in Table 7 are taken from the result in the configuration: *the whole system including library*.

We constructed configurations for each application (A) with the set of sub-packages in Table 7 (S) by the following steps.

1. For each $s \in S$, one sub-package configuration including A and s is created.
2. Select sub-packages $S' \subseteq S$ whose one-package configurations are analyzable within the time limit.
3. For each $k \in \{2, 3, \ldots, |S'|\}$, a configuration is created by randomly selecting k sub-packages from S'.

4.3.2 Result

Table 8 shows the distribution of analyzable programs and required time to prepare SDG. In order to show the difference of analyzability by the program size, we aggregate the result by the number of instructions. The interval is 10,000. The first column of Table 8 shows the range of the number of instructions. The column "#Config." shows the number of configurations whose size is included in the corresponding range. The column "#Analyzable (Percentage)" shows the number and ratio of the configurations which were analyzed within the limit time. The right most column shows the mean of required time to prepare SDG for the analyzable configuration programs. Note that all programs includes over 100,000 instructions always failed to be analyzed, so that their results are totaled in the last row.

As Table 8 shown, even if a program is small size, the analysis time may be over 3,600 seconds. The minimum configuration which was not analyzed within 3,600 seconds is avrora with java.text package that has 26,325 instructions. Any configurations including java.text were not analyzed within the time limit, partly because java.text includes a recursive data structure such as a container for text that increases analysis cost.

The minimum unanalyzable configuration including more than one sub-packages is the configuration which includes h2, java.io, java.lang, java.math and java.security. The configuration includes 75,552 instructions. Since individual sub-packages are analyzable, the configuration is classified as unanalyzable because of the scalability rather than the complexity of one of the sub-packages. Actually, most of configurations including more than 80,000 instructions were classified as unanalyzable.

On the other hand, the maximum analyzable configuration is sunflow with java.awt, javax.crypto, and javax.security packages. The configuration includes 477 classes, 3,409 methods and 98,073 instructions.

Table 8 shows that programs including less than 70,000 instructions are analyzable in the highly probabilities which is at least 83.33%. On the contrary, the success probability of an analysis is slightly falling down if a program includes more than 70,000 instructions. The ratio of analyzable programs including from 70,000 to 80,000 instructions is about 55%. In addition, in the case of more than 80,000 instructions, the percentage of analyzable programs is more smaller. Therefore, 70,000 instructions seems to be the limit of analyzable program size by IMP of our implementation.

Table 7 Sub packages used for scalability analysis.

Sub Package Name	#Classes	#Methods	#Instructions
java.awt	399	3,113	97,136
java.beans	20	192	5,792
java.io	92	805	21,089
java.lang	175	1,440	35,003
java.math	8	150	9,257
java.net	83	622	17,662
java.nio	127	623	10,156
java.security	108	482	11,400
java.sql	9	33	870
java.text	52	525	19,422
java.util	494	3,340	82,178
javax.crypto	27	166	5,365
javax.security	15	75	1,549
javax.swing	835	5,568	146,716

Table 8 The distribution of the analyzable programs and time to construct SDG.

Range of #Instructions	#Config.	#Analyzable (Percentage)	Mean of Time to Prepare SDG
[0,10k)	0	0 (0%)	NA
[10k,20k)	8	8 (100%)	113.39 sec.
[20k,30k)	28	25 (89.28%)	121.35 sec.
[30k,40k)	12	10 (83.33%)	255.02 sec.
[40k,50k)	11	10 (90.90%)	416.16 sec.
[50k,60k)	2	2 (100%)	307.81 sec.
[60k,70k)	3	3 (100%)	837.48 sec.
[70k,80k)	9	5 (55.55%)	807.05 sec.
[80k,90k)	9	1 (11.11%)	1,934.65 sec.
[90k,100k)	9	3 (33.33%)	1,055.56 sec.
[100k,230k)	14	0 (0%)	NA

5. Discussion

We review the results in terms of accuracy and scalability to answer to RQ1.

First, we discuss accuracy. SEB in many cases extracted from 80 to 100 percent of the entire program. This indicates that SEB may have been only slightly effective. CIS consistently extracted 70 percent for any instructions in a program. This was caused by large SDG dependence clusters, as previously mentioned. HYB slices were sometimes much smaller than CIS slices, partly because SEB removed instructions in dependence clusters caused by infeasible SDG control-flow paths. IMP extracted smaller slices than the other techniques; however, it could not compute a slice for the entire program.

Table 9 shows the median relative slice sizes among the slicing techniques. IMP extracted 22 percent of the in-

Table 9 Rates compared other techniques.

	Configuration1	Configuration 2
HYB / SEB	0.42	0.75
HYB / CIS	0.99	0.99
IMP / SEB	0.22	-
IMP / HYB	0.69	-

structions that SEB extracted and 69 percent of the instructions that HYB extracted. Therefore, IMP would be the best choice of the four slicing techniques when using the library approximation.

Binkley et al. [20] report the average backward slice size as 28.1 percent of the program, while IMP in this study had an average of nine percent. IMP might have achieved this improvement with more accurate field representations. In [20], they used another representations proposed by Liang et al. [9], the disadvantages of which are discussed compared with IMP in [11]. Differences between C/C++ and Java might also have affected the results. C/C++ programs can access arbitrary memory locations with pointers, while Java programs use only objects and fields to access memory locations. This difference could make Java program analysis easier than C/C++ analysis.

Second, we discuss scalability. IMP cannot analyze an entire system, as mentioned previously. IMP may construct a large tree of fields for an argument, because a field often contains another object. In the worst case, the size of a tree is estimated as $O(f^p)$, where f is the number of fields in an object and p is the number of object allocation sites. In other words, the number of vertices may increase exponentially according to program size. Actually, scalability anlaysis in Sect. 4.3 shows following two findings:

- If an analysis target program includes a container library, failure probability of the analysis will increase.
- 70,000 instructions is the indication of analyzable program size of IMP. If a program includes more than 70,000 instructions, the success probability of analysis will be falling down. Moreover, 98,073 instructions is the limit size for analysis by IMP. The limit is large enough to analyze various programs in the DaCapo Benchmarks without library, while it is insufficient to analyze programs with library.

On the contrary, since SEB needs only call graph and control-flow graph, the required memory space and analysis time is explicitly smaller than IMP. Similarly, the memory space of CIS/HYB depends on control/data dependences of each method, call graph, and square of the field access instructions. The results showed that this cost is also practical, because CIS/HYB can be computed even if the analysis target includes Java Platform APIs.

We answer to RQ2 based on the previous discussion. Since IMP output the best precise result, IMP is suitable if an analysis target program can be analyzed. However, if the program size is larger than 70,00 instructions, the analysis is prone to fail. In addition, approximation of container libraries may be needed. Therefore, it is suitable that developers need to analyze a middle-size application or a subsystem of a large application. Note that development of a suitable approximation for an application may present technical challenges.

On the other hand, when developers need to analyze a large program or a program including a library, e.g., Java Platform API, HYB is the most suitable technique, in terms of scalability and accuracy. CIS analyzes control and data dependencies in HYB, and SEB takes a few additional seconds to remove infeasible control-flow paths from CIS.

6. Threats to Validity

We approximated Java library code in the comparison of IMP and the other techniques, due to IMP's scalability. If a method in a Java library provided dependence via heap fields, IMP and CIS slices will not include statements that are actually related to the selected statement. The approximation assumes methods in the library do not call back application methods. This assumption may also result in false negatives. We did not directly compare the results of these two configurations, due to this threat.

Points-to analysis affects slices. We used Spark, but there are many other points-to analysis tools and methods we could have used: e.g., object-sensitive points-to analysis [14], hybrid context-sensitive points-to analysis [15], etc. The resulting slices could be more precise if more accurate points-to analysis is used. Improving points-to analysis would be effective for CIS and IMP, rather than SEB, because SEB uses only a call graph created by points-to analysis, while CIS and IMP use improved field access information, in addition to the call graph.

Reflection handling also affects analysis results. We used TamiFlex, which collects methods that were actually invoked during execution of a target program. Results are affected not only by the analysis target programs, but also by their execution. Program slices computed in the study miss some statements that could be executed through reflection, but are excluded from the default execution scenario in the DaCapo Benchmarks.

We obtained the results from six applications. The results may not be applicable to arbitrary Java programs; however the target programs included real Java applications. Therefore, we believe the result is indicative of a general trend.

Our slice implementations might contain some defects possibly affecting the results shown here. We have provided our implementation and dataset on our website, to enable other researchers to replicate the study and conduct further research.

7. Related Work

Binkly et al. [20] compare various kinds of program slicing. However, IMP and SEB were not included in the comparison. Additionally, the target applications are written in C/C++. Our targets were programs written in Java. In addition, CodeSurfer which is a program slicing tool and used in [20] can analyze C++ templates in analysis target source code, but it does not analyze templates in library and not in analysis target source code, such Standard Template Library (STL). [20] does not clear about handling STL. On the contrary, our Java bytecode analysis includes Java Platform APIs corresponding to STL can be performed.

[16] also compares program slicing and SEB and analyzes programs written in C/C++. We have compared SEB and two slicing techniques, CIS and IMP, for Java programs. Beszedes et al. [17] compared static execute after analysis with forward program slicing for C/C++ and Java programs. However, they did not evaluate SEB and the backward slicing techniques that we have compared in this study.

Binkley et.al. also evaluated optimization techniques of "massive slicing" which is an application of program slicing [31]. Massive slicing takes all program slice from each possible criteria, and is used for debugging. Optimization techniques evaluated in [31] optimize SDG or memory representation for SDG, but it is required that building full SDG for the first time. We evaluates approaches of building SDG or one corresponding SDG before optimizing.

[11] evaluates the SDG size of programs. However, most of the analyzed programs are student programs, and the analyzed program size is at most 10 KLOC, while we analyzed real Java applications. They also did not report the average size of a slice compared with other techniques.

There are slicing approaches for Object-Oriented Programming Languages (OOPL) other than IMP. Liang et al. [9] propose an SDG for OOPL that also simulates object trees of formal/actual in/out vertices. However, they expand field trees based on the object type (i.e., variable type) and use k-limit to expand a tree to stop infinite expansion. Therefore, the approach is less precise than IMP. Larsen et al. [10] also proposes an SDG for OOPL that treats heap field input/output as formal/actual in/out. This means that the Larsen approach does not make an object tree, it is field-based, not object-sensitive. IMP is an object-sensitive approach; therefore, it is more precise than Larsen's approach.

8. Conclusion and Future Work

We evaluated the scalability, precision and tradeoffs of four slicing techniques; SEB, CIS, HYB, and IMP. We selected six real Java applications in DaCapo benchmarks for evaluation. We computed slices of instructions in each application and compared the slice sizes and computation times of the slicing techniques.

The results show that HYB had good scalability, which was achieved with a small cost increase over CIS. An HYB slice is 25 percent smaller than the SEB slice. Moreover, an HYB slice is sometimes significantly smaller than a CIS slice, because HYB considers feasible control-flow paths from SEB. When developers need to analyze a large program or a program including a library, our results indicate that HYB is suitable.

On the other hand, our results show that IMP is the most accurate of the four slicing techniques. An IMP slice contains 22 percent of SEB and 69 percent of HYB. However, IMP does not have the scalability required to analyze a large program, such as a whole system including a JDK library. When developers need to analyze a middle-size program or a subsystem, our results indicate that IMP is suitable.

We plan to analyze how the library approximation affects slicing technique accuracy, in future research. When developers can approximate arbitrary application subsystem behavior, our results show that IMP would be applicable to a large scale programs' subsystem. Another direction for our future work would be to include additional slicing techniques, e.g., slicing for concurrent programs.

References

[1] M. Weiser, "Program slicing," IEEE Trans. Softw. Eng., vol.SE-10, no.4, pp.352–357, July 1984.

[2] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue, "Experimental evaluation of program slicing for fault localization," Empirical Softw. Eng., vol.7, no.1, pp.49–76, March 2002.

[3] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," Int. J. Inf. Secur., vol.8, no.6, pp.399–422, Oct. 2009.

[4] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," Proc. ICSE, pp.746–755, 2011.

[5] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," Proc. PLDI, pp.35–46, 1988.

[6] S. Danicic, C. Fox, M. Harman, R. Hierons, J. Howroyd, and M.R. Laurence, "Static program slicing algorithms are minimal for free liberal program schemas," Comput. J., vol.48, no.6, pp.737–748, Nov. 2005.

[7] M. Allen and S. Horwitz, "Slicing java programs that throw and catch exceptions," Proc. PEPM, pp.44–54, 2003.

[8] N. Walkinshaw, M. Roper, M. Wood, and N.W.M. Roper, "The java system dependence graph," Proc. SCAM, p.5, 2003.

[9] D. Liang and M.J. Harrold, "Slicing objects using system dependence graphs," Proc. ICSM, pp.358–367, 1998.

[10] L. Larsen and M.J. Harrold, "Slicing object-oriented software," Proc. ICSE, pp.495–505, 1996.

[11] C. Hammer and G. Snelting, "An improved slicer for java," Proc. PASTE, pp.17–22, 2004.

[12] B. Steensgaard, "Points-to analysis in almost linear time," Proc. POPL, pp.32–41, 1996.

[13] L.O. Andersen, Program Analysis and Specialization for the C Programming Language, Ph.D. thesis, Department of Computer Science, University of Copenhagen, May 1994.

[14] A. Milanova, A. Rountev, and B.G. Ryder, "Parameterized object sensitivity for points-to analysis for java," ACM Trans. Softw. Eng. Methodol., vol.14, no.1, pp.1–41, Jan. 2005.

[15] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," Proc. PLDI, pp.423–434, 2013.

[16] J. Jász, A. Beszedes, T. Gyimothy, and V. Rajlich, "Static execute after/before as a replacement of traditional software dependencies," Proc. ICSM, pp.137–146, Sept 2008.

[17] A. Beszedes, T. Gergely, J. Jasz, G. Toth, T. Gyimothy, and V. Rajlich, "Computation of static execute after relation with applications to software maintenance," Proc. ICSM, pp.295–304, Oct. 2007.

[18] L.A. Meyerovich and A.S. Rabkin, "Empirical analysis of programming language adoption," Proc. OOPSLA, pp.1–18, 2013.

[19] iTR co. Inc., "Press release," http://www.itr.co.jp/company_outline/press_release/130919PR/index.html

[20] D. Binkley, N. Gold, and M. Harman, "An empirical study of static program slice size," ACM Trans. Softw. Eng. Methodol., vol.16, no.2, pp.1–32, April 2007.
[21] "Homepage of grammatech's codesurfer." http://www.grammatech.com/research/technologies/codesurfer
[22] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: Java benchmarking development and analysis," Proc. OOPSLA, pp.169–190, 2006.
[23] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay, "Speeding up slicing," Proc. FSE, pp.11–20, 1994.
[24] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," Proc. CC, pp.153–169, 2003.
[25] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," Proc. CASCON, pp.125–135, IBM Press, 1999.
[26] E. Ruf, "Context-insensitive alias analysis reconsidered," Proc. PLDI, pp.13–22, 1995.
[27] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," Proc. ICSE, pp.241–250, 2011.
[28] J. Jász, L. Schrettner, A. Beszedes, C. Osztrogonác, and T. Gyimothy, "Impact analysis using static execute after in webkit," Proc. CSMR, pp.95–104, 2012.
[29] D. Binkley and M. Harman, "Locating dependence clusters and dependence pollution," Proc. ICSM, pp.177–186, 2005.
[30] C. Parnin and S. Rugaber, "Resumption strategies for interrupted programming tasks," Software Quality Control, vol.19, no.1, pp.5–34, March 2011.
[31] D. Binkley, M. Harman, and J. Krinke, "Empirical study of optimization techniques for massive slicing," ACM Trans. Program. Lang. Syst., vol.30, no.1, Article no.3, Nov. 2007.

Yu Kashima is now a Ph.D. student at Osaka University. His research interests include program analysis and program comprehension.

Takashi Ishio received the Ph.D. degree in information science and technology from Osaka University in 2006. He was a JSPS Research Fellow from 2006-2007. He is now an assistant professor of computer science at Osaka University. His research interests include program analysis and program comprehension. He is a member of the IEICE, IPSJ, JSSST, IEEE, and ACM.

Katsuro Inoue received the B.E., M.E., and D.E. degrees in information and computer sciences from Osaka University, Japan, in 1979, 1981, and 1984, respectively. He was an assistant professor at the University of Hawaii at Manoa from 1984–1986. He was a research associate at Osaka University from 1984–1989, an assistant professor from 1989–1995, and a professor beginning in 1995. His interests are in various topics of software engineering such as software process modeling, program analysis, and software development environment. He is a member of the IEEE, the IEEE Computer Society, and the ACM.