WHOLE PROGRAM ANALYSIS OF JAVA PROGRAMS FOR VIRTUAL CALLS AND EXCEPTION HANDLING

by

Simrandeep Nagra

Bachelors of Technology, Dr. B.R.Ambedkar NIT, Jalandhar, 2011

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public for the purpose of scholarly research only.

Abstract

Whole Program Analysis of Java Programs for Virtual Calls and Exception Handling

Simrandeep Nagra

Master of Science, Computer Science

Ryerson University, 2014

Java Programs suffer performance degradation due to the presence of virtual calls and the lack of an efficient exception handling mechanism. In this dissertation, we show how virtual calls can be statically resolved to one or two target methods. The resolved calls can then be potentially inlined and hence improve the performance of the program. Analyzing the whole program (including the Java runtime library) instead of only user code has a positive effect on the performance of the program.

We present two exception handling mechanisms, Direct Path Analysis and Display Catch Exception Handling, that improve the performance of programs as compared to the existing popular techniques, Stack Unwinding and Stack Cutting. The first analysis shows that the number of the stack frames needed to be unwound is lower in our analysis than Stack Unwinding. In the second analysis, we propose the Display Catch Exception Handling mechanism which is better than Stack Cutting in terms of operations required to catch exceptions.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 My Thesis

Java programs can be simplified by resolving virtual calls statically to monomorphic or bi-morphic calls. Exception handling can also be implemented efficiently by studying patterns obtained from both static and dynamic analysis.

1.2 Problem Background

Java is a popular language used for writing various applications. Java programs with few method calls performs well but Java programs having a large number of method calls run slowly. One of the major advantages of object oriented languages over traditional languages is the feature of abstraction. This feature is supported by dynamic method dispatch, which is defined as the calling of a method on the basis of the run time type of the object. However, this advantage has turned out to be a factor in performance degradation as most of the calls in Java are dynamically dispatched. Therefore, if we can resolve the method call to one or two target methods, we can eliminate the dynamic dispatch for that call. This will result in a decrease in execution time and performance improvement.

Java suffers more from this run time dispatching than C++ or other similar languages. Therefore, resolving some of these virtual calls at compile time gives

more leeway for performance improvement in Java programs. Other advantages of resolving virtual calls are reduced code size and potential for code inlining.

We are doing whole-program (including the Java runtime library) code generation in our compiler which allows us to access more information than traditional compilers. This helps us in resolving virtual calls for the whole program instead of only user code.

This work resolves the virtual calls using a conservative call graph. A call graph is a conservative call graph only if it has all the edges which we cannot definitely prove to be dead in a program. The virtual call resolution analysis will look for methods in the class hierarchy of a declared type of the call and its subclasses. If we find only one or two possible methods in the hierarchy, the call is deemed to be resolvable.

An application does not always exhibit the expected behavior while running. To make this application robust, there needs to be a mechanism in place to deal with these anomalies. This mechanism to deal with exceptions is called exception handling. Exceptions in programming languages are considered low-frequency events, however there are certain applications where they are used to implement short-circuit evaluation. Java is probably the most widespread language with exceptions as part of the language definition. The Java language provides programmers with a try-catch-finally construct to catch exceptions thrown while running an application. .

Although exception handling improves the performance of programs as compared to previous mechanisms of checking flags or return codes, it still has considerable room for improvement. In conventional Java compilers, exceptions are handled by unwinding stack frames until a matching handler for the thrown exception is encountered. For applications with a large number of exceptions, unwinding stack frames one-by-one can result in performance degradation. Thus, a mechanism which doesn't unwind stack frames one-by-one will make the exception handling process more efficient.

Our work analyzes exception handling. In our static analysis, we find exception catch blocks at compile time and then look for a direct path through the call graph from the exception throw code to the catch block. We also analyzed dynamic patterns of exceptions in the program leading to a new mechanism that

further short-circuits stack cutting.

1.3 Objectives and Proposed Methodology

In this work, we analyze Java programs at static time as well as at run time to obtain statistics about resolved virtual calls. We use the Rapid Type Analysis (RTA) [3] algorithm to resolve virtual calls at static time. We extend the RTA algorithm to analyze whole-program virtual calls. We also look for potential bi-morphic calls using the algorithm. We compare the number of resolved calls including the library with the number of resolved calls without the library.

We analyze exception handling by creating a catch environment around each call site of the program. The catch environment consists of a list of exception handlers having a direct path from the call site. Whenever an exception is thrown, we find the matching handler from the list of the current method's call site, in order to directly unwind the existing stack frame to the stack frame of the matching handler. If no matching handler is found in the catch environment for the current call site, we unwind one stack frame and reinstate the invoking method's stack frame. In this way, we will unwind the stack frame only when there is no direct path available from the call site. We also analyze the number of catch blocks and the exception classes in the program. We propose a new mechanism based on the patterns obtained from these results.

The Java program is then instrumented with suitable code using the Byte Code Engineering Library (BCEL) [4] to obtain the run-time statistics. We use these run-time statistics to compare the results of the static analysis with actual run of a program.

1.4 Contribution

We have extended RTA to resolve virtual calls to monomorphic (one target) or bi-morphic (two targets) calls. We have also extended RTA to include library calls in the analysis, i.e, to analyze the whole program. We have determined the number of unrolls needed to catch a particular exception throw using our catch environment

at static time. We have instrumented the code to generate the run-time statistics of virtual calls, try-catch blocks and the stack depth of the exception handler. We have also proposed a new exception handling mechanism based on statistics gathered by instrumenting the code.

1.5 Dissertation Outline

This dissertation has the following structure:

- Chapter 2 presents some background and related work. This includes discussion of previous work in areas of resolving virtual calls and exception handling.
- Chapter 3 discusses the methodology used for analyzing virtual calls and exception throw-catch patterns statically. This chapter also includes methodology to instrument Java code to enable comparison of static analysis results with an actual run of program.
- Chapter 4 compares results obtained from static analysis with works of other researchers as well as with the run-time statistics.
- The dissertation closes with a chapter on future direction and conclusions.

Chapter 2

Related Work

In this chapter, we first review some virtual call resolution techniques. Then, we review some literature related to exception handling.

2.1 Compiler and Optimization Terms

Some of the compiler and optimization terms we use in our work are defined below:

- Inlining is a process of replacing a call site with the called method's body. This results in improving a program's performance as overhead associated with the method calling such as saving registers, instruction pipelining etc. is reduced [18].

- Short-circuit evaluation is defined as a process of stopping further evaluation of an expression as soon as we determine its value [28]. It is implemented by exceptions in some applications. In these type of applications, an exception is thrown as soon we determine an expression's value. Therefore, the expression's further evaluation is stopped and the exception is caught in the stack frame where this expression's evaluation started.

- Monomorphic calls are the calls having only one possible target method. Bi-morphic calls are the calls having two possible target methods. Polymorphic calls are the calls having more than two possible target methods.

- Benchmark code, also called user code, it refers to the part of the code of a program excluding the library.
- In our work, the termDirect Path refers to the path in which we can directly unwind a stack frame to a matching exception handler in case of an exception thrown.

2.2 Resolving Virtual Calls

2.2.1 Overview

Virtual Method Call, also called Dynamic Dispatch, is defined as calling a method of a class type which is determined at run time. The polymorphism feature of Object Oriented Programming enables an object to refer to an instance of any of multiple classes. The object, on which a method is called, is either an object of the declared type or an object of a subclass of the declared type. The process of selecting the correct method for the object at run time is called dynamic dispatch. The process of determining the correct method consists of vectoring through the Virtual Call Table at run time which results in performance degradation as compared to knowing statically which code to be called.

Consider the code in Figure 2.1, in the `a.foo()` method call, the declared type of the object is class A. But due to polymorphism, the actual type of the object can be A or B or C (subclass of A). Similarly for the `b.foo()`, the object can be of class type B or C and for `c.foo`, the object can only be of class type C.

The Virtual Call analysis will try to determine the possible method(s) that can be called. This analysis must be conservative, i.e. every remotely possible method has to be considered as a possible target method. If a method call can be resolved to a monomorphic call (one target), then it can be turned into a direct call or possibly inlined in the program . A method call resolved to a bi-morphic call (two targets) can be implemented using conditionals. If a method call is not resolved to one or two targets, then we do the dynamic dispatch for this method call. The effective code including inlining for the **main** method after the virtual call analysis is shown in Figure 2.2.

As a result of virtual call resolution, we have simplified the code so it has

```
1  public class A{
2   void foo(){
3    System.out.println("A's foo method");
4   }
5  }
6  public class B extends A{
7   void foo(){
8    System.out.println("B's foo method");
9   }
10 }
11 public class C extends B{
12  void foo(){
13   System.out.println("C's foo method");
14  }
15 }
16 public class Resolve{
17  public static void main(String[] args){
18   A a = new B();
19   B b = new B();
20   C c = new C();
21   a.foo();
22   b.foo();
23   c.foo();
24  }
25 }
```

Figure 2.1: Code for an example program

fewer method calls. Virtual call resolution also results in more efficient hardware implementation of the call as explained in the following section.

2.2.2 Hardware Implication of Virtual Call Resolution

The compiler implements a method call by either an indirect jump or a direct jump. All virtual calls are implemented by an indirect jump while all other calls are implemented by a direct jump since we know the target method of the non-virtual calls at static time. When the compiler encounters a virtual call, it implements the indirect jump by vectoring through the Virtual Call Table for the class of the

```
 1 public static void main(String[] args){
 2   A a = new B();
 3   B b = new B();
 4   C c = new C();
 5   a.foo();
 6   if( b instanceof B)
 7     System.out.println("B's foo method");
 8   else
 9     System.out.println("C's foo method");
10   System.out.println("C's foo method");
11 }
```

Figure 2.2: Figure showing effective code including inlining

object. The compiler gets the called method's address from the table and then jumps to the first instruction of the method.

However, if we resolve the call at static time, the compiler can implement a direct jump. The calls that are resolved to one target can be implemented by a direct jump while the calls that are resolved to two targets can be implemented by a conditional jump. These direct and conditional jumps are friendly to instruction look-ahead. An instruction look-ahead improves the performance of the program as compared to an indirect jump. This will result in a decrease in the execution time of the program.

2.2.3 Virtual Call Resolution Techniques

In this section, we review some of the popular techniques available. The basic requirement of all these virtual call resolution techniques is the construction of a best possible call graph [2][1]. Constructed call graphs must be conservative i.e. the graph must include an edge even if there is only a remote possibility of a program taking that path. Some of these techniques are described in the next few subsections

2.2.3.1 Unique Name Analysis

Calder and Grunwald came up with the first technique to resolve virtual calls [5]. This technique was implemented and tested on C++ programs. They got the idea from the fact that in many cases, there is only one definition for a particular method signature in the whole program. Therefore, any possible call to this method has only one possible target. This enables us to possibly inline that definition in the code body and eliminate the virtual call. The biggest advantage of this technique is that it can even be implemented on bytecode and does not need a flow analysis. But there is also a disadvantage of not using flow analysis. There can be a case in which there are multiple definitions of a method but these definitions are not in the same class hierarchy. In that case, we should be able to resolve the call, but this technique does not facilitate this.

2.2.3.2 Class Hierarchy Analysis

Craig Chambers et al. worked on the idea of Unique Name analyis and modified it to give rise to the analysis called Class Hierarchy Analysis(CHA) [10]. They used the concept of using class hierarchy to determine whether the classes in a hierarchy have only one definition of the method called. They estimated the run time type of an object by rules given below.

- The possible run time types of an object, having a declared class type, are this class plus all subclasses of that class.
- The possible run time types of an object, having a declared inteface type, are the classes implementing this interface or its subinterface and all subclasses of these implementing classes.

To implement this analysis, they construct a call graph based on the class inheritance relationship. Each class has a set of methods defined in that class. The possible run time types are inferred statically using the rules given above. Of the possible run time types inferred, if there is only one definition of the method called, then the method can be statically inlined in the program and the virtual call can be eliminated. They showed that results obtained from this approach were fairly satisfying when compared to other approaches available at that time although this

approach was applied only on C++ programs which have a fairly small number of virtual calls.

2.2.3.3 Rapid Type Analysis

Bacon and Sweeney extended Class Hierarchy Analysis to give rise to a new analysis called Rapid Type Analysis(RTA) [3]. They used the fact that to have a method of a class as a receiver, that class must be instantiated. It can be either instantiated through its subclass constructor or by its own constructor. They refined the call graph by eliminating those classes in the call graph which are not instantiated. This analysis uses a pessimistic approach to construct a call graph. It takes in a more conservative call graph from CHA and then removes non-instantiated classes.

The methodology used to implement RTA starts with the construction of a call graph similar to the one constructed in CHA. Using this graph, visible methods in each class are found. These methods can be those that this class defines or are inherited from the base class. After this step, override and inherit sets are computed. An override set is the set of classes that override a particular class's method while an inherit set is the set of classes that inherit a particular class's method. Then, a program virtual graph is constructed which records method calling. This graph shows calls of both virtual as well as non-virtual type. If a call is virtual, each edge has an extra label which contains possible run time types with which this edge can be executed. In the last step, we will eliminate those possible-class types from the labels which are not instantiated. If there is a virtual edge with only one possible type in its label, we will statically inline that method and erase the edge.

This approach performed better than Class Hierarchy Analysis. They showed by doing experiments using seven benchmarks that Rapid Type Analysis is better than CHA on most occasions. However,they did not analyze the whole application i.e. including the library calls which makes this analysis incomplete as we will show in our work.

2.2.3.4 Variable Type Analysis and Declared Type Analysis

A main aim of designing virtual call resolution is that it should be simple but still should only scale linearly to the size of the program. Sundaresan developed a Variable Type Analysis(VTA) technique [31] which fulfilled this aim and was more precise than Class Hierarchy Analysis and Rapid Type Analysis . In Variable Type Analysis, each variable of object type represents a node.They implemented this technique on the SOOT framework which is an intermediate representation of the code similar to Java bytecode [32]. Intermediate representation of code is converted from source code by compiling it to a class file.

This technique is a flow insensitive technique in the sense that it does not worry about the order of the statements. VTA uses a flow graph where nodes are variables of object type. Each node has its reaching type information with it. Reaching type information is the set of possible types this object may represent. A flow graph is constructed by first adding nodes to it. Every field in a class that has an object reference will be added as a node. Then for every method in a class add to the graph as nodes: every formal paramater and every local variable having object type . For methods having return type, a node labeled return will be added. Once all nodes are created, we add directed edges to this graph. A directed edge is added from right hand side to left hand side whenever an assignment statement is found in the program. Directed edges are also added with every method call. After the flow graph is created, they initialize types for nodes. For intializing types, we track statements of the type lhs= new C(). In this case, the C type is added to the reaching type information of the node labeled lhs. When initializing process ends, we propagate these types along the directed edges. In the end, we will have the type information for every variable.

Another variation of this analysis is Directed Type Analysis(DTA) [31]. This analysis is much simpler than Variable Type Analysis but is less precise than VTA. In DTA, we use declared type of a variable as the node instead of using the variable as the node. This puts every variable having the same declared type into one node. Although DTA constructs smaller graph than VTA, its final output is not as precise.

Both of these analyses have benefits when compared with RTA. These analyses were able to reduce the size of the call graph by removing some methods. When

compared with RTA in resolution of virtual calls, VTA is way ahead. In some cases, VTA removes almost twice as many virtual calls.

2.2.3.5 Iterative Type Analysis and Extended Message Splitting

Chambers and Ungar proposed these techniques to compute the type of a variable in a loop by repeatedly compiling them until a fixed point is reached [7]. They implemented the techniques on their compiler called SELF [6]. Extended Message Splitting is used to preserve the type information, which otherwise would have been lost in the control flow merge, by duplicating all code between merge and places that use the type information.

The compiler builds a mapping from variables names to types at each point in the program. This mapping is changed or computed at each node in a control flow merge. In a merge node, type information is constructed from the type mappings for the incoming branch. The merge type is formed by merging incoming types for this node. The merge type records identities of its constituent types instead of just recording union of types. Whenever a message send is encountered in the loop, if the type of the receiver of this message is a merge type then the message is split across two copies of node from send node back to the merge point. The receiver of each copy of split message send node has a more specific type allowing the compiler to do message inlining.

In Iterative Type Analysis, type binding is used at the head of a loop. We compare binding at the end of the loop to the head. If both of them matches, then we have got to the fix-point. Otherwise, we will recompile the node with new type information received from the end of the loop. If there is a message send inside a loop, then both of these techniques are employed at the same time.

2.2.3.6 Type Feedback

The basic idea of type feedback is to get type information about a variable from previous executions [12]. This information is fed back to the compiler. The type feedback technique adds some code to a program to record the program's type profile for each call site. Based on information received from the feedback, the compiler can easily estimate the type of a variable. This technique is more precise

than earlier techniques discussed beacuse we get all the information by running a program instead of only statically guessing it. Type feedback does not need any data flow analysis.

When and what to recompile are the main questions in type feedback technique. To choose when to recompile, every method has its own counter. If the counter value is larger than a certain limit, we will recompile the method. Besides the counter value, some other factors in choosing what to recompile are increase in information about the method we wish to recompile, how much more we want to optimize the code. The results obtained from this technique shows huge improvement in resolution of virtual calls as compared to other techniques but speed of analysis still remains a big issue.

2.3 Exception Handling Analysis

This section reviews the work done by other researchers in the area of exception handling. We review popular exception handling techniques and the work done to handle exceptions using control flow analysis.

Ramsey and Jones[24] implemented four of the best known techniques for exception handling in a single framework using an intermediate language C--. They categorized these techniques into two groups, Stack Unwinding and Stack Cutting.

2.3.1 Stack Unwinding

This technique walks through the stack method [11]. In this technique, when an exception is raised, the compiler looks for a matching exception handler in the current stack frame. If the matching exception handler is not found, the current stack frame is cleared and the invoking method's stack frame is restored. This process continues until we find the matching exception handler or bottom of the stack is reached. In case of the compiler reaching the bottom of the stack without finding the matching handler, compiler's default exception handler will catch that exception. In this technique, when a method call is made, the compiler saves all non-volatile registers for the calling method. When an exception is thrown and

if we do not find the match handler in the current method, the compiler restores these saved non-volatile registers or callee-saved registers of the invoking method and releases any lock on the objects locked in the current frame. This technique is also known as walking through the stack as it unwinds stack frames one by one to find the matching exception handler.

2.3.2 Stack Cutting

This technique does not unwind the stack frames one by one. It uses a list of try-catch block to find the matching exception handler and therefore directly restores the stack frame of that handler. In this technique, a list is created which records all the try-catch blocks. When an exception is raised, the compiler searches this list to find the matching exception handler. After that, the compiler clears the current stack frame, restores the callee-saved register of that handler and releases locks in the current stack frame. This technique can be faster than the Stack Unwinding if the distance between exception catch and exception thrown point is large. As the Stack Cutting technique doesn't walk through the stack, we do not need to restore the callee-saved registers of every stack frame and can directly jump to the matching exception handler's stack frame. However, in applications throwing few exceptions, Stack Unwinding performs better due to the increased overhead in the Stack Cutting technique owing to creating the list of try-catch blocks.

2.3.3 Exception-Directed Optimization

Ogasawara et al.[22] observed that Stack Cutting is better when exceptions are frequently thrown while Stack Unwinding is better when exceptions are rare and there is no single technique available which is better in both the cases. They proposed a new technique, called Exception-Directed Optimization (EDO),based on profiling the program. EDO starts by profiling the program to find exception paths that are hot. An exception path is the stack trace from the exception throwing method to the exception catching method. All the exception paths are recorded in a list.

When an exception is caught by a path already in the list, the counter for that path is incremented by 1. Otherwise we add the path to the list. After the program has finished its execution, all the hot exception paths are analyzed and a decision is taken about inlining the exception path into the code. Therefore, in the next run of the program, the exception thrown will be caught without unwinding the stack frames. They showed that their technique optimized the exception handling process by a considerable amount.

2.3.4 Exceptional Flow Analysis

Choi et al.[9] implemented a variant of traditional control flow graph, factored control flow graph (FCFG), which is more compact and contains larger basic blocks. In traditional flow graph, a basic block ends when an instruction throwing exception is encountered. But in FCFG, the basic block can contain multiple exception throwing instructions and a control flow edge from this basic block is factored to an appropriate block. As a result, basic blocks are larger and there is more scope for local analysis. They performed local and global analysis using factored control flow graph to find reaching definitions and live variables to handle exceptions. They have incorporated standard data flow analysis into their representation.

Jo and Chang[16] separated the exceptional flow analysis from the normal flow analysis. They reasoned that the programs having mutual dependency between normal flow and exceptional flow are rare and as a result, we can decouple both these flows. They created an exception propagation graph which is similar to the traditional control flow graph. In the exception propagation graph, nodes represents the methods. Edges are labeled with the exception thrown and there is edge from the exception throwing method to the exception catching method. After creating this exception propagation separately, they merged the graph into standard control flow graph. Therefore, if a method throws an exception, the exception propagation graph is used, otherwise the standard control flow graph is used for the analysis.

However, Robillard and Murphy[25] argued that that separating these flows may have consequences. To improve robustness of a program, the developers need

to know which exceptions can flow to a point in the program which is difficult for large programs and the presence of unchecked exceptions in some languages. They described a static analysis tool, called Jex, that provides a view of all exception types and the handlers present at different points in the program. The input of Jex is a java file and it outputs a .jex file which is easily readable. The tools determines the point where an exception is generated, the points to which it is propagated and the point where it is raised. Hence, using this tool we can easily analyze the exception raising and the corresponding try-catch blocks in the program.

Chatterjee and Ryder[8] also proposed an approach to do points-to analysis for programs containing exceptions. They also gave an algorithm that calculates the definition-use pairs for all variables in the program incorporating exceptional control flow path. The complexity of their algorithm is computed to be a polynomial time with worst case complexity as $O(n^5)$. They proved that their analysis is precise when the exceptions do not contain subtyping.

Shelekhov and Kuksenko[27] proposed an approach to make exception structures such as exception blocks at run time while doing data flow analysis. Prior to their paper, all the researchers were either ignoring exceptional data flow or were roughly merging data flow for exceptions with normal data flow. In their approach, implicit control flow for raised exceptions is represented explicitly in data flow analysis.

Sinha and Harold[29] discussed the importance of control flow analysis for various application development related tasks and how an analysis can produce incorrect results if the exceptions are not considered. They presented a technique to construct representations for programs with explicitly thrown exceptions and exception handling constructs. They presented an algorithm to use these representations to perform control flow analysis. They have included the exceptions in the standard control flow graph. This is done by creating an outgoing edge from the exception throwing node and then creating exceptional-exit nodes for the exception types computed using type inference. They proved that the control dependence analysis is affected by the presence of the exception handling constructs. Hence, the control flow analysis must consider the exceptions to produce correct results.

Chapter 3

Materials and Methods

We covered two areas of Java programs in our work: Virtual Call Resolution and Exception Handling. In virtual call resolution, we extended a popular existing algorithm named Rapid Type Analysis to analyze all calls in the programs. We also researched patterns in the programs regarding exception handling. Based on those results, a new mechanism to handle exceptions is formally proposed.

In our work, we analyzed the programs statically to simplify them by resolving virtual calls to monomorphic or bi-morphic calls. We also analyzed programs to implement the exception handling mechanism more efficiently than the existing mechanisms.

In the following section, we give an overview of the compiler we worked on. Then we explain about how we have extended Rapid Type Analysis algorithm in Section 3.2. After that we describe our setup to analyze exception handling statically in Section 3.3. In Section 3.4, we explain the process of instrumenting code to get run time statistics of programs. This chapter closes by proposing a formal mechanism to handle exceptions.

3.1 Our Compiler

We are working on a compiler called OptiJava. OptiJava is an upfront compiler that reads `.class` files and then compiles them to native executable code. It uses code coagulation technique for compilation [20][17]. It is currently under active development.

```
 1 public class A{
 2  void foo(){
 3   ....
 4  }
 5 }
 6 public class B extends A{
 7  void foo(){
 8   .....
 9  }
10 }
11 public class C extends A{
12  void foo(){
13   .....
14  }
15 }
```

Figure 3.1: Class hierarchy and methods example code

OptiJava first loads in the `.class` file and converts the code into basic blocks. A basic block is a sequence of instructions in which the first instruction is an entry point while the last instruction is an exit point. The compiler will not load any method which is not reachable during some possible run of the program, i.e., a method in source code having no call to a method similar to its name or outside the hierarchy logic explained in the next paragraph will not be loaded. This loading process is completed before the code coagulation compilation step. Code coagulation gives us a sequence of compiled instructions which are then converted into machine code using the code generation step.

While loading a virtual call instruction, every possible method is loaded, i.e., the compiler loads methods of all subclasses of the class type referenced by that instruction. For example, if we have the class hierarchy and methods as shown in the code in Figure 3.1. Consider an INVOKEVIRTUAL instruction to method `foo` with declared type of A. We will load the `foo` method of class A as well as the `foo` methods of classes B and C. The loading of all possible methods is required to maintain a conservative call graph, which will be used in the virtual call resolution algorithm. We also record all classes instantiated by NEW instructions. This helps in determining live call instances in the call graph.

3.2 Virtual Call Analysis

We have used the Rapid Type Analysis(RTA) algorithm to resolve svirtual calls. RTA looks at class hierarchy and resolves all those calls that have only one or two definitions of the same method in a single hierarchy. It uses the additional fact that a call to a method is only possible if that class is instantiated.

One limitation of Rapid Type Analysis, as described in [3], is that it does not analyze library calls. The reason for the exclusion of library calls is that the authors of RTA were analyzing source code to create a call graph. This would result in analyzing huge numbers of methods if they were to include library calls in their technique. We do not have this problem since we are analyzing using bytecode. As mentioned in the overview of our compiler, the compiler doesn't load all the methods in a class and loads only those methods that are possible to reach in a run of the program. This has helped us in extending Rapid Type Analysis to analyze library calls as well. We have also extended the algorithm to resolve calls that have two possible targets using type intersection.

We will now explain the creation of data structures using steps from the original RTA algorithm. Then, we explain our extension of the second and third steps of that algorithm, eliminating non-live instances and resolving calls.

3.2.1 Constructing Data Structures

The first step of virtual call analysis is to create conservative call graph instances. Algorithms 1, 2 and 3 create the call graph instances. Algorithm 1 creates a set of visible methods using the class hierarchy. The visible method set is a set of tuples, $< c, m, d >$, where m is a method, c is a class in which m is visible and d is a class in which m is defined. Initially, the visible-methods set is empty. Line 2 of Algorithm 1 assigns topological numbers to all classes in such a way that the number for a class is greater than number for its parent class. Line 4 adds initial tuples of visible methods to the visible methods set. It goes through all the methods of every class creating tuples with visible-class and defining-class referring to the same class.

Lines 6-16 of the algorithm traverse through every class starting from the root class and check every parent-child subset represented by elements of D. In the case

Algorithm 1 Algorithm to BuildCHG (C, M, D) (as in [3])

```
1:  V ⇐ φ
2:  assignTopologicalNumbers(D)
3:  for all m ∈ M do
4:      V ⇐ V ∪ < ClassOf(m), m, ClassOf(m) >
5:  end for
6:  for i = 1 to NumberofClasses do
7:      Let c ⇐ x ∈ C : TopNum(x) = i
8:      for all b ∈ C : < b, c > ∈ D do
9:          for all m ∈ M, d ∈ C : < b, m, d > ∈ V do
10:             if ∃_{n∈M,e∈C} < c, n, e > ∈ VandSig(m) = Sig(n) then
11:                 if TopNum(d) > TopNum(e) then
12:                     V ⇐ V − {< c, n, e >}
13:                     V ⇐ V ∪ {< c, m, d >}
14:                 end if
15:             else
16:                 V ⇐ V ∪ {< c, m, d >}
17:             end if
18:         end for
19:     end for
20: end for
```

of both child and parent class having a visible method with the same signature but defined by different classes, we use the topological number assigned to each class to find out the class which defines the visible method. The class with a greater number is deemed to be the class with more specific definition, i.e. the method defined in a class with greater number overrides the method defined in a lower numbered class and hence tuples are modified as shown in lines 12-16.

The next step in creating virtual call graph instances is to create inherit and override sets. The inherit set contains all those classes that inherits a particular method while the override set contains all those methods that overrides a particular method. These sets define boundaries for a particular method. The inherit set is used to find out possible types that can invoke a method. The **BuildFrontier** algorithm is used to create these sets. Initially, for every visible method tuple constructed in the previous algorithm, the inherit set contains the visible class of that tuple and the override set is empty.

Algorithm 2 Algorithm to BuildFrontier (C, D, V) (as in [3])

```
1: for all v ∈ V do
2:    Let< c, m, d > = v
3:    Override(v) ⇐ φ
4:    Inherit(v) ⇐ {c}
5: end for
6: for i = NumberofClasses to 1 do
7:    Let c ⇐ x ∈ C : TopNum(x) = i
8:    for all m ∈ M, d ∈ C :< c, m, d > ∈ V do
9:       Let v =< c, m, d >
10:      for all b ∈ C :< b, c > ∈ D do
11:         if ∃_{n∈M,e∈C} < b, n, e >∈ VandSig(m) = Sig(n) then
12:            Let w =< b, n, e >
13:            if d = e then
14:               Inherit(w) ⇐ Inherit(w) ∪ Inherit(v)
15:               Override(w) ⇐ Override(w) ∪ Override(v)
16:            else
17:               Override(w) ⇐ Override(w) ∪ {v}
18:            end if
19:         end if
20:      end for
21:   end for
22: end for
```

Algorithm 2 starts from a leaf node and goes all the way up to the root node. Lines 8-13 compare visible method tuples in the parent class with tuples in the child class and if a pair of tuples is found having the same method signature, we check the defined-class value of that tuple. If the defined class is the same for both the tuples, then the inherit set for the parent class is updated by adding the child class to the set. Otherwise the inherit set remains the same. The override set is updated in both the scenarios as shown in lines 15 and 17.

After constructing inherit and override sets, we move on to construct a Program Virtual Graph(PVG). The Program Virtual Graph consists of call-instance edges which define the characteristics of a particular call. A call instance is a tuple $< s, f, t, P >$ where

- s is a call site in the program.

Algorithm 3 Algorithm to BuildPVG $(\sum_D, \sum_V)$ (as in [3])

1: $I \Leftarrow \phi$
2: **for all** $< s, f, g > \in \sum_D$ **do**
3: $\quad I \Leftarrow I \cup \{< s, f, g, \perp\}$
4: **end for**
5: **for all** $< s, f, v > \in \sum_V$ **do**
6: $\quad$ **addVirtualInstances**(s, f, v)
7: **end for**
8:
9: **addVirtualInstances**$(s \in S_V, f \in F, v \in V)$
10: Let$< c, m, d >= v$
11: **if** $< s, f, m, Inherits(v) > \in I$ **then**
12: $\quad$ **return**
13: **end if**
14: $I \Leftarrow I \cup \{< s, f, m, Inherits(v) >\}$
15: **for all** $w \in Override(v)$ **do**
16: $\quad$ **addVirtualInstances**(s, f, w)
17: **end for**

- f is the calling method at call site s.
- t is a target method that can be possibly reached from the call site s.
- P is a list of possible class types upon which this instance can be invoked i.e. t is a possible target method only if the actual object type is among this list of classes. $P = \perp$ for direct calls.

Algorithm 3 divides calls into two groups: direct calls and virtual calls. Call Instances of direct calls are easy to make as the target method is already known for these calls. But for virtual calls, we use inherit and override sets to create call instances. Lines 9-17 of the algorithm create call instances for all the call sites that are virtual.

v, in Line 10, is a visible method tuple depending upon the static type of the call i.e a tuple having visible-class as static-type of the call or its subclass. Line 11 checks whether the call-instances set has the particular call instance or not. If true then no action is taken, otherwise it adds the particular call instance to the

Algorithm 4 Algorithm to RemoveNonLiveInstances (I) (derived from[3])

```
1:  flag ⇐ false
2:  Method ⇐ φ
3:  for all i ∈ I do
4:     Let < s, f, t, P >⇐ i
5:     for all p ∈ P do
6:        if isInstantiated(p) then
7:           flag ⇐ true
8:        end if
9:     end for
10:    if flag = false then
11:       I ⇐ I − {i}
12:       Method ⇐ Method ∪ t
13:    end if
14:    flag ⇐ false
15: end for
16:
17: for all i ∈ I do
18:    Let < s, f, t, P >⇐ i
19:    if Method ∩ f ≠ φ then
20:       I ⇐ I − {i}
21:    end if
22: end for
```

set. Line 16 recursively calls **addVirtualInstances** method for all elements of the override set corresponding to tuple v.

3.2.2 Removing Dead Instances and Resolving Calls

We have constructed a virtual call-instances set in the previous sub-section. But this call-instances set contains some instances in which no class (in possible class types element) is instantiated. A class is said to be instantiated only if **new** method is called for that class. Class instantiation due to instantiation of child class is not deemed as instantiation in our algorithm. In our compiler, whenever we encounter a **new()** instruction, we add an instantiated class tag to that class. This tag is used to remove dead instances in Algorithm 4.

To remove dead instances, we iterate over all the call instances set I. For every element of that set, we get a list of possible class types, i.e. P. If at least one of

Algorithm 5 Algorithm to ResolveCalls (S_V, I) (derived from[3])

```
1: Let S1 ⇐ φ, S2 ⇐ φ,inferredType⇐ φ, Q ⇐ φ
2: for all s ∈ SV do
3:    inferredType ⇐ s.DeclaredTargetMethodType().getAllSubClasses()
4:    for all s :< s, f, t, P >∈ I do
5:       if P∩inferredType ≠ φ then
6:          Q ⇐ Q + 1
7:       end if
8:    end for
9:    if Q = 1 then
10:      S1 ⇐ S1 ∪ {s}
11:   else if Q = 2 then
12:      S2 ⇐ S2 ∪ {s}
13:   end if
14: end for
```

the classes in P has an instantiated tag, then the call instance element is live. If we determine that the call instance is dead i.e. no class in P is instantiated, then we remove that call instance from the set and also add the target-method value of this instance to the *Method* set.

After we have iterated over the call-instance set once, we iterate over that set a second time. This time we check for calling-method value of call instance elements. If we encounter a calling method value of the call instance that is contained in the *Method* set, then we remove that call instance from the original call instance set. This is because if a method is deemed to be unreachable from the first iteration, then all the call sites that are in the unreachable method are dead as well.

After we have eliminated dead instances, we implement the final step of virtual call analysis, i.e. type intersection to find out the resolved call sites. In this step, we check for intersection between inferred types from declared static type of object at call sites and list of possible class types for a call instance. Line 3 of Algorithm 5 calculates the inferred types for a virtual call site. Inferred types consist of declared object type at call site and all subclasses of this declared object type. The **getAllSubClasses** method in that line returns subclasses of the declared type as well as the transitive closure of the subclasses of the declared type.

The call instances set contains multiples call instances with the same call site. So, for every virtual call site, we first find all the call instances with the same

call site. Then for instances with the same call sites, we intersect inferred types with the list of possible class types obtained from the call instances individually as shown in Lines 4 and 5. If we find only 1 instance for a particular call site with non-empty intersection result, then that call site is recorded as resolved to one target method. If we find 2 instances for a particular call site with non-empty intersection result, then that call site is recorded as resolved to bi-morphic call site i.e two target methods. All the calls sites with more than 2 instances having non-empty intersection result are not resolved. The resolved call sites can then be converted to monomorphic or bi-morphic calls (and possibly inlined) to reduce the number of dynamic method calls in the program, thereby improving performance and reducing code size of the program.

3.3 Exception Handling Analysis

Java programs do not always behave as they are supposed to. There can be anomalies while running a Java program. These anomalies are called exceptions. If exceptions happen in an application, they tend to crash the whole application. Therefore, it is important to catch these exceptions and do some safe operations that prevents the application's crash. This mechanism of catching exceptions is called exception handling.

Exceptions are considered to be low frequency events in Java programs, but there are some applications such as searches where exceptions are considered as a tool to do short circuit evaluations. Hence, there is a need to implement exception handling more efficiently to improve the performance of programs with large number of exceptions.

3.3.1 Exception handling in JVM

The Java programming language has try-catch-finally blocks to handle exceptions. These blocks are placed around the code which is expected to raise exceptions. If code inside a try block raises an exception, the JVM checks for a matching catch block. A matching catch block is a catch block whose handling catch type is either the exception raised or a super class of the exception raised. If a matching block

is not found in the current method, then it searches for matching catch block in the calling method and so on until it finds a matching catch block. If a matching catch block is not found in the user written program, then the default catch block of the runtime environment handles it by printing out the call stack.

According to the JVM specifications[19], if a matching exception handler is not found in the current method, then JVM clears the current method's stack frame and reinstates the invoking method's frame and the exception is re-thrown in the reinstated stack frame's context. If a matching handler is not found even in the invoking method then its stack frame is also cleared and the stack frame of its invoking method is restored. This process goes on till we find a matching handler or bottom stack frame is reached. This process of clearing stack frames one by one is called stack unwinding. Conventional compilers use this stack unwinding approach. Stack unwinding is a very inefficient approach if the distance between the exception raising stack frame and the exception handling stack frame is large.

In the following subsection, we will statically analyze the exception handling. We have set up experiments to determine the number of times we have to cut the stack to find a matching exception handler. We have also performed experiments to find the feasibility for a new mechanism which is proposed in section 3.5.

3.3.2 Stack Unwinding and Stack Cutting Approaches

In this subsection, we give an overview of what happens inside the compiler in both of the techniques. In the Stack Unwinding technique, during the normal execution:

- If there is a method call, the compiler saves registers and then goes to the called method.

When an exception is thrown:

- The compiler saves the current register state and searches for a suitable handler in the current stack frame.
- If a suitable handler is not found in this method, the current stack frame is cleared and the invoking method's stack frame is reinstated.
- At each unwind, the saved register state is restored to the state associated with frame which is being reinstated.

- If the matching handler is found in a stack frame, the compiler starts execution from the start of the shandler PC.
- If the bottom stack frame is reached and no matching handler is found, control is passed to the run-time environment to handle the exception using the default handler.

In the Stack Cutting technique, during normal execution:

- On entry to a try-catch block, the block is added to the list of exception handlers.
- If there is a method call, the compiler saves registers and then go to the called method.

When an exception is thrown:

- The compiler saves the current register state and searches in the list for the matching exception handler starting with most recent try-catch block.
- If try-catch block does not match with the exception thrown, the block is popped from the list and the next try-catch block in the list is compared for the matching handler.
- If the matching handler is found in the list, the current stack frame is cleared and the stack frame associated with that handler is reinstated.
- The saved register state is restored to the state associated with frame having the handler.
- If no matching handler is found in the list, the control is passed to the run-time environment to handle the exception using the default handler.

3.3.3 Static Exception Handling Analysis

We have divided this analysis into two categories. The first category analyzes whether there is a direct path from where the exception is raised to the exception handler. A direct path is defined as a path where there is only one path available

from the exception point to the catch point and there are no methods in that path that are recursive (strongly connected components of the call graph). This analysis helps us in understanding the patterns related to an approach similar to stack cutting. The second category will analyze the number of catch blocks in a single program. The patterns obtained from second category analysis is used to propose a new mechanism.

3.3.3.1 Direct Path Analysis

Direct Path analysis determines if there is a direct path from the exception point to the exception handler. If there is no direct path, then we find out how many strongly connected components (SCC)[1] of the call graph or methods with multiple incoming edges there are between the exception handler and the exception raising point.

The first step of this analysis is to construct a directed call graph. This call graph has its root at **main()** and method calls represented as edges. If there are multiple calls from a single method, then that method will have multiple outgoing edges. Once the call graph is constructed, we identify SCC's in the graph. The reason for identifying SCC's is that since methods in a strongly connected component are recursive, we can not be sure about the number of stack frames in the SCC and cannot trace a definite path to the catching environment. Therefore, we need to unwind the stack frames one by one until we find a matching handler or we get out of the SCC. We have used the Depth First Search (DFS) algorithm to identify SCC's. Firstly, we do the depth first search in reverse order. We reverse the edges of the graph and perform DFS on the reversed path. In the second pass, we do the DFS on the original graph with decreasing post number for nodes obtained from DFS on the reversed graph. The post number is a number given to every node in the DFS. The post number follows a property that "if C and D are strongly connected components, and there is an edge from a node in C to a node in D , then the highest post number in C is bigger than the highest post number in D". The strongly connected component algorithm is summarized as follows:

[1] A strongly connected component of a directed graph G is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v, there is a directed path from u to v and a directed path from v to u"[21].

- Perform DFS(G^R) ,where G^R is reversed graph, to find ordering of nodes.
- Perform DFS(G) on graph G by considering vertices in decreasing order of the post numbers obtained from previous step.
- A set having number of vertices greater than 1, obtained from each tree after DFS in previous step, is an SCC for our analysis.

The next step in our analysis is to create a catch environment around each method node. Fig. 3.1 shows an example of catch environment created for the call graph. A, B, C, D etc. are nodes representing the methods. C1, C2, C3 etc. are call sites while ce1, ce2, ce3 etc. are the exception handlers for these call sites respectively. In the graph there are 3 categories of lists:

- **Catch Classes**: is a list of all exception handlers to which a call site have a direct path .
- **Union Classes**: is a list of exception handlers for a node having multiple incoming edges out of which at least one edge is not from the node contained in the same SCC as this node.
- **SCC Catch Classes**: is a list of exception handlers available in a SCC corresponding to a particular node.

The process to create these lists are as follows:

- Add an exception handler object for a call site to the Catch Classes list of that call site.
- Starting from **main** method, for every call, if the called method doesn't have multiple incoming edges and is not a part of a SCC, add all elements of the catch classes list of the calling site to the catch classes list of all the call sites of called method.
- But if the called method has multiple incoming edges and the called method is not in the same SCC as the calling method, then add the exception handler of the calling site only to the union classes of the called method's call sites.

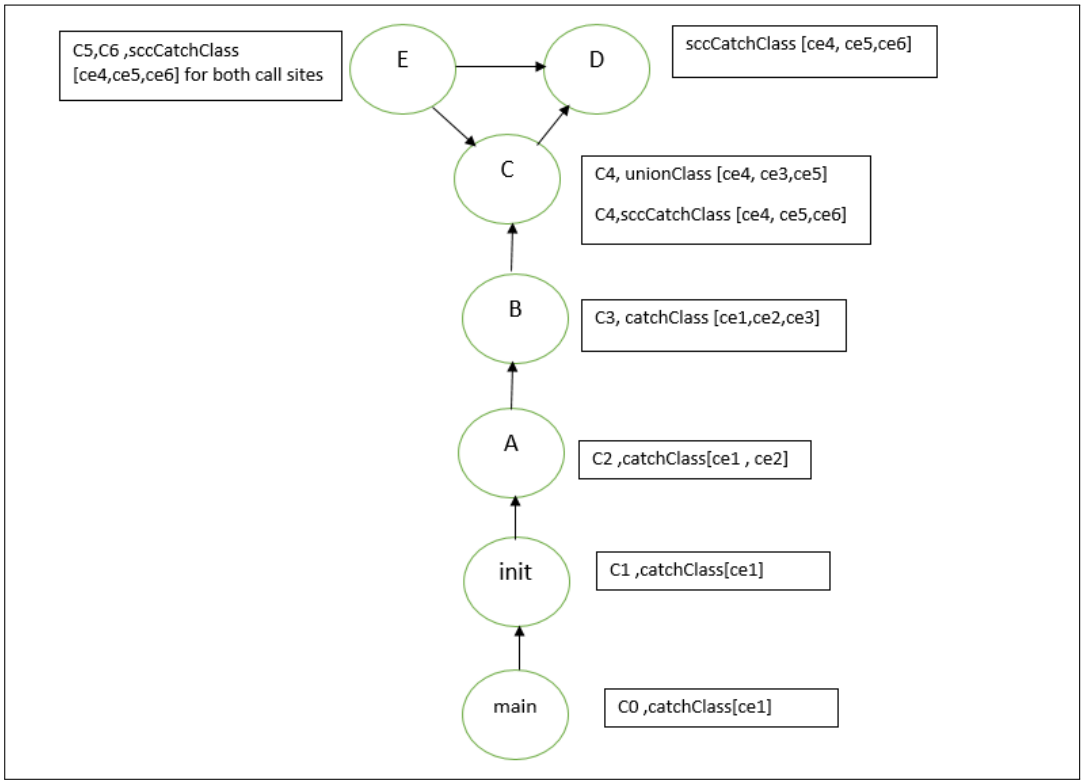

Figure 3.2: Figure showing catch environment needed for our analysis

- Else if both the calling method and the called method are in same SCC, then add all the elements of scc catch classes of calling method to the scc catch classes list of all the call sites of called method.
- Traversing of call edges should be breadth first to maintain the order of the exception handler's occurence.

Suppose we have an exception raised in method D and we have a matching handler in the main method. As D is a part of a SCC, it will check for SCC catch classes i.e. whether any of the exception handlers ce4,ce5 or ce6 is a matching exception handler. As a matching handler is not found, we unwind one stack frame and move to call site C4 of method C. Now as it is a node having multiple incoming edges out of which at least one edge is not from the node contained in same SCC as this node, we check for **Union Classes** recorded at this node for a matching exception handler. As a matching handler is still not found, we unwind another stack frame. We clear the current stack frame and move on to the stack frame of handlers recorded in **Union Classes** of that method. This means we now have a stack frame of either ce3 or ce5. Suppose we have unwound to ce3, then we have a direct path to the matching handler in main method as found in **Catch Classes** of call site C3.

For the above example, the stack unwinding approach would have unwound through 1 SCC and 4 other simple unwindings. The Stack Cutting would have unwound 6 stack frames to find a matching handler. Our analysis shows we need to unwind stack frames through 1 SCC and 2 other simple unwinds. Thus, our approach would have saved some time to unwind to the matching handler even in this example with atypically many try-catch handlers.

3.3.3.2 Analyzing catch categories

Analyzing catch categories determines the number of catch blocks initialized in a program. We have proposed an exception handling mechanism described in Section 3.5 based on patterns obtained from this analysis.

The analysis starts with recording catch block characteristics as we encounter a catch block while traversing the call graph. The characteristics recorded are the catching class for that block and the number of similar catch blocks already in the

program. After we have recorded all the catch blocks in the program into our data structures, we analyze the patterns in these characteristics. The analysis finds the following patterns in the program:

- How many categories of exception classes are there in the whole program.
- How many catch blocks are there for a particular exception class in the whole program.
- How many catch blocks of the subclasses are there for a particular exception class in the whole program.

We will describe the point of finding these patterns in Section 3.5, where we explain our proposed exception handling mechanism.

3.4 Instrumenting Java Code for Run-Time Analysis

We have described our approaches to the static analysis in the previous sections. But we need to compare our static analysis statistics with run time statistics to prove the actual usefulness of our static analysis[2]. For this purpose, we have instrumented the code to get the run time analysis. We have added extra code to the actual code in a java program. This extra injected code gives us statistics when the program runs. We have used the Byte Code Engineering Library (BCEL)[4] to instrument the code. The instrumentation process for our code is divided into two parts: instrumenting for virtual calls analysis and instrumenting for exception analysis.

We have made our own class, **StatRecord**, to record the information from the instrumentation process. The StatRecord class has five static methods namely **recordCatchInitialize**, **recordCatch**, **recordThrow**, **VirtualCallsRecord** and **print**. The recordCatchInitialize method is called whenever a try-catch block is encountered. The recordCatch method records the call stack at the point of exception catch while the recordThrow method records the call stack at the point

[2]Our compiler is not yet generating code

of exception thrown. The VirtualCallsRecord method is called whenever a virtual method call happens and records the information as to whether the call is resolved or not. The print method is called at the end of a program execution to print out statistics about the analysis.

Firstly, we have instrumented code to get the virtual call analysis results at run time. We get a list of call sites resolved to one target, two targets or more than two targets from our static analysis of virtual calls. To instrument the code we need to go through the instruction list of the calling method and wherever we find **INVOKEVIRTUAL** or **INVOKEINTERFACE** instruction, we add code before the instruction.

BCEL provides us with **ClassGen** and **MethodGen** classes to manipulate a class or a method of a given program. The MethodGen class has a method called **getInstructionList()** which gives the instruction list of that method. Now, we go through our static analysis result and instrument the code into the calling method of each call site which is statically analyzed. The code snippet for making a MethodGen object and getting instruction list is shown in Figure 3.3.

After getting the instruction list of the method, we iterate through the instruc-

```
1 MethodGen theMethod = new MethodGen(method, className
    ,constantpoolgen);
2 InstructionList ilist= theMethod.getInstructionList()
    ;
```

Figure 3.3: Code showing how to get instruction list using BCEL

tion list. When we find an instruction that is equal to the instruction representing our analyzed call site, we add an instruction which calls the **VirtualCallsRecord** of class StatRecord. The VirtualCallsRecord method has an argument which represents whether the call site has been resolved to one, two or more than two targets. The code showing the instructions added is shown in Figure 3.4:

In the above code, the LDC instruction pushes a resolvedCount variable to the stack which is then used by INVOKESTATIC instruction as its argument. If a call site has been statically resolved to one target, then resolvedCount = 1 or if a call site has been resolved to two targets, then resolvedCount = 2. Resolved Count = 3 for more than 2 targets possible for a call site. The VirtualCallsRecord

```
1 Iterator iterator= ilist.iterator();
2 while(iterator.hasNext()){
3  InstructionHandle ih = (InstructionHandle)iterator.
     next();
4  if(ih.getInstruction()==callSite.getInstruction()){
5   ilist.append(new LDC(pgen.addInteger(resolvedCount)
     ));
6   ilist.append(new INVOKESTATIC(pgen.addMethodref("
     StatRecord","virtualCallsRecord","(I)V")));
7  }
8 }
```

Figure 3.4: Code showing how to instrument a call site

method keeps a counter which keeps track of number of calls resolved to one, two or more than two targets and we print out these numbers when the program has finished its execution.

After instrumenting the code for the virtual call analysis, we then instrument the code for exception handling analysis. For instrumenting exception handling, we extract the exception table for a method using the MethodGen class. MethodGen class has a **getExceptionHandlers** method, which gives a starting instruction handle and an ending instruction handle for a try block. When going through the instruction list if we find an instruction handle equal to the starting instruction handle of an exception handler of that exception table, we instrument the code at that point to indicate an initialization of the catch block. The code instrumented is an instruction calling **recordCatchInitialize** of our analysis code . This method has one argument, which is the type of exception class handled by the catch block initialized.

After instrumenting the catch block initialization, we instrument the code at the point where the exception is thrown and at the point where exception is caught. The code for that is shown in Figure 3.5.

In the first `if` statement, we record the thrown exception object in our code while in the next `if` statement, we have made a new exception object at every catch block with the same exception catch type as this block can handle. We then record this newly created exception object in our analysis code. To get the number

```
1  if(ih.getInstruction() instanceof org.apache.bcel.
     generic.ATHROW){
2   ilistNew.append(new DUP());
3   ilistNew.append(new INVOKESTATIC(pgen.addMethodref("
      analysis.StatRecord", "recordThrow", "(Ljava/lang
      /Throwable;)V")));
4  }
5  if(ih==ceg.getHandlerPC()){
6   ilist.append(new NEW(pgen.addClass("java.lang.
      Throwable")));
7   ilist.append(new DUP());
8   ilist.append(new INVOKESPECIAL(pgen.addMethodref("
      java.lang.Throwable", "<init>","()V")));
9   ilist.append(new INVOKESTATIC(pgen.addMethodref("
      analysis.StatRecord", "recordCatch", "(Ljava/lang
      /Throwable;)V")));
10 }
```

Figure 3.5: Code showing how to instrument exception handling process

of stack frames unwound in the actual program while handling an exception, we subtract the length of call stack for the newly created exception object from the call stack length of the object thrown by instruction ATHROW.

We do not need to do any extra instrumentation to get run time results of our 'direct path' static analysis. We have used the call stack obtained from the above recorded throw and catch objects to do the direct path analysis.

3.5 Display Catch Exception Handling

In this section, we discuss our proposed mechanism the aim of which is to make exception catch cheaper. Our mechanism uses a table containing an entry for each catch category. This entry is a pointer that points to Catch Save Block(CSB) on the stack. The CSB contains information about the stack to be used on a throw. The table used by our mechanism is shown in Figure 3.2.

Our mechanism starts with reserving a space on the display catch table for every exception class which has a specified catch block in the program. In a run of the

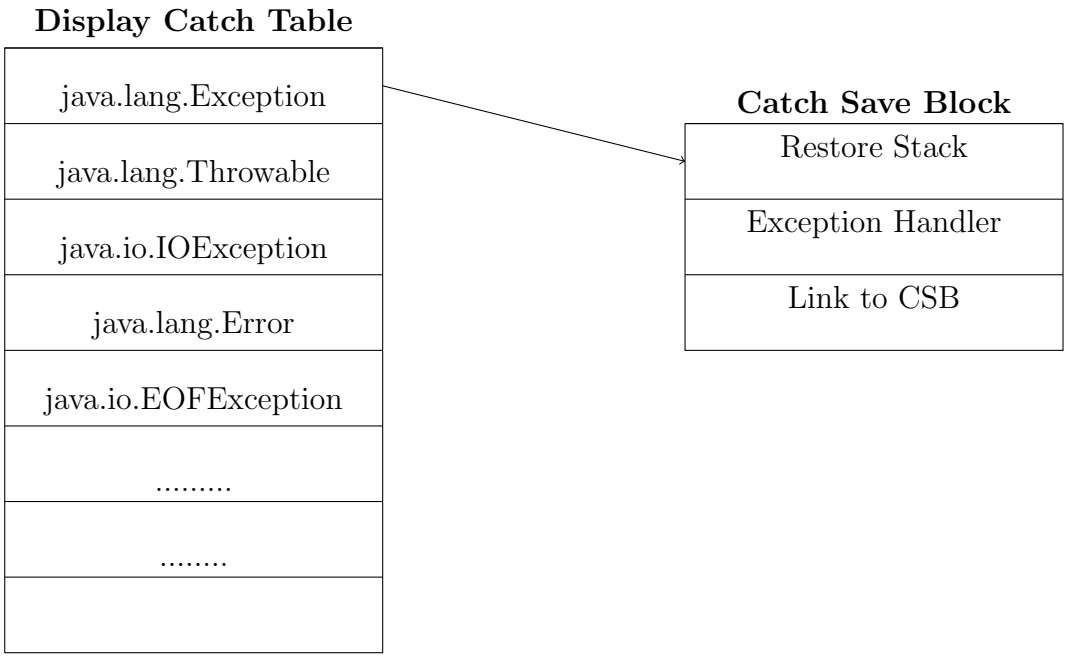

Figure 3.6: Figure showing Display Catch Table

program, whenever we execute a try-catch block, we initialize a CSB on the stack. The CSB contains the stack pointer for restore, an exception handler address and a link to the previous CSB which is the current entry in the display catch table for the particular class handled by this catch block. Then, we store the address of the Catch Save Block into the class's spot in the table.

Now, when we encounter an exception throw, we reference the display catch table entry for the exception thrown. The address pointed to by the entry is then compared with the address pointed to by the Display Catch Table entry for all super-classes of the thrown exception class. The address which is highest on the stack i.e. nearest catch block of the class or its super-class is determined to be the exception handling block.

The previous CSB link saved in the Display Catch Table is used to reconstruct the display catch table once we have restored the stack to the matching exception handler. We update the entry in the table for the current exception thrown class to the address found at the back link portion of the saved address. All other entries in the table are also updated using back link of their respective entries, until we get all entries in the table pointing to an address lower than the current stack address.

This mechanism aims to make catch handling more efficient than both stack cutting and stack unwinding. As we can go straight to the matching exception handler's stack frame without unwinding the stack frames in between, it makes this approach more efficient for exception catching. Although it results in more overhead due to registering of catch blocks than the stack unwinding approach, will be faster due to less stack unwinding. When compared with stack cutting, overhead for registering catch blocks is the same because stack cutting also registers a list of all catch blocks. But our approach will result in fewer stack frames unwound resulting in performance improvement. We can analyze the resulting overhead using a static analysis of catch categories.

As mentioned in section 3.3.2.2, we look for the following information:

- Number of the exception classes in the program for which a catch block exists.
- Number of the catch blocks for a particular exception class.
- Number of the super classes for each exception class.

The number of the exception classes determines the size of the display catch table. A small number of exception classes will indicate that we have to reserve less memory for the table. The dynamic number of catch blocks for a particular exception class indicates the maximum number of times we will have to update the entry for a particular class in the table. The overhead will remain low if the number of catch blocks is small as we then have to update the entries less frequently. The last part of catch categories analysis is establishing the number of super classes for each exception class. This gives an estimate of how many comparisons we have to do to get a matching handler. As discussed earlier, when an exception is thrown, we compare the address found at a particular exception class Display Catch Table entry with the addresses at the entry of super classes to find the address nearest the top of stack. So, a small number of super classes for the entry class will result in fewer comparisons.

3.5.1 Display Catch Exception Approach

In section 3.3.2, we discussed what the compiler does in the Stack Unwinding and Stack Cutting techniques. Now, we will discuss what the compiler will do in our proposed mechanism.
In our proposed mechanism, during normal execution:

- If the execution enters a try-catch block, the address of the Catch Save Block created for this try-catch block is stored in the Display Catch Table at the spot of the exception class handled by this block.
- If there is a method call, the compiler saves registers and then goes to the called method.

If an exception is thrown, our mechanism handles the exception by the following steps:

- The compiler saves the current register state and references the Display Catch Table for the thrown exception class.
- The address at the entry of the thrown exception class is compared with the address at the entry of its super classes and the handler having address closer to the current top of stack address is deemed to be the matching handler.
- The Display Catch Table is restored to the state associated with the re-instated frame by using the back link of the Catch Save Block for every exception class in the Table.
- The address of the back link is compared with the target stack frame address and the back links are traversed recursively until all the entries in the Display Catch Table have lower address than the target stack frame address.
- After that the current stack frame is cleared and the stack frame at the target address is reinstated.
- If no matching handler is found in the table, control is passed to the run-time environment to handle the exception using the default handler.

3.6 Benchmarks

We have experimented on the following Java applications to obtain results discussed in the next chapter:

- **JavaCC** is a Java parser generator [14], previously known as **jack**.
- **Pizza** is a compiler which compiles Java programs [23].
- **SableCC** is a compiler front end written in Java[26].
- **Compress** is a compression method based on Lempel-Ziv method and is part of the specJVM2008 benchmark [30].
- **Jlex** is a lexical analyzer generator written in Java [15].
- **JavaBinHex** is a binhex decompressor in Java [13].
- **Javac** is also a compiler which compiles Java programs. It is a part of specJVM98 benchmark set.

We have used the first six benchmarks to experiment with the virtual call analysis. The reason for not experimenting with `Javac` in the virtual call analysis is that `Javac` is a part of the library in Java programming. Therefore, it is not possible to get the comparison of only the benchmark code v/s the whole application.

We have ignored the `JavaBinHex` and `JLex` benchmarks for the experiments with Direct Path Analysis as these benchmarks throw no exceptions. We have ignored only the `JavaBinHex` benchmark for the analyzing catch categories analysis due to the lack of a sufficient number of catch blocks needed to analyze in the benchmark.

Chapter 4

Results

In this chapter, we discuss results of the analysis described in chapter 3. This chapter is divided into two sections; the first section discusses the results of virtual call resolution analysis while the second section discusses the results of exception handling analysis.

4.1 Virtual Call Analysis Results

In this section, we discuss the results obtained after experimenting with the virtual call analysis on the benchmarks discussed in Section 3.6.

Table 4.1 summarizes the static results of the virtual call analysis. The 'Benchmark Code Only' column contains the statistics for analysis without including library calls while the 'Whole Application' column has the analysis results including the library calls. The `JavaBinHex` and `Compress` programs after analyzing only the benchmark code have all their call sites, except one, resolved to one target statically. With the inclusion of library calls in the analysis, the call sites resolved to one target drops to about 80% of the total call sites.

All other programs exhibited behavior similar to each other. The number of call sites resolved to one target ranges between 80-90% without including the library calls. The static analysis including the library calls experiences a drop in resolving to one target percentage but still the percentage remains in the 80's even after the drop. This means that libraries are little more polymorphic statically than the benchmark's code.

Name	Benchmark Code			Only Library			Whole Program		
	mono	bi	poly	mono	bi	poly	mono	bi	poly
`JavaBinHex`	118	0	1	10383	615	2130	10501	615	2131
`Compress`	100	0	1	10632	618	2125	10732	618	2126
`JavaCC`	8831	135	1177	10481	617	2147	19312	752	3324
`Pizza`	4790	200	450	14247	691	2674	19037	891	3124
`SableCC`	7761	124	1182	10255	770	2168	18016	894	3350
`Jlex`	855	32	25	10399	615	2131	11254	647	2156

Table 4.1: Static numbers for virtual call analysis

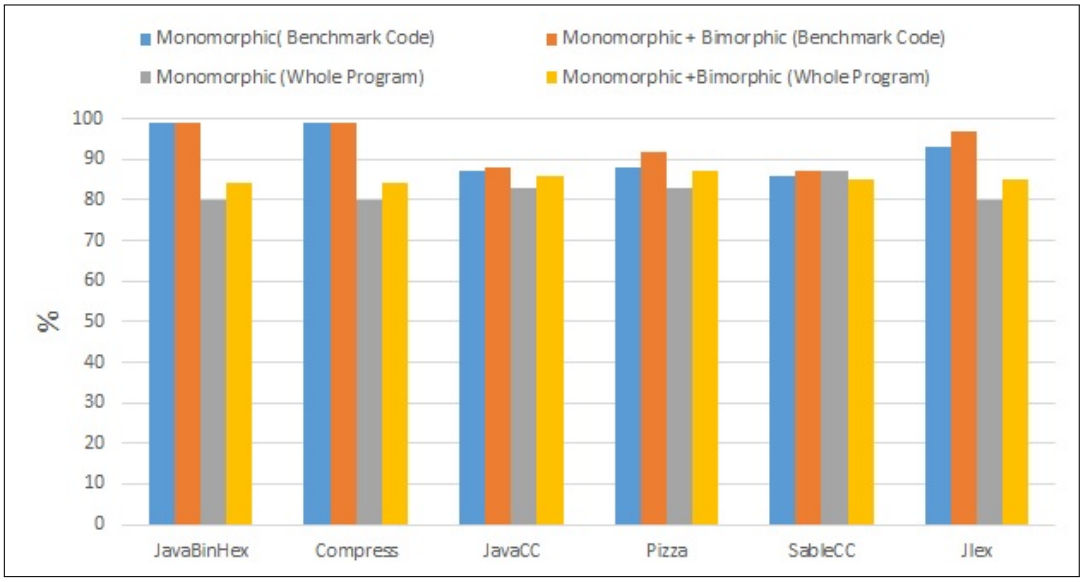

Figure 4.1: % statically resolved call sites

Another interesting statistic that needs to be observed in the static analysis is the number of bi-morphic call sites in the program. The bi-morphic call sites in the benchmark code only ranges from being about half the number of call sites not found to be monomorphic in `Jlex` to being about 10% of the total call sites not found to be monomorphic in `JavaCC` and `SableCC`. When we include the library calls in the analysis, the bi-morphic calls sites are about 20% of the polymorphic calls for all the benchmarks. This means that a considerable number of call sites not found to be monomorphic can be resolved to two targets. Figure 4.1 visualizes the numbers in Table 4.1.

Figure 4.2 shows the number of monomorphic or bi-morphic call sites during a run of the program. This figure shows the percentage of call sites resolved in the program. In the figure 4.2, we see that the analysis resolves all of the call sites to one target in `JavaBinHex` and `Compress` and about 94% of the call sites to one target in `Jlex`, when analyzing only the benchmark code. The percentage of call sites resolved to one target is between low to mid 80's for all the other benchmarks. The figure 4.2 also shows that there are 7% call sites out of the 18% non-monomorphic call sites in the `Pizza` benchmark, for only the benchmark code, that are bi-morphic in nature. For `Jlex`, the percentage of bimorphic call sites is about 4% out of the total 7% non-monomorphic call sites.

Figure 4.2 shows less variance in the percentage of resolved call sites in the benchmarks after including the library calls. All the benchmarks including the library have the percentage of the monomorphic call sites in the 80's. For `JavaCC`, the percentage of resolved call sites is same for both the library included and the library excluded analysis. For `JavaBinHex`, `Compress` and `Jlex`, analyzing the whole application results in a decrease in the percentage of resolved to one or two targets call sites. However, this behavior is expected as they have a very high percentage of resolved call sites when library calls are not included. So, even if some of the call sites in the library are bi-morphic in nature, the percentage will go down. However, even after the inclusion of the library call sites, the percentage still doesn't go down under 80% for any of the benchmark.

Figure 4.2 also shows that for all the benchmarks except `Pizza` and `JavaCC`, there exists more number of bi-morphic call sites when library is included as compared to when library is excluded. Hence, we can deduce from it that there exists more bi-morphic call sites in the library than in the benchmark .

Figure 4.3 shows how many truly monomomorphic and bi-morphic calls exist in the program . The number of calls resolved to one target in the benchmark code only ranges from 100% in `JavaBinHex` and `Compress` to 74% in `Pizza`. The `Jlex` benchmark has the highest number of bi-morphic calls in the benchmark code only among all the analyzed benchmarks. Figure 4.3 shows that the resolution algorithm is doing well in resolving virtual calls when analyzing only the benchmark code.

We have also analyzed the performance of the virtual calls resolution analysis after including the library calls as well. Figure 4.3 also gives the percentage of

Name	Benchmark Code			Only Library			Whole Program		
	mono	bi	poly	mono	bi	poly	mono	bi	poly
`JavaBinHex`	50	0	0	477	19	64	527	19	64
`Compress`	37	0	0	915	31	129	952	31	129
`JavaCC`	1609	38	378	866	35	145	2475	73	523
`Pizza`	438	39	58	487	16	66	925	55	124
`SableCC`	3400	90	712	769	30	120	4169	120	832
`Jlex`	575	26	13	493	19	78	1068	45	91

Table 4.2: Number of resolved run-time call sites

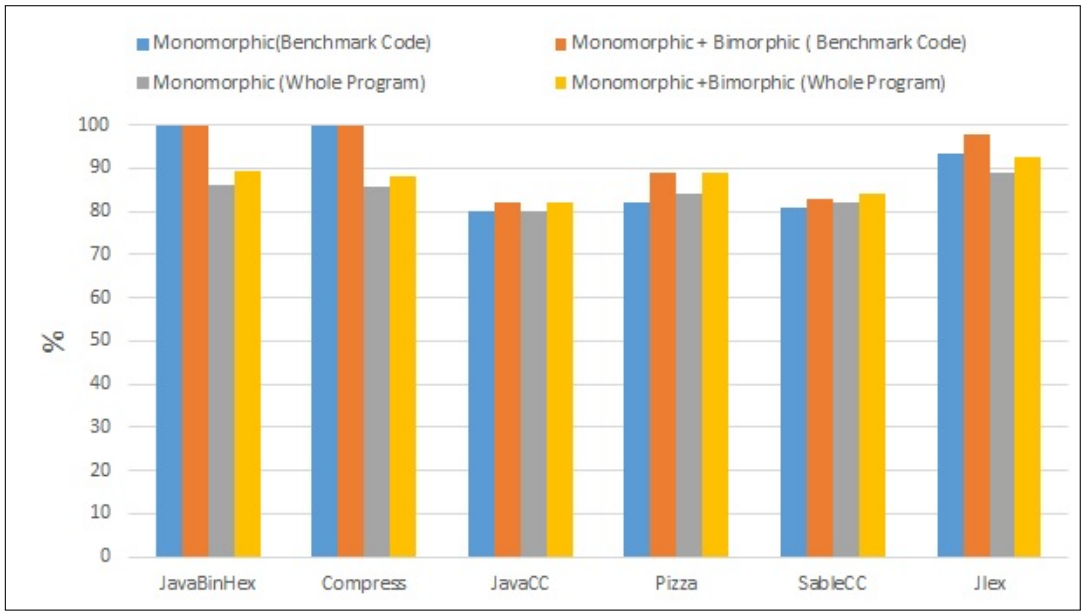

Figure 4.2: % Dynamic Monomorphic and Bimorphic Call Sites

Name	Benchmark Code			Only Library			Whole Program		
	mono	bi	poly	mono	bi	poly	mono	bi	poly
`JavaBinHex`	14414	0	0	6146	317	1691	20560	317	1691
`Compress`	592	0	0	15242	815	7875	15834	815	7875
`JavaCC`	36155	78	3227	76235	2493	24586	112390	2571	27813
`Pizza`	245597	1330	84182	330876	821	55719	576473	2151	139901
`SableCC`	719303	21306	135721	291846	3877	153039	1011149	25183	288760
`Jlex`	10029	373	479	13896	408	4434	23925	781	4913

Table 4.3: Number of resolved run-time call events

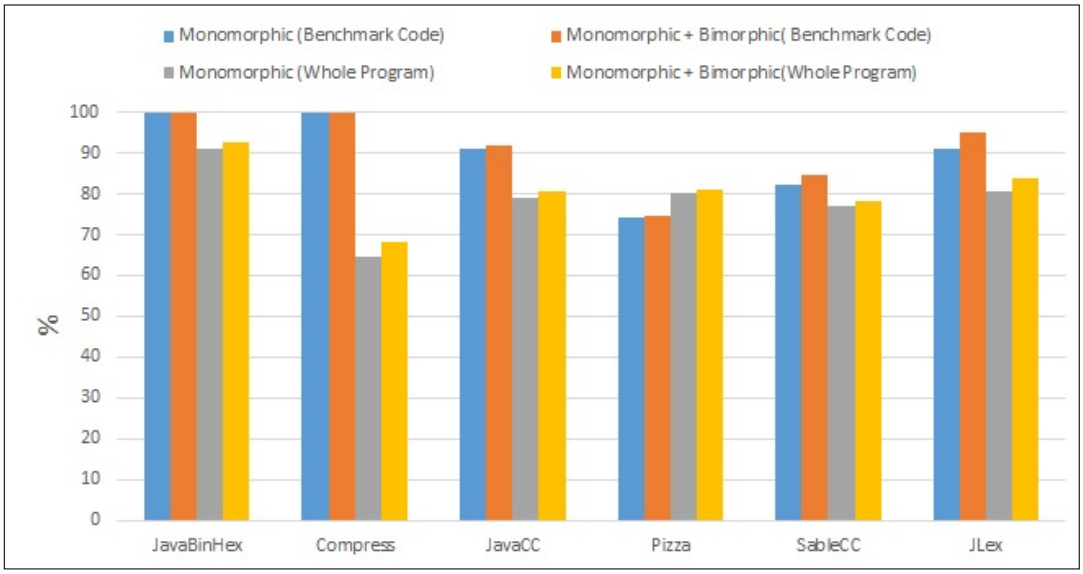

Figure 4.3: % Dynamic Monomorphic and Bimorphic Call Events

monomorphic and bimorphic calls that are made during a run of the program after including the library. The `Compress` benchmark has the highest drop in the percentage of monomorphic calls when compared to the analysis without including the library. The figure shows that the calls in the included library for `Pizza` are more bi-morphic. We notice that the percentage for all the benchmarks except `Compress` is more than 80% even after including the library calls. This proves that virtual call analysis algorithm is effective even after analyzing the whole application .

After comparing the three graphs, we notice that the percentage of bi-morphic calls are highest in the static analysis, to a lesser degree in the dynamic call events and least in the dynamic call sites. This means that the static analysis resolves

more percentage of calls to bi-morphic calls that are never called in a run of the program as compared to resolving calls to monomorphic or polymorphic calls.

Tables 4.1, 4.2 and 4.3 lists the number of call sites or events resolved in the benchmark code and the library code. We observe from the tables that the call sites or events in the libraries are more than the call sites or events in the benchmark code. This leads us to the conclusion that the if we analyze only the benchmark code, more than half of the calls in the whole program are not analyzed. This will result in the dynamic dispatch of the calls that are not analyzed. Therefore, it becomes important to analyze the whole program to improve the performance of the program. The authors of the original Rapid Type Analysis did the analysis only on the benchmark code due to their limitation of analyzing the source code. However ,as we are using bytecode, we do not have that limitation and after analyzing the whole application found that the algorithm is equally effective when analyzing the whole program.

4.1.1 Summary of Virtual Call Analysis Results

The results of analysis on the benchmarks shows that the Rapid Type Analysis Algorithm works effectively to resolve virtual calls in a program as reported in [3].

The results also shows that a considerable number of non-monomorphic calls can be resolved to bi-morphic calls. Thus, implementing them using conditionals can improve the performance of the program. The bi-morphic calls can be implemented using the conditional jump instead of the indirect jump as in the case of polymorhic calls. Therefore, resolving calls to two targets also improves the performance of the program.

The analysis also works well even when the libraries are included in the analysis. The analysis shows that if the libraries are not analyzed, we are considering all the calls in the libraries to be polymorphic which in fact are mostly monomorphic as the whole program analysis shows.

4.2 Exception Handling Analysis Results

This section is divided into two subsections. The first subsection discusses the results of the Direct Path Analysis while the second subsection discusses the results after analyzing catch categories. The benchmarks used for experimenting with the Exception Handling Analysis is discussed in Section 3.6. `JavaCC` and `SableCC` throw a large number of exceptions i.e. more than 400 exceptions; `Pizza` and `Javac` throw exceptions in the range of 30-60 exceptions while `compress` throws less than 10 exceptions. This variable number of exception thrown is used to understand the behavior of analysis for different number of exceptions.

4.2.1 Direct Path Analysis

In this analysis, we set up experiments to count the number of stack frames that needs to be unwound to find a matching handler. As discussed in section in Section 3.3.3.1, we have set up a catch environment around each call site. The catch environment consists a list of exception handlers to which a particular call site has a direct path. We need to unwind a stack frame only when the current stack frame's method is a part of a strongly connected component (SCC) of the call graph or has multiple incoming edges. As methods in a strongly connected component are recursive, we can not be sure about the number of stack frames in the SCC. Therefore, we need to unwind the stack frames one by one until we find a matching handler or we get out of the SCC. Similarly for methods with multiple incoming edges, we need to unwind the stack frame to get the invoking method's stack frame.

Table 4.4 and Table 4.5 shows the number of stack frames that needs to be unwound to find a matching handler for the stack unwinding technique and our analysis. In the case of having more than 5 values for the number of stack frames unwound, the table lists the top five stack frames unwound values.

As evident from tables 4.4 and 4.5, the number of stack frames unwound during exception handling is lower for the Direct Path Analysis. We observed from the tables that the Direct Path Analysis has an advantage in cases where the stack unwinding technique has to unwind more than 3 stack frames as an in the case of

Stack Frames Number	Javacc	SableCC	Pizza	Javac	Compress
0 Stack Frames	52	0	2	0	0
1 Stack Frames	108	4	6	10	4
2 Stack Frames	4	1862	33	0	0
3 Stack Frames	210	0	0	11	0
4 Stack Frames	63	0	0	0	0
5 Stack Frames	6	2	4	0	2
More than 5	23	0	8	0	0
Total	466	1868	53	21	6

Table 4.4: Number of Stack Frames Unwound in Stack Unwinding Technique

Stack Frames Number	Javacc	SableCC	Pizza	Javac	Compress
0 Stack Frames	52	0	2	0	0
1 Stack Frames	110	6	37	10	6
2 Stack Frames	165	1862	2	3	0
3 Stack Frames	112	0	0	8	0
4 Stack Frames	0	0	12	0	0
5 Stack Frames	4	0	0	0	0
More than 5	23	0	0	0	0
Total	466	1868	53	21	6

Table 4.5: Number of Stack Frames Unwound in Direct Path Analysis

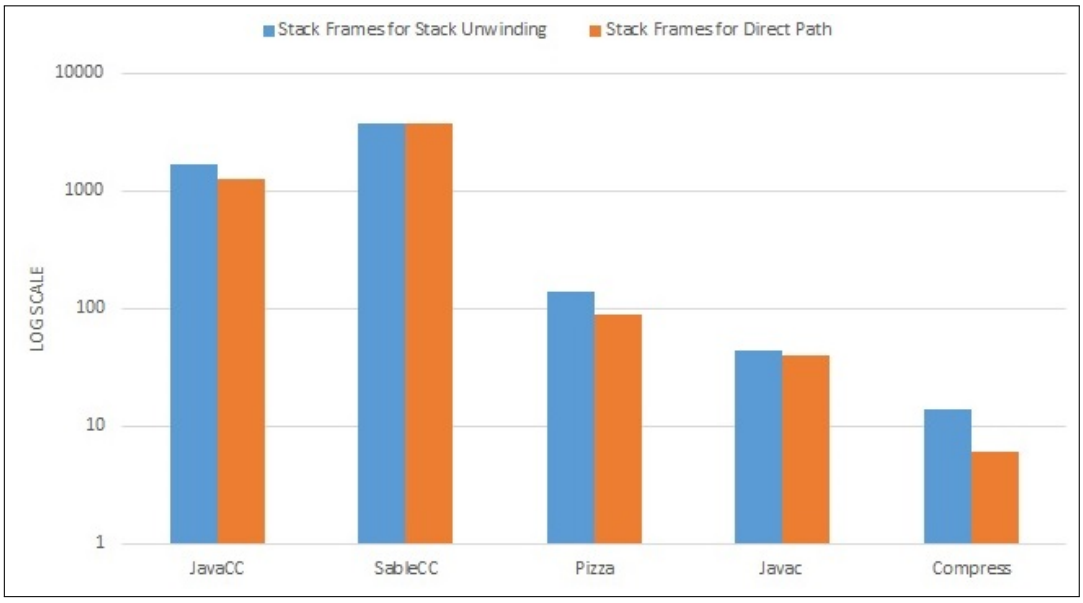

Figure 4.4: Total Stack Frames Unwound in Whole Program

`JavaCC` and `Pizza`. The total number of stack frames unwound for the exceptions in the whole application for stack unwinding technique and direct path analysis is shown in figure 4.4. The figure compares the total number of stack frames unwound in the whole application for the two techniques i.e. if an application needs to unwind 5 stack frames to catch every exception and 3 exceptions are thrown, the total number of stack frames unwound in the whole application is $5 * 3 = 15$.

Figure 4.4 clearly shows that the total number of stack frames that are unwound in a full run of the program is lower for the Direct Path Analysis than the stack unwinding technique in all the benchmarks. The difference between the stack frames unwound is highest for `JavaCC` benchmark while `Javac` has the least difference among all the benchmarks we have experimented upon. The Direct Path Analysis reduces the number of stack frames unwound by more than half in `Compress` while in `Pizza`, the number is reduced by about 40%. Hence, from the evidence from above data, we can say that the Direct Path Analysis performs better than the Stack Unwinding technique in terms of number of stack frames needed to unwind.

Benchmark	Number of Exception Classes
JavaCC	32
SableCC	24
Pizza	22
Javac	30
Compress	28
JLex	15

Table 4.6: Number of Exception Classes in the Program

4.2.2 Analyzing catch categories

We have analyzed catch block characteristics to support the exception handling mechanism proposed in section 3.5. We have set up experiments to get the information regarding the following points:

- Number of the exception classes in the program.
- Number of the catch blocks for a particular exception class in the program.
- Number of the superclasses for a particular exception class in the program.

Table 4.6 shows the number of exception classes in the program while Table 4.7 shows the number of the catch blocks for a particular exception and the superclasses for that class.

Table 4.6 shows that the exception classes in an application are small in number. This means that the memory that needs to be reserved for the Display Catch Table is very small.

The next part of analysis is to find the number of the catch blocks for a particular class in the application. Table 4.6 shows the dynamic number of the catch blocks for a particular exception class for every benchmark. The number of catch blocks is the number written in parenthesis besides every exception class. As evident from the table, the number of the catch blocks is high for about 2 or 3 classes in every benchmark. Apart from those 2 or 3 classes, try-catch blocks for other exception classes are small in number. This number determines the number of times we have to update each entry in the Display Catch Table. So, we need to update the entry for those 2 or 3 classes frequently while we need to update the entry for other classes not so frequently. The number of times we need to update the entry

JavaCC	java.io.Exception (34482) finally (18171) java.io.InterruptedIOException (3213) org.javacc.parser.JavaCCParser$LookaheadSuccess (3092) java.lang.Exception (206) java.nio.BufferUnderflowException(190) All Other Exceptions(896)
SableCC	finally (147620) java.lang.Exception (13819) java.nio.BufferUnderflowException (3905) java.nio.BufferOverflowException (3905) java.lang.IllegalArgumentException (3827) java.io.IOException (1161) All Other Exceptions (1184)
Pizza	finally (79780) java.io.IOException (469) pizza.support.ExceptionWrapper (36) java.lang.RuntimeException (34) java.lang.ClassNotFoundException (11) java.lang.reflect.InvocationTargetException (9) All Other Exceptions (78)
Javac	finally (48743) java.io.IOException (130) java.lang.IllegalArgumentException (67) java.security.PrivilegedActionException (49) java.nio.BufferUnderflowException (38) java.nio.BufferOverflowException (38) All Other Exceptions (168)
Compress	finally (853) java.io.IOException (83) java.nio.BufferUnderflowException (58) java.nio.BufferOverflowException (58) java.io.InterruptedIOException (52) java.lang.IllegalArgumentException (40) All Other Exceptions (179)
JLex	finally (2585) java.io.IOException (818) java.io.InterruptedIOException (799) java.lang.IllegalArgumentException (61) java.nio.BufferUnderflowException (61) java.nio.BufferOverflowException (61) All Other Exceptions (56)

Table 4.7: Number of Catch Blocks in the Program

is equal to the number of times the Stack Cutting technique needs to register a try-catch block. So, we do not have any advantage over the Stack Cutting in this part of analysis.

The last part of the analysis is to find the number of superclasses for a particular class in the program. As discussed in section 3.5, when an exception is thrown, we reference the entry for that exception class in the Display Catch Table. After that we compare the address stored at that entry with the address stored at the superclasses of that entry to find the nearest exception handler. Therefore, this number determines the number of comparisons we have to do to find a matching handler. In the analysis we find that the class hierarchy is very shallow for the exception classes. The number of superclasses for the exception classes in the benchmarks ranges from 0-3 . Therefore, the number of comparisons we have to do find a matching handler is very few for the Display Exception Catch mechanism.

When an exception is thrown, Stack Cutting will go through every try-catch block registered and check whether the exception thrown is an instance of the catch block's handling class. This requires the compiler to invoke an `instanceof` method. In comparison, our mechanism has an advantage during the exception handling process as we only have to compare the address, at the most 3 times, for a particular thrown exception instead of checking through the `instanceof` method.

After we have caught the exception, we need to reconstruct the Display Catch Table. For this purpose, we will check the back link of Catch Save Block (CSB) saved at each entry. We will check if the address pointed by back link is lower than the current address or not. In case of the address being higher than the target address, we will recursively go through CSB's back link to find a CSB lower than the current address. The number of comparisons needed to reconstruct the Display Catch Table is equal to the number of stack frames unwound by Stack Cutting. Hence, our mechanism does not have an advantage in the number of comparison needed to reconstruct the Display Catch Table. However, it has an advantage in the number of instructions required to catch a thrown exception because it uses an address comparison instead of using the `instanceof` method.

4.2.3 Summary of Exception Handling Results

The Direct Path Analysis shows that the number of stack frames needing to be unwound is lower compared to the number of stack frames needing to be unwound in the Stack Unwinding Technique.

After analyzing the catch categories, we have determined that the Direct Catch Exception Handling mechanism is better than Stack Cutting in terms of the number of instructions required when catching an exception thrown.

The Stack Unwinding works better than the Stack Cutting in the programs throwing few exceptions and vice versa as argued in [22]. Therefore, our Direct Path Analysis works better when the exceptions are thrown rarely as it is better than the Stack Unwinding technique. However, in the programs throwing frequent exceptions, the Display Catch Exception handling mechanism will perform better as it is proved to be better than the Stack Cutting technique.

Chapter 5

Conclusions

In this dissertation, our focus has been to analyze Java programs to improve their performance, by examining their virtual calls and exception handling. The virtual calls analysis showed that the number of calls resolved to the monomorphic calls was more than 80% for almost all the benchmarks. The analysis also showed that the bi-morphic calls represents a considerable number of the non-monomorphic calls. The results also showed that the analysis of solely the benchmark code resolves less than half of the calls in the whole program. Therefore, analysis of the whole program is important to improve performance.

The performance of the programs throwing exceptions partially depends upon the exception handling mechanism. Thus, implementing an efficient mechanism improves the performance of these programs. Our work gave two ways to handle the exceptions efficiently. In the programs throwing fewer exceptions, the Direct Path Analysis improves performance by lowering the number of stack frames needed to be unwound to catch an exception. For the programs throwing frequent exceptions, the Display Catch Exception Mechanism works better as we can handle the exceptions by unwinding directly to the specific stack frame. However, as some overhead is associated with the Display Catch Exception Mechanism for creating and reconstructing the Display Catch Table, it may not be better than Direct Path Analysis in programs that rarely throw exceptions.

5.1 Future Work

Although the results of our analysis are really encouraging, we can improve the analysis to get even better results.

For Virtual Call Analysis, we can incorporate full data flow analysis in the call graph. Incorporating data flow analysis will help in understanding the type propagation through the call graph. This will make the virtual call analysis more effective. To simplify the analysis, we have not analyzed calls where the declared type of an object is an interface. Therefore, in future, we need to include these calls as well to make our virtual call analysis complete.

We have gathered statistics about exception handling mechanisms by instrumenting the code. In future, we need to implement these mechanisms in our compiler, which is currently not generating code, and measure the performance improvement these mechanisms give.

We have ignored `synchronized` methods while analyzing the programs for the exception handling. In future, we need to include these methods as well in our analysis to make the analysis complete.

We also need to gather statistics after comparing the Direct Path Analysis with the Display Catch Exception Handling to determine the conditions in which either of them is better than the other. In future, we will also measure tradeoffs between the cost of analysis and the actual performance improvement.

Bibliography

[1] Alfred V Aho and Jeffrey D Ullman. *Principles of compiler design*. Addision-Wesley Pub. Co., 1977.

[2] R Allen et al. "Advanced Compiler Design and Implementation". In: ().

[3] David F Bacon and Peter F Sweeney. "Fast static analysis of C++ virtual function calls". In: *ACM Sigplan Notices* 31.10 (1996), pp. 324–341.

[4] *Byte Code Engineering Library*. `http://commons.apache.org/proper/commons-bcel/manual.html`.

[5] Brad Calder and Dirk Grunwald. "Reducing Indirect Function Call Overhead in C++ Programs". In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '94. Portland, Oregon, USA: ACM, 1994, pp. 397–408. ISBN: 0-89791-636-0. DOI: `10.1145/174675.177973`. URL: `http://doi.acm.org/10.1145/174675.177973`.

[6] C. Chambers and D. Ungar. "Customization: Optimizing Compiler Technology for SELF, a Dynamically-typed Object-oriented Programming Language". In: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*. PLDI '89. Portland, Oregon, USA: ACM, 1989, pp. 146–160. ISBN: 0-89791-306-X. DOI: `10.1145/73141.74831`. URL: `http://doi.acm.org/10.1145/73141.74831`.

[7] Craig Chambers and David Ungar. "Interactive Type Analysis and Extended Message Splitting; Optimizing Dynamically-typed Object-oriented Programs". In: *SIGPLAN Not.* 25.6 (June 1990), pp. 150–164. ISSN: 0362-1340. DOI: `10.1145/93548.93562`. URL: `http://doi.acm.org/10.1145/93548.93562`.

[8] Ramkrishna Chatterjee, Barbara G. Ryder, and William Landi. "Complexity of Concrete Type-Inference in the Presence of Exceptions". In: *ESOP*. Ed. by Chris Hankin. Vol. 1381. Lecture Notes in Computer Science. Springer, 1998, pp. 57–74. ISBN: 3-540-64302-8.

[9] Jong-Deok Choi et al. "Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs". In: *PASTE*. Ed. by William G. Griswold and Susan Horwitz. ACM, 1999, pp. 21–31. ISBN: 1-58113-137-2.

[10] Jeffrey Dean, David Grove, and Craig Chambers. "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis". In: *Proceedings of the 9th European Conference on Object-Oriented Programming*. ECOOP '95. London, UK, UK: Springer-Verlag, 1995, pp. 77–101. ISBN: 3-540-60160-0. URL: http://dl.acm.org/citation.cfm?id=646153.679523.

[11] S Drew, K John Gough, and J Ledermann. "Implementing zero overhead exception handling". In: *Faculty of Information Technology, Queensland University of Technology, Australia, Tech. Rep* (1995), pp. 95–12.

[12] Urs Hölzle and David Ungar. "Optimizing Dynamically-dispatched Calls with Run-time Type Feedback". In: *SIGPLAN Not.* 29.6 (June 1994), pp. 326–336. ISSN: 0362-1340. DOI: 10.1145/773473.178478. URL: http://doi.acm.org/10.1145/773473.178478.

[13] *JavaBinHex*. http://www.zrenard.com/javabinhex/.

[14] *JavaCC Version 5.0*. https://java.net/projects/javacc/downloads.

[15] *Jlex*. https://www.cs.princeton.edu/~appel/modern/java/JLex/.

[16] Jang-Wu Jo and Byeong-Mo Chang. "Constructing control flow graph that accounts for exception induced control flows for Java". In: *The 7th Korea-Russia International Symposium on Science and Technology, Volume 2*. Vol. 2. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1222596. July 2003, pp. 160–165. ISBN: 89-7868-617-6.

[17] Michael Karr. "Code Generation by Coagulation". In: *SIGPLAN Not.* 19.6 (June 1984), pp. 1–12. ISSN: 0362-1340. DOI: 10.1145/502949.502875. URL: http://doi.acm.org/10.1145/502949.502875.

[18] R. Leupers and P. Marwedel. "Function inlining under code size constraints for embedded processors". In: *Computer-Aided Design, 1999. Digest of Technical Papers. 1999 IEEE/ACM International Conference on*. 1999, pp. 253–256. DOI: 10.1109/ICCAD.1999.810657.

[19] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201432943.

[20] W. G. Morris. "CCG: A Prototype Coagulating Code Generator". In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. Toronto, Ontario, Canada: ACM, 1991, pp. 45–58. ISBN: 0-89791-428-7. DOI: 10.1145/113445.113450. URL: http://doi.acm.org/10.1145/113445.113450.

[21] Esko Nuutila and Eljas Soisalon-Soininen. "On finding the strongly connected components in a directed graph". In: *Information Processing Letters* 49.1 (1994), pp. 9–14.

[22] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. "EDO: Exception-directed optimization in java". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28.1 (2006), pp. 70–105.

[23] *Pizza Compiler*. http://wwwipd.ira.uka.de/~pizza/.

[24] Norman Ramsey and Simon Peyton Jones. "A single intermediate language that supports multiple implementations of exceptions". In: *ACM SIGPLAN Notices*. Vol. 35. 5. ACM. 2000, pp. 285–298.

[25] Martin P. Robillard and Gail C. Murphy. "Static analysis to support the evolution of exception structure in object-oriented systems". In: *ACM Trans. Softw. Eng. Methodol.* 12.2 (Apr. 2003), pp. 191–221. ISSN: 1049-331X. DOI: 10.1145/941566.941569. URL: http://doi.acm.org/10.1145/941566.941569.

[26] *SableCC*. http://www.sable.mcgill.ca/sablecc/.

[27] Vladimir I. Shelekhov and Sergey V. Kuksenko. "Data flow analysis of Java programs in the presence of exceptions". In: *In Proceedings of the third International Andrei Ershov Memorial Conference on Perspectives of System Informatics. Lecture Notes in Computer Science*. SpringerVerlag, 1999.

[28] *Short-Circuit Evaluation*. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`.

[29] Saurabh Sinha and Mary Jean Harrold. "Analysis and Testing of Programs with Exception Handling Constructs". In: *IEEE Trans. Softw. Eng.* 26.9 (Sept. 2000), pp. 849–871. ISSN: 0098-5589. DOI: `10.1109/32.877846`. URL: `http://dx.doi.org/10.1109/32.877846`.

[30] *SpecJVM Benchmarks*. `https://www.spec.org/benchmarks.html`.

[31] Vijay Sundaresan et al. "Practical virtual method call resolution for Java". In: *ACM SIGPLAN Notices* 35.10 (2000), pp. 264–280.

[32] Raja Vallée-Rai et al. "Soot - a Java Bytecode Optimization Framework". In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, pp. 13–. URL: `http://dl.acm.org/citation.cfm?id=781995.782008`.