IOSR Journal of Engineering
Apr. 2012, Vol. 2(4) pp: 544-554

Minimization of Time & Cost Factors with Optimized Program-States Using Exception-Handling Constructs in Java (During Analysis and Testing of Programs)

Pawan Nagar
Department of Computer Science
Lingaya's University
Faridabad, India

Nitasha Soni
Department of Computer Science
Lingaya's University,
Faridabad, India

***Abstract*:** It has been observed that 80% of development time and a lots of memory space is spend over handling the abnormal conditions and exceptions in any programming Environment. It also affects the cost factor of program development also. The support for precise exceptions in optimizing Java program states, combined with frequent checks for runtime exceptions, leads to severe limitations on the compiler's ability to perform program optimizations that involve reordering of instructions. This paper presents a basic framework that allows the application programmers to recognise the constraints of application programs in instruction scheduling. We first present a formulated way for analysing the problem caused in coded program and its related Exception-handling constructs, and a methodology, to identify the subset of program state that needs to be preserved if an exception is thrown. This allows many spurious dependence constraints between potentially excepting instructions (PEIs) and writes into variables to be eliminated. Our planned methodology is particularly suitable for dynamically dispatched methods in object-oriented languages, where static analysis may be quite conservative. We then present the first software-only solution that allows dependence constraints among PEIs to be completely ignored while applying program optimizations, with no need to execute any additional instructions if an exception is not thrown. With a preliminary implementation, by this we observe that for many benchmark programs, a large percentage of methods can be optimized (while honouring the precise exception requirement) without any constraints imposed by frequent runtime exceptions. Finally, we show that relaxing these reordering constraints can lead to substantial improvements (up to a factor of 7 out of 10 for small codes) in the performance of programs

Keywords: *Optimization, Compiler-ability, instruction –scheduling, Exception-handling constructs, potentially excepting instructions (PEIs), spurious dependence, dispatched methods, dependence constraints, abnormal situations ,protected code, Handler-association, instruction-boosting ,potentially implicit exceptions.*

I. INTRODUCTION

Java already has been embraced as a Web programming language and is beginning to gain acceptance as a language for general applications. In response to the acceptance and use of this and other OO languages, compile-time analysis and code transformations that produce optimized code for features found in such languages (e.g., polymorphism, exceptions, etc.) are being studied. This paper summarizes how exceptions are used in current Java codes. The information gathered shows that exceptions are ubiquitous and that their use often falls into specific patterns. Among different program statements, more of software-engineering tasks, such as test-coverage analysis, test-case generation, impact analysis, regression testing, static and dynamic slicing and dynamic execution profiling (e.g., [1]) require information about the control flow, data dependence, and the control dependence also. Previous researches have addressed the problems of computing such analysis information at intra procedural level (for individual procedures) and at inter procedural level (for interacting procedures). Some of this research has addressed the problems of performing analyses for programs with transfers of control, such as **continue** and **goto** statements, that can affect the analyses at the intra procedural level[2]. Some other research has addressed the problems of performing analyses for programs with transfers of control, such as **exit ()** statements, that can affect the analyses at the inter procedural level [3]. To be applicable to programs written in programming languages, such as Java and C++, however, these analysis techniques should, to the extent possible, account for the effects of exception-handling constructs.

A mechanism for raising exceptions and a facility for designating protected code is provided by Exception-handling constructs by attaching exception handlers to blocks of code. Any type of failure to account for the effects of exception-handling constructs in performing analyses can result in incorrect analysis information, which in turn can result in unreliable or inefficient software tools. For example, a branch-coverage testing tool for C++ that fails to recognize the flow of control (for data or execution) among exception-handling constructs cannot adequately measure the branch coverage of a test suite [1]. As a further example, a slicing tool for Java that fails to recognize the flow of control among exception-handling constructs cannot accurately compute control and data dependence, which may result in incorrect slices. The additional expense that is required to perform analyses that account for the effects of exception-handling constructs may not be justified unless these constructs occur frequently in practice [1].

In a number of conducted studies [4] in previous research the occurring result shows that the method or classes used in any programming language uses some form of Exceptional-Handling Constructs most of the times. These results provide support for our belief that, in practice, the use of exception-handling constructs (mostly in Java programs) is significant enough that it should be considered during various analyses. Recently, several researchers have considered and observed the effects of exception-handling constructs on various types of analyses. One approach constructs control-flow representation for exception-handling constructs, and uses the representation to perform data-flow analysis [5]. Another approach considers the control flow caused by exceptions while performing points-to and data-flow analysis [6]. Some other research has analyzed the flow of exceptions, and built tools to facilitate understanding of the exceptional behaviour of programs [7]. Due to this reason the Exception-Handling mechanism has become a necessary way for handling abnormal situations in software development, especially since the Java language came in the existence, where exceptions are part of the language, libraries and frameworks. In the Java exception-handling paradigm, an exception can be raised explicitly through a throw statement, or implicitly, through a call to a library routine or by the runtime environment. The main concern behind the popularity of this mechanism the developers need not to be aware of exceptional situations throughout the entire development process (starting from modelling end of development of a software project to till the implementation end).

An Overview of effect of Exception-Handling Constructs on various analysis and testing techniques

This section posse an overview of exception-handling constructs in Java. For this the language model; details of the Java language can be found in Reference [8].

In Java, an exception is an object: each exception is an instance of a class that is derived from the class java.lang.Throwable. In a program an exception can be raised at any point through a throw statement. The expression associated with the throw statement denotes the exception object. A throw statement can appear anywhere in the program—it may or may not be enclosed in a try statement and the expression can be a variable (e.g., throw e), a method call (e.g., throw m()), or a new instance expression (e.g., throw new E()).

A try block contains statements whose execution is monitored for exception occurrences. A catch block, which may be associated with each try block, is a sequence of catch clauses that specify exception handlers. Each catch clause specifies the type of exception it handles, and contains a block of code that is executed when an exception of that type is raised in the associated try block. A catch clause also specifies a variable that is initialized with the handled exception, and whose scope is limited to the block of code for that catch clause. A try statement can have a finally block. The code in a finally block is always executed, regardless of how control transfers out of the try block. Control may exit a try block by reaching the last statement in the try block, through an exception that may or may not be handled in the associated catch block, or because of break, continue, or return statements.

Java follows the non-resemble model of exception handling: after an exception is handled, control does not return to the point at which the exception was raised, but continues at the first statement following the try statement where the exception was handled. A Java exception can be propagated up on the call stack: if a method raises but does not handle an exception, the exception is re-raised in the context of the caller of that method [1] [2].

Due to an efficient and reliable Exception-Handling mechanism Java continues to gain importance as a popular object-oriented programming language for general-purpose programming [8]. Although some aspects of Java, such as strong typing, simplify the task of program analysis and optimization, other aspects, such as support for precise exceptions, can hamper program analysis and optimizations. The Java language specification requires that exceptions be precise, which implies that;

1. Exception(s) must be thrown in the same order as specified by the original (un-optimized) program; and
2. When an exception is thrown, the program state observable at the entry of the corresponding exception handler must be the same as in the original program [9].

To satisfy the precise exception requirement, Java compilers disable many important optimizations across instructions that may throw an exception (we refer to these potentially excepting instructions as PEIs [10]). This hampers a wide range of program optimizations such as instruction scheduling, instruction selection (across a PEI), loop transformations, and parallelization. Furthermore, PEIs are quite common in Java programs – frequently occurring operations such as reads and writes of instance variables, array loads and stores, method calls, and object allocations may all throw an exception. Hence, the ability of the compiler to perform any program transformation that requires instruction reordering is severely limited, which impedes the performance of Java programs. This paper presents a basic methodology to optimize program transformations in the presence of precise exception. Exception handling is common in Java programs for the Web, but it is also true that exceptions will play a significant role in general purpose applications. The use of exceptions in general applications is due to several emerging trends. Key among these are: the development of automated systems with complex control paths, and the shift toward exception-based programming paradigms seen in most introductory language and data-structures texts.

II. RELATED WORK

There is much previous research relevant to this work in: fault-injection testing, dataflow testing coverage metrics, exception-handler analysis and compilation, points-to analysis (for reference variables) and infeasible path analysis. Choi and colleagues [5] describe an intra procedural control-flow representation called the factored control-flow graph (FCFG) to analyse efficiently programs written in languages, such as Java, that may have frequently occurring exceptional control flow. The FCFG represents exceptional control flow caused by both explicit and implicit exceptions. For explicit exceptions, the approach creates edges that are similar to the edges created in our approach. For implicit exceptions, however, the approach does not create edges from each potentially exception-throwing instruction (PEI) because such instructions occur very frequently. Instead, the approach merges several such instructions in the same basic block, and creates factored control-flow edges from the basic block to catch handlers to summarize the exceptional control flow for that basic block. The approach creates one factored edge for each type of implicit exception that can be raised by the statements in a basic block. The approach derives the target of the implicit exceptional exits from each PEI in a basic block on demand. Choi and colleagues also describe modifications to data-flow analysis techniques, such as reaching-definition and live-variable analysis, that allow the techniques to work correctly on the FCFG. That work differs from ours in several ways. First, the work does not model the propagation of exceptions across methods. Although Choi and colleagues discuss alternative representations for inter procedural control flow, their current tool does not construct inter procedural representations. Second, the work does not describe the behaviour of, and representations for, finally blocks. Third, the work does not discuss issues relating to inferring exception types, and how they affect precision of the FCFG and the analyses performed on the FCFG. Finally, the scope of the work is limited to data-flow analysis, and it does not consider the effects of exceptions on control dependence, slicing, and structural testing. Chatterjee and Ryder [6] describe an approach to performing points-to analysis that incorporates exceptional control flow in languages such as Java. Their approach derives the exceptional control flow during the points-to analysis, and does not represent it explicitly in an inter procedural control-flow graph. Their approach does not consider implicit exceptions. In subsequent work [11], Chatterjee and Ryder provide an algorithm for computing definition- Use pairs that arise because of exception variables, and along exceptional control-flow paths. In this work, however, they ignore the control flow within finally blocks.

Chatterjee and Ryder do not describe representations for exceptional control flow, and the scope of their work is limited to points-to and data-flow analysis. Schaefer and Bundy [12] analyze the flow of exceptions in Ada programs, and extract information that describes how exceptions are propagated across modules. They define several relations that let them specify formally the set of exceptions propagated by different blocks of code. The goal of their analysis is to identify potential violations in

the code of application-specific guidelines that govern the usage of exception handling. Robillard and Murphy [13] have similar goals for Java programs. They describe a tool that extracts the flow of exceptions in a Java program, and generates views of the exception structure. These views enable a developer to reason about the flow of exceptions across modules, and identify program points where exceptions are caught unintentionally, or where finer-grained exception handling may be possible. The tool extracts potential implicit exceptions by examining module interface and documentation. The techniques described by both Schafer and Bundy [12] and Robillard and Murphy [13] omit reporting several common implicit exceptions because including them can generate too much information, which adversely affects the usability of their tools. Their techniques are primarily intended for program understanding and detection of inconsistencies in coding. Therefore, they do not consider the effects of exceptions on various program-analysis techniques and testing. Using our control-flow representations, we can generate information that is similar to the information generated by their techniques. Melski and Reps [14] present techniques for inter procedural path profiling, and briefly discuss how path profiles for inter procedural exceptional control flow may be generated. Their work neither describes representations for exceptional control flow, nor analyses the effects of exceptional control flow on program-analysis techniques. Other researchers have addressed the problem of computing accurate slices for programs that contain arbitrary intra procedural control flow [15], [16], [17]. Such control flow is caused by intra procedural goto statements and statements such as break and continue. Because statements, such as break and continue, neither control other statements nor use data values, they are never included in a slice. References [15], [16], [17] present solutions in which the statements are included in the slices, when necessary. The same problem can occur in the presence of exception-handling constructs: statements, such as throw and catch can be excluded from slices. Our slicing technique for handling constructs [18] ensures that throw and exception catch statements are included in the slices, when necessary. Ryder and colleagues [4] conducted a study of the usage patterns of exception-handling constructs in Java programs. They studied a suite of thirty-one Java programs, which contained from two to 2,096 methods. They examined 10,161 methods, and found that, on average, 16% of the methods contained either a throw statement or a try statement. Our subjects contain four of the subjects that were included in their study. For those four subjects, our results are consistent with theirs. Their study thus offers further evidence to support our belief that exception-handling constructs are used frequently in Java programs.

III. PROBLEM FORMULATION

The problem formulation of the proposed work mainly depends on the different exceptional-handling constructs. A particular exceptional-handling construct is a formulation of following blocks in the respect of completeness of its functionality (Figure 1):-

A. *Problem*

This block specifies that the particular exceptional handling construct defined or constructed for which problem (i.e. exception or error).

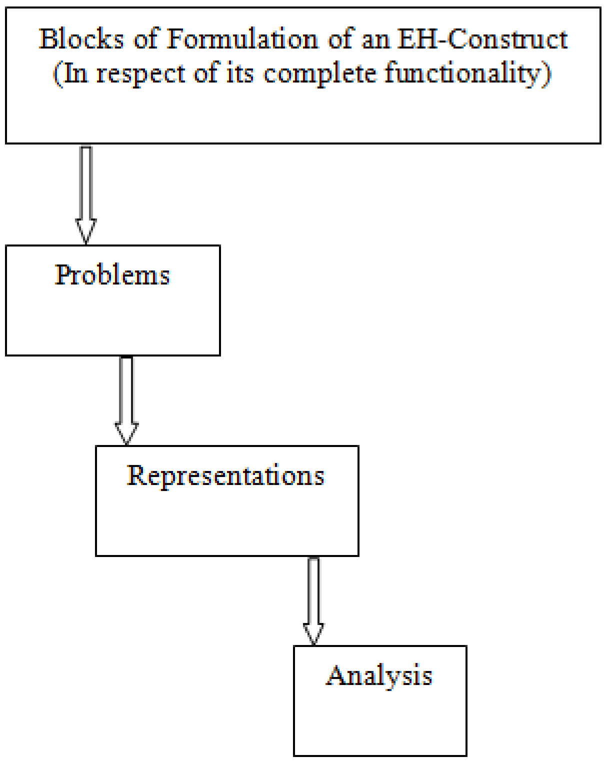
Blocks of Formulation of an EH-Construct
(In respect of its complete functionality)
Problems
Representations
Analysis

Figure 1: Formulation blocks of EH-Construct

The problem of debugging optimized programs also has some similarities with optimizing programs in the presence of exceptions, in that the notion of preserving the currency of variables at a breakpoint [23] is similar to that of preserving program state at exception points. Holzle et al. [24] restrict optimizations across interrupt points and use dynamic de-optimization of code to support debugging. Other researchers have proposed various approaches to detect variables whose value may not be current at a breakpoint due to program optimizations [25,26] and possibly recovering the original value of variables that are not current [26]. In contrast, our work must ensure that the value of each relevant variable is "current" at an exception point, and focus on determining the minimal set of such relevant variables. Furthermore, we deal with the problem of ensuring the correct ordering of exceptions (without adversely affecting program optimizations), which does not occur in the context of debugging optimized programs.

B. *Representation*

This block specifies how a particular EH-Construct represent semantic (including throw statement and try, catch or finally blocks).

We can use a form of program dependence graph [20] to model control and data dependences in the program, with a special representation for exception-related dependences. This representation exploits ideas from the factored control-flow graph (FCFG) [22] to model control flow due to exceptions. The FCFG representation does not terminate basic blocks at PEIs, which results in larger basic blocks and fewer edges than a regular control flow graph. The low level intermediate representation (LIR) of the Jalapeno optimizing compiler [19] we employ for our optimization uses condition registers to represent the exception-conditional dependences. An instruction that is exception-conditionally dependent on exception-check instructions should execute only if none of these exception-check instructions indicates that an exception should be thrown.

C. *Analysis*

This block specifies weather a particular newly built EH-Construct is more efficient or not in handling a no. of exceptions as compare to previously use those can handle less no. of exceptions.

Exception-handling constructs belong to a class of control structures that cause arbitrary inter procedural control flow, and affect program-analysis techniques in similar ways. Other examples of such control structures include inter procedural jump statements, such as the setjmp()–longjmp() calls in C, and halt statements, such as the exit() call in C. Such constructs affect the flow of control across procedures, and in doing so, affect all analyses that are derived from control-flow analysis. The common effect of such control structures is that, at a call site, control may not return from the called procedure back to the call site. Instead, control may return to a different point in the calling procedure, or control may not return to the calling procedure at all. Through such an effect, the control structures influence program-analysis techniques, such as control-flow analysis, data-flow analysis, and control-dependence analysis [1].

An EH-Construct is the basic building aspect(or reason) to define an exception/exception-type. For e.g. The EH-Construct behind the Null Pointer Exception is that we cannot assign an object to a variable which is declared as null (i.e. x= 0 ;). An EH-Construct includes the overall semantic representation of throw statement and try, catch and finally blocks.

IV. METHODOLOGY

Our approach relies on the following facts:-
First, the program state that needs to be preserved when an exception is thrown is often a very small subset of the program state that is conservatively preserved by compilers to support precise exceptions. By identifying this subset, many spurious constraints on instruction reordering can be removed (Figure-2).

Second, exceptions are rarely thrown by correctly executing Java programs, and so it is desirable to optimize the program for this expected case even at the expense of some inefficiency when an exception is thrown.

So for achieving our goal of optimization of program states in analysis and testing phase of program development with the help of Exceptional-Handling we are going to implement following step by step evaluation:-

1. Recognise the most time consuming block of program states during the occurrence of an error exception.
2. Recognise how these program states affect the flow of Program execution and how we can overcome this situation with a suitable exception-handler. The flow of program execution decides with the help of different Path-Finder Algorithms(For example;

Factored Control Flow Graph (FCFG) Algorithm decides the actual path of flow of program Execution

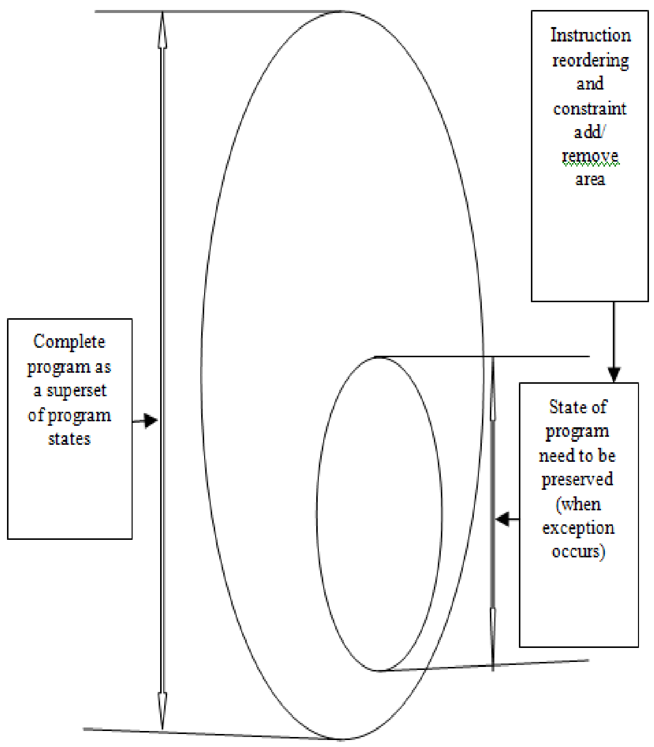
Instruction reordering and constraint add/ remove area
Complete program as a superset of program states
State of program need to be preserved (when exception occurs)

Figure 2: States of program in execution area (when any exception occurs).

3. Then we have to recognise that the occurring error/exception is of which type and this belong to which class.
4. If no exception type or class is defined for the occurring error/exception a new exception type or class have to be defined for it and then decide which exception-Handler is applicable for it. This process is totally based on the EH-constructs behind the occurrence of the exception.
5. In the next step if the there is a requirement to build new construct for handling this new kind of Exception it should be built (or New Exception-Handler can also be define).
6. Now we have to develop new subsets of program states those are require to improve time factors in the implementation phases (i.e. analysis and testing) These can be related to the Exception-Handling Mechanism also. These subsets can be developed by the reordering of instructions and by removing and adding some constraints in the subset of program states.
7. In the next step we have to put these subsets of the program states in the complete superset of program execution state.
8. The efficiency of program states can be checked with the implementation of different Path-Finding Algorithms.

Previous work on speculative code motion for superscalar and VLIW processors [28, 31, 30, 29] has some similarities with our work, in that it involves aggressive code motion and recovery from exceptions thrown by speculative instructions. The general percolation scheduling model [28] uses hardware support for *silent* exceptions, but possibly fails to detect an exception that should be thrown. The instruction-boosting scheduling model [31] avoids this drawback, but requires greater hardware support in the form of shadow register files and shadow store buffers, which hold the results of speculative instructions. Sentinel scheduling [30] and its variants use less expensive hardware support. Broadly, our work differs in at least two ways. First, it does not require any hardware support, while these approaches rely on special hardware to support silent exceptions or to store the results of speculative instructions. Second, (since we are looking at a different problem, that of handling precise exceptions in Java) our work is unique in addressing the problem of reducing the program state that must be preserved at a possible exception point.

Automated systems are being built around legacy codes that were designed to be controlled by humans and are now being controlled by programs. Not surprisingly, human-friendly codes are proving to be program-unfriendly, necessitating the adaptation of these codes for use under program, as opposed to human, control. These legacy codes have been built over several years by several people, have been validated against extensive test suites, and are now trusted tools. Revalidation of such codes is very expensive in time and money thus constraining adaptations of these codes to minimize the need for revalidation. One of the most promising strategies to facilitate the adaptation of legacy codes under the constraint of limited revalidation is the introduction of wrappers [32,31,33] that handle unexpected situations. Wrappers provide a mechanism that detects when a code has failed and passes control to a module designed to manage the failure. Java exceptions, in conjunction with their catch and throw operators, provide an ideal mechanism for implementing wrappers.

The advantages of Possible Optimizations

Knowing statically the binding between a thrown exception and the catch clause that will process the exception has two advantages that can be exploited by an optimizing compiler:

- No runtime check is necessary to determine if the next current method will handle the exception.

- The code contained in the corresponding catch clause can be moved to the site where the exception is thrown, allowing better code locality. In addition, the runtime stack can be updated with a single multi-frame pop operation. This optimization assumes that no statement such as a finally occurs in any intermediate method invocation between the throw and catch methods. This safety property can be easily verified by a compile-time analysis.

V. CONCLUSION

Precise exception support imposes constraints on optimizations such as instruction reordering. The novel methodology presented in this paper enables the relaxation of these constraints. By identifying the subset of program state that needs to be preserved if an exception is thrown, the framework enables the removal of many spurious dependences between writes and potentially excepting instructions. The static and dynamic analysis algorithms we presented can be used separately or together to improve the effectiveness of the analysis. Our framework further allows aggressive optimization of the program, ignoring all dependences between potentially excepting instructions, by generating compensation code that is executed only when an exception is thrown. This code ensures that the same exception is raised as in the original un-optimized code. The solution presented requires no hardware support and can be implemented in both static and dynamic compilers. The various algorithms are applicable to any language that constrains program transformations due to dependences involving exceptions. The preliminary implementation using a conservative version of various algorithms(aseptically path-Finding and control-flow algorithms) shows promising results: using the static analysis, in 11 out of 13 benchmarks, over 65% of methods are recognized as targets of our aggressive optimization techniques for ignoring program state dependences at potentially excepting instructions; using the dynamic analysis, over 96% of the method invocations can be aggressively optimized in 9 of those benchmarks. We have also demonstrated that significant speedups, up to a factor of 7 on small programs, can be obtained using our techniques and well-known transformations. We expect the importance of technique presented in this paper to grow further, as Java is used more heavily in application areas that require high performance, and also as Java compilers become more mature, and run into the limitations imposed by precise exception semantics while applying aggressive optimizations that involve code reordering. This will also posses the cost effectiveness and minimal time requirement of a java programming environment.

VI. ACKNOWLEDGEMENT

We thank Sandhya Nagar and Gajender Nagar for useful technical discussions. We thank Kajal Sharma, Anamika Sharma and Sonu Kr. Singh for their help in Java Run Time Environment and in Programming related Issues. We also thank Saurabh Kackar, E.H.Ansari for their useful feedback on earlier drafts of this work and Nakul Nagar for his support of this work.

REFERENCES

[1] Saurabh Sinha and Mary Jean Harrold" Analysis and Testing of Programs With Exception-Handling Constructs" IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 26, NO. 9, SEPTEMBER 2000.

[2] H. Agrawal, "On slicing programs with jump statements," in Proc. of the ACM SIGPLAN '94 Conf. on Prog. Lang. Design and Impl., June 1994, pp. 302–12.

[3] M. J. Harrold, G. Rothermel, and S. Sinha, "Computation of interprocedural control dependence," in Proc. of the ACM Int'l. Symp. on Softw. Testing and Analysis, Mar. 1998, pp. 11–20.

[4] B. G. Ryder, D. Smith, U. Kremer, M. Gordon, and N. Shah, "A static study of Java exceptions using JSEP," Tech. Rep. DCS-TR-403, Rutgers University, Nov. 1999.

[5] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, "Efficient and precise modeling of exceptions for analysis of Java programs," in Proceedings of PASTE '99 ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, September 1999, pp. 21–31.

[6] R. K. Chatterjee, B. G. Ryder, and W. A. Landi, "Complexity of concrete type-inference in the presence of exceptions," Lecture Notes in Computer Science, vol. 1381, pp. 57–74, Apr. 1998.

[7] M. P. Robillard and G. C. Murphy, "Analyzing exception flow in Java programs," in Proc. of ESEC/FSE '99 Seventh European Softw. Eng. Conf. and Seventh ACM SIGSOFT Symp. On the Found. of Softw. Eng. September 1999, vol. 1687 of Lecture Notes in Computer Science, pp. 322–337, Springer-Verlag.

[8] J. Gosling, B. Joy, and G. Steele, The Java Language Specification, Addison-Wesley, Reading, MA, 1996.

[9] "Optimizing Java Programs in the Presence of Exceptions"Manish Gupta, Jong-Deok Choi, Michael Hind IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598.

[10] S. Mahlke, W. Chen, R. Bringmann, R. Hank, W.-M. Hwu, B. Rau, and M. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, 11(4):376–408,November 1993.

[11] R Chatterjee and B. G. Ryder, "Data-flow-based testing of object-oriented libraries," Tech. Rep. DCS-TR-382, Rutgers University, Mar. 1999.

[12] C. F. Schaefer and G. N. Bundy, "Static analysis of exception handling in Ada," Software—Practice and Experience, vol. 23, no. 10, pp. 1157–1174, Oct. 1993.

[13] M. P. Robillard and G. C. Murphy, "Analyzing exception flow in Java programs," in Proc. of ESEC/FSE '99 Seventh European Softw. Eng. Conf. and Seventh ACM SIGSOFT Symp. On the Found. of Softw. Eng. September 1999, vol. 1687 of Lecture Notes in Computer Science, pp. 322–337, Springer-Verlag.

[14] D. Melski and T. Reps, "Interprocedural path profiling," in Proceedings of the 8th International Conference on Compiler Construction. March 1999, vol. 1575 of Lecture Notes in Computer Science, pp. 47–62, Springer-Verlag.

[15] T. Ball and S. Horwitz, "Slicing programs with arbitrary control flow," in Proc. of 1st Int'l Workshop on Automated and Algorithmic Debugging. Nov. 1993, vol. 749 of Lec. Notes in Computer Science, pp. 206–222, Springer-Verlag.

[16] H. Agrawal, "On slicing programs with jump statements," in Proc. of the ACM SIGPLAN '94 Conf. on Prog. Lang. Design and Impl., June 1994, pp. 302–12.

[17] J-D. Choi and J. Ferrante, "Static slicing in the presence of goto statements," ACM Trans. on Prog. Lang. and Sys., vol. 16, no. 4, pp. 1097–1113, July 1994.

[18] S. Sinha, M. J. Harrold, and G. Rothermel, "Systemdependence-graph-based slicing of programs

with arbitrary interprocedural control flow," in Proc. of the 21st Int'l Conf. On Softw. Eng., May 1999, pp. 432–441.

[19] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano,V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalape˜no dynamic optimizing compiler for Java.In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.

[20] Craig Chambers, Igor Pechtchanski, Vivek Sarkar, Mauricio J. Serrano, and Harini Srinivasan. Dependence analysis for Java. In *12th InternationalWorkshop on Languages and Compilers for Parallel Computing*, August 1999.

[21] P. Chang, S. Mahlke, W. Chen, N. Warter, and W.-M. Hwu. IMPACT: An architectural framework for multipleinstruction-issue processors. In *Proc. 18th International Symposium on Computer Architecture*, pages 266–275,1991.

[22] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, September 1999.

[23] John Hennessy. Program optimization and exception handling. In *8th Annual ACM Symposium on the Principles of Programming Languages*, pages 200–206, 1981.

[24] U. Holzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.

[25] M. Cooperman. Debugging optimized code without being misled. *ACM Transactions on Programming Languagesand Systems*, 16(3):387–427, 1994.

[26] D. Dhamdhere and K. Sankaranarayanan. Dynamic currency determination in optimized programs. *ACM Transactions on Programming Languages and Systems*, 20(6), November 1998.

[27] P. Chang, S. Mahlke, W. Chen, N. Warter, and W.-M. Hwu. IMPACT: An architectural framework for multipleinstruction-issue processors. In *Proc. 18th International Symposium on Computer Architecture*, pages 266–275,1991.

[28] K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatbility. In *Proc. 24th International Symposium on Computer Architecture*, pages 26–37, June 1997.

[29] S. Mahlke, W. Chen, R. Bringmann, R. Hank, W.-M. Hwu, B. Rau, and M. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, 11(4):376–408, November 1993.

[30] M.D. Smith, M.S. Lam, and M.A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proc. 17th International Symposium on Computer Architecture*, pages 344–354,May 1990.

[31] J. Keane. Knowledge-based Management of Legacy Codes for Automated Design. PhD thesis, Rutgers University, October 1996.

[32] J. Keane and T. Ellman. Knowledge-based re-engineering of legacy programs for robustness in automated designs. In Proceedings of the Eleventh Knowledge-Based Software Engineering Conference, 1996.

[33] Andrew Gelsey, Don Smith, Mark Schwabacher, Khaled Rasheed, and Keith Miyake. A search space toolkit. Decision Support Systems - special issue on Unication of Arti_cial Intelligence with Optimization, 18:341-356, 1996.

Pawan Nagar
M.Tech. CSE (Lingaya's University, Faribabad)
B.E., CSE (LIMAT, Faridabad)
Diploma, ECE (G.P. Nilokheri, Karnal)

Nitasha Soni Asst. Professor, Dept. of CSE (Lingaya's University, Faridabad)