A Static Study of Java Exceptions using JESP*

Barbara G. Ryder, Donald Smith, Ulrich Kremer, Michael Gordon, and Nirav Shah

Department of Computer Science
Rutgers University
110 Frelinghuysen Road
Piscataway, NJ 08854-8019, USA
fax: 732-445-0537

`ryder, dsmith, uli, fabfour, nshah @cs.rutgers.edu`

Abstract. JESP is a tool for statically examining the usage of user thrown exceptions in Java source code. Reported here are the first findings over a dataset of 31 publicly available Java codes, including the JavaSpecs. Of greatest interest to compiler writers are the findings that most Java exceptions are thrown across method boundaries, `try`s and `catch`es occur in equal numbers, `finally`s are rare, and programs fall into one of two categories, those dominated by `throw` statements and those dominated by `catch` statements.

* The research reported here was supported, in part, by NSF grant CCR-9808607.

1 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.

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.

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-***un***friendly*, 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[12, 11, 8] 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.

In addition to the development wrapper-based automated systems, a general shift toward exception-based programming paradigms is being observed. There are several reasons for this shift including program correctness, program clarity, and elimination of side effects. These reasons have been noted by many in the community. Texts and references now contain persuasive arguments, such as the following.

> Exceptions provide a clean way to check for errors without cluttering code. Exceptions also provide a mechanism to signal errors directly rather than use flags or side effects such as fields that must be checked. Exceptions make the error conditions that a method can signal an explicit part of the method's contract. *p.151*[1]

These two trends, along with the growth in popularity of object-oriented languages such as Java, make it clear that exceptions will play an increasing role in program design, and thus present a new challenge to optimizing compilers.

The Java exception model is very strict as to what program state must be made available to the user when an exception occurs.

> Exceptions in Java are *precise*: when the transfer of control takes place, all effects of the statements executed and expressions evaluated before the point from which the exception is thrown must appear to have taken place. No expressions, statements, or parts thereof that occur after the point from which the exception is thrown may appear to have been evaluated. If optimized code has speculatively executed some of the expressions or statements which follow the point at which the exception occurs, such code must be prepared to hide this speculative execution from the user-visible state of the Java program. *p.205*[9]

In order to produce efficient optimized code that meet these specifications, safety conditions must be guaranteed for any optimizing transform used by the compiler. Since exceptions produce new and distinctive control-flow patterns, new static analysis techniques must be developed that accurately estimate control flow[5, 4, 6, 10, 14]. Speculation of instructions must be consistent with what is required by this exception model[2, 13, 7]. Practical compile-time analyses and code transformations should take advantage of common patterns of exception usage, especially in calculating possible profitability of transformations or the appropriateness of approximation in analysis.

We partition exceptions into two classes: those that are not usually thrown by a user and those that are usually thrown by a user. The former are usually thrown by the runtime system and are objects in the *RuntimeException* class; for example, *NullPointerException* and *ArrayBoundsException* are among these. The latter usually appear in a `throw` statement of a user's Java program and are often, but not always, a user-defined subclass of *Exception*. Both kinds of exceptions complicate program control flow. The empirical study reported here is an examination of Java program source codes using the Prolangs JESP tool to determine their patterns of user thrown exceptions. These initial results (and further planned static and dynamic studies) will serve as a basis for focusing compiler writers' attention on those uses of exceptions which actually occur in real codes. Additionally, it will allow compiler writers to de-emphasize those that may be difficult to transform, but do not occur very often (e.g., nested `finally`s).

Outline. Section 2 presents the dataset used in this study. The **JESP** tool and empirical measurements obtained in the study are explained and interpreted in Section 3. Ramifications of the findings for optimizing compilers are listed in Section 4. Related work is mentioned in Section 5. Section 6 states the conclusions and briefly describes future work.

2 Description of Data

As an initial study, a representative set of popular Java applications and applets were gathered. Included are a large number of programs from JARS, the Java

Review Service,[1] rated in their top twenty percent by utility. These programs include:

– **IceMail** - a Java email client
– **JFlex** - a lexical analyzer generator
– **RabbIT** - a web proxy
– **WebHunter** - a file downloader
– **LPR**- a class library that allows printing to a standard LPD (Unix) printer over the network.
– **Creature** - an artificial life simulator
– **Statistician** - lists method statistics for a class
– **Vigenere** - encrypts text based on the vigenere algorithm
– **JavaBinHex** - a BinHex decompresser
– **JHLZip** - compresses an uncompressed zip file
– **JHLUnzip** - decompresses a compressed zip file
– **Othello** - a Reversi applet

A large number of Java programs that are currently being developed also were included. Among these programs are McGill University's **Soot beta 4**[15], a Java bytecode analysis and transformation framework and its accompanying tools. Additional programs, such as the **Hot Path Browser**[2] and **JavaCC**[3] came from Lucent Technologies/Bell Laboratories and Sun Microsystems, respectively. The **Hot Path Browser** allows the user to view the most frequently executed paths of a program given the correct path information. **JavaCC** is a parser generator. The final two programs are from SBKTech; **Jas** a Java bytecode assembler and **Jell** a parser generator.

3 Findings

3.1 Experimental Framework

To gather the static measurements of the sample set of applications, a set of tools collectively known as **JESP** (Java Exception Static Profiler) were built using the **JTrek** library from Compaq/Digital[4] The **JTrek** library is a Java API that allows developers to instrument, profile, and decompile Java classes. It allows for rapid development of both static and dynamic profiling tools. Specifically, the Trek class library models Java code as a hierarchy of objects. The Trek API allows a developer to scan through a program's bytecode instructions, with classes representing the program's class files, and the fields, methods, local variables, statements, and instructions within them.

3.2 Data gathered

[1] http://www.jars.com
[2] http://www.bell-labs.com/project/HPB/
[3] http://www.suntest.com/JavaCC/
[4] http://www.digital.com/java/download/jtrek/index.html

	Program	Lines of Code	Bytes	Files Analyzed	Implements	Total Methods	% of Meth. w/ Exceptions
1	_227_mtrt	48	859	1	1	4	0%
2	JHLUnzip	187	3,262	1	0	3	67%
3	spec	283	5,420	2	0	11	18%
4	JavaBinHex	300	4,851	1	0	4	50%
5	Vigenere	305	9,333	3	2	22	18%
6	_999_checkit	332	5,899	3	1	8	25%
7	Statistician	387	6,996	1	0	2	50%
8	JHLZip	425	5,473	2	0	11	27%
9	Javacc-JJTree-JJDoc	602	21,573	6	0	22	14%
10	LPR	777	14,731	3	0	52	25%
11	_201_compress	927	17,821	12	1	44	2%
12	_209_db	1028	10,156	3	1	34	21%
13	WebHunter	1610	36,114	6	9	81	11%
14	_200_check	1818	40,381	17	7	107	19%
15	io	2631	58,057	8	0	98	13%
16	reporter	2743	71,703	18	3	114	21%
17	Othello	2804	41,427	18	0	164	9%
18	_205_raytrace	3751	57,000	25	1	176	4%
19	harness	4882	83,899	23	2	185	12%
20	Jas	4920	210,807	121	89	422	14%
21	RabbIT	5276	128,552	43	15	267	17%
22	Joie	6773	147,757	63	3	630	8%
23	Soot-SableUtil	7155	86,006	67	23	484	12%
24	_202_jess	10579	396,536	151	106	690	11%
25	Jell	10747	140,252	51	20	337	1%
26	Creature	11142	98,932	46	15	152	4%
27	HotPath Browser	11512	259,697	108	43	891	5%
28	Jflex	11796	192,839	53	16	420	7%
29	Soot-Jasmin	17152	253,292	99	22	664	8%
30	ICEMail	25974	457,246	135	72	966	13%
31	Soot beta 4	96144	1,033,630	651	276	3096	4%

Fig. 1. Programs in dataset

The 31 Java programs evaluated in this study are shown in Figure 1. They were obtained from JavaSpecs[5] as well as from public sources on the Internet and ranged in size from 48 to 96,000 lines of code. **JESP** analyzed each program and produced summaries of exception usage and structure based on JDK 1.1.5. The usage and structure of exceptions and exception constructs are reported in the sections that follow.

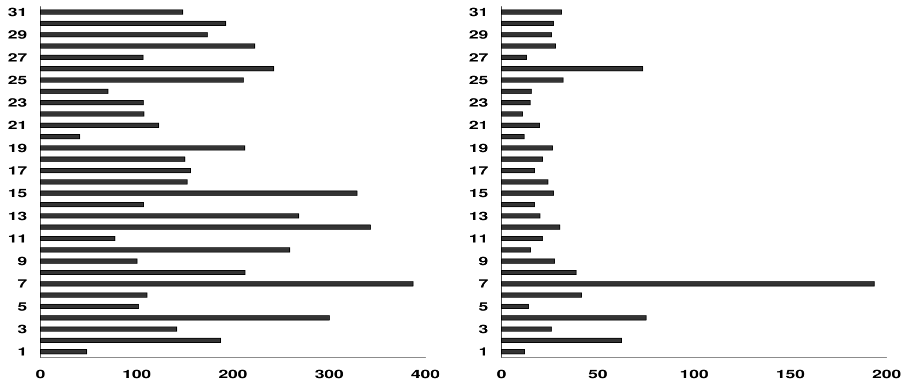
General Descriptive Data Figure 1 describes the programs that were evaluated. The codes are ordered from smallest to largest based on number of lines of code. Additional fields giving size in bytes, number of classes[6], number of classes with an `implements` clause, total number of methods, and percentage of methods with exception constructs (i.e., `throw, try, catch, finally`) are also shown.

Examination of Figure 1 shows that multiple inheritance, as indicated by an implements clause, is used infrequently in smaller programs of the dataset (i.e., under 5,000 lines of code). Figure 2(a) shows that lines of code per class

[5] http://www.specbench.org

[6] Figure 1 list number of files. In Java, files and non-internal classes are 1-1 mappable.

has little or no correlation with program size and Figure 2(b) corroborates the prevailing view that Java methods are short. For the programs examined the number of lines of code per method had little or no correlation with program size and averaged 33 (median 26) over all programs. Note: **Statistician** (id 7, with 6996 lines of code) is an outlier because it has only one class.



(a)Lines per Class (b) Lines per Method

Fig. 2. Average number of lines of code

`Try`, `catch`, `throw`, and `finally` statements This section reports on constructs that play a role in exception processing. **JESP** totals the number of `try`, `catch`, `throw` and `finally` statements along with the number of `throws` declarations used in a program. Methods declared to throw multiple exceptions are handled the same as methods declared to throw only one exception - each are counted only once.

Across the entire dataset, Figure 3 shows that on average 16% (median 13%) of the methods contain some kind of exception construct and that this percentage is insensitive to program size. This confirms the belief that exceptions will be ubiquitous in Java programs and highlights the fact that optimization transformations which are effective vis-a-vis exceptions will be necessary for efficient code generation.

Figures 4(a) and 4(b) display how many `catch`, `throw` and `try` statements occur in the dataset. Results for `finally` statements are not presented in the figure since they are rarely used; 26 of the 31 programs do not contain any `finally` clause. Similarly, `throws` were rare in the smaller programs; none of the programs smaller than 1615 lines of code contained a `throw`. Note that the number of `catch` and `try` statements in each program seem equal.

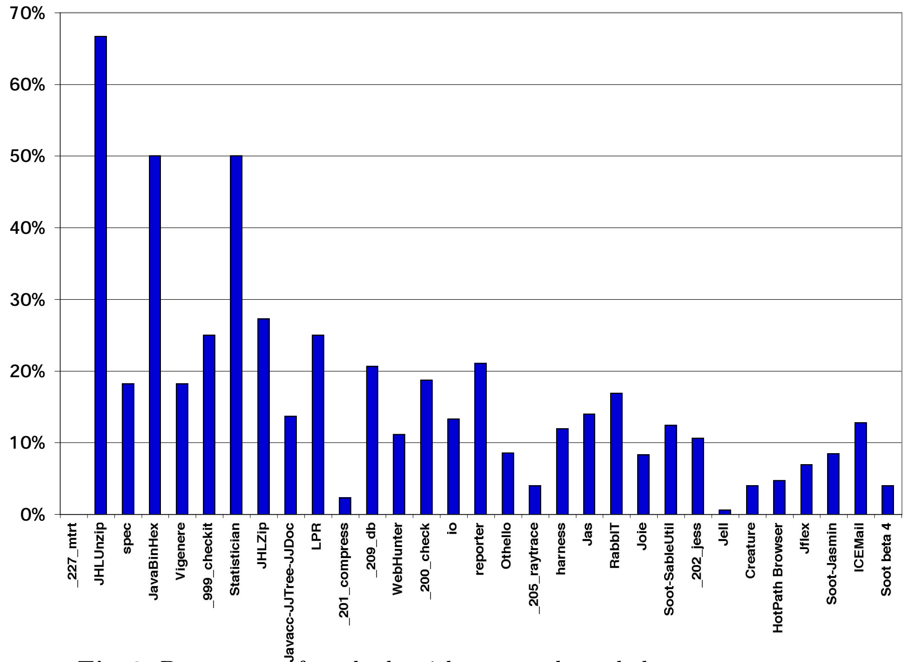
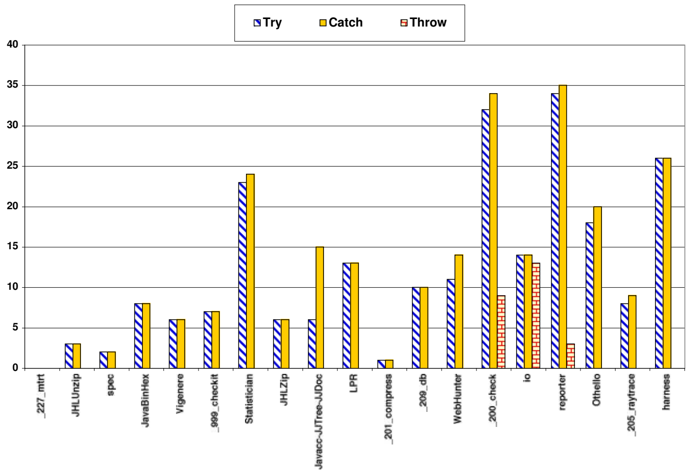


Fig. 3. Percentage of methods with try, catch, and throw constructs

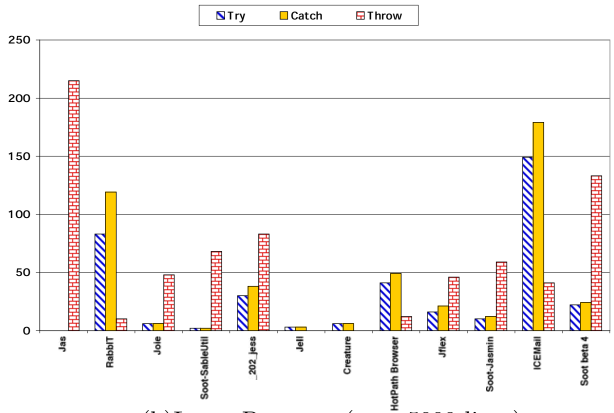
Figure 5 displays the ratio of `try` to `catch` statements. 50% of the programs have exactly 1 `catch` per `try` and only 1 program exceeds 1.5 `catch`es per `try`. This implies that `try` clauses almost always contain 1 `catch` statement and very rarely contain 2 or more `catch` statements. The fact that `try`s usually contain only 1 `catch` provides strong evidence that specialization and inlining transformations may be effectively employed to improve performance.

A surprising feature of our dataset is that almost every program is dominated by either `catch`es or `throw`s and seldom have similar numbers of each. Figure 6 shows the breakdown by percent of `catch` and `throw` statements. Only one procedure, **io** (id 16, with 2632 lines of code), has the number of throws and catches nearly equal. In all other cases the ratio of catches to throws is greater than 3-to-1 or less than 1-to-3. This is indicative of a difference in the kinds of applications being measured. Applications with more `throw`s than `catch`es are hypothesized to be libraries, which must be robust over all possible uses of their methods. Similarly, applications with more `catch`es than `throw`s are probably clients of libraries that are handling the exceptions returned by library routines rather than passing them up the Java runtime stack. This, along with lack of `finally` statements, indicates that exceptions are fully processed where, and when, they are first caught.

We were very interested in seeing how exceptions are used in `finally` clauses and to see if potential complications to control flow were observed in practice. Unfortunately `finally` statements were so rare in our dataset that reporting on exceptions in `finally` statements is impossible.



(a)Small Programs(under 5000 lines)



(b)Large Programs(over 5000 lines)

Fig. 4. Try, catch, throw counts per program

Distance between a `throw` and its corresponding `catch` JESP was used to measured specific patterns of exception constructs within `try` statements and to report the number of `throw`s not enclosed in a `try` statement. We considered four distinct usage patterns for `try` statements. `Try`s that contain:

– one or more `throw`s and a corresponding `catch`,
– no `throw`s and a `catch`,
– one or more `throw`s and no `catch`es, and
– no `throw`s and no `catch`es

Each `try` construct could optionally contain a `finally`; however for the dataset evaluated this had little impact. We found, as expected, that `throw`s do not occur in `try` statements but rather occur outside of `try` statements. Figures 7(a) and 7(b) summarize our data. These observations support our belief that

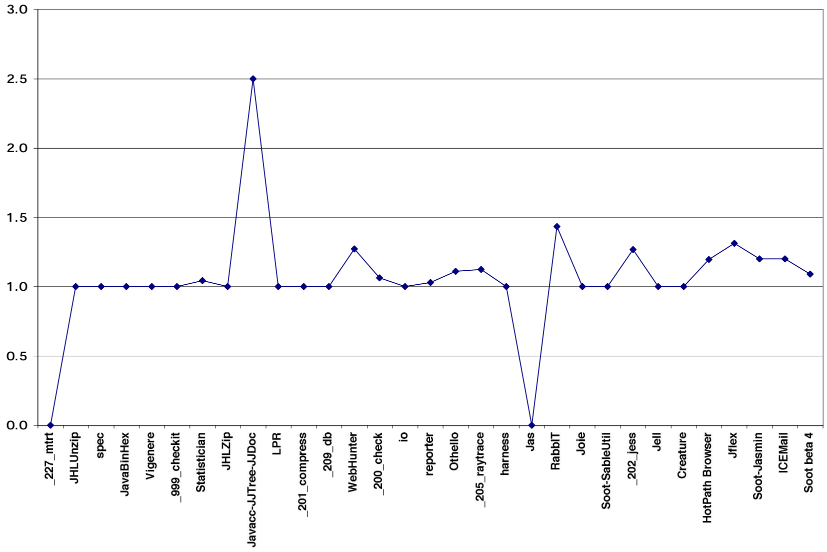


Fig. 5. Average number of catches per try

exception handling seems to be done by passing the exception up the current call stack. A `try` without a `throw` probably contains a method call that may throw an exception.[7] A `throw` not in a `try` is probably in a method called directly or indirectly from a `try` block. Thus, the next question to be investigated both statically and dynamically is "How far up the call stack is an exception object passed, on average?" This knowledge will aid in tailoring optimizations of this exception usage, especially for non-aborting control-flow exceptions.

Characterizing Exceptions JESP was also used to categorize the class of the object that is being thrown as either a Java-defined exception, User-defined exception, or Unknown. A Java-defined exception is an instance of an exception class that is defined in the JDK 1.1.5. User-defined exceptions are classified as exception objects that are instances of classes not defined in the JDK. These can include exception classes defined in any library code that the program uses. Figure 8 reports these results and clearly shows that user defined exceptions are much more prevalent than Java-defined exceptions. In two cases, library code was not available and we found programs that did not define an exception class yet did throw one or more user-defined exceptions. These are not included in Figure 8.

The prevalence of user-defined exceptions is strong supporting evidence that exceptions are not only ubiquitous but also gaining acceptance as a mainstream control flow mechanisms in Java. The implications are clear for optimizing com-

[7] It is possible for a try block to be used to catch exceptions such as *ArrayBoundsException* or other subclasses of *RuntimeException* which can be generated without a method call; however, we believe this use of `try` blocks is very rare.

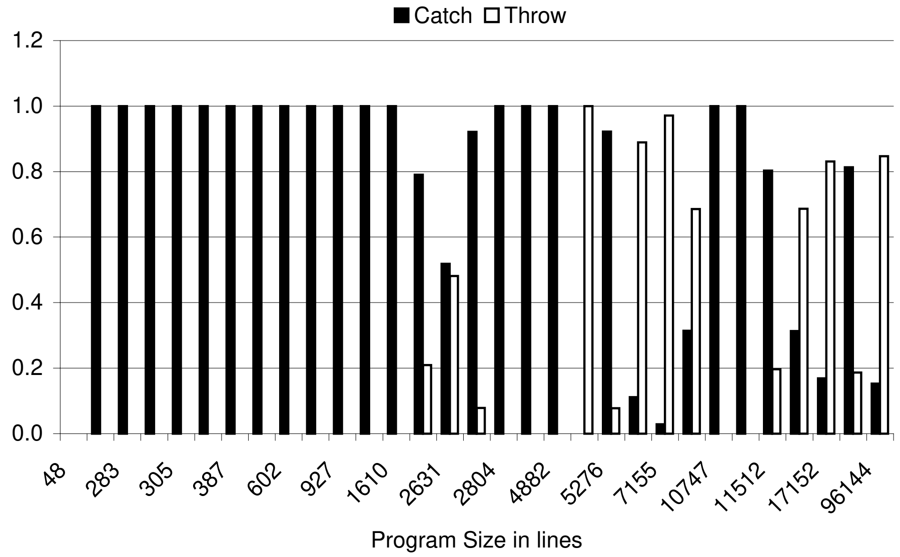


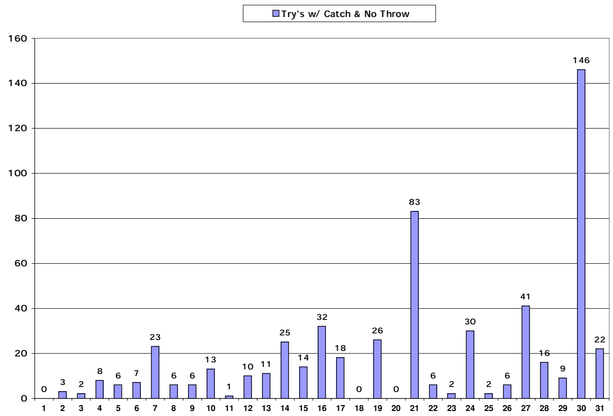
Fig. 6. Catch vs. Throw

pilers. They will have to produce efficient code from source that contains user-defined exceptions and `catch`/`throw` patterns that manage control flow.

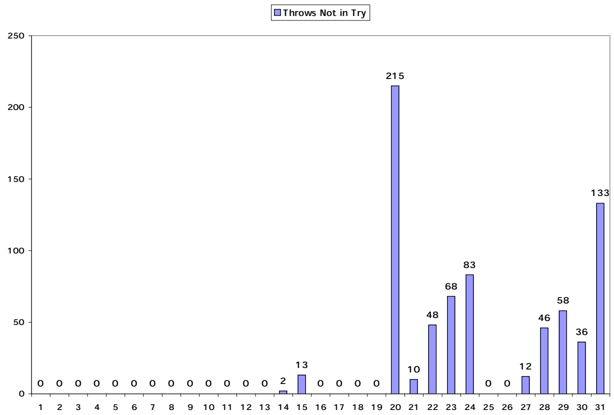
The final category of data that **JESP** gathered is information regarding the number of exception classes that the program contains as well as the shape of the exception tree hierarchy that the program defines. An exception class was considered to be any class that extends *Throwable* or any of its subclasses, excluding classes that extend *Error*. To analyze the shape of the exception tree, **JESP** gathers the maximum breadth and depth of the tree. Results of this analysis indicate that user-defined exceptions change the exception tree very little. Without any user-defined exceptions, the JDK's base exception tree has a maximum breadth of eleven and a maximum depth of four. With user-defined exceptions the breadth and depth changed very little if at all. More work is needed to separate the user- and Java-defined trees so representative figures can be reported. Preliminary indications are that user-defined exception hierarchies are very shallow (i.e., in our dataset no program had a user-defined hierarchy deeper than 2).

4 Ramifications

The gathered statistics about the use of exceptions in Java are preliminary. More experiments, in particular dynamic program instrumentation, are needed to give a more detailed picture of exceptions and their use in Java. However, some interesting conclusions and conjectures can already be made.



(a)Trys with catches but no throws



(b)Throws not in try block

Fig. 7. Exception constructs

– `finally` clauses are only used rarely in our benchmark suite. This suggests that the development of techniques to optimize `finally` clauses and handle the potential complexity they introduce into exception handling may not be profitable.
– The typical `try` blocks contains only one `catch` clause. This suggests that users have a particular exception event in mind when writing a typical `try` block and indicates a *local* treatment of exceptions may be possible. In such cases a compiler, supported by the appropriate analysis, may be able to determine which `catch` clause will process an exception thrown from a specific program point.
– The combination of `throw` statements being rare in `try` blocks, `try` blocks containing only one `catch` each, and the user-defined exception hierarchy

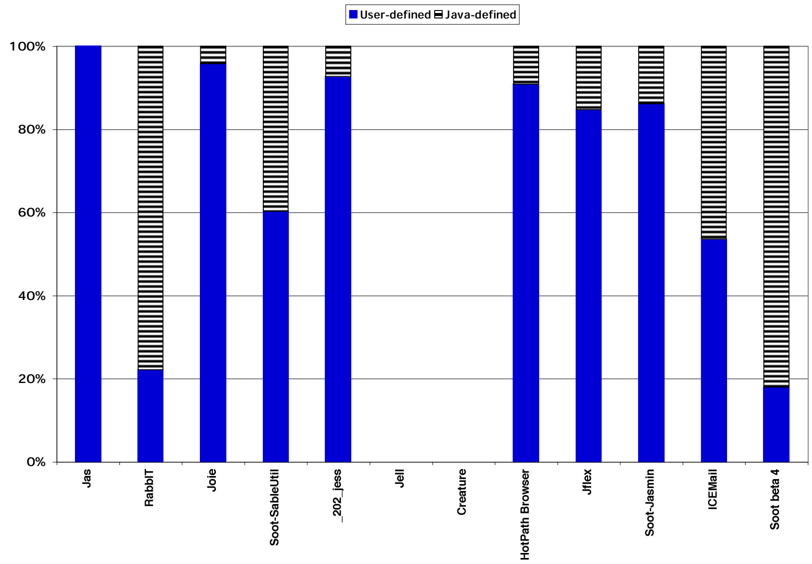


Fig. 8. User-defined versus Java-defined exceptions

being shallow indicates that the *static* and *dynamic* distance between `throw` and `catch` events in the call graph or on the run time stack may be short.

– There are a significant number of explicit `throw` statements outside of `try` blocks. This suggests that exception handling cannot be dealt with inside a single method. However, more work is needed to determine if exceptions are handled within methods of the same class or passed across classes. If exceptions are thrown and caught in the same class more aggressive analyses and optimizations are possible.

In Section 1 exceptions were classified as either *usually thrown by a user* or *usually not thrown by a user*; this study gathers information about the former. There is another useful classification of exceptions as either *abortive* or *control* exceptions. Abortive exceptions do not allow program recovery, but lead to the termination of the program, possibly after some clean-up or diagnostic analysis has been performed in the corresponding `catch` clause. In contrast, a control exception signals some abnormal, but anticipated program condition from which the program is expected to recover. In this case, the corresponding `catch` clause contains recovery code and initiates the continuation of program execution in an updated program state.

Since abortive exceptions lead to program termination, the benefit of optimized code for these exceptions is minimal. Since they are abortive, their expected frequency should be rather low in debugged and tested codes and any optimization will only speedup the final cleanup stages before termination. Therefore, efforts to optimize the execution of an abortive exception itself are not beneficial. This is not true for control exceptions since they represent a likely control flow path in the program. An aggressive optimizing compiler should optimize the execution of control exceptions and should try to limit the effects of all exceptions on the safety and benefits of other transformations.

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.

5 Related Work

There has been much recent interest in the problems presented to compilation and analysis of programs by Java exceptions. Of course, any Java compiler must handle exceptions; for space reasons all Java compilers under development cannot be presented here. Therefore, this discussion focuses on research which reports exception usage.

Sinha and Harrold[10] present new control-flow representations to model exception flow for data-flow and control-dependence analyses. They performed a quick static study of a seven Java programs and found that that on average 23% of their methods contained a `try` or a `throw`. Their dataset included: **jacorb, javacup, jdk, jlex, swing, tdb**, and **toba**. They use this data to substantiate the hypothesis that analysis algorithms will have to take account of exceptions in Java because of their prevalence.

Chatterjee, Ryder and Landi[5] presented a new flow-sensitive analysis, called *relevant context inference (RCI)* for a subset of Java that can model exception control flow. This analysis stresses optimized memory usage by only requiring the code for a method to be in memory three times during the analysis. Exception control flow is modeled as another context preserved by the algorithm (with alias and type contexts). Chatterjee's thesis[4] shows how to use RCI to calculate def-use associations in the presence of exceptions.

Krall and Probst[13] performed a small static study of five Java programs (i.e., **JavaLex, javac, espresso, Toba, java_cup**) used as data for their *CACAO* compiler. They measured number of try blocks versus number of method calls in these programs and found that the latter was two orders of magnitude larger than the former. They also reported on the number of null pointer checks in these programs which was more comparable to the number of method invocations.

Choi *et.al.*[6] present a new intermediate representation, a factored control flow graph, used in the IBM *Jalapeno* dynamic Java compiler that accommodates exits caused by possible implicit exceptions such as *NullPointerException*. They report static counts of the number of basic blocks with exception handler edges for implicit exceptions, both for the traditional basic block and their new design.

The *Marmot* optimizing Java compiler from Microsoft[7] also optimizes implicit exceptions. Their transformations make it possible to catch *NullPointerExceptions* using hardware.

Robaillard and Murphy[14] report on **Jex**, a static tool for identifying user-defined exception control flow in Java programs. Given a Java source file and the user-called packages (denoted by the user), **Jex** derives a representation of exception flow through the program. Exceptions generated by the Java API were also counted. For each Java method, **Jex** produces a summary of those exceptions possibly thrown by its execution. The empirical study reported focused on the use of subsumption in `catch` clauses. About 44% of the exceptions within `try` blocks were not caught with the most precise exception type available. The goal was to give this information to a programmer to increase their understanding of possible exception flow in their program.

Brookshier[3] measured and reported on the cost of exception processing for control exceptions when the handler is not close to the method throwing the exception on the runtime stack. He describes the cost of a `throw` as due to creation of an exception object, an invocation of the handler, a search of the stack for the current method, a search for a `try` block range, and then the search for a matching `catch`. He notes that cost of a `throw` is related to the depth of nested frames, so that speed can be improved if the `catch` clause is in the same method or relatively close on the runtime stack. Measured on a 300MHz Pentium II with Symantec's 1.1 JIT, there was a slowdown of 80 milliseconds per exception in a program written so that the handler is several stack frames from the throw.

6 Conclusions and Future Work

The studies reported here with the Prolangs **JESP** tool are an attempt to discern the generality with which explicit exception constructs (i.e., `try, catch, finally, throw`) occur in Java codes. The most important results were that a substantial percentage of methods (on average 16%) in a Java program contain exception constructs, so exception usage is ubiquitous, but exception constructs are fairly sparse (the number of `try`s match number of `catch`es). `Finally`s are rare and thrown exceptions are usually not caught within the same method. User-defined exception hierarchies are shallow. Another interesting finding was the observed dramatic categorization of programs which display many more `throw`s than `catch`es and *vice versa*.

As with all initial studies, this one raises more questions than it answers. Future work includes measurement of the:

– dynamic behavior of user-thrown exception,
– use of control and abortive exceptions,
– number of user- and Java-defined exceptions caught, and

– number of times a `catch` clause catches an exception thrown from within the `try` block.

Acknowledgments. We sincerely thank Seth Cohen from Compaq Corporation whose help with **JTrek** was invaluable.

References

1. K. Arnold and J. Gosling. *The Java Programming Language, Second Edition.* Addison-Wesley, 1997.
2. M. Arnold, M. Hsiao, U. Kremer, and B.G. Ryder. Instruction scheduling in the presence of java's runtime exceptions. In *Proceedings of Workshop on Languages and Compilers for Parallel Computation (LCPC'99)*, August 1999.
3. D. Brookshier. Exception handling: Simpler, faster, safer. *Java Report*, 3(2), February 1998.
4. R. Chatterjee. *Modular Data-flow Analysis of Statically Typed Object-oriented Programming Languages.* PhD thesis, Department of Computer Science, Rutgers University, October 1999.
5. Ramkrishna Chatterjee, Barbara G. Ryder, and William. A Landi. Relevant context inference. In *Conference Record of the Twenty-sixth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, January 1999.
6. J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 1–11, September 1999.
7. R. Fitzgerald, T.B. Knoblock, E. Rif, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compilr for java. Technical Report Technical Report MSR-TR-99-33, Microsoft Research, June 1999.
8. Andrew Gelsey, Don Smith, Mark Schwabacher, Khaled Rasheed, and Keith Miyake. A search space toolkit. *Decision Support Systems - special issue on Unification of Artificial Intelligence with Optimization*, 18:341–356, 1996.
9. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
10. M.J. Harrold and S. Sinha. Analysis of programs with exception-handling constructs. In *Proceedings of the International Conference on Software Maintenance*, November 1998.
11. J. Keane. *Knowledge-based Management of Legacy Codes for Automated Design.* PhD thesis, Rutgers University, October 1996.
12. 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.
13. A. Krall and M. Probst. Monitors and exceptions: How to implement java efficiently. In *Proceedings of 1998 ACM Java Workshop*, 1998.
14. M. Robillard and G. Murphy. Analyzing exception flow in java programs. In *Proceedings of the 7th Annual ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 1999.
15. Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lamand Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *CASCON99*, Toronto, Ontario, September 1999.