Application of Aspect-Oriented Programming to Calculation of Program Slice

Takashi Ishio, Shinji Kusumoto, Katsuro Inoue
Graduate School of Information Science and Technology,
Osaka University
1-3 Machikaneyama, Toyonaka,
Osaka 560-8531, Japan
+81 6 6850 6571
{t-isio, kusumoto, inoue}@ist.osaka-u.ac.jp

Abstract

Aspect-Oriented Programming (AOP) is a new technology for separation of concerns in program development. Using AOP, it is possible to modularize crosscutting aspects of a system. Common examples of crosscutting aspects are design or architectural constraints, systemic properties or behaviors (e.g., logging and error recovery), and features. Since such crosscutting aspects are usually distributed among objects in Object-Oriented Programming, it is difficult to maintain them consistently. In AOP, they can be written in a single aspect and thus easy to maintain. One useful application of AOP is to modularize collecting program's dynamic information for program analysis. Since collection of dynamic information affects over all target program, this functionality becomes typical crosscutting concerns. In this paper, we intend to evaluate the usefulness of AOP in the area of program analysis. At first, we examine the application of AOP to collecting dynamic information from program execution and calculating program slice. Then, we develop a program slicing system using AspectJ, and describe benefits, usability, cost effectiveness of the module of dynamic analysis based on AOP.

1. Introduction

Aspect-Oriented Programming (AOP) proposes a new module unit named *aspect* for encapsulating crosscutting concerns, such as logging, synchronization, and so on [1]. Since such concerns crosscut objects, program codes implementing such concern must be distributed among objects in Object-Oriented Programming. In AOP, it can be written in a single aspect.

AOP seems to be usable and useful, but there are not so many actual examples that show the usefulness of applying AOP to program development. One useful application of AOP is to modularize collecting program's dynamic information for program analysis. Dynamic information, to be short, are series of program execution. Collecting dynamic information from program execution is needed in calculating program slice and measuring dynamic complexity of a program [2, 10].

Program slicing is a very promising approach for program debugging, testing, understanding, and so on [17]. Given a source program p, *program slice* is a collection of statements possibly affecting the value of *slicing criterion* (a pair $<s, v>$, s is a statement in p and v is a variable defined or referred to at s). Also, we call program slice simply *slice*.

In recent software development, not only procedural languages like C and Pascal but also Object-Oriented languages like Java [7] and C++[8] have become to be used. Since Object-Oriented languages have new concepts such as *class*, *inheritance*, *dynamic binding* and *polymorphism*[9], Object-Oriented programs have many dynamically determined elements.

In slice calculation process, it is effective to observe program execution, and to use information about statements actually executed. *Dependence-Cache (DC) slicing* has been proposed to use dynamic data dependence analysis and static control dependence analysis to calculate accurate slices with lightweight costs [2, 14]. Ohata $et\ al.$ extends DC slicing method for Object-Oriented languages [12].

In process of DC slice calculation of Java, it is an important issue how to analyze dynamic data dependence. An analyzer may implement function that observes target program to track and to collect information about dynamic data dependence. In the past research, such function have not been encapsulated in a single module. Actually, the function was implemented as a pre-processor which inserts analysis operations in the target program code [12], or as a customized Java Virtual Machine (JVM) [3]. But, the former

approach is hard to implement and to maintain the rules of conversion, the latter approach is expensive because we must re-customize a JVM when new versions are released.

In this paper, we propose to introduce AOP for encapsulating dynamic program analysis into an aspect and achieve a cost-effective DC slice calculation. We implement a DC slice calculation system using AspectJ [15], and conduct experiment to evaluate the usefulness of our approach comparing to a customized JVM approach. As the result, it is confirmed that AOP approach can reduce the cost greatly to calculate DC slice and get the practical precision of the slice.

The structure of this paper is as follows: In Section 2, we will briefly overview Aspect-Oriented Programming. In Section 3, we describe DC slice and our approach to calculate it using AOP. In Section 4, we evaluate the proposed method comparing to the customized JVM approach and discuses experimental results. In Section 5, we conclude our discussion with a few remarks regarding plans for future work.

2 Aspect-Oriented Programming (AOP)

2.1 Features of AOP

The goal of Aspect-Oriented Programming (AOP) is to separate concerns in software. While the hierarchical modularity of object-oriented languages are extremely useful, they are inherently unable to modularize crosscutting concerns, such as logging, synchronization, and so on. AOP provides language mechanisms that explicitly capture the crosscutting structure. It is possible to encapsulate the crosscutting concerns as module unit *aspect* that is easier to develop, maintain and reuse. Aspects separated from an object-oriented program are composed by *Aspect Weaver* to construct the program with the crosscutting concerns.

AspectJ is an aspect weaver for Java. AspectJ provides language constructs to write aspects. *Join points* are well-defined points in the execution of the program. Programmer chooses collections of join points as *pointcuts*, and define method-like construct named *advice*, additional behavior at the join points. Examples of join points which programmer can use are shown in Table 1. Advices can be united by three kinds of forms, *before* (immediately before join points), *after* (immediately after), and *around* (before and behind).

2.2 Example of Aspect

Here, an observer pattern (Observer Subject Protocol) [16] is shown as an example of the aspect. The Observer pattern is consists of Observer object which watches the state change of objects and Subject object which is watched

Table 1. Pointcut Designators of AspectJ

kind of join point	meanings
call	method or constructor is called.
execute	an individual method or constructor is invoked.
get	a field of object is read.
set	a field of object is set.
handler	an exception handler is invoked.

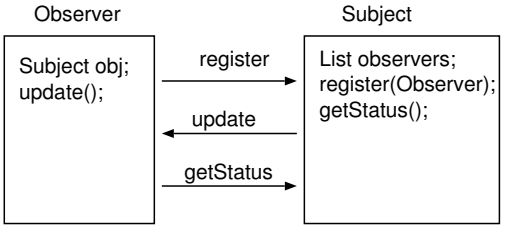

Figure 1. Class relations of Observer pattern (Java)

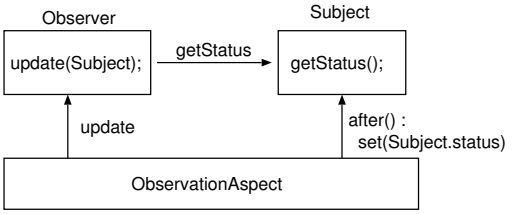

Figure 2. Class relations of Observer pattern (AspectJ)

by the Observer. Since the concern of "Observer observes state change of Subject" is crosscutting objects, this makes inter-dependence relationships between Observer and Subject. It is shown in Figure 1, and implemented as follows.

1. Observer object registers Subject which Observer wants to observe (an arrow labeled "register" in Figure 1).
2. When state change of Subject is occurred, Subject notifies Observer ("update").
3. Observer gets the latest information about Subject ("getStatus").

In AOP, the observation aspect is described as follows.

- A requirement of "When new value is set to Subject.status field, Observer gets an update message" is implemented as an aspect ("after() : set(Subject.status)" and "update").
- Observer gets the latest information about Subject ("getStatus").

In AOP, since extra codes are not required to be written in Subject, interdependence between Subject and Observer is removed. So, they become simpler and easier to reuse.

2.3 Dynamic Analysis of Program

Analysis of dynamic information from program execution is a technology needed for program slice calculation and for measurement of dynamic software metrics.

In the past, the following methods of dynamic analysis have been used for Java programs:

(a) Using preprocessor to insert analysis operations in target program [12].

(b) Using *Java Virtual Machine Profiler Interface* (JVMPI) to collect dynamic information [13].

(c) Using customized *Java Virtual Machine* for dynamic analysis [3].

In method (a), preprocessor and conversion rules on abstract syntax tree are made to insert operations for analysis in target program. But, it is hard to make generic conversion rules because such conversion is too low level operation. There are some problems about maintainability and reusability of it, conflict with other preprocessors, multi-threaded program handling.

In (b), JVMPI is used to observe program execution. JVMPI is a non-standard interface of JVM for profiling CPU and memory usage. It is possible to collect detailed events on program execution, e.g. method call, thread control, memory allocation and garbage collection. But, events are too primitive to analyze, so its overhead of analysis is expensive. Moreover, JVMPI is not standardized interface. Also, information obtained from JVMPI strongly depends on implementation of JVM.

(c) is a method that customizes JVM to observe and analyze program execution. An advantage of this approach is that JVM can access all of information in Java runtime environment. However, JVM customization depends on its implementation. Whenever a new version of JVM is released, it must be re-customized.

In (b) and (c), program has to be analyzed at Java bytecode level. So, the bytecode optimization by *Just In Time* (JIT) compiler usually affects the analysis result.

On the other hand, in AOP approach, dynamic analysis aspect can be composed by join points, which is more abstract than syntax tree conversion rules. It achieves good modularity, maintainability and reusability. It also achieves handling complex control elements, such as multi-threading and exception, by well-organized way. Moreover, AspectJ composes source codes of objects and aspects, so it does not depend on implementation of specific JVM.

3 Program Slicing

Program slicing is one of the methods such that dynamic program analysis is effective.

Slice calculation is based on dependence analysis between program statements in a source program, and dependence analysis consists of two components, data dependence analysis and control dependence analysis.

Though many slice calculation algorithms have already been proposed, we use *program dependence graph* (PDG) in this research [5].

3.1 Program Dependence Graph

A PDG is a directed graph whose nodes represent statements in a source program, and whose edges denote dependence relations (data dependence or control dependence) between statements. An edge drawn from node V_s to node V_t represents that "node V_t depends on node V_s". PDG also includes special nodes which represent method call and parameter passing [6].

Control dependence and data dependence are defined as follows.

***Control Dependence* (CD)** Consider statements s_1 and s_2 in a source program p. When all of the following conditions are satisfied, we say that a *control dependence (CD)*, from statement s_1 to statement s_2 exists:

1. s_1 is a conditional predicate, and

2. the result of s_1 determines whether s_2 is executed or not.

This relation is written by $CD(s_1, s_2)$ or $s_1 \dashrightarrow s_2$.

Data Dependence **(DD)** When all of the following conditions are satisfied, we say that a *data dependence (DD)*, from statement s_1 to statement s_2 by a variable v, exists:

1. s_1 defines v, and
2. s_2 refers to v, and
3. at least one execution path from s_1 to s_2 without re-defining v exists (we call this condition *reachable*).

This relation is denoted by $DD(s_1, v, s_2)$ or $s_1 \xrightarrow{v} s_2$.

Program slicing calculation consists of the following four phases:

Phase 1: Defined and Referred Variables Extraction
We identify defined variables and referred ones for each statement in a source program.

Phase 2: Data Dependence Analysis and Control Dependence Analysis
We extract data dependence relations and control dependence relations between program statements.

Phase 3: Program Dependence Graph Construction
We construct PDG using dependence relations extracted in Phase 2.

Phase 4: Slice Extraction
We calculate the slice for the slicing criterion specified by the user. In order to calculate the slice for a slicing criterion $<s, v>$, PDG nodes are traversed in reverse order from V_s (node V_s denotes statement s.). The corresponding statements to the reachable nodes during this traversal form the slice for $<s, v>$.

We can obtain sufficient information about control dependence from static analysis (from only source code). However, in static analysis, information about data dependence which we can get contains redundant part because we analyze all execution paths, includes paths which may be never executed. If we use program slicing for debugging and program understanding, it is effective to analyze detailed information about one program execution path with a specific input. Dependence Cache (DC) slicing has been proposed to realize such requirement [2, 12, 14].

In DC slice calculation, the data dependence analysis is performed during program execution, and the information of dynamically determined elements is collected. Control dependence is analyzed statically from source code since it needs much cost to analyze control dependence during program execution. It is known that the DC slicing takes reasonable cost for calculation of practical programs [2, 12, 14].

3.2 Dynamic Data Dependence Analysis in DC Slice Calculation

When variable v is referred to at statement s, dynamic data dependence (DD) relation about v from t to s can be extracted if we can resolve v's defined statement t. We create a table named *Cache Table* that contains all variables in a source program and most-recently defined statement information for each variable. When variable v is referred to, we extract dynamic DD relation about v using the cache table. The following shows the extraction algorithm for the dynamic DD relations.

Step 1: We create a cache $C(v)$ for each variable v in a source program.
$C(v)$ represents the statement which most-recently defined v.

Step 2: We execute a source program and conduct the following processes on each execution point.
On executing statement s,

- when variable v is referred to, we draw an DD edge from the node corresponding to $C(v)$ to the node corresponding to s about v, or
- when variable v is defined, we update $C(v)$ to s.

For example, Figure 3 is a program using array. Table 2 shows a transition of cache $C(v)$ of each variable v at each statement when program is executed with input $c = 0$.

It becomes $C(a[0])$ = 1, $C(a[1])$ = 2, $C(a[2])$ = 3, $C(a[3])$ =4, $C(a[4])$ = 5, $C(c)$ = 6 when statement 6 is executed. When variable $a[0]$ is referred to at statement 7, data dependence $statement1 \xrightarrow{a[0]} statement7$ is extracted because statement 7 refers to $a[0]$ and $C(a[0]) = 1$.

Table 2. Cache transition of Figure 3

Statement number executed	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	b	c
1	1	-	-	-	-	-	-
2	1	2	-	-	-	-	-
3	1	2	3	-	-	-	-
4	1	2	3	4	-	-	-
5	1	2	3	4	5	-	-
6	1	2	3	4	5	-	6
7	1	2	3	4	5	7	6

```
1: a[0] = 0;
2: a[1] = 1;
3: a[2] = 2;
4: a[3] = 2;
5: a[4] = 2;
6: read(c);
7: b = a[c] + 5;
```

Figure 3. Example program using array

Figure 4 shows another example of the DC slice. DC slice with input = 2 and slice criteria $=< 37, d >$ is the non-shaded part of Figure 4.

3.3 Dynamic Analysis Using AspectJ

AspectJ is an Aspect Weaver which composes objects and aspects at source code level. AspectJ generates normal Java code which includes the aspects. At this time, since AspectJ knows where aspect is built in, AspectJ can generate codes accessing the information of source codes as context of aspect, e.g. join points' position in the source code, signature of methods, and so on. Programmers can write the dynamic analysis aspect using this feature of AspectJ.

An algorithm of the data dependence analysis and polymorphism resolution can be described as follows using AspectJ.

- **Data Dependence Analysis**

  When new value is set to a field: The aspect logs signature of the field, and the position of assignment statement.

  When field is referred to: The aspect gets the statement information of last assignment to field, and logs a data dependence from the assignment to the reference.

- **Polymorphism Resolution**

  When method is called (before call): The aspect pushes the method signature and the position of calling into call stack prepared for each thread of control (it is for multi-threaded program).

  When method is invoked (before execution): The aspect checks the top of the call stack, and generates a control dependence from the caller to the actually invoked method.

  After method call: The aspect removes the top of the call stack.

  When exception is thrown: The aspect removes the top of the call stack.

```
 1:  #include <stdio.h>
 2:  #define SIZE 5
 3:
 4:  int cube(int x) {
 5:      return x*x*x;
 6:  }
 7:
 8:  void main(void)
 9:  {
10:      int a[SIZE];
11:      int b[SIZE];
12:      int c, d, i;
13:
14:      a[0] = 0;
15:      a[1] = -1;
16:      a[2] = 2;
17:      a[3] = -3;
18:      a[4] = 4;
19:
20:      for (i=0; i<SIZE; i++) {
21:          b[i] = a[i];
22:      }
23:
24:      printf("Input: ");
25:      scanf("%d", &c);
26:
27:      if (c >= SIZE) {
28:          c = c % SIZE;
29:      }
30:
31:      d = cube(b[c]);
32:
33:      if (d < 0) {
34:          d = -1 * d;
35:      }
36:
37:      printf("%d\n", d);
38:  }
```

Figure 4. Source program and DC slice (non-shaded part, slice criterion $=< 37, d >$, input $= 2$)

3.4 Implementation Details

3.4.1 Statical Analysis Supplement

In AOP, aspect may be limited by usable join points and the applicable operation to the join points. The join points of AspectJ does not include local control structures (e.g. *if, while, for* statements) nor access to local variables. Because such join points are fine grained, it needs remarkable cost to implement, and very few cases would need them.

Though usual dynamic analysis requires to observe the behavior of all variables and control structures, we cannot implement in AspectJ. Instead, we statically collect information about local variables and control structures for the compensation. This seems sufficient because data dependence of local variables and execution paths of local control structures are limited in one method, and they are affected only a little from dynamic determined elements in OOP. Later, we will discuss this issue based on the result of experimental evaluation.

3.4.2 Analysis of Libraries

Since AspectJ links the aspects to target source code, it cannot link them into library classes. Here, the library classes indicate reusable components which are not included as source codes here.

In this research, libraries are excluded from analysis by the following reasons:

Library class is reliable. Since libraries are repeatedly reused, it can be assumed that defects in the libraries are already removed. Therefore, we do not need to conduct the detailed analysis to the library classes.

Amount of code of library is numerous. The cost of the dynamic analysis of libraries is generally more numerous than main program.

When a program uses callback from library side, hidden dependence via the library might be caused. It can be known by the dependence analysis at bytecode level [3].

However, even if we use the bytecode analysis, a dependence analysis to important objects, such as file I/O and basic data structures, cannot be done because of limitation in the Java language described later. Therefore, we cannot say that the range where it actually influences is wide even with the bytecode handling. It is enough to analyze only the range where the source code exists.

3.4.3 Loop Caused by Aspect

AspectJ has an advantage that programmers write aspects in Java easily. But, it causes dependences from aspects to classes which are used to collect and log information.

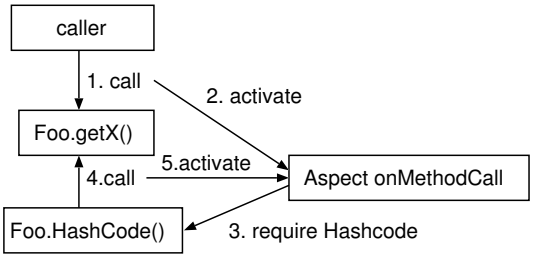

Figure 5. loop by aspect

Therefore, if some aspect is built into such classes to analyze, an loop might be caused.

The example of such a loop is shown in Figure 5. In Figure 5, the aspect operates corresponding to a method call $Foo.getX$. The aspect calls $Foo.hashCode$ to get hash code of the object, and calling $Foo.getX$ occurs in $Foo.hashCode$.

It is not possible to solve it essentially in Java language. Only the approach like customized JVM approach solves this problem.

Since we have implemented the data analysis module using Java standard library, a loop might be caused if the target program has the methods called from standard library. As long as we have examined, there are only two methods that might be called in our implementation. One of them is $Object.toString$ which is a method that converts an object into character string to make data readable. Another is $Object.hashCode$, a method that calculates the hash code for fast access to data structures. It is possible to avoid the loop by not joining the aspect to them. It causes decrease of completeness of the information, but we consider that it does not give the influence on practical use because the role of these methods is usually independent of the other part.

4 Experimental Evaluation

4.1 Overview

We have implemented a dynamic analysis module using AspectJ, and then developed a DC slice calculation system for Java. Figure 6 shows system overview.

Using this system, a user can calculate DC slice by the following steps:

Step 1: Compile target Java program and the dynamic analysis aspect using AspectJ compiler.

Step 2: Execute the program as usual Java program. Then, dynamic analysis aspect in the program generate a file contains dynamic information of the program execution.

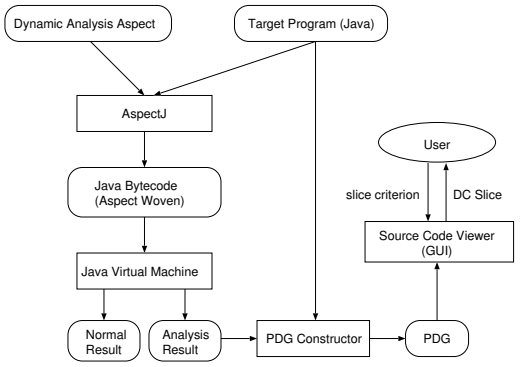

Figure 6. DC slicing system

Table 3. Target programs

	Program	# of classes	Size (LOC)
P1	Simple database	4	262
P2	Sorting	5	228
P3	DC slice calculation	125	16207

Step 3: Execute the DC slice calculation tool with source code of target program and a dynamic information file generated by Step 2. The tool extracts static information from the source code, constructs PDG, and opens a window of a source code viewer.

Step 4: Specify the slice criterion and view DC slice by a graphical user interface.

In order to evaluate the proposed DC slice system, we have compared it to the system developed using the customized JVM approach [3] from the viewpoint of cost and module size necessary for the dynamic analysis. In the evaluation, we have used the programs shown in Table 3 as the input of the systems.

P1 is a simple database program which contains few elements of object-oriented language. P2 is a program which uses polymorphism to switch sorting algorithms. P3 is the DC calculation system presented in this paper. It includes many features of Java, e.g. polymorphism, classes and package hierarchy, exception handling, and interactive user interfaces.

We have executed these programs with some input data, and calculated DC slice for arbitrary slice criterion.

In Section 4.2, we evaluate and discuss about DC slice size. We discuss time cost and module size necessary for DC slice calculation in Section 4.3 and Section 4.4.

Table 4. Slice size [LOC]

Slice criterion	Customized JVM	Aspect
S1 (P1)	29	36
S2 (P2)	28	50
S3 (P3)	708	839

4.2 Resulting Slice Size

Here, we compare the two slice tools from the viewpoint of resulting slice size.

Table 4 shows the size of DC slice for slice criterion S1 in P1, S2 in P2, and S3 in P3.

Actually, the DC slices calculated by both of two systems included the correct DC slice that is obtained by manually. But, some redundant statements were included. So, it can be said that the difference of the slice size shows the difference of accuracy.

With respect to the redundant statements, in our approach, we have to statically analyze the target program to collect information about local variables and local control structures. Therefore, statements which are possibly dependent but are actually non-dependent may be included in the slice result. For example, assume that there are some conditional clauses in the program and one of them is not executed because of the corresponding conditional predicate is not satisfied. Then, the statements in the conditional clause which were are not executed may be included in our approach, but they are not included in customized JVM approach.

For the program P1 with a slicing criterion S1, the sizes of DC slice by the customized JVM approach was 29 (LOC) and one by our approach was 36 (LOC), respectively. This is not a substantial difference because the program size of P1 is small and does not include the characteristics of object oriented program.

On the other hand, for the program P2 with a slicing criterion S2, the size by our approach became about twice the size of one by the customized JVM approach. It is considered that because the program P2 is small but contains several methods which use many local variables and nested control structures.

For the program P3 with a slicing criterion S3, the difference is not so huge though the size of P3 is much larger than other programs P1 and P2. The reason is that the program P3 is skillfully decomposed into modules with proper sizes, and each method has few local variables and simple control structures.

As we expected, the result shows that the size of DC slice by our approach is larger than one by the customized JVM approach for the programs that include many local variables and local control structures. However, for the size of the tar-

Table 5. Execution time (JIT disabled) [sec.]

Target program	Normal	Customized JVM	Aspect
P1	0.18	1.8	0.26
P2	0.19	2.8	0.39
P3	1.2	81.0	10.3

Table 6. Execute time (JIT enabled) [sec.]

target program	Normal	Aspect
P1	0.24	0.34
P2	0.24	0.41
P3	1.1	9.9

get program (especially P3), the difference of the resulting slice size between the two approach is insignificant. So, we consider that our approach would be effective for the large scale programs.

4.3 Analysis Cost

Here, we evaluate the time for calculating the DC slice.

Table 5 shows the time to execute Java program with normal JVM, with customized JVM, and the program which aspect has been inserted with normal JVM (our approach) for the same input. These values are measured in JIT disabled environment. The execution time with enabled JIT is shown in Table 6.

In general, our approach shows good performance compared with the customized JVM approach. We consider that the cost of a dynamic analysis of the local variables is very expensive, because of little use of the library in P1 and P2. Moreover, in P3, analyzing internal processing in the library required further cost. As program size becomes larger, analysis cost must increase further because more libraries become to be used.

Our aspect approach has an advantage that we can use JIT compiler to improve performance. In small programs such as P1 and P2, performance of program without optimization by JIT compiler is better, because the optimization is not effective in this case. However, in practically-large scale program like P3, JIT compiler is very effective to improve performance. Though effect of JIT compiler is unequal in runtime environment, it has experimentally been shown that JIT makes a crucial difference on system performance [11].

4.4 Effort to Implement the Slicing Tool

Here, we examine the effort of implementing the slice tool.

A dynamic analysis module implemented as an aspect became about 400 lines of code. The DC slice calculation tool totally became about 16,000 lines in Java.

In our approach, the aspect can be described in high abstraction level and has good readability compared with the pre-processor approach. Moreover, because the aspect is small and simple, it is easy for the programmer (user) to switch other implementation to adapt each runtime environment.

On the other hand, in the customized JVM approach, it was necessary to add about 16,000 lines of code to JVM and Java compiler that is totally consisted of about 500,000 LOC. Furthermore, the overall programs must be re-customized when the original JVM is updated. Therefore, it is unrealistic to keep it consistent. Our aspect approach, which uses the aspect written once, is applicable to any platform where the aspect weaver is available. Since AspectJ is written in Java, the aspects achieve good reusability. It is much cheaper than the customized JVM approach to implement.

5 Conclusion and Future Work

In this paper, we have examined an application of the aspect-oriented programming to collect dynamic information in program slicing calculation. Then, through the implementation of a dynamic program analysis module in AOP, we have developed a DC slice calculation system and evaluated the usefulness of it.

Since we make join points of the aspect to be generic form, the dynamic data dependence analysis aspect can be woven into various object-oriented programs without its changes. We can improve maintainability and reusability of the module.

Here, we have chosen AspectJ to implement the module. AspectJ has an restriction that we cannot analyze local variables and local control structures. However, such a difference influences little for the result of slice, and we observes that the our aspect approach reduces cost for dynamic analysis effectively. Compared with the customized JVM approach, we could achieve cost reduction and maintainability improvement with practical precision of the slice size. Our aspect approach is not dependent to Java specific factor. It is possible to implement dynamic analysis using appropriate aspect weaver for other languages.

In future work, we are going to evaluate our slicing system for large programs. Also, we will examine the applicability of aspect-oriented programming to other application in software development.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier and J. Irwin: "Aspect Oriented Programming", Proceedings of ECOOP, vol.1241 of LNCS, pp.220-242(1997).

[2] Y. Ashida, F. Ohata and K. Inoue: "Slicing Methods Using Static and Dynamic Information", Proceedings of the 6th Asia Pacific Software Engineering Conference, pp.344-350, Takamatsu, Japan, December(1999).

[3] K. Konda, F. Ohata, K. Inoue: "Extraction Method for Dynamic Dependence Relations between Bytecodes Using Java Virtual Machine", JSSST Computer Software, Vol.18, No.3, pp.40-44 in Japanese (2001).

[4] H. Agrawal and J. Horgan: "Dynamic Program Slicing", SIGPLAN Notices, Vol.25, No.6, pp.246-256(1990).

[5] K. J. Ottenstein and L. M. Ottenstein: "The program dependence graph in a software development environment", Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp.177–184, Pittsburgh, Pennsylvania, April (1984).

[6] R. Ueda, K. Inoue and H. Iida: "A Practical Slice Algorithm for Recursive Programs", Proceedings of the International Symposium on Software Engineering for the Next Generation, pp.96–106, Nagoya, Japan, February (1996).

[7] J. Gosling, B. Joy, and G. Steele: "The Java ™ Language Specification", Addison-Weseley (1996).

[8] B. Stroustrup : "The C++ Programming Language (Third edition)", Addison-Wesley (1997).

[9] G. Booch: "Object-Oriented Design with Application", The Benjamin/Cummings Publishing Company, Inc (1991).

[10] S. Yacoub, H. Ammar and T. Robinson: "Dynamic Metrics for Object Oriented Designs", Proc.of the 6th International Symposium on Software Metrics (METRICS99), Boca Raton, Florida USA, pp. 50-61 (1999).

[11] Performance Comparison of JIT, http://www.shudo.net/jit/perf/index.html

[12] F. Ohata, K. Hirose, M. Fujii, and K. Inoue: "A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information", In Proc. of APSEC2001, pp.273-280(2001).

[13] S. Kusumoto, M. Imagawa, K. Inoue, S. Morimoto, K. Matsusita and M. Tsuda: "Function point measurement from Java programs", Proc. of the 24th International Conference on Software Engineering, pp. 576-582 (2002).

[14] T. Takada, F. Ohata, K. Inoue: "Dependence-Cache Slicing: A Program Slicing Method Using Lightweight Dynamic Information", Proceedings of the 10th International Workshop on Program Comprehension (IWPC2002), pp.169-177, Paris, France, June (2002).

[15] AspectJ Team, "The AspectJ Programming Guide", http://aspectj.org/doc/dist/progguide/

[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley (1995).

[17] M. Weiser: "Program slicing", IEEE Transactions on Software Engineering, SE-10(4):352-357(1984).