1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

Dynamic Java Program Corpus Analysis

Chung-Chien Hwang[a], Shih-Kun Huang[b], Min-Shong Lin[c], Chorng-Shiuh Koong[a] and Deng-Jyi Chen[a]

[a] Computer Science and Information Engineering Department, National Chiao Tung University, Hsin-Chu, Taiwan
[b] Institute of Information Science, Academia Sinica, 128 Academic Road, Section 2, Nankang 115, Taipei, Taiwan
[c] Computer and Communication Research Laboratories, Industrial Technology Research Institute, Hsin-Chu, Taiwan
Email: cchwang@csie.nctu.edu.tw, skhuang@iis.sinica.edu.tw, {minslin,csko,djchen}@csie.nctu.edu.tw

ABSTRACT

Program corpus analysis is important in optimization of run-time systems. Conventional linguistic analysis is static in nature and can't reflect dynamic behaviors revealed by versatile object-oriented programming languages. Pattern based run-time profiler is proposed and realized in this paper. Unlike conventional profiler or run-time visualization tools, representative program corpora accumulated and benchmarked not only show monolithic functions that introduce excessive run-time overhead but also reflect their correlated code patterns. We proposed pattern-based analysis to address program run-time bottleneck in a sequence of method invocations. It will reveal more semantic meanings in performance bottleneck rendered by object-oriented programming systems.

Keywords: Object-Oriented Computing, Java VM, Program Corpus Analysis, Run-time Profiler and Visualization, Design Patterns, Code Patterns

1. INTRODUCTION

Linguistic analysis of text corpus is done by parsing strings of token sequentially due to the spoken nature of human beings. However, program execution is not sequential and polymorphic operations largely complicate execution trace. For example, a string of tokens A, B and C is a program corpus. $P_f(A|B)$ is the condition probability that A will occur consecutively before B and $P_b(C|B)$ with C occurred after B. If $P_f(A|B)$ and $P_b(C|B)$ relatively small, B may be a candidate code patterns that are frequently used. However, program behavior is not static and varied due to dynamic binding and polymorphic operations associated with the context where B resides. If we want to fit the lexicon model into program corpus, we must consider using run-time trace to detect dominant converge of code patterns.

Code patterns (also named Idioms) are low-level patterns specific to a programming language. They reflect the style experienced programmers frequently apply in their routine work. The same recurrent structures are used many times and may embed in certain design patterns like Singleton design pattern in C++ or Smalltalk. It's hard to find a specific pattern from a large of program corpus due to diversity of dynamic binding and inheritance. We have built a Java run-time profiler in a pattern based approach to detect specific control structures that are central to the code patterns. We call them the control patterns[4]. These control patterns can be performance bottleneck and will be highlighted in our visualization tool. They can be recurrent structures in some problem domain and can be used for a pattern finding system. If applied in a software vulnerability penetration tool, it is a good testing tool for finding interface incompatibility.

The rest of this paper is organized as follows. In section 2, the design and implementation of the profile analyzer are discussed. The meanings of the evaluation results of the programs are also discussed in this section. In section 3, a suite of Java programs is collected for our benchmark programs. The analyzing results of these benchmark programs by our analyzer are discussed. Finally, our system is compared with the profile produced by the original software implementation of the JVM[6]. Conclusions and further work will be given in section 4.

2. THE DESIGN AND IMPLEMENTATION OF THE ANALYZER

The run-time information of Java programs can be obtained by running the Java programs on our modified JVM software implementation. In this section, we will design and implement an analyzer to analyze several object-oriented program behaviors for seeking possible patterns and find potential performance improvement.

2.1 Object-Oriented Program Behaviors

2.1.1 Method Invocation Localities

Hardware caches in a computer system can improve performance[11]. It is because there is existed reference locality of memory space in programs. Reference locality of memory space means that given a period of time, references to memory space are confined to a small range of memory space. Similarly, we wonder if there exists method invocation locality during object-oriented program execution. Method Invocation locality means that during a period of time, method invocations are confined to a small set of methods, or classes.

Take the segment of program in Figure 2-1 as an example. It traverses a tree and gives each node a number to represent the traversal order. In spite of how many classes and methods in the whole programs, during the execution of this program segment, only 2 classes and 7 methods are involved. They are *stack* and *ce* classes, and *stack.empty()*, *stack.push()*, *stack.pop()*, *ce.setOK()*, *ce.setValue()*, *ce.hasoreChildren()* and *ce.nextChild()* methods. As a result, method invocation locality might be a behavior to characterize object-oriented programs.

```
while ( !stack.empty() ) {
   ce = stack.pop();
   if( ce.traverse() ) {
        ce.setOK();
   }
   else {
        ce.setValue( value++ );
        stack.push( ce );
        while( ce.hasMoreChildren() )
             stack.push( ce.nextChild() );
   }
```

Figure 2-1 Segment of A Tree Traversal Program

A method invocation is consisted of three parts: receiver class, method class, and method. In the analyzer, localities of receiver class, method class, and method are evaluated respectively. To define the localities, a window size is first chosen to be the range to count the number of classes or methods. For receiver class locality and method class locality, the total number of classes of the evaluated programs is chosen as the window size. For method locality, the total number of methods of the evaluated programs is chosen as the window size. The receiver class locality {method class locality, method locality} is then defined as the ratio of the receiver class count {method class count, method count} in this window to the window size. By the definition of locality values, one can easily realize the phenomenon that method invocation sequences with smaller value of locality exhibit better overall locality. Figure 2-2 shows the method invocation sequence produced by running the program segment of Figure 2-1. We hypothetically define the window size as 12. Then the receiver class locality {method class locality, method locality} of this window is 2/12 {2/12, 7/12}.

For the whole method invocation sequence of a program execution, the window is shift from the beginning to the end of the method invocation sequence, and respective localities of each shift are evaluated. And then the average

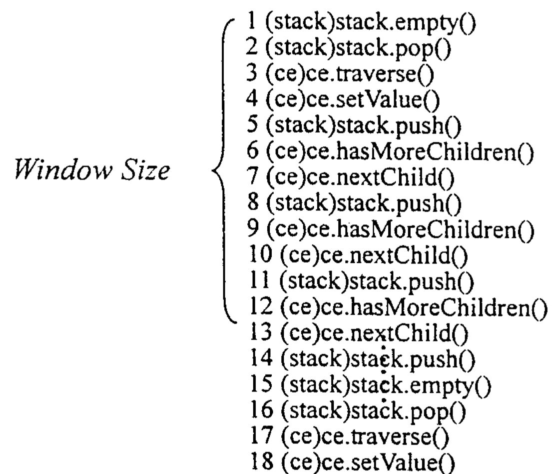

Figure 2-2 Method Invocation Sequence of the Program Segment in Figure 2-1

of these locality values is used to represent the locality of a program execution.

2.1.2 Control Patterns

During object-oriented program execution, the action of invoke a method to execute is known as a control transformation. A method invocation sequence keeps track of all the control transfers occurred during program execution. There might exhibits some recurrence patterns in the method invocation sequence, and these recurrence patterns of control transformation are called control patterns. In this thesis, we evaluate three kinds of control patterns, and they are consecutive patters, hierarchy consecutive patterns, and loop-N patterns. Examples are used to explain these patterns in the following paragraphs.

Consider the method invocation sequence in Figure 2-2. The 1st and 2nd method invocation is an instance of receiver class consecutive pattern and method class consecutive pattern, because they are consecutive, and their receiver classes are the same (stack), and their method classes are the same (stack). Consider the 6th to 14th method invocations, the same method invocation is repeated every three method invocations. As a result, they are an instance of receiver class loop-3 pattern, method class loop-3 pattern, and method loop-3 pattern.

Let's look at another example in Figure 2-3. At the left of this figure is a class hierarchy diagram, and at the right is a program segment. *TextArea* class is a subclass of *TextComponent* class, *Component* class and *Object* class. Method *appendText()* is defined in *TextArea* class, and method *enable()* is defined in *Component* class. The correspondent method invocation sequence of the last four

statements is listed in Figure 2-4. The 1st and 2nd method invocations are both invoke the *enable()* method of class *Component*, so they are an instance of method consecutive pattern. Now let's focus on the method classes of these four method invocations. All of the method classes belong to the same class hierarchy, although two of them are *Component* class, and two of them are *TextArea* class. As a result, these four method invocations are an instance of class hierarchy consecutive pattern. When the situation happens on the receiver class, then it is called a receiver hierarchy consecutive pattern.

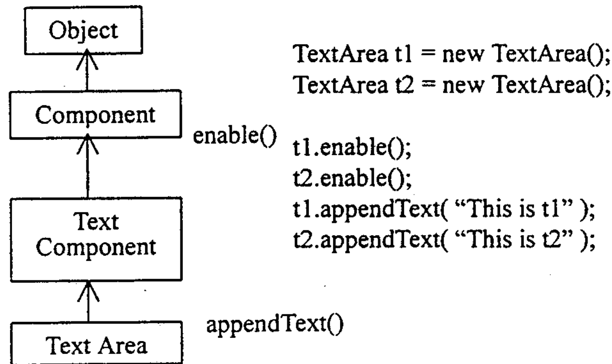

Figure 2-3 UI Program Example

1 (TextArea)Component.enable()
2 (TextArea)Component.enable()
3 (TextArea)TextArea.appendText()
4 (TextArea)TextArea.appendText()

Figure 2-4 Method Invocation Sequence of the Program in Figure 2-3

2.1.3 Other Behaviors

An object can receive a message (method invocation) either defined in the class of the receiver or defined in the superclasses of the receiver. When an object execute the method defined in the class of the object, then the method class is the same with the receiver class. When an object execute the method defined in the superclasses of the object, then the method class is not the same with the receiver class, but they are in the same class hierarchy. The analyzer we designed can measure the distance between the receiver class and method class when they are in the same class hierarchy. If the method lookup algorithm is implemented by searching the class hierarchy, the efficiency of method lookup will be affected with the distance between the receiver class and method class[1,3,10]. Take the method invocation sequence in Figure 2-4 as an example. The receiver class of the 1st and 2nd method invocations is *TextArea*, while the method class of them is *Component*. *Component* is a direct superclass of *TextComponent*, and *TextComponent* is a direct superclass of *TextArea*, so the distance between *Component* and *TextArea* is 2.

Other behaviors of Java program that our analyzer measures and analyzes are listed as follows: average method size of all methods in a program, method size distributions, the number of native method exist in a program, and the invocation count of native methods during program execution. How these behaviors affect the program performance are explained in the section 3 of this paper.

2.2 Analyzer Architecture

The architecture overview of the analyzer is shown in Figure 2-5. The static information file is first read in line by line to construct the database. After reading the whole static information file, the database contains the classes and methods information that is used during the program execution, and the events produced during the program execution.

In the database the methods that defined in the same class are linked together, and pointed by the class that defines them. Each method node contains its name, signature and size. Classes are linked by their inheritance relationships. Each class node contains its name, a pointer pointed the class node of its superclass, a pointer pointed to the methods which defined in it, and two integer values (left-value and right-value) which are used in class inclusion tests. The left-value and right-value of the class node will be further discussed in 2.3 *Implementation Notes*. The database also contains an event list in which the event entries are indexed by the data in the dynamic information file. Each event entry contains three pointers. The *ReceiverClass* points to a class node that represents the receiver class of this event. The *MethodClass* also points to a class node that represents the method class of this event. The *Method* points to a method node that represents the method of this event.

After the database is constructed, it is ready for the analysis engine to access. The analysis engine is designed to analyze the various program behaviors described in section 2.1.

2.3 Implementation Notes

2.3.1 Class Hierarchy Encoding

Class inclusion tests[5,12] are usually needed during our analysis. For example, when analyzing the class hierarchy consecutive pattern, two classes should be tested to see if they are in the same class hierarchy. Java is a single inheritance programming language, so the relative numbering method is adopted for the class inclusion tests in our analyzer implementation.

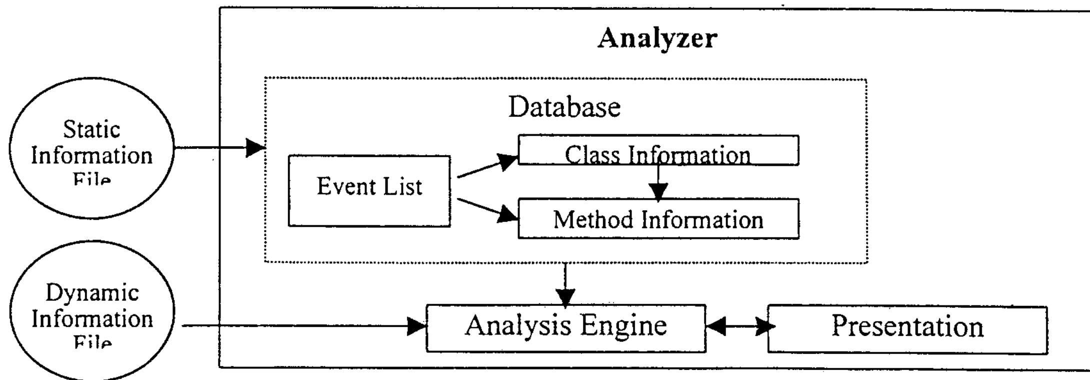

Figure 2-5 Analyzer Architecture Overview

2.3.2 Control Pattern Evaluation

In this section, we describe how the percentages of control patterns in a method invocation sequence are calculated. Different control patterns are calculated separately. Only one control pattern is calculated for each pass of the method invocation sequence. Figure 4-9 illustrates the way our analyzer used to calculate the percentages of control patterns in a method invocation sequence.

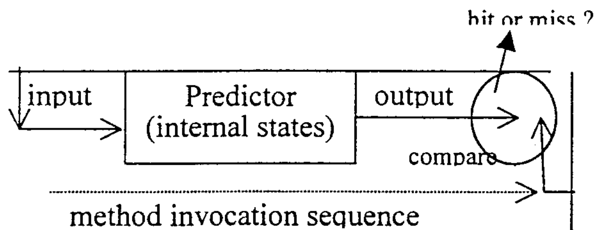

Figure 2-6 Predictor for Evaluating Control Patterns

The method invocation sequence is fed into the predictor. The predictor keeps several internal states to predict the next method invocation. The output of the predictor is compared with the next method invocation input from the method invocation sequence. If the output of the predictor is same with the input method invocation, then the input method invocation together with the method invocations in the predictor is an instance of the evaluated control pattern. The input method invocation is then fed into the predictor to update the internal states of predictor.

The predictor is configurable. Setup the predictor with different internal state configuration correspond to different control pattern evaluation. To evaluate the consecutive patterns, setup the predictor with one internal state to record the previous method invocation. The internal state is used as an output to compare with next input method invocation. To evaluate the loop-3 pattern, setup the predictor with three internal states to record the previous three method invocations. The third internal state is used as an output to compare with the next input method invocation. With these techniques, the percentages of control patterns in a method invocation sequence can be easily evaluated.

3. PATTERN-BASED CORPUS ANALYSIS

In order to see if there exist any particular behaviors in typical Java programs, we collected a suite of Java programs to analyze. These programs are first executed on the modified JVM to get the run-time invocation sequence, and then analyzed by the analyzer to obtain various statistics. In this chapter, the benchmark programs are described and the results are discussed.

3.1 The Program Corpus: Benchmark Programs

We have collected 18 Java programs for our analyzer to analyze. Most of these programs are from two sources. One is the sample programs included in the JDK, the other is the winner programs of JavaCup program contest, which was held by Sun Microsystem in 1996. Javac program is included in the JDK API. LinpackJava is a Java version of Linkpack benchmark[2]. It is hoped that these programs can represent the application domains of java programs and exhibit the typical java program behaviors.

Below are the overview and descriptions of our benchmark programs. In the **# of Classes** field, the number in the parentheses is the number of classes exist in the program, while the number outside the parentheses is the number of classes that are actually used in the run-time of the program execution. Most of these programs are user-intervention programs. In other words, it needs users to terminate the execution of these programs. We always terminate their execution after the execution behaviors have reached a steady state, or after proceeding a meaningful work. For example, in the Animation program,

we kept the program running for the animation repeating two or three times before terminating it. In the WebDraw program, we drew a Mickey Mouse face and saved it before exiting the program.

Table 3-1 Overview of the Benchmark Programs

Name	# of Lines	# of Classes	# of Events
Javac	2,570	156(8)	272,193
Animation	361	139(1)	70,942
Molecule Viewer	705	132(4)	558,202
ScrollText	307	121(1)	32,907
Blink	94	111(1)	59,977
Fractal	385	115(4)	134,158
Dither Test	332	141(3)	303,727
TicTacToe	306	146(1)	40,391
Tubes	617	149(8)	585,210
Background Thread	367	135(5)	159,120
ThreadX	278	118(3)	74,449
CardTest	113	118(2)	31,547
MapInfo	4,277	192(26)	306,904
TrafficSim	669	125(6)	563,661
TuringMachine	991	167(1)	156,045
WebDraw	5,170	156(23)	248,353
DigSim	10,293	225(64)	993,350
LinpackJava	629	39(1)	11,180

These benchmark programs can be classified into the following eight categories:

1. Text Processing

1.1 Javac A java compiler which is included in the JDK. This program is run to compile the 2.1 Animation program.

2. Image Processing

2.1 Animation An animator program which can show a sequence of pictures circularly.

2.2 MoleculeViewer The program draws 3D molecule models. Users can use the mouse to navigate the 3D model to different viewpoints.

2.3 ScrollText This program show a sentence on the screen, and scroll it around.

2.4 Blink Several words are shown on the screen, and their places and colors changes with time randomly.

2.5 Fractal Calculate the fractal graph, and draw it on the screen

2.6 DitherTest Choose two colors in the program, the DitherTest will mix the two colors gradually, and show the results.

3. Game

3.1 TicTacToe A little game program.

3.2 Tubes A puzzle game program.

4. Multi-Thread Program

4.1 BackgroundThread Two threads run simultaneously, and communicate with each other to exchange their computed data.

4.2 ThreadX Three threads in the program, and users can control when to start or stop the running of any of the three threads.

5. Interactive Program

5.1 CardTest It is a GUI demonstration program. Users can choose a layout by mouse, then the program will show the layout demonstration to the users.

5.2 MapInfo A simple geographic information system. The map of University of British Columbia is shown on the screen. Users can get detailed information about any particular building by clicking the position of the building on the map.

6. Simulation

6.1 TrafficSim The program simulates the effect of the traffic lights on the traffic flow, and demonstrates the simulation to users by graphics.

6.2 TuringMachine The program simulates the operations of the Turing machine. Sample programs can be loaded into the simulator, and the execution process will be shown by graphics.

7. System

7.1 WebDraw A graphics editor. Users can draw graphics and save/load them to/from the files.

7.2 DigSim A digital circuit editor and simulator. Users can choose the build-in digital circuit components, connect them, and simulate the operations of the built digital circuits.

8. CPU Intensive program

8.1 LinpackJava It is an artificial benchmark to test the CPU performance.

3.2 Discussions

3.2.1 Method Size

The average method sizes of most of our benchmark programs are between 30 and 40 bytes (in bytecode). Except the *Javac*, the others are below 50 bytes. In the Java compiler program, many codes are dealing with the text parsing, bytecodes generation, there will be a lot of if-then-else statements in a method. That is the reason the average method size of *Javac* is bigger compared to other programs.

From the aspect of the method size distribution we observed that the sizes of most methods in Java programs are small, and nearly half of them are between 0 and 20 bytes. This is one of the features of object-oriented programming. It is recommended to access object variables through methods. Methods for accessing object variables are usually contain only one return statement.

Moreover, methods in object-oriented programs aren't very complicated. Complicated methods are usually divided into several simple methods. These are the reasons that most methods are relatively small in our benchmark programs.

Too many small methods induce frequent control transfer during program execution. A control transfer is very expensive compared to sequential execution. It has to search the destination address, and prepare the execution context of target method. Thus, the implication of small method sizes is inefficiency of program execution.

3.2.2 Native Method

Though Java Virtual Machine can be implemented on a chip, it doesn't contain any I/O instructions. Therefore, when implementing the JVM by software on a platform, all I/O operations must be executed by the native codes of that platform. Some of the methods of the JDK API are written in C language, and compiled into native codes rather than Java bytecodes.

By the analysis of our benchmark programs, we have found that most of our benchmark programs, the execution percentages of the native methods do not exceed 20%. The native method execution percentages of TrafficSim and TuringMachine are 25% and 32%, which are higher than other programs. The reason is that the two programs contain lots of I/O operations. While the DitherTest and LinpackJava programs involve many arithmetic operations, so the native method execution percentages are relatively very low.

It is believed that if a Java program execute more native methods, then the execution speed of that program will be faster. But execute more native methods contradict the design philosophy of JVM, which is designed for platform-independent execution. If the execution depends heavily on the native methods, then the Java program won't be platform-independent anymore.

3.2.3 Method Invocation Localities

The receiver class locality and the method class locality are smaller than 0.1 or around 0.1. This means that during the execution of a program, given a short period of time, the accesses to classes are confined to a small subset of classes rather than all of the classes in the program. A very interesting phenomenon is the receiver class localities are always smaller than the method class localities. That is because a message to an object may cause the object to execute the methods of its super classes. All of these method localities are smaller than 0.07, which are slightly better than the receiver class localities and method class localities. The receiver class, method class, and method localities of DitherTest program are much smaller compared to other programs. The reason is that there is a very big loop which involves intensive computations. Only several methods are involved in that loop. Consequently, the localities of the DitherTest program are much smaller. By the analysis of method invocation localities during program execution, given a period of time, the method invocation behaviors are confined to a small set of classes or methods. Java Run-time System (object-oriented run-time system) can make use of this feature to support run-time method invocation prediction. With accurate prediction, the performance of programs can be improved.

3.2.4 Consecutive Patterns

The percentages of receiver class consecutive pattern and method class consecutive pattern vary between different programs. The percentages of DitherTest and LinpackJava programs are very high compared to other programs. They are computation intensive programs. The codes for intensive computations are in one or several methods of a particular class. As a result, the dynamic execution behavior focuses on these methods or classes, and produces high percentages of consecutive pattern. Except the ThreadX program, the percentages of receiver class consecutive pattern are always higher than the percentages of method class consecutive pattern.

Compare with the percentages of class consecutive patterns, the percentages of method consecutive pattern are very low. Three programs with high percentages of method consecutive pattern are worth to be mentioned. They are DitherTest, LinpackJava and DigSim. The reasons of DitherTest and LinpackJava with high percentage of consecutive patterns have been discussed. The DigSim program uses a lot of *Vector* objects to store the circuit information. When simulating the circuits, these *Vector* objects' sizes must be retrieved for simulation. As a result, lots of the method consecutive patterns are from

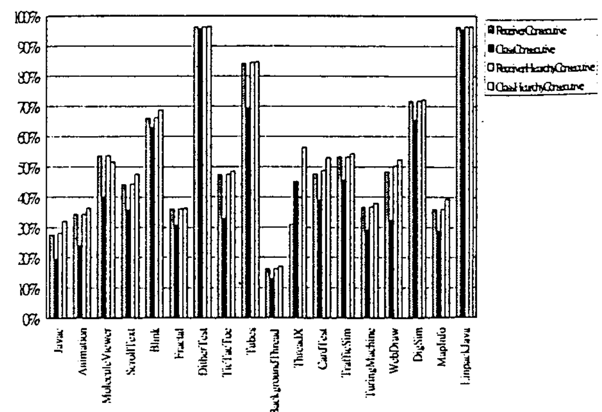

Figure 3-1 Percentages of Receiver/Class Hierarchy Consecutive Patterns

invoking the *Vector.size()* method. There are four values drawn in Figure 3-1. The percentages of receiver-consecutive pattern and class-consecutive pattern are listed here for comparing with the percentages of receiver-hierarchy-consecutive pattern and class-hierarchy-consecutive pattern.

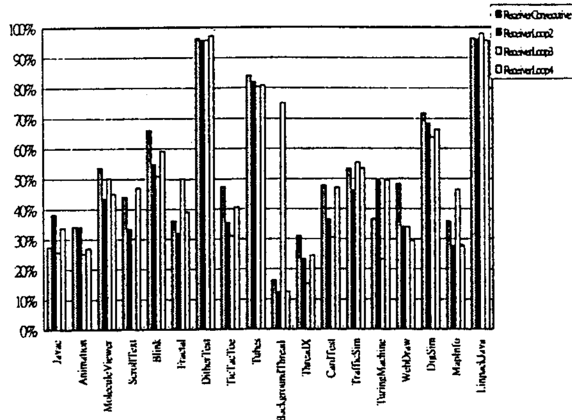

Figure 3-2 Percentages of Receiver Loop-2,3,4 Patterns

3.2.5 Loop-N Patterns

In Figure 3-2, four values are drawn for each program. The first is the percentage of receiver-class-consecutive-pattern that is listed here for comparison with other three values. The second is the percentage of receiver-loop-2 pattern, the third is the percentage of receiver-loop-3 pattern, and the fourth is the percentage of receiver-loop-4 pattern. By Figure 3-2, it is observed that programs with high percentages of receiver-consecutive-pattern usually have high percentages of receiver-loop-N-Pattern. BackgroundThread program is the only one exception. During the execution of BackgraoundThread program, the percentages of receiver-consecutive-pattern, receiver-loop-2 pattern, and receiver-loop-4 pattern are 16%, 12% and 13% respectively, but the percentage of receiver-loop-3 pattern is 75%. From this comparison, we can easily conclude that during the execution of BackgroundThread program, there are many method invocation destinations are the same with its following third method invocation destination. The behaviors and values of the class-loop-N pattern are nearly the same with the receiver-loop-N pattern. Compare with receiver-loop-N and class-loop-N pattern, the percentages of method-loop-N pattern are lower.

General speaking, the occurrence of the same consecutive method invocation are rare. But the occurrence of the same method invocation every two, three or four are more frequent than consecutive situations. Take the Javac program as an example, its percentage of method-consecutive pattern is 7%. While the percentages of method-loop-2, 3, 4 are 24%, 13%, and 24% respectively, which is higher than the percentage of method-consecutive pattern.

Except the consecutive pattern discussed in the previous section, Loop-N pattern is another kind of control pattern. By our analysis, we have found that these Loop-N patterns do exist during the Java program execution, and the percentages of these patterns of some Java programs are high. This also provides another opportunity for compilers or run-time systems to optimize the Java programs (object-oriented programs) execution.

3.2.6 Receiver Class Versus Method Class

There are three kind of relationships between receiver class and method class. When the receiver class and method class are in the same inheritance path, but not the same class, then there will be a distance along the inheritance path between the two classes. In Figure 3-3, the average distances between receiver class and method class of each benchmark programs are shown (the situation that receiver class and method class are the same are not included in the calculation of average distance). Most of the average distances are between 1 and 2. This means that more than half of the messages are sent to the receiver's direct superclass, and others are sent to the superclasses of receiver's direct superclass.

Use the methods defined in superclasses is a feature of object-oriented programming paradigm. It let the programmers reuse the programs that already written by others. But this feature is also a source of run-time overhead. When a method invocation occurs, the run-time system has to search the applicable method upward along the inheritance path. A solution of this overhead is to gather all the applicable methods in a table, then there is no need to search upward the inheritance path. But this solution incurs excessive memory usage.

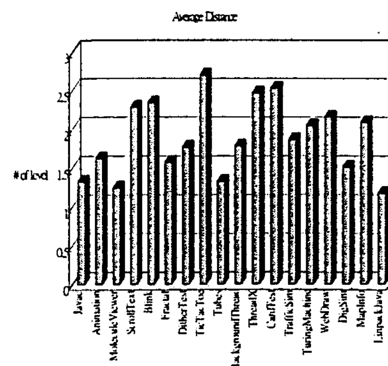

Figure 3-3 Average Distance between Receiver Class and Method Class

By our analysis of the relationships between receiver class and method class, and their respective percentages during

execution, we have found that in most of our benchmark programs, more than half of the method invocations are that receiver class is the same class as method class. For superclass-method invocations, the average search upward levels are below 2. The result of these analysis hint us that merging the searching upward method and table method mentioned in the previous paragraph can improve the performance of method lookup with acceptable increased memory usage.

4. CONCLUSIONS

The behavior of an object-oriented program can be characterized by their method invocation sequence produced during program execution[7,8]. In this paper, Java is chosen as our target programming language, and run-time information is analyzed. The method invocation sequence of a Java program execution is obtained by running the Java program on the modified JVM software implementation. The JVM software implementation of Sun Microsystems is written in C language. The major parts of the implementation we modified are the dynamic class loader and the execution engine. We also designed and implemented an analyzer to analyze the run-time information obtained by running Java programs on the modified JVM software implementation.

We collect 18 Java programs as our benchmark programs. These 18 Java programs is consisted of 8 application domain categories. We expect these 18 programs can be typical representatives of Java applications. After obtaining the run-time information of these benchmark programs, we use our analyzer to analyze the method sizes, native method percentages, method invocation localities, and control patterns.

Method sizes of Java programs are usually very short. Above 50% of them are less than 20 bytes. This indicates that inline these short methods will reduce many method invocation overheads and improve the performance. JVM does not define any I/O instructions, so JVM software implementation needs native codes to deal with program I/O operations. By our analysis, less than 20% of the method invocations are native method invocations in most Java programs.

We evaluate whether exists locality of method invocation during Java program execution. Our analysis reveals that given a short period of time, the method invocations are confined to a small set of classes and methods. The immediate application of this analysis result is to help the method invocation prediction. With more accurate method invocation prediction during program execution, the performance of object-oriented programs can be improved. In this paper, we define two control patterns and evaluate their percentages in the method invocation sequence of the benchmark programs. These two control patterns are consecutive pattern and loop-N pattern. Although the percentages of these patterns vary between different programs, they do exist in the method invocation sequence during program execution. The existence of control patterns provides opportunities to optimize the run-time behaviors of object-oriented programs. Another result of our analysis on the benchmark programs is that most of the method classes of method invocations are the same class with the receiver class rather than the superclasses of the receiver class.

The contribution of this paper is that the run-time behaviors of Java programs are analyzed, and the analysis results provide directions for improving the execution efficiency of object-oriented programs.

5. REFERENCES

[1] David F. Bacon and Peter F. Sweeney, Fast Static Analysis of C++ Virtual Function Calls, OOPSLA'96, San Jose, Calif., pp324-341, October 1996
[2] Jack Dongarra and Reed Wade, Linpack Benchmark -- Java Version, http://www.netlib.org/benchmark/linpackjava/.
[3] Urs Holzle, Craig Chambers and David Ungar, Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches, ECOOP'91, Geneva, Switzerland, pp21-38, July 1991
[4] Shih-Kun Huang, Optimizing Run-Time Behaviors in Object-Oriented Programming Systems, PhD Dissertation of Institute of Computer Sicence and Information Engineering, National Chiao-Tung University, HsinChu Taiwan. 1996
[5] Andreas Krall, Jan Vitek and Nigel Horspool, Near Optimal Hierarchical Encoding of Types, ECOOP'97, Jyvaskyla, Finland, pp128-145, June 1997
[6] Tim Lindholm, Frank Yellin, The Java Virtual Machine Specification, Addison-Wesley, 1997
[7] Wim De Pauw, Doug Kimelman, John Vlisides, Modeling Object-Oriented Program Execution, Proceedings of ECOOP'94, Bologna, Italy, pp163-182, July 1994
[8] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides, Visualizing the Behavior of Object-Oriented Systems, OOPSLA'93, Washington, D.C., USA, pp326-337, October 1993
[9] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, Object-Oriented Modeling and Design, Prentice-Hall Inc., 1991
[10] David Ungar, Randall B. Smith, Craig Chambers and Urs Holzle, Object, Message, and Performance: How They Coexist in Self, Computer, pp53-64, October 1992
[11] David Ungar, Ricki Blau, Peter Foley, Dain Samples, and David Patterson, Architecture of SOAR: Samlltalk on a RISC, The Proceedings of the 11th International Symposium on Computer Architecture, pp188-197, 1984
[12] Jan Vitek, R. Nigel Horspool and Andreas Krall, Efficient Type Inclusion Tests, OOPSLA'97, Atlantas, GA, USA, pp142-157, October 1997.