The Case for Exception Handling

by

Michael M. J. Zastre
B.Sc., Simon Fraser University, 1993; M.Sc., University of Victoria, 1996

A Dissertation Submitted in Partial Fulfillment of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Supervisor: Dr. R. Nigel Horspool

ABSTRACT

Mechanisms for handling exceptions within programming languages are now nearly forty years old, and attitudes towards exception handling betray the original association of "exception" with "error". There have been several attempts to justify the use of exception handling as a general programming tool for expressing control flow, but the current consensus is that "exceptions should be rare, therefore their implementation need not be efficient". This thesis argues the opposite view by making several contributions, roughly categorized as: (a) examination of the use of exceptions in the Java programming language, and in turn using these results to identify opportunities for improving performance in a Java Virtual Machine (JVM); (b) identifying and discussing several programming idioms which lend themselves to an exception-handling style; and (c) introducing a program transformation to improve performance of regular code by using exceptions to eliminate redundant run-time tests. A research VM has been modified to explore the effectiveness of these ideas, and experimental results are presented. The overall aim of this thesis, however, is to present the viewpoint that exceptions should and can be used more often, and that significant opportunities for improving exception-handling performance do exist within purely-interpreted (*i.e.*, non-JIT) VM implementations.

Table of Contents

List of Figures

List of Tables

Acknowledgements

Without the patience and encouragement of my advisor, Dr. Nigel Horspool, I could not have undertaken and completed this dissertation—I thank him for being so tenacious and seeing potential where I sometimes saw none. Many other faculty members in the Department have also given me words of encouragement at many points through my career as a student at the University of Victoria: Drs. Eric Manning, Hausi Müller, Micaela Serra (who has harried me endlessly), Dale Olesky and Dan Hoffman. My special thanks go to Drs. Jon Muzio and Michael Miller who have been very supportive in the last several months of this work, especially in lifting for a short while some administrative burdens from my shoulders. Several others of my colleagues have also been of wonderful assistance in helping me manage my teaching load: Dr. Daniel German made available to me his excellent slides for courses that we have both taught, and Dr. Yvonne Coady took on much extra work in Fall 2003 (not to mention her taking the time to cheer me home!); all this helped free up precious hours and days. Several professors from my time as a student at Simon Fraser University have been an inspiration, and have also spoken the right words at the right time, especially when I needed to hear them: Drs. Ronald Harrop, Stella Atkins and Basil McDermott. My deepest thanks also go out to Dr. Bernhard Scholz with whom I had many interesting, varied and joyful conversations when he visited Victoria in 2002 as part of his sabbatical from Technische Universität Wien.

Many past and present fellow graduate students have lightened my days and communicated to me both their enjoyment of computer science and the fun they have had practicing it. In no particular order they are: Jim Uhl, Robert Bryce, Watheq El-Kharashi, Gordon O'Connell, Claudio Costi, Gord Brown, Duncan Hogg, Jackie Rice, Raj Panesar, Piotr Kaminski and Ken Kent. My special and heartfelt thanks are given to John Aycock and Shannon Jaeger (plus Melissa and Amanda!) who gave me a place to hide from the world in February 2002—this short break provided me with just the right amount of energy to begin the final stage of my doctoral work. I also thank Jan Vitek for his constant encouragement and interest through the years.

My friends and family have also given me support through all these past few rollercoasterish years. Jim & Bertha Gunson have always had an open ear and an

open door, and our conversations from way back in 1989 helped give to me the confidence to arrive here. Several past and current members of St. David-by-the-Sea Anglican Church have often been good sounding boards and sage advisors: The Revd. Canon Andrew Gates, and Drs. Denis Brown, Peter Taitt, and Willard Allen. I give thanks for having the pleasure of Saturday morning coffees with Ken Shilson as this gave me a chance to lift my mind to other sights and ideas—the result has been that I have returned to my work refreshed; the same is true for our visits with Don & Alison Good and Emile & Muriel Lacroix. My fellow members of Hexaphone have shown me that "Music for awhile" shall indeed "all your cares beguile," and they have also taken on cheerfully some of the tasks that I simply could not find the time to do. My parents and siblings have, once a year without fail, checked if I'm finished, and then spurred me on when I have answered "not yet." Carol Benoit has stepped in to our home and been a loving friend to our son and ourselves, and has also cheerfully taken care of wee Robert at those times when I simply was too overwhelmed.

I had dearly wished that my friend, Joel Morris, would have lived to see the completion of this work. He believed I could do this long before anybody else did—even and especially myself—and with Darrel Seyler made the beginning of this long journey possible. I am thankful that Darrel will be able to share with me the joy and excitement of the completion of this degree.

Finally I could not have completed any of this work were it not for the love, companionship, and presence of my wife, Susanne, and my son, Robert. They have put up with a distracted husband and father for so long that I'm sure they will be puzzled to greet the man who has finished his doctorate.

Dedication

Für Susanne ...

"Mein guter Freund, das wird sich alles geben;
Sobald du dir vertraust, sobald weißt du leben." (Goethe—Faust I)

... und Robert.

Chapter 1

Introduction

This thesis aims to rehabilitate *Exception Handling* (or *EH*) from its current low station. While in the 1970's it showed great promise, it has since then descended—through various incarnations—into the current set of slow and bulky (*i.e.,* large code size) realizations amongst various programming languages. It is my view that the relatively high run-time cost of using exceptions, when compared with other programming constructs, actively discourages the use of exceptions even when their use may result in clearer and more easily-maintained code. A three-pronged approach is used to address this: (1) clearly identify the sources of high run-time costs for various EH mechanisms; (2) introduce new compile-time and run-time analyses for reducing this cost; and (3) provide a "dictionary" of solutions to programming problems using EH-based idioms.

But what is "exception handling"? Most modern languages now support exceptions—the list includes Ada-95, C++, C#, Eiffel, Java, Haskell, Standard ML, Modula-3, Python, and Smalltalk. Some of these are imperative-style languages, a few are functional languages, and one is widely used for scripting. The syntax and semantics of each EH implementation differs in subtle and not-so-subtle ways amongst languages. The "Survey" section of this thesis contains a brief discussion of termination- *vs.* resume- *vs.* retry-semantics; here is a (non-exhaustive) list of other differences:

- *Exception handlers* correspond to program text mapping *exception types* to program behavior, *i.e.,* handlers specify the code that must be executed (the *catch-block*) when an instance of the exception type occurs in a particular—but disjoint—region of code (the *try-block.*) However, while Java try-blocks may be as short as a single statement, Eiffel try-blocks may only enclose a whole procedure.

- *Exception-dispatch mechanisms* transfer flow-of-control from an exception *throw site* (*i.e.*, the program point at which the exception occurs) to the start of the matching handler. Imperative-style languages such as Ada-95 invoke the exception-dispatch functionality provided in the language's run-time system. Functional-style languages such as Standard ML, however, model exceptions as algebraic types where "throwing an exception" is the same as "return a value of this exception type." The former control-transfer mechanism has a higher run-time cost than the latter.
- *Exception types* may belong to hierarchies of built-in types, user-defined types, or classes. For instance, a particular C++ catch-block may provide code to handle an exception of some type T. Any instance of a subtype of T will also be handled by the same catch-block. Of interest here, however, is that the exception type must be specified by the catch-block at compile-time. Smalltalk also allows catch blocks to specify a set of types to be handled (*i.e.*, exception class plus its subclasses) but this set cannot be known in general until run-time.

The proposals of this thesis must take into account the existence of these differences. Analyses which work well for Java and Ada may not work well for Eiffel and perhaps not work at all for Smalltalk. Programming idioms suitable for Haskell and Standard ML may require too much extra phrasing in C++ to be easily writable, readable and maintainable. Our rule of thumb will be to cast most of the research contributions toward Java, but should benefits result from the application to other (mostly imperative) languages, they will be highlighted.

1.1 A concrete example of EH (Java)

Figure 1.1 contains several Java-like code fragments which use exception handling. The two methods are part of some contrived class; methods `computeIndex()` and `computeSentinel()` are defined elsewhere. The first method contains two catch blocks and the second method contains none. There are a few program lines illustrative of important cases:

1. The only point at which an exception is *explicitly thrown* is at line 23. A new instance of `NumberFormat-Exception` is created and passed as the argument

to `throw`.

2. The signature for `transform()` indicates that a `NumberFormatException` may be *propagated out* of the method; this follows from the absence of any catch block in the `transform`. Since `NumberFormatException` is considered to be a *checked exception*, any possible propagation outside of a method must be noted via the "throws" signature modifier. If the immediate call to `transform()` at line 8 leads to the execution of line 22, then control will be transferred from line 22 to line 15.

3. The handler at line 11 catches an exception caused when some array is indexed outside the array's lower and upper bounds. This is an example of an exception considered to be *unchecked*; such exceptions may be propagated out of a method regardless of whether or not a `throws` modifier is used for that method. For example, if the value `B.length` (*i.e.*, the size of `B[]`) at line 18 is 10, and if the result of the call to `computeIndex()` is 20, then the semantics of the Java language require an `ArrayIndexOutOfBoundsException` be thrown.

   Combining checked with unchecked exceptions will complicate exception analysis. In order to determine as precisely as possible which catch blocks are "reachable" from a given throw site, we must perform some inter-procedural analysis, *i.e.*, analyzing method signatures will not be sufficient. The use of such an analysis is motivated by the next item.

4. Neither handler in `process()` uses the exception object passed into the handler (`e` at line 11, `n` at line 14). Therefore an exception caught by either of these handlers can be considered the same as a non-local goto from the throw-site to the exception handler. In the case of Java there is a significant cost for creating an exception object—creating the object, constructing a stack trace, etc. —which will be wasted in these two cases. Some analysis as suggested the previous case (case 3) is desirable to determine if we can avoid the cost of exception-object creation.

   There are a few additional wrinkles, however. Given the support for polymorphic method calls in Java, it may be either too expensive or virtually impossible to obtain an inter-procedural analysis precise enough to optimize away exception object construction. Removing the creation of an exception object may

itself eliminate an exception thrown within the object constructor—and Java semantics require that the constructor's exception be thrown. Finally, class loading in Java (or dynamic linking in other languages) could invalidate any analysis if a class method was added that both provides a possible handler for a throw site and also uses the exception object caught by this handler.

5. Lastly, dispatching a thrown exception in Java requires examination of both the static and dynamic context of the throw. The static scope must be searched for a local handler (*i.e.*, a catch-block within the same activation of the method throwing the exception); if no local handler is found, then methods comprising the call stack must be examined and local handlers found within them. This mix of static and dynamic scoping of exceptions contributes to their complexity and to the cost of dispatch.

The example code of Figure 1.1 uses Java exceptions in a straightforward way, but the code is admittedly contrived. A more prosaic and useful example taken from Smalltalk is shown next.

1.2 A concrete example of EH (Smalltalk)

Figure 1.2 shows a short Smalltalk message using exception handling. This message is actually a wrapper around access to a matrix; the matrix is a two-by-two array of characters representing a crossword puzzle. Black squares are represented by the asterisk, and to ease programming we would also like to use the asterisk as a sentinel value to indicate the boundaries of the matrix. By itself "`matrix in: row in: col`" accesses an element in the array (*i.e.*, "`in:in:`" is the name of the message, "`row`" and "`col`" are the formal parameters).

The access to the array in line 5 is surrounded by square brackets—in Smalltalk terminology this is a block context, and block contexts are themselves objects. Exception handling in Smalltalk is achieved by sending the "`on:do:`" message to a block context; the first parameter to this message is an object (usually a class object representing the Exception or Error of interest) and the second parameter is the action to be performed.

What this `on:do:` message catches, amongst other things, is an index-out-of-

```
1   float process ( int A[] )
    {
        int result = 0;

5       try {
            // ... some code
            A[computeIndex()] = 1;
            result = transform (A);
            // ...
10      }
        catch (ArrayIndexOutOfBoundsException e) {
            result = -1;
        }
        catch (NumberFormatException n) {
15          result = 0;
        }
        return (float)result;
    }
```

```
    int transform ( int B[] ) throws NumberFormatException
20  {
        // ...
        if ( B[computeIndex()] < 0 ) {
            throw new NumberFormatException();
        }
25      // ...
        return B[computeSentinel()];
    }
```

Figure 1.1. *EH in Java (1)*

```
class Crossword
message !!
in: row in: col

    [ ^matrix in: row in: col ]
              on: Error
              do: [^ '*']
```

Figure 1.2. *EH in Smalltalk (1)*

range error. Now any access to an element of `matrix` where **row** or **col** is out of the appropriate range will result in the asterisk being returned as a value. We have therefore obtained sentinel values without modifying the array.

Two observations can be made about this code:

1. An objection can be made that the intent of the message implementation is not apparent from the code. If checking for out-of-range indexes is required, then this is what should be written. However, the example handles far more cases than just out-of-range errors—if the **matrix** instance variable happens to be uninitialized, then the message shown will still return an asterisk as a result. The code shown is, in fact, idiomatic Smalltalk and would be understood by a Smalltalk programmer. An argument can be made that the code shown is somewhat more efficient than explicitly checking out-of-range errors—the Smalltalk run-time checks for them anyway, so work is duplicated.
2. Unlike languages such as Java, Smalltalk exceptions are not built into the language. Exceptions are implemented as messages to block contexts and as a consequence they have the same overhead as any other object message dispatch. The relative cost of exceptions *vs.* "regular" operations is quite small in comparison to that of Java or C++; therefore the style of programming shown in the example would not be considered inefficient. Modern Smalltalk environments also support both purely interpreted code and Just-In-Time compilers (or *JITs*).

Several other languages which do not have support for exceptions have added them via meta-programming and reflection—Oberon-2 is one such example [31]. In such cases, the EH mechanism uses facilities such as call-stack traversal and activation-record examination.

1.3 Classical use of exceptions

The first popular implementation of exception handling was designed to handle operating-system exceptions in the IBM OS/360—hence the term "exceptions." PL/I **ON**-conditions were used and a comparison of OS/360 exceptions to the number of supported conditions shows a close match [64, p. 503]. Since the early 1970s, a variety of other uses have been made for EH. Amongst others, these include:

- **Separate normal-case code from error-handling code:** One rule-of-thumb is that 90% of all code deals with the 10% of the input cases involving errors. One consequence for programming is that error-handling code is often deeply intertwined with normal-case code. EH enables the clear separation of the code—adding a new error-handling case can now be as simple as adding an exception handler, and the normal-case code is left untouched.
- **Library support:** Authors of code libraries can detect run-time errors (*e.g.,* memory-allocation errors, out-of-bounds array indexing, null-pointer dereferences, etc.), but do not know how best to deal with errors in all contexts. Users of libraries know best how to respond to such errors, but are usually unable to detect them. EH provides an error-communication interface between libraries and users of those libraries [66, p. 355].
- **Error return values:** A function may be required to return values other than those from a successful computation. Such multiple outcomes as *success, normal result, failure* and *impossible* are more easily represented by constructing datatypes which represent these outcomes as different values. In this case, EH is a way to deal with failure: exceptions are raised where failure is discovered, and the mechanism allows the error to be handled elsewhere (*i.e.,* possibly far away) [57, p. 134].
- **Support for language semantics:** In *contract-based* programming languages such as Eiffel [48], the user of an object is expected to obey the requirements for accessing the object's services as set down by the designer of the object's interface (*i.e.,* match pre-conditions). In a similar fashion, the implementors of the object are expected to obey the requirements on values returned by the object's services (*i.e.,* match post-conditions). If any of the requirements are

not met at run-time then an exception is raised, which would require that the exception handler either modify program state to satisfy the condition or return some value indicating the computation represented by the service could not be completed.

- **Asynchronous behavior:** Exceptions can be used as a way to deliver operating system signals to a program. In an extension to Concurrent Haskell, exceptions may be delivered from one thread to another [46]. This mechanisms supports programming techniques such as: speculative computation (two or more computations are started, with the earliest completing computation throwing exceptions to the others in order to stop them); time-outs; user interrupts; and dealing with resource exhaustion.
- **Meta-programming:** In languages such as Smalltalk where everything is an object, it is rather easy to send messages to objects which do not understand the messages. For all code other than early prototypes, this can be a serious problem. In this case an exception handler can catch such events (*i.e.,* `MessageNotUnderstood`) and either perform some other operation or query the object's class to find a suitable operation and dispatch to the corresponding message [74].
- **Support for regular programming:** Java-like exceptions allow a programmer to code multi-level returns from deeply-nested calls. Exceptions can also be used by algorithms to transfer control when important conditions occur—as an example, a compiler may use exceptions for actions involving missing symbols from symbol tables, syntax errors, semantic errors, etc.

The above examples range from providing mechanisms for handling errors to use of exceptions as an additional control-flow construct. In all cases they should be used to clarify code. However, one result of the past twenty-five years of programming language research is that exceptions are still thought of as rare, and therefore their implementation is usually quite slow. Programming language implementors usually assume that exceptions will be relatively infrequent events when compared to regular program instructions. Also considered acceptable is that throwing and catching exceptions may result in many more instructions executed at run-time than that of a regular procedure call. One of the few places in the literature where this assumption

is explicitly stated is in the "Modula-3 Report" [11, p.17]:

> Implementations [of Modula-3] should speed up normal outcomes at the expense of exceptions (except for the return-exception and exit-exception). Expending ten thousand instructions per exception raised to save one instruction per procedural call would be defensible.

It is unclear how the 10,000:1 ratio was determined, but many other language implementors appear to have taken this to heart. For example, throwing and catching a `NullPointer` exception in Java is about 5,000 times more expensive than an `if-then` statement testing a reference for `null`; in Smalltalk the ratio is 800:1; and in C++ the ratio is about 200:1 for Visual C++ and 80:1 for GNU's `g++` (cf. Appendix A for a description of how these numbers were obtained).

When exceptions are rare then such ratios are reasonable. Even in this rarity, compiler implementors have applied great ingenuity to both reduce the cost of throws and catches [13, 37, 39, 67]. However, there still appears to be something of a disdain for exceptions.

1.4 Attitudes to EH

The range of opinions toward EH could be broken down into four groups: (1) never use them; (2) rarely use them; (3) use them where sensible; and (4) use them where clear.

1.4.1 Never use them

There has been great reluctance to embrace exceptions regardless of their cost. In his 1980 Turing Award address, C.A.R. Hoare reflected on his experiences with the PL/I and Ada language projects [30]:

> I have been giving the best of my advice to [the Ada language] project since 1975. At first I was extremely hopeful. The original objectives of the language included reliability, readability of programs, formality of language definition, and even simplicity. Gradually these objectives have been sacrificed in favour of power, supposedly achieved by a plethora of

> features and notational conventions, many of them unnecessary, *and some of them, like exception handling, even dangerous.* [emphasis added]

At the time of Hoare's speech there were a wide range of EH mechanisms, including several with interesting and rather complex semantics (Chapter 2 explains why `retry`-semantics are no longer used).

Several years later, Andrew Black wrote in his dissertation (entitled "Exception Handling: The Case Against"):

> [Exception handling] mechanisms are included in the programming languages Pl/I, CLU and Ada ... All the mechanisms are "high-level" in the sense that they can be simulated by conventional language features. Their designers offer only the vaguest indication of their range of applicability, and when the motivating examples are re-written without exception handling there is often an improvement in clarity. This is partly because of the reduced weight of notation, and partly because the exception handling mechanisms obscure what is really happening. [6, abstract]

At about the same time as Hoare and Black made these comments, John Hennessy proposed that code using EH could be made more readable by limiting EH mechanisms to use termination semantics [29]. Since then both C++ and Java have been introduced, have achieved widespread use, and both have only termination-style exceptions.

Given the developments in programming languages since Hoare's lecture, he has appeared to have something of a change of heart. In 1999 he co-authored a paper on exceptions [32] in which they are described in positive terms.

1.4.2 Rarely use them

Another philosophy suggests that we should use exceptions when forced to do so, but if at all possible, their high cost should cause us to eschew their use. This is very much applicable to Java exceptions [62]. Using some of the core library functionality requires that exception handlers be written before code using the functionality will even compile.

While run-time cost is a concern, object-oriented languages once faced similar concerns when they were first introduced. Dispatching methods were seen to be

expensive and so the use of widespread overridden methods was considered unacceptable. Much time has passed and much excellent research into improving the run-time performance of dynamic dispatch has now made the transition to production-quality compilers. We can now use such powerful tools as the C++ Standard Template Library. Given that high cost of dynamic dispatch has been reduced as a result of extensive research—and that this is no longer an argument against the use of object-orientation—we believe a similar argument can eventually be made for exception handling. As exceptions' run-time cost is reduced, we expect they will be used more frequently by programmers.

1.4.3 Use them where sensible and clear

Bjarne Stroustrup devotes a few pages to "Exceptions and Efficiency" in his book "C++: The Programming Language." In it he writes:

> In principle, exception handling can be implemented so that there is no run-time overhead when no exception is thrown. In addition, this can be done so that throwing an exception isn't all that expensive compared to calling a function [66, p. 381].

Later in the chapter on "Exception Handling" he writes:

> Use exceptions for error handling; don't use exceptions when more local control structures will suffice [66, p. 386].

Given that C++ has been designed to support the design, implementation and maintenance of large programs, Stroustrup's advice would seem to imply the exceptions should be used when they make code clearer. Unfortunately, exceptions are still expensive in most C++ implementations, but not at the behest of the language's creator.

1.5 Structure of dissertation

The next chapter provides a survey of the varying EH semantics and EH implementations. Chapter 3 explores the various analyses—both compile-time and run-time—for identifying the catch-block for given throw sites. It also includes a description

of several exception-handling implementation techniques. Chapter 4 discusses several idioms which take advantage of exception-handling control flow, and which will result in good run-time performance (in comparison to version not using exceptions) if given a good EH implementation. Chapter 5 explores some transformations of Java programs that use exceptions to improve run-time performance; these transformations are applied to code written by a programmer. Chapters 6 and 7 present more contributions to program analysis and Java Virtual Machine implementation. Chapter 8 presents experimental results. The dissertation concludes in Chapter 9 with remarks on future work.

Chapter 2

Exception Handling Survey

One goal of this thesis is to convince the reader that exceptions are not only a helpful way to structure programs, but that there exist compile-time techniques to ensure program performance is not degraded as a result. Here we have a problem in exposition: exception handling mechanisms in the past were suggested and implemented because of the problems that they solved, yet present implementations of modern mechanisms are relatively slow compared to other control-flow mechanisms. What prevents widespread use of exceptions is not their ability to solve problems, but rather (1) their overhead at run-time, and to some extent (2) difficulties arising from combining exceptions with other language features. These two aspects of the EH problem are separate, and so this survey is broken into two parts. The current chapter surveys a range of mechanisms—not all, but the most significant—and especially focus on the problems of programming that they were designed to solve. One interesting mechanism that was proposed but not widely used will also be examined. The next chapter focuses on actual implementations of several mechanisms and explores the handful of EH analyses proposed for compilers. Many of these analyses are specializations of other standard analyses (*e.g.*, array-bounds-check removal) in that they attempt to reduce the occasions for which exceptions are thrown; this is different from what we propose as techniques to improve exception performance.

The arrangement of material in this chapter is chronological—as the range of mechanisms is sometimes wide (and at times particular mechanisms appear to be essential), this order appears as good as any. There have been several surveys with other treatments and arrangements of this material (in particular Buhr & Mok [8] and Lang & Stewart [38]) which are cited in later chapters. Soffa & Ryder describe the history of interactions between software engineering research and the introduction

of new exception handling mechanisms in [60]. We also survey some of the proposals made for the use of exceptions as a support in constructing software systems whether they be termed "robust", "fault-tolerant", or some such. Our statement that exceptions need to be rehabilitated is not a new one—it has been stated by others in the past. But first we turn to cover some basics.

2.1 Terminology and concepts

There is no one definition of an exception—which seems reasonable as there are so many different semantics and mechanisms—so we will defer presenting one until the end of this chapter. There is, however, a general acceptance of the meaning of these terms:

- *exceptional control flow*: Transfer of control from one program point to another program point that is only possible via a language's EH mechanism (*i.e.,* is not the direct result of sequencing, looping, function invocation, function return, or any other form of structured control flow).
- *throw, raise*: When used as a keyword in a programming language, represents a program point which, when executed, will begin exceptional control flow. When used as a verb, describes the act of transferring control from one program point to another using exceptional control flow.
- *catch, handle*: When used as a keyword, represents a syntactic construct (keyword, block, notation, etc.) that may be the destination of a control flow transfer. When used as a verb, describes the act of re-establishing normal control flow from the exceptional control flow.

While the terms "normal control flow" and "exceptional control flow" are somewhat awkward, they convey the notion amongst many programmers of control flow that is "relatively easy to understand" versus that which is "relatively hard to understand."

- *handler*: A syntactic construct which is intended to be the destination of some transfer of control via an exception. Handlers may themselves contain "normal" code, explicit or implicit throws, or even nested handlers.

- *handler binding*: The syntactic or semantic association of handlers with program text.
- *propagation*: A sequence of exceptional control flow transfers which may either terminate in the resumption of normal control flow or in the termination of a program.

The exact meaning of each term for a specific language supporting EH will vary; attempts to capture the semantics of the difference between normal and exceptional control flow are many and varied ([32, 68, 7, 44]). It appears to be true in every case that exceptions do not fit comfortably in the language, *i.e.*, the difficulty many programmers have in switching between normal and exceptional control flow is reflected in the semantics.

Despite the difficulties mentioned above, however, exceptions have been a part of nearly every major programming language since the late 1960s. To understand why, we must examine the EH mechanisms provided by these languages along with what they suggested to designers of newer languages.

2.2 PL/I

When IBM began to plan the rollout of the System 360 Series in the early 1960's, they also anticipated a large set of new applications which they termed the "New Product Line" [58, p. 552]. In the same manner with which separate hardware lines for scientific and commercial processors were to be combined into one single line, IBM determined that the needs of many system administrators and programmers could also benefit from having a single programming language. Even with a single hardware product line, however, the boundary between FORTRAN, COBOL and JOVIAL programmers was becoming blurred within many organizations using IBM equipment, and a consequence was that one single mainframe installation would need several operating systems and compilers. If a single language could support the needs of the three different programming communities within the same company, then the benefits of a single hardware architecture for that company would be significant, perhaps even dramatic.

It was in this context that PL/I was first proposed in 1963 (at the time called the

```
...
ON zerodivide, underflow BEGIN;
    X = 0;
    GOTO Continue;
    END;
    ...
    ... some program path in which "Z = 0" could appear ...
    ...
X = Y / Z;
Continue:;
    ...
REVERT zerodivide, underflow;
```

Figure 2.1. *ON condition in PL/I*

"New Programming Language") [58, p. 556]. From the initial explorations of language definition through to the first implementation out of England, both programming practitioners and programming language theorists were involved in crafting the language. One of the important tasks was that PL/I serve as a systems programming language—*i.e.,* languages used to implement systems software such as compilers, device drivers, operating system kernels, operating system services, etc. A major challenge in developing such software is ensuring the programs are reliable and safe: cases include correctly dealing with I/O events (end-of-file, undefined file), arithmetic events (overflow, underflow, division-by-zero), and other events either generated by the underlying hardware or generated programmatically [58, p. 565]. The language designers enumerated nearly two dozen different "conditions" and introduced a new kind of executable statement named the "ON condition" which associated a block of code (the *action*) which would be executed given the occurrence of the condition.[1] For example, Figure 2.1 contains an example involving arithmetic conditions.

When control flow reaches and executes the `ON` statement, the code between the `BEGIN` and `END` (otherwise called the *action*) is associated with the two conditions (*zerodivide* and *underflow*). From this program point onwards, any occurrence of the conditions will immediately transfer control to the statement `X = 0;`. When control flow reaches and executes the `REVERT` statement, the association between the conditions and the action block is broken, such that the previous association (if any)

[1]The conditions were: `area`, `attention`, `condition`, `conversion`, `endfile`, `endpage`, `error`, `finish`, `fixedoverflow`, `invalidop`, `key`, `name`, `overflow`, `record`, `size`, `storage`, `stringrange`, `subscriptrange`, `transmit`, `undefinedfile`, `underflow`, `zerodivide`, and `anycondition`.

is restored [45].

The principal benefit provided by the powerful ON control structure is that infrequently executed code, such as error conditions, can be kept apart from "regular" code, *i.e.*, statements for normal processing. Regardless of the control-flow path from an ON statement to some other statement generating the condition, PL/I ensures the action associated with the condition is executed. This yields our first principle:

- **Principle 1**: *Exception handling decouples temporal properties of a program statement from the logical consequences of executing the program statement.*

Worded differently, exceptions in PL/I free the programmer from needing to specify the details of precisely *when* a condition is true, and instead allows the programmer to focus on the details of how the program *must* behave if that condition *is* true.

There was resistance to the adoption of ON-conditions. Their power, when combined with other features in PL/I (such as label variables for GOTO statements and asynchronous executing tasks), resulted in (a) programs which were hard to undestand and (b) compilers which were hard to write. (To be fair, the team which set down the main features of PL/I were working under tremendous time pressures—the initial design took place between October 1963 and February 1964 during meetings of the same group, with that group consisting of members working on the West and East Coasts of the U.S. [58, p. 560].)

As an example of some of the complexity, `ON` statements are executed at run time, and are not associated with any lexical scope, *i.e.*, the actions invoked at some program point as the result of a condition are determined by which ON condition was last invoked on a control-flow path to that program point. In the example of Figure 2.1, nothing prevents another ON statement from appearing between that in the figure and the assignment to X. This *dynamic binding* of action blocks to conditions yielded programs that were hard to undestand. Another difficulty was in the use of *GOTO* statements to transfer control from the end of the action back to normal control flow; the undisciplined use of GOTOs combined with dynamic binding led many to dismiss the merits of *ON* conditions. One group which did *not* dismiss them was the group of operating system researchers spearheading the MULTICS project [56]—a variant of PL/I called EPL was developed in early 1965 (it did not have complex data types, aggregate exceptions, or I/O) [58, p. 561]. Nearly all of

the MULTICS operating system was written using mostly EPL. The success in the use of a high-level language for writing such a complex piece of systems software was an influence in the creation of C during development of early versions of UNIX [59]; C uses a simpler *signal* mechanism than ON-conditions (of which more is written below).

2.3 Goodenough's proposals

One outcome of the introduction of ON-expressions was the proliferation of various new mechanisms in different languages produced by different organizations. The range of power amongst these mechanisms—and the difficulty in translating programmatic ideas between them—led John Goodenough to propose a new notation for exception-handling concepts. These are contained in his seminal 1975 CACM paper, "Exception Handling: Issues and a Proposed Notation" [27]. His contributions were:

- A notation for associating handlers with operations, especially dealing with the case of detecting programmer errors in the use of exceptions by reporting compile-time errors;
- Clear representations of the control-flow produced by an exception, *i.e.*, whether or not control-flow returns to the program point causing the exception-handled condition;
- A notion of *default exception handlers* for conditions occurring at some program point but for which no handler is specified; and
- A notation for specifying hierarchies of exceptions given the relationship between the invoker of some code and that code itself, particular when both pieces of code are associated with handlers.

Of the four contributions, the most influential has been his suggestion for the transfer to and from normal control flow in the context of an exception, namely *EXIT*, *RESUME* and *ENDED* semantics. Most extant exception handling mechanisms left these details to the programmer in the form of GOTO statements (as seen above in the previous section on PL/I); the consequence was that control-flow transfer semantics could not easily be determined from reading code. Goodenough's suggested commands are:

```
sum = 0;
try {
    scan (p, v);
}
catch (Value v) {
    sum = sum + v.data;
    resume;
}
```

Figure 2.2. *Goodenough's example of RESUME (caller)*

```
void scan (SomePointer p, Value v) throws Value
{
    ValuePointer  *a = (somecast) p;
    Value         v;
    int          i;
    for ( int i=0; i<100; i++ ) {
       v.data = (anothercast) a[i];
       throw (v);
    }
    return;
}
```

Figure 2.3. *Goodenough's example of RESUME (callee)*

- *EXIT*: to be used to terminate the operation raising the exception (*i.e.*, control-flow permanently leaves the handler);
- *RESUME*: to be used to force control to return to the program point immediately following the operation raising the exception; and
- *ENDED*: to specify a syntactic unit always executed upon the termination of a code block in which an exception my be raised.[2]

One intriguing suggestion made in the paper was with respect to the *RESUME* command. Consider a structure which is traversed via a function `scan(p, v)`, where `p` is a pointer to the data structure and `v` is some value in it. The code in Figure 2.2 computes the sum of all elements; code for `SCAN()` itself appears in Figure 2.3. A C++-ish notation is used even though C++ semantics are different.

As values in the structure referenced by `p` are found, the values are thrown back to the handler surrounding the call to `scan`; the handler will be invoked for all 100

[2]In Java parlance, *EXIT* is the built-in semantics for Java exceptions, while *ENDED* is represented by a *FINALLY* clause. There is no notion of *RESUME* in Java.

items. When all items are summed together, `scan` is terminated normally and code proceeds to the normal-flow program point following the invocation of `scan`. A similar suggestion is made by Levin in his Ph.D. dissertation [41] with the observation that this program organization is sensible when the data structure imposes a high initial cost to lookup into that structure, *i.e.*, the cost of the first lookup can be amortized across the remaining lookups. Such use of exceptions and *RESUME* semantics has a similar flavour to co-routines. This yields our second principle:

- **Principle 2:** *Exception handling can be used to structure programs in the absence of errors.*

Many other ideas and code examples are provided throughout Goodenough's paper. Even today it should be required reading for any programming language designer or implementor.

2.4 Mesa

Mesa was designed, developed and used at the Xerox Palo Alto Research Centre around 1974 [26]. Amongst other concepts, it attempted to put into practice some of the latest research into structured programming techniques (modules, information hiding, data-structuring facilities, typing). The compiler was especially designed to perform a large amount of compile-time checking, the purpose ultimately being to improve the process of programming. Unlike languages such as Pascal, however, which also implemented these research ideas but for a teaching environment, Mesa was meant to support systems programming at Xerox (PARC). It was used to program the first Alto workstation—its user interface inspired the team at Apple who created the Lisa (and from there the first Macintosh), and subsequently influenced Microsoft and their Windows interface.

The following idea implemented in Mesa stems from Goodenough's paper: Handlers for exceptions (or "signals" in Mesa terminology) were specified by *catch phrases*, with each catch phrase returning one of three values [50, p. 140]:

- *Reject*: Normal flow of control begins at the first statement following the end of the handler;

- *Unwind*: Used to propagate the exception up the chain of calls, and notifies the run-time that one or more activation frames are about to be deallocated, hence triggering some cleanup; and
- *Resume*: Returns values from the catch phrase to the routine which generated the signal as if the exception was a procedure call used to return some results.

There is little published literature on the experiences using exceptions in Mesa at Xerox PARC, specifically those involving the use of *Resume* in catch phrases. However, what exists is fascinating.

One item comes from Stroustrup's "Design and Evolution of C++" [65] in which he describes the years of discussions over whether C++ exceptions should have termination semantics or resumption semantics or both. There were some advocates of using a keyword like `resume`, but they were greatly outnumbered by the group advocating the banishment of `resume`. Anecdotal evidence in support of resumption semantics came out of the OS/2 programming community, but unfortunately nothing more concrete was ever presented. The final decision to include only termination semantics was the result of a presentation by a former user of Mesa, who stated that:

> "... termination is preferred over resumption; this is not a matter of opinion but a matter of years of experience. Resumption is seductive, but not valid." [65, p. 392]

The speaker (Jim Mitchell) was one of the main designers and implementors of a flavour of Mesa called "Cedar", and in every case involving `resume`, cleaner and faster code was the result of removing it. Stroustrup summed up the presentation by writing that "every use of resumption had represented a failure to keep separate levels of abstraction disjoint."[65, p. 392]

The other item is a reference to the Mesa language manual and its suggestions for dealing with "signals calling signals"—*i.e.,* recursive exception handlers. Resumption-style semantics, despite their power, are confusing and error prone in this particular situation. Buhr & Mok explore this further in their survey paper [8]. Figure 2.4 shows their code based on their example and Figure 2.5 several different snapshots of the "call stack" as computation proceeds. A Java-like notation is used along with the keyword `resume` (which does not appear in the Java language). The confusing semantics of resumption results from the mix of lexical and dynamic scoping used

```
void process (void)
{
      ...
      try {
            try {
                  try {                                    P
                        resume new R1();
                  }
                  catch (R2 r2) {                          3
                        // Handler 3
                        resume new R1();
                  }
            }
            catch (R1 r1) {                                2
                  // Handler 2
                  resume new R2();
            }
      }
      catch (R2 r2) {                                      1
            // Handler 1
            // Code 1
      }
      // Code 2
}
```

Figure 2.4. *Mesa termination semantics: example code*

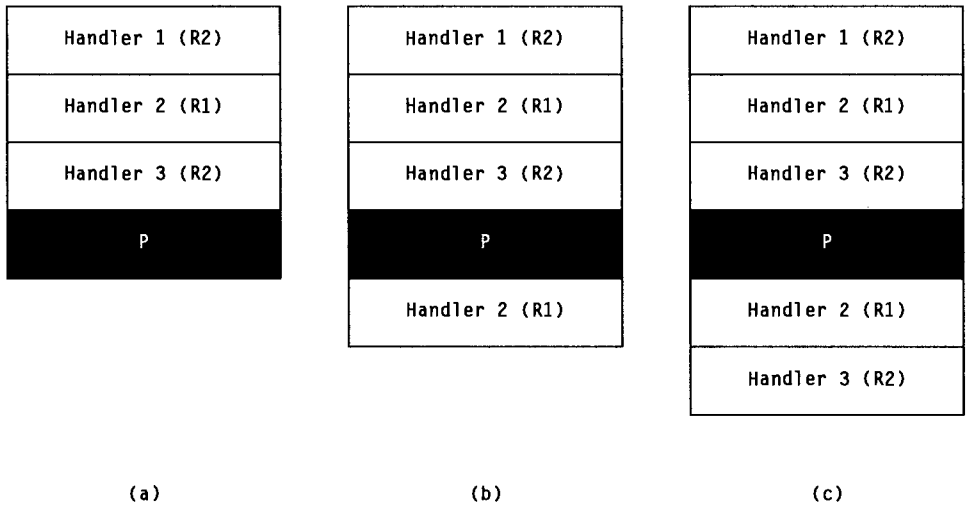


Figure 2.5. *Mesa termination semantics: catch frames for example code*

by Mesa. Once a `resume` instruction is completed, control-flow continues from the instruction following the resume (in this case, that at `// Code 2`, but to accomplish it an activation frame corresponding to the `resume` instruction is kept (that for code `P`). The nesting of exception handlers at the point in time just before `P` is executed is shown in Figure 2.5 (a). `P` then causes an instance of exception `R1` to be thrown, and in this case the lexical scope and dynamic scope produce the same handler as a result—this being the code at `2`. Transferring control to the handler does not discard the activation frame for `P`, however, as it is needed for the resume; instead the system adds an activation frame for the code in `2` as seen in Figure 2.5 (b). Code in `2`, in turn, throws a new exception of type `R2`, but control is not transferred to `1` as might be expected with local scoping rules. Mesa uses dynamic scoping in the presence of `resume`; the handler invoked for this new `R2` is that closest to it in the activation frame, in this case that of `3`—Figure 2.5 (c). Now an infinite loop has been established.

The designers of Mesa solved the "infinite loop" problem by using a concept that Buhr & Mok call an "unhandled handler." Key to the solution is that Mesa exceptions are caught and dispatched to the proper handler by an instance of the subroutine `Signaller`. The solution described here occurs at runtime when a later instance of `Signaller` on the call stack, called `Signaller'`, walks up the call chain only to find some earlier instance of `Signaller`. If this earlier instance of `Signaller` involves a different exception type than `Signaller'`, the signal is propagated right through the earlier `Signaller`. However, if both instances handle the same exception type, the Mesa runtime performs the following two steps:

- `Signaller'` copies some state variables from the earlier `Signaller`, and then
- the modified `Signaller'` skips over the frame *containing the catch phrase for* `Signaller` *after which it continues to propagate the signal.* [50, p. 142]

This means that when the code in `3` is executed as would be the case in Figure 2.5 at the bottom-most activation frame, Mesa dictates that control would not be transferred to *any* frame currently on the stack as `Signaller` would begin looking for a handler from the topmost activation frame, *i.e.,* the earlier `Signaller` was at `2`, so skipping over this frame yields the frame for `1`, or the topmost frame in (a), (b) and (c). These semantics are confusing because sometimes there is a handler for the thrown `R1` in

code 3, and sometimes there isn't one—and this unclear from reading the code. All of this yields our next principle:

- **Principle 3:** Exception handling with termination semantics is less powerful, but easier to understand, then exception handling with resumption semantics.

2.5 CLU and Levin

Many of the languages designed and implemented in the 1970s explored support for structured programming. Barbara Liskov and others at the Massachusetts Institute of Technology were part of this trend and produced the language `CLU`. `CLU`'s version of exception handling was influenced by Goodenough's proposals, but its design was also influenced by the importance attached in ensuring code robustness and readability [43]. CLU used the static binding of exception handlers as was originally intended for Mesa (*i.e.,* the local handler for an exception thrown as some program point can be determined by examining the lexical scope of the `try`-block). However, the language designers introduced two additional restrictions:

- The exception mechanism was limited to one level of propagation.
- Only termination semantics were supported.

Both of these were intended to improve readability of code.

Limit to one level of propagation: CLU's syntax for subroutine declarations requires that the programmer specify all exceptions propagated out (or "thrown out") of a subroutine. (Java's requirement for specifying exceptions thrown out of a method in the method signature is influenced similarly.) The reason is to improve code comprehension: the exceptions specified to be propagated out of the procedure must be known from reading the text of the program, not by simulating code execution. An important consequence is that an exception of type `q` thrown in subprogram `Q()`—itself called by subprogram `P()`—cannot be propagated out of `P()` unless `P()` specifies that instances of `q` may be propagated out. Another consequence in this case is that `q` must be re-thrown by `P()`—exceptions thrown by `Q()`, itemized by `Q()`'s header, must be caught in `P()`, and therefore cannot propagate out without `P()`'s involvement. `CLU` ensures details of `Q()`'s implementation (such as the exceptions it throws) cannot be seen by users of `P()`.

Only termination semantics: The authors of [43] wrote that:

> The resumption model is more complex than the termination model ... The termination model requires a simpler linguistic mechanism for its support than does the resumption model ... We conjecture that the expressive power of termination semantics is adequate: that situations handled awkwardly by the termination model and simply by the resumption model are not frequent.

The "situations" to which they refer include several such suggestions in Ron Levin's doctoral research on an exception handling mechanism for a multiprocessor operating system (Hydra). The code in Figure 2.6 contains one of Levin's ideas for handling low-memory situations [41, p. 131]: it shows that a complex data structure supports a fast form of insertion which consumes a relatively large amount of memory, but the structure may be "compacted" into a representation using less memory. The compaction is invoked when necessary—in this case, when `allocate()` (or one of its called procedures) throws an instance of `LowPoolException`. The `resume` keyword in the handler transfers control back to the instruction which threw the exception; when this occurs, the allocator (which is using the memory pool) can continue its work.

Liskov and Snyder suggest that the above problem can be solved in CLU by passing a procedure to the `allocate()` function which would be invoked when compaction is necessary. They also remark that their solution may be more error-prone than Levin's, but come close to special pleading when they doubt the existence of a large number of such cases justifying resumption semantics, *i.e.,* widespread availability of the mechanism might yield more cases justifying resumption semantics. This suggests the following:

- **Observation**: *Opinions on the merits of exception handling mechanisms are often the outcome of software engineering concerns or programmer temperament or both.*

CLU has had a big influence on the design and implementation of imperative programming languages. It is significant that Java's exception handling mechanism does not keep the "single-level" propagation model, but does require the `throws` clause for methods from which are propagated "checked" exceptions. Restrictions to CLU's

```
// Assumption: list is some global variable 'm' that is
// accessed via the method add_to(); the memory 'pool'
// is managed by the in allocate().

void hairy_list_inserter (InfoType info)
{
    try {
        HairyNode hn1 = allocate (pool, SIZEOFNODE);
        HairyNode hn2 = allocate (pool, SIZEOFNODE);

        // fill in hn1 and hn2 using data within info

        try {
            add_to (hn1);
            add_to (hn2);
        }
        catch (LowPoolException lpe) {
            // Compaction inhibited during addition
            // to structure.
        }
    }
    catch (LowPoolException lpe) {
        // Perform data-dependent compaction of 'm',
        // using 'release (p, d)' to release into the pool
        // removed by compacting 'm'.
        resume;
    }
}
```

Figure 2.6. *Resumption semantics: Managing a memory pool*

EH mechanism also led to an efficient implementation [5, p. 489], and its creators refused to treat exceptions as errors—they even suggest that with an efficient enough implementation, exceptions could be used to convey information about normal and usual situations [43, p. 547]. This, of course, is the central theme motivating the new work presented later in this thesis.

2.6 Ada

The original version of the Ada programming language ("Ada 83") was designed in a time of great confidence in the ability of languages to solve difficult problems in software development. Two of these problems were (1) achieving fault tolerance and (2) constructing software systems using machine-readable specifications [19]. The challenges in constructing fault-tolerant software are beyond the scope of this thesis—and the naïve attempts to obtain it solely via exception handling have been noted and critiqued ([12] in the context of C++). The second problem is the one that is the focus of this section.

Separating a module interface from its implementation was already supported by Modula-2, a language invented by Niklaus Wirth in 1980. What still remained, however, was the difficulty in determining how best to determine *what* should handle errors—the service detecting the error or the module using the service? One solution allows exceptions to decouple the responsibilities. Text from the Ada 95 language standard suggests this approach; the example involves the interaction between a `File_System` package and a `Main` procedure which uses the former's services:

> In the ... example, the `File_System` package contains information about detecting certain exceptional situations, but it does not specify how to handle those situations. Procedure `Main` specifies how to handle them; other clients of `File_System` might have different handlers, even though the exceptional situations arise from the same basic causes. [51, p. 204]

The module providing the services *detects* the error, and throws an exception. The module using the services *catches* the exception and determines what should be done with the error. From this follows our next principle:

- **Principle 4**: *Exceptions can be used to provide a clear separation between policy*

and mechanism.

The Ada 95 specification and its Annex describes a set of built-in exceptions, including:[3]

- `Constraint_error`—thrown for a wide range of situations, such as when an assignment attempts to store a value outside the range declared for the variable, *e.g.,* a negative number into a variable of type `Positive`.
- `Storage_error`—thrown when a heap memory allocator attempts to access more memory than is available.
- `Program_error`—thrown when a particular Ada package is not yet properly initialized (or *elaborated*) before its first use.

Ada also supports user-defined exceptions, although these are actually variables declared within packages that have type `exception`. An example from [61] appears in Figure 2.7. Exception handlers may be attached to any `begin` ... `end` code sequence, with handler clauses indicated using the `when` reserved word.

Ada supports only termination semantics, but there is an Ada idiom which appears to accomplish exactly what resumption semantics makes possible. The one `Get` is protected by the exception handlers; as long as `Get` fails as a result of a thrown exception, the loop will ensure `Get` is retried. If `Get` succeeds, then the call to `exit` terminates the loop and code continues. (Strictly speaking this example illustrates a flavour of resumption called *retry* semantics.)

2.7 Eiffel

An intriguing but constrained form of exception handling was introduced into `Eiffel`, a language designed by Bertrand Meyer. The core idea of this language is the support of *contract-based* programming. Components are guaranteed that their users will provide data according to a contract published by the service; conversely the users

[3]The Ada 95 specification includes two sections, 11.5 and 11.6, addressed to developers of Ada compilers and runtime systems. The first section specifies that some of the checks yielding an exception on failure can be suppressed, albeit with the resulting code having undefined behavior. The second section provides some opportunities for compiler optimizations, of which more is written in the next chapter [51].

```
package Dir is
    type Directory is private;
    Directory_Not_Empty : exception;
    procedure Delete_Directory (D : Directory);
    function Is_Empty (D : Directory) return Boolean;
end Dir;

package body Dir is
    ...
    procedure Delete_Directory (D : Directory) is
    begin
        if Is_Empty(D) then
            -- delete directory D
        else
            raise Directory_Not_Empty;
        end if;
    end Delete_Directory;
end Dir;
```

Figure 2.7. *Example of Ada's user-defined exceptions*

```
loop
    begin
        Get(...);
        exit;
    exception
        when ...
        when ...
    end;
end loop;
```

Figure 2.8. *A resumption-like Ada code idiom*

```
try_once_or_twice is
    -- Solve problem using method 1 or, if unsuccessful, method 2
    local
        already_tried : BOOLEAN
    do
        if not already_tried then
            method_1
        else
            method_2
        end
    rescue
        if not already_tried then
            already_tried := true;
            retry;
        end
    end -- try_once_or_twice
```

Figure 2.9. *Eiffel's "rescue" and "organized panic"*

of the component are assured that data will be produced according to the contract. The former is simply a pre-condition, the latter a post condition.

One of Meyer's novel ideas was to combine the notion of pre- and post-conditions with exceptions. Any condition which prevents a component from completing its task is an "exception" [47, p. 249]. He describes three basic classes of exceptions:

- *False alarms*: Exceptions raised by the environment but not preventing completion of the task.
- *Resumption*: Exception is anticipated as a possibility, and an alternative way is found to fulfill the contract, *i.e.,* execution will try the alternative method.
- *Organized panic*: Similar to termination semantics, but where there is no way in which the contract can be fulfilled, and the client of the component is signalled.

This suggests yet another principle associated with the use of exceptions:

- **Principle 5***: Exceptions can represent the use of executable specifications.*

An example of an Eiffel "feature" using both "rescue" and "organized panic" appears in Figure 2.9. The default value of boolean variables in Eiffel is `false`, so this code will first attempt `method_1`. If it fails as a result of an exception, the `rescue` clause ensures `method_2` is executed when the feature it itself re-attempted. A failure in `method_2` will still transfer control to the `rescue` clause, but since the

`else` portion is empty, control leaves the feature and the exception is raised in the caller of `try_once_or_twice`.

Meyer was quite determined to support contract-based programming, and hence he constrained the scope of exceptions. For instance, the `retry` reserved word causes the whole feature to be re-executed, *i.e.,* rescue clauses may be only associate with whole procedures and functions. This approach—of limiting the scope or dynamic semantics or both of exceptions—has been proposed in many places, with one of the most notable being Knudsen's proposal for static exceptions [34].

2.8 C++ and Java

The C programming language as developed at Bell Labs (and when as standardized by ANSI/ISO) supported one mechanism for "error" or "exceptional" conditions, albeit not using exceptions as understood. *Signals* were introduced to "provide facilities for handling exceptional conditions that arise during execution, such as an interrupt signal from an external source or an error in execution" [33, p. 255]. Events such as a division by zero or illegal access to storage (*i.e.,* accesses deemed illegal by the run-time organization) are "signalled" by the operating system. A signal is "delivered" to the correct process, and if the programmer has specified a "signal-handler" function for the signal, control in the process / program is transferred to the signal. If control flow leaves the handler function normally then control is transferred to the point in the process where execution was interrupted. (The other technique used for handling errors in C is *status variables.*)

Using signals to implement exception handling is similar to exceptions in PL/I: a signal handler is programatically associated with a particular signal by executing the `signal` function.[4] Like PL/I, therefore, understanding the mapping of handlers to signals depends upon knowing the control flow through the program. C signals do differ from PL/I signals in that handlers must be reassociated with a signal after that signal occurs.

One proposal for adding exceptions to C is "Exceptional C", an extension to C

[4]Given that the creators of UNIX and C, Ken Thompson and Dennis Ritchie, participated in the Multics project, which itself used PL/I as an implementation language, the similarity between signals and `ON`-conditions seems natural.

```
int NoReply = 1;
{
    SendMessage (RequestMsg) ;
    while (NoReply) {
        ReceiveMessage (& ReplyMsg, 10) ; /*10 second timeout*/
        if (InReplyTo (ReplyMsg) == Message Id (RequestMsg))
            NoReply = 0;
        else
            printf ("Bad reply received\n") ;
    }
    except {
        when TIMEOUT: printf ("timeout\n") ;
            retry(5);
            printf ("Retry count expired\n") ;
            raise RPC_FAILED;
        when BADSEND: printf (" Bad send\n") ;
            raise RPC_FAILED;
        when BADRECEIVE: printf ("Bad receive\n") ;
            raise RPC_FAILED;
    }
}
```

Figure 2.10. *Exceptions in "Exceptional C" (Example 1)*

suggested by Gehani in 1992 [25]. Exceptions are introduced into C and subsume signals: regular exceptions are written with termination semantics, while exception-signals use resumption semantics. Keywords such as `raise`, `retry` and `resume` are used to indicate the desired semantics for the exception handler (with `resume` only available for "signal exceptions"). Figure 2.10 gives an example of the syntax used by Gehani. The scope of the exception handlers is the block in which the `except` clause is nested, and here would correspond to the code within the outermost braces.

At about the same time in the early 1990s, Stroustrup and Koenig proposed an exception handling mechanism for the then new C++ language [36]. Stroustrup describes the ANSI standard's committee deliberations over the design of EH in C++ [65], and this short chapter makes for fascinating reading. Of interest to us here, however, is the committee's idea of combining exceptions with the object-oriented notion of inheritance. In all of the other languages we've examined so far, exceptions exist in a hierarchy with two levels—the lower level contains all of the exceptions, and the highest is simply "all" exceptions (for use when specifying a "catch-all"

```
class Netfile_err : public Network_err, public File_system_err
{
  ...
}

// Function dealing with network exceptions can catch Netfile_err
void f()
{
    try {
        // code
    }
    catch (Network_err& e) {
        ...
    }
}

// Function dealing with file system exceptions can also catch Netfile_err
void g()
{
    try {
        // more code
    }
    catch (File_system_err& e) {
        ...
    }
}
```

Figure 2.11. *Exceptions in "Exceptional C" (Example 2)*

handler).[5] When mixed with C++'s multiple-inheritance, exceptions may now be derived not only from one base class but from multiple classes (known as *composite exceptions*). The example shown in Figure 2.11 [66, p. 360] introduces an exception of class `Netfile_err`; it can be caught by functions containing a catch handler for `Network_err` or in functions containing a catch handler for `File_system_err`. In this particular example, the arrangement of the composite exception frees the developer of `g()` from needing to know that the file system is implemented over a system such as Sun's NFS—all that the developer wants to know is if an error was raised as a result of using the filesystem, regardless of its implementation.

There have been some criticisms made of C++'s exception handling mechanism, but these tend to reflect the interaction of EH with other system features (*e.g.*,

[5]This idea was not new at the time—Smalltalk-80 supports both inheritance and exceptions, and theoretically a programmer could specify a true hierarchy of exceptions [74].

throwing exceptions in destructors; interaction between exceptions and code used to support concurrency and resource locking). Stroustrup provides advice for many of the cases involving such interactions. Of interest to us, however, is that much thought has been put into creating a hierarchy of built-in exceptions [66, p. 385].

- **Principle 7**: *Exceptions can leverage the expressibility provided by inheritance.*

In a later chapter we explore some of the design patterns made possible by such hierarchies.

Java's exception handling mechanism is closely modelled on that of C++ (as is much of Java's syntax and semantics), with the exception that multiple inheritance is not supported and hence composite-type exceptions cannot be defined. Java has a hierarchy of built-in exception classes (as well as built-in error classes which are raised by the Java runtime) all rooted in the class `Throwable` [42, p. 38]. Programmers may define their own classes as a part of this hierarchy. Java exceptions are also split into two groups: `checked` and `unchecked`. *Checked* exceptions use an idea from CLU; these exceptions may not be propagated out of a method unless the method signature states this. (All user-defined exceptions belong to this group.) *Checking* refers to a compile-time analysis to ensure a method containing `throw` statements also specifies these exception classes in the method signature. (*Unchecked* exceptions such as those for array bounds checks and null pointers need not be mentioned in a method signature.) In essence, this analysis is similar to type-checking. In later chapters we discuss more aspects of Java's exception handling semantics.

2.9 Data-Oriented Exceptions

The range and complexity of exception handling mechanisms—both proposed and implemented—are large, and as stated at the beginning of this chapter, an exhaustive listing is beyond the scope of this thesis. However, amongst all of these is one very interesting proposal which came out of the Ada programming community. *Data-oriented exceptions* are an attempt to address the impact of readability on normal code by the use of `try-catch` blocks [20]. As an example, consider the code in Figure 2.12 (adapted from Cui and Gannon by rewriting it in a Java-like code style using a syntax for parameterized types proposed by Sun). There are two instances of

the generic class `Stack`, where one object (`S1`) grows as new values are added, and the other object (`S2`) keeps 90% of the old values plus the new value pushed onto the stack. Despite the intent of exceptions as a mechanism for moving "error" related code outside of the normal-line code, this example demonstrates how readability is decreased. From the structure of the code it is not clear that the *only* difference between `S1` and `S2` is the handling of `MyStackOverflow`.

The idea behind a data-oriented exception is to move the handler specifications into the elaboration of the generic type (in this case instances of the generic class `MyStack`). The code of Figure 2.13 displays a possible syntax for the elaboration of a generic type with exception handlers specified. An assumption here is that handlers associated with instances of generic types have *retry* semantics (*i.e.*, the exception handler is executed and code retries the method raising the exception). The benefit is that normal code is now clearly shown in the method `process()`, and the difference between the two instances of `MyStack` are clearly shown during their elaboration.

- **Principle 8**: *Exception mechanisms must aid code understanding while also encouraging code reuse.*

Besides Ada, I am aware of no other languages for which data-oriented exceptions have been implemented. Cui and Gannon also devote part of their paper to an analysis of typical exception usage seen in a library of Ada programs.

2.10 What is an exception?

Systems programmers tend to be idiosyncratic, and in almost every environment wish to complete tasks "their way", and this was suggested by our Principle 4. The same holds true for their conception of typical programming tasks, problems and solutions. Some have temperaments which find exceptions to be odious and a "hack": they view errors as system state which must be explicitly examined. Others find exceptions appealing as a program structuring mechanism: exceptions clarify the code by literally keeping error concerns separate from normal code. In both cases, however, the view is that exceptions help deal with "bad" code, and this itself is derived from the meaning of "exception" when describing computer architectures.

An alternate view of an exception is based on the dictionary definition of "excep-

```
import java.util.Stack;

public class DOE_Example {
    ...
    MyStack<Integer> S1 = new MyStack<Integer>();
    MyStack<Integer> S2 = new MyStack<Integer>();

    void process (void)
    {
        Integer I = new Integer();

        // Some code assigning a value to I.
        // ...
        try {
            while (/* some condition */) {
                try {
                    S1.push (I);
                }
                catch (MyStackOverflow e) {
                    S1.expand (growth_rate);
                    S1.push (I);
                }

                try {
                    S2.push (I);
                }
                catch (MyStackOverflow e) {
                    S2.retain (90);
                    S2.push (I);
                }
            }
        }
        catch (MyStackNoSpace me) {
            error ("PUSH");
        }
    }
}
```

Figure 2.12. *Motivating example for Data-Oriented Exceptions*

```
import java.util.Stack;

public class DOE_Example2 {
    ...
    MyStack<Integer> S1 = new MyStack<Integer>() with
        handler (MyStackOverflow e)
        {
            expand (growth_rate);
        };
    MyStack<Integer> S2 = new MyStack<Integer>() with
        handler (MyStackOverflow e)
        {
            retain (90);
        };

    void process (void)
    {
        Integer I = new Integer();

        // Some code assigning a value to I.
        // ...
        try {
            S1.push (I);
            S2.push (I);
        }
        catch (MyStackNoSpace me) {
            error ("PUSH");
        }
    }
}
```

Figure 2.13. *Proposed Data-Oriented Exception syntax for Java generics*

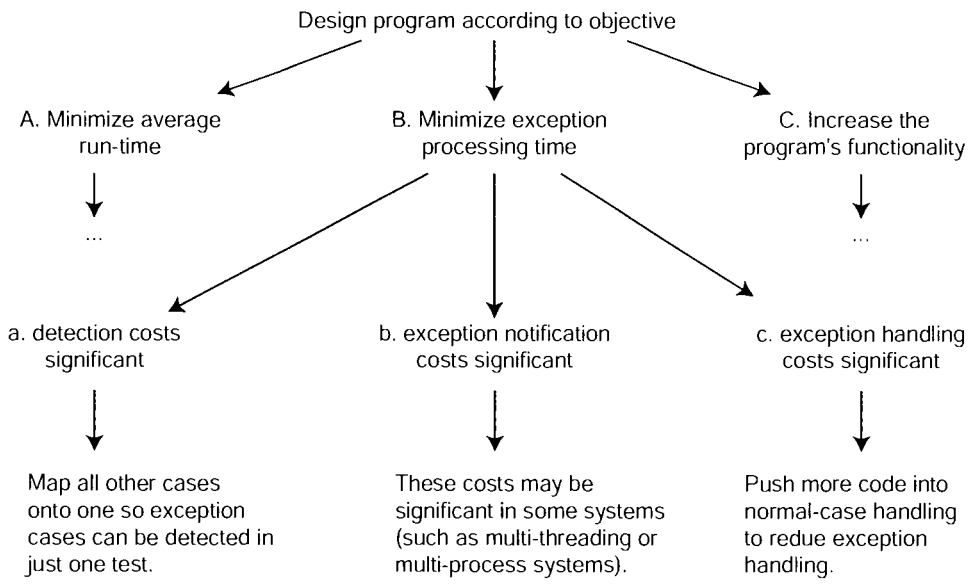


Figure 2.14. *Models for minimizing exception processing time*

tion", or as Cheriton writes in [14]:

> In contrast to both computing and the real world, the dictionary takes the ambivalent view that an exception is simply an unusual event, not necessarily a failure or an unexpected event. Thus, one might simply designate some probability P and declare that any event with probability of occurrence less than P to be an exception.

The above definition does not treat exceptions as rare events for which little consideration is given by way of performance. Instead it describes a spectrum of design and use techniques, and even suggests alternatives to the implementation of exceptions. Given that most new compiler optimization techniques are based on profile-based feedback, probabilities of occurrence of code can now be either estimated or obtained at run-time and used to guide optimization of exception handling.

Even without such optimizations, there are principles which can be applied in which exception mechanisms are used to improve performance. Figure 2.14 contains a diagram base on those in Atkins' dissertation [3, p. 55, 63], a work which describes models for system design which involve exceptions as defined above.

In the rest of this dissertation we accept Cheriton's definition of an exception, but with the realization that one class of system programmers may still find the idea

unappealing. Our hope is that we can convince some members of this group to view exceptions in a new and more constructive light, but part of this task involves improving the performance of exception handling mechanisms. To do this, we must next examine several different implementation techniques, and from this learn why many current implementations perform so poorly. We also examine compiler optimizations that have attempted to improve this performance, some of which do so as a result of solving a different problem.

Chapter 3

EH Implementations and Optimizations

3.1 Overview

3.1.1 Implementation framework

Earlier we referred to the *throwing* or *raising* of exceptions and the *catching* or *handling* of them. The previous chapter described various semantics for exceptions, and any implementation must support the basic algorithm given in Figure 3.1 when an exception is thrown, either explicitly or implicitly, by a program statement.

Of course, the devil is in the details (*i.e.,* what is meant by *context*? what is meant by *exiting*? what happens when a handler itself throws an exception?) but our concern here is with general strategies for implementation. We will eventually focus on a specific implementation from amongst those presented here.

As an example, consider the code in Figure 3.2 which uses Java syntax: Let us assume that control enters `some_function()`, executes the code at line 11 (*i.e.,* a call to `massage_object()`). As a result of processing earlier in the called function, line 41 is executed. Given our EH algorithm, at this point step A is performed. The *context* in this case is the *static scope* surrounding line 41: there exists no *catch block* or *local handler* for the exception. Therefore we skip step B and proceed to C. As the current context also happens to be the scope of the procedure `massage_object()`, *exiting the context* consists of transferring control to the program point calling the function. (If termination semantics are used, the activation record for `massage_object()` is also discarded.)

A. Does a handler local to the current statement's *context* exist for the exception?

B. If A is true, then execute the handler and transfer control to the first statement immediately following the current context. At this point normal processing is resumed, and this algorithm is exited. Given termination semantics we can assume here that a compiler or interpreter would emit, after the last instruction in a handler, a `jump` to the first instruction after the `try`-block boundary. Therefore some of the description in this step may seem redundant.

C. If A is false, and nesting contexts exist, then exit the current context and go to step A, otherwise proceed to D.

D. No further contexts exist, so invoke the default handler for the exception (if one exists).

Figure 3.1. *Algorithm EH1*

Processing of the exception now returns to step A at line 11. Here the context is a bit more complex as we are within a `try-catch-finally` block. There exists in this context a handler for exception type `A`, but none for `B`. Step C requires that we exit the current context: this means code in the `finally` block is executed (line 20) and control is transferred to the immediately enclosing context. This latter context happens to be the `try` block starting at line 7 and includes a handler starting at line 25 for exceptions of type `B`. (Exception types `A` and `B` are not related in the class hierarchy.)

Processing now returns to step A, and as there happens to be a handler for the thrown exception in this context, we proceed to step B. The code within the handler (consisting of the single line at 25) is executed. The statement immediately following the context is at line 28, and normal processing resumes at this point.

Much of this chapter is spent exploring the major implementation techniques for this EH algorithm.

```
1   int some_function (int i, Object o, StatusObj s) {
        int retval = 0;

        if (process_object(o) > i) {
5          retval = 1;
        } else {
            try {
                s.init();
                try {
10                  open_entity(i, o);
                    retval = massage_object(o);
                    s.log_message (Integer.toString(i));
                }
                catch (A a) {
15                  s.some_field = 2;
                    s.log_exception(b);
                    retval = 3;
                }
                finally {
20                  close_entity(i, o);
                }
                s.complete();
            }
            catch (B b) {
25              s.log_exception(b);
            }
        }
        s.some_field = retval;
        return retval;
30  }

    int massage_object (Object o) throws A, B {
        int s;

35      // Some code for manipulating input parameter
        //
        ...

        switch (s) {
40      case 0: throw new A;
        case 1: throw new B;
        default: return s;
        }
    }
```

Figure 3.2. *Exceptions: Example A*

3.1.2 Extant optimizations

The set of optimizations specifically aimed at improving exception-handling performance is quite small. Most optimizations which refer to EH are actually attempts to expose more opportunities for standard optimizations, especially as they are often hidden due to the possibility of exceptions. For example, if too many edges are added to a control-flow graph because of EH, then various optimizations become difficult to perform. (Example: a *reaching definitions* analysis on the variable `retval` will be unable to determine if the definition at line 5 is live at line 27 because of the possibility that line 17 will be executed on every invocation of the method.) While the larger set of optimizations is important, we focus mostly on the smaller set as a way of setting the stage for our own contributions.

3.1.3 A general implementation principle

One of the goals of this thesis is to present techniques for improving the run-time performance of exceptions, *i.e.,* when an exception is thrown, its corresponding handler is found as quickly as possible, and control is transferred to the handler as quickly as possible. One of the reasons why this is challenging is because of the "user-pay" principle which has guided implementers over the past several decades. This principle asserts that an EH implementation *must not* incur a run-time cost to a running program which does not throw an exception. The program itself may contain possible control-flow paths where an exception will be thrown, but if the invocation of the program throws nothing, then the program should be no slower than if the compiler or interpreter were unaware of exceptions. There may be concerns about the potential for increasing the size of an executable as a result of EH, but this is not considered as big an issue.

We have brought attention to this principle now if only because there are certain schemes which come to mind that could result in much faster exception handling if only we could build certain data structures, record control data, etc. at execution time. The implementations presented in the following sections suggest such schemes to the reader, but we ask the reader to continue focusing on the "user-pay" principle as it provides motivation and context for the following chapters.

3.2 Simple implementations

Straightforward implementations of exception handling are similar to the ways in which programmers attempt to understand source code containing exceptions. For instance, the person reading the code notes that control flow now enters a `try`-block, and so the exceptions listed in the associated `catch`-phrases are noted. Similarly when code is traced to a point just after the end of the `try`-block, a note is made of the handlers no longer active. If the code trace within the block comes across a function call, the programmer notes from the start of that function which handlers are active from the point at which the call is made. If the function itself contains a `try`-block with a handler for an exception already on the programmer's "mental list", then this new handler is given priority. This simulation of the program's execution, with its addition and removal of handlers, suggests a possible implementation. What follows is a description of the technique as based on Christensen's thesis [16] (which provides details on the implementation of exceptions in a Pascal-like language called `MEHL`). While there is no single accepted name for this technique, some refer to it as "dynamic registration". The code in Figure 3.3 along with that in Figure 3.2 will be used as examples throughout the rest of this chapter.

The key data structure is a stack of exception handlers maintained by the run-time system. In this simple scheme, each "active" handler has a stack node associated with it. Each such node contains:

- information about the exception type,
- a pointer to the handler's code, and
- data needed to maintain the list.

Searching for an exception handler is now a linear search through the stack starting at its head. Each node may also contain some extra information such as:

- whether or not the node indicates the start of an activation frame context, or
- whether there is a `try`-block boundary, etc.

In Figure 3.4 are seen two stacks. The top shows the stack as it would appear just before the call to `s.init()` in line 8 of Figure 3.2. The bottom shows the result when control flow is at the point indicated by line 52 of Figure 3.3. Vertical dashed lines indicate a call boundary. A solid line corresponds to a `try`-block boundary. The

```
45  class StatusObj {
        // Various instance variables and method declaration
        // appear here
        // ...

50      void init (void) {
            // ...
            try {
               // ...
            }
55          catch (NullPointerException npe) {
               // ...
            }
            catch (SecurityException se) {
               // ...
60          }
            // ...
        }

        void log_message (String message) {
65          // ...
            try {
                // ...
            }
            catch (FileNotFoundException fnfe) {
70              // ...
            }
            catch (WriteAbortedException wae) {
                // ...
            }
75      }

    }
```

Figure 3.3. *Exceptions: Example B*

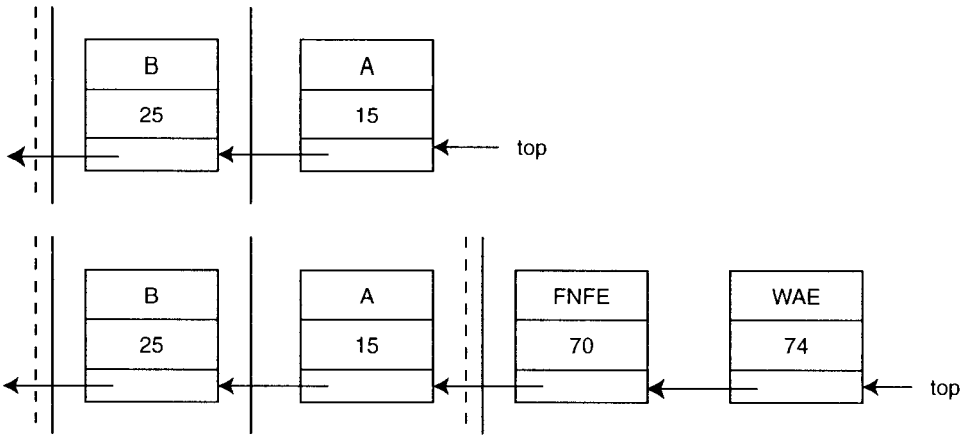


Figure 3.4. *Dynamic Registration: example*

line number corresponding to the start of a handler's code is used to represent the "pointer" mentioned in the list above. Nodes are added to and removed from the list as control-flow enters and leaves `try`-blocks.

The algorithm given in 3.1.1 can now be expressed in terms of this implementation scheme; the result can be seen in Figure 3.5. The type of the thrown exception is denoted as `T`.

Variants of this scheme exist, with one of the best-known used in the first C++ implementation supporting exceptions [36]. It took advantage of the fact that C++ code was converted into ANSI C, and used `setjmp` and `longjmp` as a way of capturing handler context. As noted by Chase in [13] this technique was also used by those developing portable implementations of Eiffel and Modula-3. This particular implementation of dynamic registration is called "threaded exception stacks". Lists must still be maintained, nodes must contain exception handling information, and a list must still be searched for a handler, but the non-local transfer of control from a throw site to its handler is now largely performed by `setjmp/longjmp`. Figure 3.6 describes code that would be emitted for a `try`-block. Note that the normal processing code (*i.e.*, that within the `try`-block) is nested within an `if` statement. Figure 3.7 describes code that would appear at a `throw` statement. Cameron *et al.* describe a portable implementation of C++ exception handling in their 1992 USENIX paper [10], in which they deal with many of the issues involving constructors and destructors, storage for thrown objects, and determining subtype relationships amongst exception classes.

A. While the `top` pointer is not at a call or `try`-block boundary, pop off nodes from the stack; if some node `N` amongst those popped has a type matching `T`, then proceed to step B, otherwise proceed to step C

B. Transfer flow-of-control to the statement indicated in node `N` and exit this algorithm.

C. As there is no exception handler, we must exit the current context.

- If the next boundary is that for a `try`-block, then there is no extra work and the algorithm proceeds to step A.
- If the next boundary is that for a call boundary, then the current activation record must be discarded, with any other processing resulting from this also performed, after which the algorithm proceeds to step A.
- If neither a `try`-block nor a call boundary exists, then there are no further contexts in which we can search for a handler. Proceed to step D.

D. Transfer flow-of-control to the default handler for exception type `T`, and exit this algorithm.

Figure 3.5. *Algorithm EH2—stack-based technique*

```
1   /*
     * Code needed at entry to a try-block; assume a new
     * scope is entered.
     */
5
    handler_info_t info;
    stack_entry_t  node;
    int            which_ex;

10  /*
     * Assign to "info" the correct info for handlers associated
     * with this try-block, and then initialize "node" to contain a
     * pointer to info.
     *
15   * Instances of "stack_entry_t" also contain a field named
     * "context" into which the a stack environment may be saved.
     *
     * "node" is added to the list of handlers.
     */
20
    which_ex = setjmp (node.context);

    if (which_ex == 0) {
        /*
25       * Code from within the try-block.
         */

        /*
         * If we arrived at this point, then execution
30       * was completed normally.  At this point there
         * is code to pop "node" off the list of handlers.
         */
    } else {
        /*
35       * Code corresponding to the handlers.  The
         * "switch" statement used "which_ex" to determine
         * which handler body is to be invoked.  Note that
         * if control arrives at this point, the handler
         * is guaranteed to be here.
40       */
         switch (which_ex) {
            ...
         }
    }
```

Figure 3.6. *Threaded-execution stacks—code for try-catch block (after Chase [13])*

```
45  /*
     * Code needed to properly dispatch an exception:  This
     * code would be called from the point at which a "throw"
     * appears.  The variable "ex" contains data describing
     * the thrown exception.
50   */

    stack_entry_t entry = top_of_exception_stack;
    int           match;

55  while ("entry" does not match handler needed for "ex")
       /* Advance to the next item in the list */

    if (nothing matched)
      abort();
60  else
      /*
       * Pop the topmost item off "top_of_exception_stack",
       * and perform a longjmp to the stack environment
       * held in "entry".
65     */
```

Figure 3.7. *Threaded-execution stacks—code for throw site (after Chase [13])*

These techniques are relatively easy to understand and implement, and for someone interested in improving the performance of exception handling they lend themselves to ideas. For instance, if a threaded-execution stack could be replaced by a different structure that supports very fast handler lookups, then immediately there is a gain that can be exploited. Regardless of how handler lookups are accelerated, however, any technique based on those in this section will punish executions of code fragments that contain handlers but which do *not* always have control transferred to them. For instance, consider the code in Figure 3.2 from lines 9 to 21. If this code fragment existed within a `for`-loop that iterates a thousand times, and if no exception was ever thrown for any iteration during a specific execution of the program, then the `try`-block entry and exit code would still have been executed a thousand times despite being unneeded (*i.e.,* no exceptions thrown). Given that compiler optimizations are often designed to squeeze several instructions out of a block of code, the insertion of a large number of these potentially wasteful instructions would reduce performance. Even keeping `try`-block entry-code to a minimum is considered "too much" by most compiler writers (see Section 3.1.3), who instead prefer implementations such as those

Range	Exception	Handler Start
10 – 12	A	15
8 – 22	B	25

Table 3.1. *Handler table for* `some_function()`

described in the next section.

3.3 "User-pay" techniques

Most modern EH models depend upon handlers having a lexical scope within a function, *i.e.,* every control-flow path to a particular statement results in the same set of active local handlers. In Figure 3.2, for instance, the local handlers at line 11 will always be that for exception `A` starting at line 15 and that for exception `B` starting at line 25. Someone reading the code for this function can determine immediately from the text which local handlers deal with either exception type, so there is no need to simulate code execution to identify handlers in this case. (We are referring here to local handlers only, however.) If some other exception type is raised as a result of executing line 11, and the exception is thrown up to this point, then someone reading the code must know what functions called `some_function()` to determine what handler from which function is invoked.

All of this suggests techniques for finding local handler that are *program-counter based.* In each procedure we can build a table having three columns: one column contains ranges of line numbers, another column contains exception types, and the third column has a list of starting lines for handlers. Each individual handler corresponds to a row in the table. An example table for `some_function()` appears in Table 3.1. Similar examples can be constructed for the functions in Figure 3.3. No special analysis is required at compile-time to construct such tables as the ranges can be directly determined from the program text. The technique also depends upon `try`-block ranges being contiguous and that nested handlers appear earlier in the search order applied to the table. The Java Virtual Machine specification uses this *exception table* or *range table* approach [42, p.112—113]—the table is stored as an array within

the `Code` attribute of a method.

These are *user-pay* techniques in that a code invocation which does *not* raise an exception will not incur a run-time penalty (albeit with a very small space penalty). Only code *using* the exception will cause an exception-table lookup and dispatch. Our algorithm is reworded in Figure 3.8 for this technique. The exception thrown is of type `T`, with tables satisfying the following:

- A row R_i in the table is denoted by tuple $((b_i, e_i), E_i, h_i)$ where:
  - b, e, and h are line numbers, and
  - E is from the set of exceptions types
- Row R_i appears before row R_j in the table if either of these conditions hold:
  - $(b_i = b_j \wedge e_i = e_j \wedge h_i < h_j)$ (multiple handlers for the same `try`-block)
  - $((b_i = b_j \wedge e_i < e_j) \vee (b_i > b_j \wedge e_i = e_j) \vee (b_i > b_j \wedge e_i < e_j)$ (nested handlers)

Note that there is no requirement that handlers for subtypes appear before handlers for supertypes.

Although this scheme's cost is nil for portions of a program which do not throw exceptions, it may be less than the *registration* scheme when exceptions are thrown. Examining nodes in an exception stack is not necessarily more expensive than performing table lookups. What is no longer possible, however, are the obvious techniques for reducing the cost of dispatching an instruction such as searching through a list of possible handlers. These handlers must now be discovered by explicitly unwinding the call stack; the cost of constructing a list of handlers "on the way down" to the throw site is not acceptable.

A slight variant of the "static table" approach differs in its placement of the table within the code. For example, if only *synchronous exceptions* are allowed (*i.e.*, exceptions occur either at a function callsite or with an explicit `throw` or `raise` statement) then handler info may take the form of *in-code markers* [13]. This variant is neither more nor less expensive at run-time than the "separate table" approach.

3.4 Other issues and techniques

While the literature on EH implementations is not large, what is described tends to include much detail. There are also a few other intriguing techniques which we

A. Using the value of the *program counter* (or PC) at the throwing instruction, find the first row R where $R = ((b, e), E, h)$ in the context's exception table such that $b \leq PC \leq e$ and $E \preceq T$ (where $T_i \preceq T_j$ denotes that T_i is either the same type or a subtype of T_j). Note that the table ordering means we need not search for the two cases suggested in the partial ordering $R_i < R_j$. (In the case of a context with no handlers, we can assume an exception table with 0 rows.)

B. If such a row exists, then transfer control to the statement at h, and exit this algorithm.

C. If no such row exists, then no handler exists within this context for the exception. The current activation record must be discarded, with any other processing resulting from this step also performed. If no other enclosing contexts exist, proceed to step D; otherwise proceed to step A.

D. Transfer flow-of-control to the default handler for exception type T, and exit this algorithm.

Figure 3.8. *Algorithm EH3—Table-based technique*

```
1   VAR r: Ref.Rider;

    Ref.OpenStack(NIL, r);          (* a rider is placed on the topmost
                                     * frame of the procedure stack *)
5   WHILE r.mode # Ref.End DO
        Out.String(r.name); Out.Ln; (* output corresponding procedure *)
        Out.String(r.mod); Out.Ln;  (* output module declaring proc. *)
        r.Next;                     (* proceed to next frame *)
    END
```

Figure 3.9. *Iterating through activation frames using Oberon* Riders

briefly describe here. This section also mentions other issues that practitioners deem important for many implementations.

3.4.1 Metaprogramming approaches

Hof *et al.* describe the addition of an exception-handling mechanism into the Oberon system [31]. This is both a language and a small operating system, but exceptions are not part of the language definition. Their chosen approach uses the *metaprogramming* facilties in Oberon, particular the *reflection* capability named *Riders*. These *Riders* may be used to iterate through, amongst others, activation frames of active procedures. The code which appears in Figure 3.9 taken from [31] demonstrates how this is expressed in Oberon.

Using *Riders* the implementors are able to achieve what they term *zero-overhead* exception handling (*i.e.*, code which does not throw exceptions has zero overhead). Given that Oberon allows nested procedures, handlers for a procedure `P` correspond to appropriately named procedures nested within `P`'s scope. Code shown in Figure 3.10 contains a nested procedure named `HandleEof`, and this procedure is invoked if `Exceptions.Raise(eof)` is executed any time starting from when `Foo` is called to when `Foo` returns.

- `Exceptions` is a class added to the system for supporting exceptions, and all exceptions are objects in the system.
- `Raise(eof)` causes a *Rider* to be created for activation frames on the stack.
- As the *Rider* exposes each frame, the system determines which procedure created the frame. Given this procedure `P`, the algorithm then proceeds to iterate

```
 1   PROCEDURE Foo() : INTEGER;
         VAR f : File; ch : CHAR;

         (* A handler for "eof" exceptions *)
 5       PROCEDURE HandleEof (VAR eof: EofException) : INTEGER;
         BEGIN
             Close(f);
             RETURN 1 (* Error code for eof *)
         END H;
10
     BEGIN (* Foo *)
         Open(f, "...");
         REPEAT Read(f, ch); ... UNTIL ...;
         Close (f);
15       RETURN 0 ("no error")
     END Foo;

     PROCEDURE Read (f : File; VAR ch : CHAR);
         VAR eof : EofException
20   BEGIN
         IF .. end of file .. THEN Exceptions.Raise(eof) ELSE ... END
     END Read;
```

Figure 3.10. *Example of Oberon exception handling*

(via another *Rider*) through all procedures R local to P (*i.e.,* those nested within P).

- If some procedure R has the same return type as P and accepts as a parameter a type or supertype of the thrown exception, then R is the handler for this exception.

While this approach is not necessarily as readable as Java or C++ exceptions, it can support termination, resumption and retry semantics if the `Exception` class is coded appropriately (or, equivalently, if the suitable subclass of `Exception` is chosen). The authors make no claims for the efficiency of their implementation beyond not needing the extra space required by range-table techniques.

3.4.2 Object creation and destruction

Transfer-of-control issues are most important when implementing an EH mechanism, but another difficult set of issues involves the effect of thrown exceptions when regular control-flow moves into or out of variable scopes. Languages without garbage collection, for instance, require both the explicit invocation of variable destructors (if they exist) and the return of variable memory to heap. Christensen lists four different cases which the mechanism may encounter [16]:

1. When exceptions propagate out of a function, procedure, or method, then *local variable* destructors must be invoked.
2. When a handler cannot be found for an exception (*i.e.,* unwinding has proceeded to the top context), then *global variable* destructors must be invoked.
3. When an exception handler is exited because control-flow has reverted back to normal, then the *exception object* destructor must be invoked.
4. Finally, when an exception is thrown when an object is partially constructed, all instance variables already defined must have their destructors invoked.

Registration-based EH implementations (*i.e.,* those which maintain stacks of active handlers) lend themselves to an easy solution in that some record of created variables or objects can be kept on the same stack as that for handler information. At the time an exception is thrown the list of such variables is easily traversed. Clearly, however, this adds even more overhead at execution time (both space and time) to

programs which do not throw exceptions; there must also be some way to distinguish amongst the four different kinds of variables and their cases as listed above. The list is examined only when an unhandled exception occurs [36], but otherwise it is maintained in a similar manner as with handler-info nodes.

Range-table techniques can lend themselves to "clean-up" code, especially for those cases where objects may be partially created when an exception is thrown. Here the ranges and cleanup code are generated by the compiler. In essence these handlers "undo" the effect of an operation sequence which is aborted before completion. For each atomic step in the operation sequence—and such a step may even correspond to a single machine instruction— a handler is created which "undoes" the work of the step. A handler associated with an operation is executed if an exception is delivered *before* the operation is completed [13].

3.4.3 Stack traces and debugging support

A useful attribute of Java exception instances is a *stack trace*. Dumped to the terminal as a result of an exception unhandled by the program, the trace contains an entry for each method active at the throw-point in the program along with extra information of great value to debuggers (*e.g.,* line counts from classfile sourcecode, classfile names, etc.). Gathering such information requires that the stack be walked from the throwsite up to the topmost `main` method; inspection of Sun's JVM confirms this to be the case. Unfortunately all of this effort is wasted when an exception is handled, especially if the handler does not inspect the exception object. As mentioned in the thesis introduction, our experiments have determined that stack-trace construction accounts for a significant portion of the time taken to throw an exception (a larger discussion of this issue, plus actual timings, can be found in Section 7.1). Therefore one way of increasing exception handling performance would be to avoid constructing such a trace: at the time the exception is thrown, some information about its handler must be known.

Debuggers also require some additional support in the presence of exceptions, specifically that for *non-destructive stack walking*. The standard technique for locating an exception handler walks up the stack through activation frames, releasing the frames if the exception is propagated out of the corresponding procedure. This is a

serious problem if we desire a debugger whose focus may be changed from the throwing context to some ancestor context—destructors and finalization routines should not be executed in these cases. Yet another additional complication is that some handlers are compiler-generated to deal with object creation and deletion (cf. previous section), while others are indeed written by a programmer [13]. Chase suggests three possible solutions to this problem:

1. If the debugger is able to interpret machine instructions, then have it interpret the unwinding operations on a copy of the machine state.
2. Construct debugging information that represents the steps required to restore a previous caller's activation record.
3. Create an alternate version of the unwinder which works on a copy of the machine state.

Any techniques and analyses used to improve the performance of exception handling would need to deal with the ability of the debugger to reconstruct program state, and it seems the easiest approach is that described by (1).

3.4.4 Type comparisons

Algorithm EH3 (Figure 3.8) used the $\preceq$ operation to determine if a handler body was either the same type or a supertype of the thrown exception. While such tests do not dominate the cost of handling an exception, reducing the cost is clearly of some interest especially as an exception hierarchy becomes more expressive (*i.e.*, larger and with more ramifications). There exist a family of techniques for object-oriented languages which attempt to determine type relationships as efficiently as possible both in time and space [71].

Of interest here, however, are the needs when trying to locate a handler. Given a list of potential handlers in some context, a dispatch algorithm must determine if a handler exists within this list, and should do this as quickly as possible. Therefore what is needed is some function which can return `true` or `false` depending on the existence of such a handler; the function would accept two parameters, one an exception type, and the other a list of types. A more expressive function could accept a list of handlers, returning the address of the handler if it exists, and returning the

address 0 otherwise. Since these lists are known at compile-time, the lists and the type information they encoded could be generated in some efficient form suitable for lookup at runtime.

3.4.5 Exception-condition detection

In their description of an implemention of Java exceptions in CACAO, Krall and Probst describe an interesting benefit they obtained from their run-time organization [37]. While this benefit does not directly improve the speed of exception handling, it does improve the speed of *detecting* a condition leading to an exception. This is important. A large group of exceptions in languages such as Java are *unchecked* (*i.e.,* asychronous from the point of view of the programmer), and the possibility of throwing exceptions of this group such as `NullPointerException` means that run-time checks are associated with each dereference of a pointer. Given that we wish to encourage the use of exceptions—including those which are unchecked—some might object not so much to the cost of throwing the exception as to the extra overhead introduced by a language requiring that these runtime checks be performed.

Krall and Probst instead eliminate the overhead of null-pointer checks by protecting the first 64K of memory against read and write accesses. This technique is used by several operating systems in that important boundaries between sections of process memory are protected by "barriers" [63, p. 414], and pointer dereferences within these barriers yield the dreaded `segmentation fault` if not caught by the program. In CACAO, null pointer dereferences raise a bus error, and after checking if this signal is the result of accessing an address in the first 64K, a `NullPointerException` is thrown. This particular scheme provides a significant time savings, especially in situations where null-pointer checks are very frequent but null-pointer exceptions are rare (*i.e.,* OS overhead in delivering relevant signals is less than the time consumed by null-pointer checks).

While the technique is not necessarily applicable to other kinds of checks (*e.g.,* whether an array index is within the array bounds; whether an object may be cast into a class, etc.), reducing the overhead of such checks is important. We will assume such techniques are readily available (*i.e.,* those described by Gupta in [28], and Midkiff in [49]) and may be implemented as needed.

3.5 Existing EH optimizations

3.5.1 Stack-unwinding overhead

In their technical report describing an implementation of Modula-2 exception handling, Drew *et al.* describe a source of unnecessary overhead within stack-unwinding routines [22]. They remark that the rather heavyweight state restoration required when transferring context from one activation frame to another is, in fact, not strictly necessary when handling an exception. For instance, what needs to be restored is the stack and frame pointers of the catcher and its display vector along with the values for callee-saved registers; such state belonging to activation frames lying between the thrower and catcher need not be restored (with suitable representations constructed of these frames as needed for debuggers). There may even be some unnecessary restoration of state from the point of view of the catching context, *e.g.,* callee-saved registers in "sandwiched" frames. Of interest to programming-language designers is that an accurate representation of system state be available to the catcher, and how that state is constructed (or re-constructed) is of interest to implementors.

Drew *et al.* instead suggest an *incremental restoration of state.* This uses a *dummy frame* upon which are applied the actions specified by *dummy epilogs.* These epilogs are emitted by a code generator at compile time and contain code for restoring callee-save registers and display-element restoration. Unwinding is now applied to a dummy frame, and only when the frame with the handler is found are the real pointers updated (*i.e.,* stack information of the unwound procedures is lost only at that time). We observe that, given some knowledge of the possible paths from specific `try`-blocks to throw site, the dummy epilogs could be optimized even further to contain less code.

3.5.2 Exception-directed optimization (EDO)

A distinction can be made between `stack unwinding` in which activation frames are examined one-by-one, and `stack cutting` in which a single operation transfers control from the throw site to the catch block by removing intervening activation frames all at once. As discussed earlier in the chapter, the former scheme imposes no costs on `normal path` code, while the latter benefits `exception path` code while imposing a overhead each time a `try`-block is entered and exited (*i.e.,* `setjmp` and

`longjmp` costs).

Ogasawara *et al.* have proposed a combination of three techniques that, combined together, attempt to unify these two approaches but without imposing a penalty on normal-flow code [54]. These are:

1. exception path profiling,
2. exception path inlining, and
3. throw elimination.

The central idea is that of constructing *exception paths* at runtime. Each path represents the sequence of methods encountered when some instance of an exception is propagated from its throw site to its handler. In order to achieve (3), the handler code must be accessible from a Java Virtual Machine `athrow` instruction such that a `goto` can transfer control to the start of the handler, *i.e.,* all of the code must be contained with the same method. (The `athrow` instruction is a JVM bytecode that takes the object on the top of the VM's operand stack—which is always an instance of some exception class—and then throws an exception based on the object's type.) By identifying `hot exception paths` via use of profiling data, a just-in-time compiler (JIT) can create a large inlined method for each exception path containing the methods listed in that path.

Tests performed using benchmarking code from various tests suites such as `SPECjvm98`, `SPECjbb2000` and the Java Grande benchmarks do indeed demonstrate a big benefit to what the authors call *exception intensive programs*, *i.e.,* those which throw tens of thousands of exceptions as a result of startup processing or regular processing or both. Optimizing for exceptions paths is a big win as 90% of the *total* exception count is covered by such paths in these programs, and the paths lengths are not so large as to produce bloated code (path lengths were less than 5).

3.5.3 Optimizing for local throws

A less aggressive but still profitable approach has been taken by the implementers of `LaTTe`, another Java JIT compiler and VM [40], [39]. As is the case with EDO, the Java bytecodes are converted into machine code via the JIT. The big difference, however, is that for a handler and throwsite to be optimized, they must be within

the same method—this particular optimization does not work when throwsites and handlers are located in different methods. Once translated as a result of the algorithm, control-flow transfer from throwsite to handler is via a `goto`. This translation is made *on-demand*, meaning that it is only performed if the `throw-catch` pair is actually used at runtime.

Another difference of this scheme from EDO is that the authors believe reducing the JIT workload itself produces a significant benefit. JIT translation time is reduced by about 4% on `SPECjvm98` benchmarks as a result of applying the optimization on-demand. Therefore a compile-time scheme (*i.e.,* translating Java into bytecode) could make the JIT treatment of exception handling more efficient by annotating bytecode with additional optimization information.

3.5.4 Representing control flow

Even if EH can be made efficient and fast, there still remains the impact of exceptions on other optimizations. One of the most serious is the effect on *control-flow graphs* or *CFGs*, a traditional representation of programs used by many compilers. One approach to modelling control-flow given exceptions is to add edges from throw sites to catch sites, but Choi *et al.* have observed that this introduces several serious drawbacks [15]:

1. the size of basic blocks is reduced, therefore reducing the scope for *local analyses*;
2. the number of nodes and edges in the CFG increases;
3. a reduced precision results from analyses performed on behalf of optimizations, in addition to increased analysis time given the large graph.

One solution observes that instructions which may throw an exception (*e.g.,* any instruction with dereferences a pointer) can be called *potentially exception-throwing instructions* or *PEIs*, and that their presence (in the form of edge-inducers) can be mitigated in a `Factored Control-Flow Graph` or *FCFG*. The high frequency of PEIs induces CFG edges (*i.e.,* explicit `throws` also generate an edge, but are not as numerous as PEIs), so FCFGs differ from traditional CFG in that they also include *factored edges*. These lead from basic blocks which contain PEIs to graph nodes representing exception handlers. By introducing this new kind of edge, the drawbacks listed above are less severe:

1. Basic blocks are no longer necessarily broken up by PEIs as basic blocks can now have multiple factored edges exiting them, leaving the regular exit edge to represent normal flow of control as it leaves the basic block.
2. With larger blocks, the node count is kept smaller.
3. Global-analysis equations now include new expressions involving factored edges such that the meaning of these new edges is kept separate from that of the traditional CFG edges.

FCFGs have been used with success in the *Jalapeño* compiler from IBM (*i.e.*, a Java JIT and JVM). Basic block sizes are indeed much larger than with a traditional CFG treatment of PEI-induced edges.

3.6 Onwards

We are now ready to investigate some of our new techniques for improving EH performance. The focus here is on Java implementations, and we observe that there are three separate phases at which we can perform analysis and code generation: compilation to bytecode; loading of classfiles or *linking*; and dynamic generation of machine code in a JIT. The next few chapters cover:

- techniques for identifying handler sets, and using these to yield new compile- and link-time approaches for building data structures that reduce exception dispatch time;
- the impact on run-time organization given the additional information above and the need to transfer control to handlers while at the same time ensure normal-flow code is not caused to run slower; and
- lazy stack trace construction, again using compile- and link-time techniques, as a way of reducing exception-object construction time.

However, we first discuss some programming idioms—old and new—which lend themselves particularly well to exception handling.

Chapter 4

Exception Handling Idioms

In earlier chapters we referred to the association of "exceptions" with "errors." This association is an easy one for programmers to make; errors raised within a CPU are themselves called "exceptions," and the original version of programmatic exception-handling in PL/I matched underlying System 360 hardware with exception types in the language (and there were also some with no relation to hardware errors). Given that errors are expected to be rare events, exception-handling mechanisms have not been considered suitable candidates for optimization. All this is both reasonable and unfortunate.

We believe, however, that the control-flow and data-flow patterns made possible by exceptions can be used for programming situations other than that of expressing error handling. The contribution of this chapter, therefore, is its description of some of these possibilities. Our hope that others will suggest themselves to the reader. Nearly all of the *idioms* or *patterns* in this chapter depend upon an efficient EH mechanism—without such a mechanism, the resulting code is simply too slow.

In this chapter we examine several different ways in which exceptions can be used that are not dependent on error detection. *Local exceptions* occur where both throw-site and handler are always within the same method; exceptions therefore add yet another mechanism for transferring control flow within the method. ("Local" here takes its meaning from programming-language design, not from compiler construction where it refers to the contents of a basic block.) *Non-local exceptions* occur where a throwsite and its handlers are in different methods; the automatic unwinding of the callstack—in effect, non-local transfer of control—allows for an expressive form of *deep returns*. *Sinking exceptions* are a flavour of non-local exceptions where the actual exception object is created within the `try` block and passed as an argument

down the callstack (*i.e.*, "sinking"); throwing the exception results in transfer to a handler specified by the creator of the exception object. Finally we compare exception handling to the semantics of the *Zahn construct*, a form of event-driven programming.

4.1 Local Exceptions

Imperative languages such as Java provide a range of structured control-flow mechanisms, `if-then-else` statements, `for` loops, `while`s and `do-while`s, `switch` statements, labelled `break`s, etc. One construct that is missing, however, is the ability to transfer control arbitrarily from several program points to a single program point. This is something we can achieve with C's `goto` statement, but there are (of course) good reasons for *not* implementing such a mechanism (and good reasons *for* implementing it, *pace* Knuth [35]).

The missing mechanisms would be especially helpful where writing straightforward `switch` or multiway `if` is awkward. These situations occur where the several originating program points (or "cases") are at different statement nestings and loop—the solution usually involves writing values to temporary state variables. Exceptions, however, can be used to implement a variant of the missing construct.

An example of this is an *interpreter* structured around an infinite loop, a loop that contains two basic steps:

- A *decode* step examines the current state of the system along with the next operation (and possibly its operands) that appears in the instruction stream. Here the goal is to determine what action the interpreter should next take on behalf of the interpreted program.
- A *dispatch* step transfers control to the part of the interpreter which performs the action determined by the *decode* step.

There are several standard realizations of such a structure. For example, the *decode* step can be used to compute a specific case in the form of a unique integer, and the *dispatch* step is a `switch` statement that uses the integer as a label for an individual `case` statement. In another example, separate functions are written to implement the interpreter actions, and dispatch consists of determining which function should be called. Both examples assume that once control returns from the dispatched code,

control is then transferred to the decode step of the next instruction (assuming that the decode step also fetches this instruction) In our example, we assume that all control-flow transfer is within the same procedure.

This two-part structure lends itself to an exception-handling approach. Figure 4.1 contains some Java code for a program that accepts a single string as input and interprets this as a reverse-polish notation expression with three operators (addition as "`+`," multiplication as "`x`," and square root as "`sqrt`"). If the keyword `var` appears in the stream of tokens, then a variable name is indicated by the following token; if it is in the system dictionary, its value is obtained, otherwise a new variable is "defined." (The various bits of code for these operations are not shown in the figure.) This last bit of functionality complicates the decode step; where the first three cases have a similar structure, this last case is somewhat more complicated. It is not that this code cannot be expressed using just `if` statements and a `switch`, but rather that the basic control flow of the decode step would be obscured.

This interpreter has several different exception types, each corresponding to a different action.

- The loop is a `while` statement that iterates through all tokens in the string (that is, all character sequences surrounded by whitespace).
- The *decode* section consists of the code within the innermost `try` block.
- The *dispatch* of operations is now performed by Java's exception handling mechanism. When the addition operator is encountered in the stream of tokens, then the code in the `AddE` handler is executed; for multiplication, the handler for `MultE` is executed; for the square-root operation, that for `SqrtE` is executed; and for all other operands, that for `PushE` is executed.

Note that there is another try-block in the code; this outermost try-block contains the scope for the handlers involving errors. The errors here are: `NumberFormatException` for strings which cannot be converted into double-precision floats; and `ArrayIndexOutOfBoundsException` for operand-stack underflows and overflows. Of course, there is nothing preventing us from catching these exceptions at the inner try; in the same vein, we could have multiple handlers for the error exceptions (one for inner `try`, one for outer `try`).

```
1   import java.util.StringTokenizer;

    public class RPN {
        private final int MAX_STACK = 20;
5       private String expr;
        public double value = Double.NaN;

        public RPN (String expr) { this.expr = expr; };

10      public void interpret() {
            double stack[] = new double[MAX_STACK];
            int sptr = -1;
            StringTokenizer st = new StringTokenizer(this.expr);
            String tok = "";
15          try {
                while (st.hasMoreTokens()) {
                    tok = st.nextToken();
                    try {
                        if (tok.equals("+")) {
20                          throw (new AddE());
                        } else if (tok.equals ("x")) {
                            throw (new MultE());
                        } else if (tok.equals ("sqrt")) {
                            throw (new SqrtE());
25                      } else if (tok.equals ("var")) {
                            tok = st.nextToken();
                            for ( /* traverse through some structure
                                   * to find var value*/ ) {
                                if ( /* found */ ) {
30                                  tok = ...; throw new PushE();
                                }
                            }
                            tok = "0"; throw new VarDefineE();
                        } else {
35                          throw (new PushE());
                        }
                    } // end inner try
                    catch (PushE le) {
                        stack[++sptr] = Float.parseFloat(tok);
40                  } // clauses continued ....
```

Figure 4.1. *Local-exception usage (decode and dispatch)*

```
                    // continued...
                    catch (AddE ae) {
                        double f = stack[sptr--];
                        double g = stack[sptr]; stack[sptr] = f + g;
                    }
                    catch (MultE ae) {
                        double f = stack[sptr--];
                        double g = stack[sptr]; stack[sptr] = f * g;
                    }
                    catch (SqrtE se) {
                        stack[sptr] = Math.sqrt(stack[sptr]);
                    }
                    catch (VarDefineE e) {
                        // Code here helps keep track of number of
                        // variables....
                        stack[++sptr] = Float.parseFloat(tok);
                    }
                } // end while
            } // end outer try
            catch (NumberFormatException nfe) {
                System.out.println("Cannot convert "
                     + tok + " into a double"); return;
            }
            catch (ArrayIndexOutOfBoundsException ae) {
                System.out.println("Stack error when "
                     + "processing token " + tok); return;
            }
            catch (NoSuchElementException ne) {
                System.out.println("Expecting more tokens after "
                     + "processing token " + tok); return;
            }
            if (sptr >= 0) { this.value = stack[sptr]; }
        }

        public static void main(String args[]) {
            if (args.length >= 1) {
                RPN r = new RPN (args[0]); r.interpret();
                System.out.println (r.value);
            }
        }

        class PushE extends Exception { }
        class AddE extends Exception { }
        class MultE extends Exception { }
        class SqrtE extends Exception { }
        class VarDefineE extends Exception { }
    }
```

Figure 4.2. *Local-exception usage (decode and dispatch)—continued*

Using exceptions for decode-and-dispatch yields several benefits:

- It is easier to ensure that flow-of-control is transferred from some decode case to a dispatch case. The decode instructions may be arbitrarily complex and nested, yet the `throw` statement will transfer control to the correct dispatch/exception clause without the use of `break` statements.
- This also applies to decode steps which require support functions, especially where a dispatch decision can be made within such a function. Control-flow transfer leads directly from the function to the dispatch/exception clause.
- Statements required for decoding interpreted instructions are kept entirely separate from statements implementing the instructions. The decode stage can then be rewritten, modified, etc. without affecting code in the dispatch section.

The above style of coding, however, requires some careful planning. If the values of variables manipulated by the decode stage must be visible to dispatch clauses, then the scope of such variables must be outside the `try`-block. The programmer must also write exception classes for each of the instruction types. Even here, however, multi-stage decoding could be modelled by a non-trivial hierarchy of instruction-type exceptions; such a hierarchy could also serve as a form of executable documentation.

Another surprising benefit from the use of exceptions within the same method is that the resulting code may run *faster* than the original. Specific conditions must hold for this performance improvement to be present, however, which are described in more detail in Chapter 5.

4.2 Non-local Exceptions

A greater range of program-structure opportunities exist when handlers and throwsites are in different methods, some of which Levin noted in his PhD dissertation [41]. One involved the use of exceptions with resumption-semantics to improve the performance of a search function. The search function had the property that starting a search was expensive, and so an exception was used to literally "throw back up" an individual search result; the next matching item would then be found as a result of the "resumed" search, *i.e.,* the resumed exception would continue where it last left off.

As described earlier in this thesis, researchers have settled on termination-semantics for exceptions, therefore Levin's proposal would not work in a language such as Java. There are, however, still many situations where exceptions can be used to structure algorithms involving search or traversing where the method locating the answer is some distance away on the call-stack from the method needing the answer. Such *multi-level returns* can be awkward to program. Methods must check return values to determine if the value needed was found or if it was not found, and method can usually only return values of one type. Part of the difficulty is caused by the conflation of control-flow with the inherent structure of search data—when all goes well, control-flow matches the data structure perfectly, but otherwise the control-flow can be dramatically different. This can be seen in recursive-descent parsers—an error must be propagated up the call-chain (*i.e.,* multi-level return), and since each intervening function must be aware of such errors, the regular- and error-handling code become intertwined.

Exceptions are a natural mechanism for implementing algorithms using multi-level returns. Figure 4.3 shows an example of its use in solving a typical programming problem (and a variant of this is explored further in Chapter 8 which presents experimental results using the variant). Normal control-flow occurs when a word is found to already be in the tree; exception-handling is used when the word is *not* in the tree and a new node must be inserted into the tree. The code necessary for searching and the code necessary for tree construction are therefore kept separate.

4.3 Sinking Exceptions

One of the motivations for modern exception handling mechanisms is that it enables a separation of *error detection* and *error handling.* For example, a filesystem library routine for `open(filename, mode)` is able to detect when the file corresponding to `filename` does not exist, or when the user does not have the permissions required for `mode` access, etc. However, the library does not know what action to take. Should another file be tried? Should the `open` itself be retried again after an interval (*i.e.,* for network-file system environments where files may be temporarily inaccessible)? Of course, the code using the filesystem routines should know what actions to take;

```
1   public class TreeSearch
    {
        private WordNode word_tree;
        public  WordNode curr_node;
5
        // Various constructors...

        public void build_frequencies() throws IOException {
            String line = null;
10          word_tree = new WordNode ("");
            while ((line = br.readLine()) != null) {
                StringTokenizer st = new StringTokenizer(line);
                while (st.hasMoreTokens()) {
                    String tok = st.nextToken();
15                  try {
                        traverse (word_tree, tok);
                    }
                    catch (NewLeftLeafException le) {
                        WordNode new_word = new WordNode(tok);
20                      curr_node.left = new_word;
                    }
                    catch (NewRightLeafException re) {
                        WordNode new_word = new WordNode(tok);
                        curr_node.right = new_word;
25                  }
                    catch (TraversalException e) { }
                }
            }
        }
30
        private void traverse (WordNode wn, String curr_tok)
            throws NewRightLeafException, NewLeftLeafException
        {
            if (wn.value.compareTo(curr_tok) == 0) {
35              wn.frequency++;
                return;
            } else if (wn.value.compareTo(curr_tok) > 0) {
                if (wn.left != null) {
                    traverse (wn.left, curr_tok);
40              } else {
                    throw new NewLeftLeafException(wn);
                }
            } else {
                if (wn.right != null) {
45                  traverse (wn.right, curr_tok);
                } else {
                    curr_node = wn;
                    throw new NewRightLeafException(wn);
                }
50          }
        }
```

Figure 4.3. *Non-local exception usage (tree search)*

```
        class WordNode
        {
            public String  value;  public int     frequency;
55          public WordNode left;  public WordNode right;

            public WordNode (String word) {
                this.value = word; this.frequency = 1;
                this.left = this.right = null;
60          }
        }

        class NewLeafException extends Exception {
            WordNode n;
65          public NewRightLeafException (WordNode n) { this.n = n; }
            public WordNode n() { return n; }
        }
        class NewRightLeafException extends NewLeafException { }
        class NewLeftLeafException extends NewLeafException { }
70  }
```

Figure 4.4. *Non-local exception usage (tree search): continued*

actions can be encoded within an exception handler, and the handlers are invoked by the `open` throwing the appropriate handler.

This model can be used for more than just error handling. What is important here is the separation of the detection (of *some condition*) from the processing (associated *with that condition*).

The processing required will depend, of course, on the code to which handlers are attached. However, the actual action to be taken may depend on more of the program's state. For example, sometimes we may want action A to be taken in the event of the condition occurring, and other times we may want action B (*e.g.*, action A may be "retry," while action B is "abort"). Usually we must encode these two events by one handler—the handler itself must then determine which of the two actions to take.

Another way to accomplish the same thing is shown in Figure 4.5. Rather than burdening the handler with the need for determining whether to print "Result 1," "Result 2" or "Result 3," we instead pass an exception object down to the called routine (in this example, the leaf routine is `subtask`). In effect we have implemented a "one-time call-back." The determination of system state in `process` is now separated from the work of handling conditions as detected. Another benefit is that the states are clearly delineated in the code (as a list of "catch" blocks).

```
1   public class Downwards {

        public Downwards() { }

5       public void process (String s) {
            try {
                if (s.equals("one")) {
                    subtask(new State1());
                } else if (s.equals("two")) {
10                  subtask(new State2());
                } else {
                    subtask(new State3());
                }
            }
15          catch (State1 oe) {
                System.out.println ("Result 1");
            }
            catch (State2 oe) {
                System.out.println ("Result 2");
20          }
            catch (State3 oe) {
                System.out.println ("Result 3");
            }
            catch (StateE oe) {
25              System.out.println ("Unexpected result");
            }
        }

        private void subtask (StateE oe) throws StateE {
30          throw oe;
        }

        public static void main(String args[]) {
            Downwards d = new Downwards();
35          if (args.length >= 1) {
                d.process(args[0]);
            }
        }

40      class StateE extends Exception{}
        class State1 extends StateE {}
        class State2 extends StateE {}
        class State3 extends StateE {}
    }
```

Figure 4.5. *Sending an exception object downwards*

The same caveats mentioned in the last section hold here (*i.e.,* variable visibility, declaration of exception types).

4.4 Zahn's *event* construct

In 1974, Donald Knuth penned an article that appeared in *ACM Computing Surveys* entitled "Structured Programming with go to Statements" [35]. He attempted to provide a counterpoint to the enthusiasm for eliminating `goto` statements—all the rage in the early 1970s—by arguing that some uses of `goto` were necessary, so much so that their elimination would otherwise result in beautifully structured but unreadable code. He also explored an "improved syntax for iterations and error exits," and championed a language feature proposed by C. T. Zahn called an *event indicator.*[1]

The code example at the top of Figure 4.6 demonstrates code for a tree-search-and-insertion algorithm that Knuth proposed; it uses event indicators. The item `x` is to be inserted into a binary tree; array `A` contains the values at tree nodes, while arrays `L` and `R` are "pointers" to the left- and right-subtrees of nodes. The code at the bottom of Figure 4.6 expresses the same thing using exceptions.

Code using event indicators takes the following form:

```
loop until <event_1> or ... or <event_n>:
    <statement_list_0>;
repeat;
then <event_1> => <statement_list_1>;
     ...
     <event_n> => <statement_list_n>;
```

Each of the `<event>`s corresponds to a new, programmer-defined keyword, and its appearance in `<statement_0>` indicates that event (or designated condition) has occurred. The scope of `<statement_list_0>` is also the scope from which events can be designated. The `<statement_list_0>` is executed over and over until one of the named events is indicated; control flow then leaves the loop and is transferred to the

[1]Unfortunately, Zahn's proposal has sunk back into relative obscurity. A recent search for `knuth ''event indicator''`, `zahn ''event indicator''` and `''zahn construct''` on Google yields about a dozen hits.

```
1   loop until left_leaf_hit or
                right_left_hit:
        if A[i] < x
        then if L[i] <> 0 then i := L[i];
5            else left_leaf_hit;
        else if R[i] <> 0 then i := R[i];
             else right_leaf_hit;
    repeat;
        then left_leaf_hit => L[i] := j;
10           right_leaf_hit => R[i] := j;
    A[j] := x;
    L[j] := 0;
    R[j] := 0;
    j := j + 1;
```

(a) using event indicators (after Knuth [35])

```
1   try {
        for (;;) {
            if (A[i] < x) {
               if (L[i] != 0) i = L[i];
5              else throw new left_leaf_hit();
            } else {
               if (R[i] != 0) i = R[i];
               else throw new right_leaf_hit();
            }
10      }
    }
    catch (left_leaf_hit e) {
        L[i] = j;
    }
15  catch (right_leaf_hit e) {
        R[i] = j;
    }
    A[j] = x;
    L[j] = 0;
20  R[j] = 0;
    j = j + 1;
```

(b) using exceptions

Figure 4.6. *Tree-search-and-insertion*

"event handler." There is also another flavour of event indicators which does not imply iteration (*i.e.,* `loop` and `repeat` are replaced by `begin until` and `end`).

The version of the algorithm using exception handlers has a strikingly similar structure. Of course, the exceptions caught by the catch blocks may be thrown from other methods (*i.e.,* the scope of the exceptions is not restricted to the infinite loop). However, it is possible to write a preprocessor which could accept a modified event-indicator-like syntax in a Java program and convert that code into the exception-handling style.

Chapter 5

Exploiting Exceptions

A version of this chapter was previously published in 2001 as "Exploiting Exceptions" in *Software: Practice & Experience* [75].

5.1 Introduction

There are situations where exceptions can be used as a standard programming pattern to make programs execute faster. These situations are opportunities to increase the speed of Java programs by changing bytecode within methods with a space cost of, at most, a few extra instructions. Such modifications apply knowledge of which run-time actions and checks are performed by a virtual machine as bytecode instructions are executed. For instance, within Sun's Java Virtual Machine (JVM), all object dereferences are preceded with a run-time check for a null value; if null, a `NullPointerException` is thrown; if not null, the object dereference proceeds. As observed by Orchard [55], this exception can be exploited in loops whose control expressions involve an explicit null check since the expressions may be redundant—the exception thrown as a result of dereferencing a null pointer may be used to transfer control out of the loop (Figure 5.1a, b).

Array-bounds checks present another opportunity. Before accessing any element of an array, the run-time system first checks if the array index is within the array's bounds; if not, the JVM throws an `ArrayIndexOutOfBounds` exception; otherwise the array access proceeds. In a similar manner to the previous example, the exception may be exploited in any loop whose control expression involves a comparison between a loop variable and an array's length. The check may be redundant where the same action is performed for every array access, and we may use the exception to transfer

```
a = 0;
p = head;
while (p != null) {
  p = p.next;
  a++;
}
return a;
```

(a) original

```
a = 0;
p = head;
try {
  for (;;) {
    p = p.next;
    a++;
  }
}
catch (NullPointerException e) {}
return a;
```

(b) transformed

Figure 5.1. *Eliminating a redundant null check*

control out of the loop. This eliminates one check on every loop iteration (Figure 5.2a, b); using Sun's HotSpot JVM on a 700 MHz Pentium 3, the transformed code is faster than the original code (13 ms vs. 20 ms) when the array is large (`A.length` > 500K). The speedup on any JVM clearly depends on the cost of throwing an exception and on the number of loop iterations.

The transformations exploit the termination semantics of Java exceptions, transferring flow of control through the use of try-catch statements. Exceptions are subclasses of the Java `Exception` class, and instances of exceptions are either thrown explicitly via the `throw` keyword, or implicitly through the failure of some run-time check. The programmer may use a try-catch statement to specify the exception handler (`catch` clause) for a specific block of code (`try` clause). When an exception is thrown within a try-block, the JVM checks if a handler for this exception class exists within a catch-block. If so, control is transferred to the first instruction in the catch-block; if not, the exception is propagated to the dynamically enclosing scope where the search is repeated.

In Figure 5.1a, the loop terminates when `p` is null, after which the linked list size is returned. The transformed code in Figure 5.1b also terminates when `p` is null:

- The expression `p.next` dereferences `p`, so the JVM checks if `p` is null.
- When `p` is null, the dereference causes a `NullPointerException` to be thrown.
- Control is transferred to the catch block for a `NullPointerException`; in this

```
i = 0;
sum = 0;
while (i < A.length) {
  sum += A[i];
  i++;
}
```

(a) original

```
i = 0;
sum = 0;
try {
  for (;;) {
    sum += A[i];
    i++;
  }
}
catch (ArrayIndexOutOfBoundsException e) {}
System.out.println(sum);
```

(b) transformed

Figure 5.2. *Eliminating redundant array-bounds check*

case, the block is empty.

- Control continues to the next statement after the end of the catch block, in this case the `return` statement.

At the level of bytecode, an exception table associated with each method contains the information represented by source-level try-catch statements. The bytecode of Figures 5.3 and 5.4 correspond to that generated for the example in Figure 5.1a and b (*i.e.,* as would be output from a class-file disassembler such as Sun's `javap -c`). At the end of the code listing is a table having a single row, with numbers referring to statements within the method and a string referring to an Exception class. The first two numbers correspond to the try block scope, the third number to the first bytecode of the catch block, and the class name is used at run-time to match exceptions with local handlers.

The example transformation was applied to Java source code, but may be applied almost as easily to bytecode given modern bytecode disassembly-and-analysis tools. Try- and catch-blocks may then each be as small as a single bytecode instruction. An advantage of working directly with bytecode is access to the `goto` instruction—a catch block may transfer control to any other point in the method that is not itself part of a catch block. However, we must ensure that the expression stack has the correct contents as required by the definition of Java semantics, regardless of the control-flow introduced by our use of `goto`.

```
 0 getstatic #26 <head>     //
 3 astore_3                 // p = head;
 4 iconst_0
 5 istore_2                 // a = 0;
 6 aload_1
 7 aload_3
 8 ifnull 19                // p == null?
11 aload_3
12 getfield #31 <next>      // fetch p.next ...
15 astore_3                 // ... and store back in p
16 iinc 2, 1                // a = a + 1
19 goto 4                   // unconditional goto
22 iload_2                  // fetch 'a' ...
23 ireturn                  // ... and return the value
```

Figure 5.3. *Bytecode for Figure 5.1a*

```
 0 getstatic #26 <head>     // p = head
 3 astore_0
 4 iconst_0                 // a = 0
 5 istore_1
 6 aload_0                  // fetch p.next ...
 7 getfield #31 <next>
10 astore_0                 // ... and store back in p
11 iinc 1 1                 // a = a + 1
14 goto 6                   // unconditional goto
17 pop                      // start of handler
                            // (discard exception object)
18 iload_1                  // fetch 'a' ...
19 ireturn                  // ... and return the value

Exception table:
   from   to  target type
     6    17    17   <Class java.lang.NullPointerException>
```

Figure 5.4. *Bytecode for Figure 5.1b*

Not all loops may be transformed to exploit exceptions quite so simply. For instance, if the order of increment and dereference statements are reversed in Figure 5.1a, then the effects of a *spurious update* would be seen at the print statement in the transformed code, giving an off-by-one error for the list size. This occurs because the semantics of the original code would not be preserved in the transformed code. Where the original never increments the variable `a` when `p` is null, *i.e.,* the loop-control expression evaluates to false, the transformed code instead increments the variable before the `NullPointerException` is thrown and control transferred out of the loop. In this latter case, the transformation is not possible without making other changes to the code.

Another instance is where the loop body may also contain a dereference of another variable. Any `NullPointerException` resulting from this dereference must not be consumed by the catch block, but propagated out of the block with a throw instruction. Figure 5.5a shows a modification of the first example: the dereference of `q` might cause a `NullPointerException` to be thrown. In this instance, the transformed code has an additional check within the catch block to re-throw any unexpected `NullPointerException`. Figure 5.5b contains the transformed code with the check in place.

Assuming we have performed the necessary code analyses, our algorithm is applied to each program statement s which tests if an object reference `R` is null and transfers control if it is.

1. If a statement t in the *false* program path (*i.e.,* the code which is executed after the loop-conditional evaluates to `false`) *must* throw a null-pointer exception for object reference `R`, and if no variables live after t are modified on any program path from s to t, then (a) create a new try-block enclosing t, and (b) create a new handler of the form `pop; goto label`.
2. If a statement u in the *false* program path *may* throw a null-pointer exception for some object **besides** R, and if no variables live after u are modified on any program path from s to u, then (a) find all try-blocks introduced in previous step that also include u and (b) modify the handler to rethrow the exception if R happens to be not null.
3. Delete all statements s.

```
a = 0;
p = head;
while (p != null) {
  q = q.next;
  p = p.next;
  a++;
}
return a;
```

(a) original

```
a = 0;
p = head;
try {
  for (;;) {
    q = q.next;
    p = p.next;
    a++;
  }
}
catch (NullPointerException e) {
  if (p != null) {
    throw e;
  }
}
return a;
```

(b) transformed with check

Figure 5.5. *Use of additional checks in transformed code*

In the next section we present a safety analysis for exploiting the `NullPointerException`. (We omit the analysis required for the `ArrayIndexOutOfBounds` case, observing that it fits within the framework presented here.) It is followed by a short description of the transformation algorithm and an example. If the cost of throwing an exception is less than the total cost of evaluating the redundant loop-control expression summed over all iterations, then a speed improvement is the result. A more efficient implementation of exceptions as discussed earlier in this thesis would provide such an improvement for any loop iterating a sufficient number of times.

5.2 Code analysis

Our transformation goal is to remove redundant programmatic null pointer checks from conditional expressions while ensuring the meaning of the transformed program is unchanged from the original. Sufficient conditions to ensure correctness are that every `true` program path following the eliminated check:

- has a dereference of the object involved in the expression;

- and previous to every dereference contains no assignments to variables which will be used (*i.e.,* live) on some program path leading from the loop, nor contains any method call.

The first condition ensures that a transfer of control out of the loop *must* occur through a null pointer dereference, and the second ensures that all extra or spurious iterations through the loop body do not change the transformed program's meaning from the original. A *spurious iteration* is a (possibly partial) extra iteration which would not have occurred with the conditional expression in place.

We must perform analysis for two nodes: that following the *true branch* of a control expression involving some null check of object `p`, and that following the *false branch.* For the true branch we must determine:

- if every program path from this point contains at least one dereference of `p` (*i.e.,* the `NullPointerException` is guaranteed to be thrown when `p` is null); and
- the location of object dereferences within the method (*i.e.,* the starting and ending bytecodes corresponding to the try block).

For the false branch, we must know:

- the names of all variables whose values are modified on any program path leading *from* the eliminated expression *to* the object dereference (*i.e.,* if such a variable is used after loop exit, then the transformed code may have a different meaning from the original code);
- whether any operations cause a side-effect (*e.g.,* method invocations which would change the value of some instance or class variable during a spurious iteration); and
- if there are any other objects which may be dereferenced before `p`'s dereference causes a `NullPointerException` to be thrown (*i.e.,* must introduce a check within the correct catch block).

We can capture the needed information by constructing a flowgraph of the code where each node in the flowgraph corresponds to a program statement and each edge represents potential flow of control from one statement to another. *Must-ref states*

are computed for each flowgraph node. Each state is a set of tuples of the form:

$$(\textit{object reference}, \{\textit{instruction number}\}, \{\textit{variable name}\})$$

For example, at some node n, a tuple such as $(p, \{8, 9\}, \{p, x, q\})$ means:

> All forward program paths from node n will dereference `p`; the first dereference on each path will occur in one of statements 8 and 9; and assignments to `p`, `x` and `q` are the only ones which might occur before the first dereference of `p`.

An `isnull` check of an object reference `p` is considered redundant if `p` appears as an object reference within a state tuple *at the flowgraph node immediately before the true branch of the check*, *i.e.,* an `isnull` check is indeed implicitly performed by the JVM on all `true` program paths (*i.e.,* all code sequences which start when the loop condition evaluates to true and end when the condition evaluates to false). Then the set of variable names is compared against the set of live variables at the program node *immediately before the false branch of the check*, *i.e.,* a variable is live at a program node if it is used on some program path starting from that node. If the intersection of the two sets is empty, then the transformed program is guaranteed to have the same meaning as the original program. (Note that we refer to programs rather than just loops.) We use the set of bytecode positions to either construct a new exception table for the method or to modify an existing table—each new row will correspond to a one-bytecode-sized try block. A catch block is also constructed for each new try block, and simply contains a `goto` instruction to the first instruction of the false branch. A developed example is shown in Figures 5.7 and 5.9.

If we know the *must-ref* state at a node immediately after some statement S, then we can compute the *must-ref* state immediately before S by using the appropriate rule for each of the program statement types (*e.g.,* assignment, dereference, conditional branch, unconditional branch, possible side-effect). In terms of data-flow analysis, we say that information flows backwards. The function σ maps nodes to *must-ref* state values.

- Unconditional branch: copies state from successor. If n is the node preceding some flowgraph node, then $succ(n)$ is the node immediately following that same flowgraph node.

$$\sigma(n) = \sigma(n'),\ n' = succ(n)$$

- Conditional branches: a *meet operation* is performed for the *must-ref* states. We consider the simple case of conditional statements having two branches; multi-way branches are a simple generalization. The only tuples which should appear in the resulting state are those whose object reference appear in both incoming states, *i.e.,* the object will be dereferenced on all outgoing branches.

$$\sigma(n) = \{(r, l \cup l', v \cup v') \mid \exists (r, l, v) \in \sigma(n_1), \exists (r, l', v') \in \sigma(n_2)\},\ n_1, n_2 \textit{ succeed } n$$

- Side-effects: our analysis is presented for intra-procedural cases only. Therefore any instruction which performs a message send or results in a side-effect will invalidate the *must-ref* state information gathered at that node.

$$\sigma(n) = \emptyset$$

- Assignments: There are two groups of cases—one for the left-hand side of the assignment, and another for the right-hand side. Each group has two sub-cases — scalar variables and object dereference. One combination may be ignored as impossible, *e.g.,* lhs is a pointer reference with rhs a scalar variable. All other combinations transform state by first applying the rhs rule to the incoming state, then the lhs rule to this result. We use $lhs(n)$ to refer to the left-hand side variable in the statement following node n; $rhs(n)$ is defined similarly; $lnum(n)$ is the bytecode position of that statement.

  1. Any pointer dereference will either generate a new tuple, which must be added to the incoming state, or if such a tuple already exists, will replace the information already gathered for that tuple. The lhs- and rhs-cases have the same rule. A pointer dereference is of the form $r.e$, where r is an object reference, and e is a field accessed by dereferencing r. The left side of the union operator is used to select out tuples from the original value $\sigma(n)$ that are unchanged in the new value of $\sigma(n)$.

$$\begin{aligned} \sigma(n) \quad = \quad & \{(r, l, v) \mid (r, l, v) \in \sigma(n'), rhs(n') \neq r.e\} \ \cup \\ & \{(r, lnum(n), \emptyset) \mid (r, l, v) \in \sigma(n'), rhs(n') = r.e\}, n' = succ(n) \end{aligned}$$

  2. Any assignment to a scalar or object reference adds to the set of variables in all state tuples.

$$\sigma(n) = \{(r, l, v \cup lhs(n)) \mid (r, l, v) \in \sigma(n')\},\ n' = succ(n)$$

3. If the left-hand side is an object reference, then an alias is introduced, *i.e.*, the lhs object reference is now aliased to the rhs object reference following the assignment. A null pointer check on the lhs variable is equivalent to a null pointer check on the rhs.

$$\begin{aligned}\sigma(n) = \; & \{(rhs(n'), l, v \cup lhs(n')) \mid (r, l, v) \in \sigma(n'), lhs(n') = r\} \cup \\ & \{(r, l, v \cup lhs(n') \mid r, l, v) \in \sigma(n'), lhs(n') \neq r\}, \; n' = succ(n)\end{aligned}$$

4. Finally, the introduction of aliasing may also result in a state with more than one tuple having the same object reference. A *simplify* operation can merge such tuples together.

$$\begin{aligned}simplify(\sigma(n)) \; = \; & \{(r, \lambda, \delta) \mid (r, l, v) \in \sigma(n), \\ & \lambda = \bigcup\{l' \mid (r, l', v'') \in \sigma'(n)\}, \\ & \delta = \bigcup\{v' \mid (r, l'', v') \in \sigma'(n)\}\} \\ & where \; \sigma'(n) = \{(r, l', v') \mid (r', l', v') \in \sigma(n), r = r'\}\end{aligned}$$

The technique above is a form of abstract interpretation [1]. We start not knowing any of the *must-ref* states, but if we initialize all states to empty and repeatedly apply the rules, then we converge to the solution.

Before using *must-ref* states in a transformation algorithm, we must still account for one other complication added by aliasing. Aliasing introduces the possibility that an object dereference, and hence the run-time `isnull` check, is applied to some object other than the one in the loop-control expression. For instance, in Figure 5.6 the loop control expression involves an explicit `isnull` check on `p`, but one program path from the check to the dereference of `p` has an assignment of the form `p=q`. A null pointer exception thrown at the dereference of `p` may be caused by a null value originating from either (1) the object pointed to by `p` at the start of the loop or (2) the object pointed to by `q`. If the latter, then our catch block *must re-throw the exception.*

We discover this possibility in our analysis by generating *may-ref* states for each node. For example, a *may-ref* state value such as $\{(p, \{8, 9\}, \{x\}), \{q, \{8\}, \emptyset)\}$, at some node n, may be read as:

> On some forward program paths from n, `p` may be dereferenced at statement 8 or 9, and `q` may be dereferenced at statement 8.

```
a = 0;
while (p != null) {
  if (s < t) {
    p = q;
  }
  p = p.next;
  a++;
}
return a;
```

Figure 5.6. *Aliasing of object references*

At statement 8, both p and q may be referenced. Therefore if our analysis indicates that the transformation is possible, we must add a run-time check for the catch block corresponding to the try block for statement 8; if p is not null, the catch block code must re-throw the exception.

The construction of *may-ref* states differs from *must-ref*s only in the meet operation for a conditional instruction. All tuples in either path must appear in the resulting state.

$$\sigma(n) = \sigma(n_1) \cup \sigma(n_2),\ n_1, n_2 \textit{ successors to } n$$

Similar rules for analysis of array-indexed loops are based upon that loops that follow chains of references. May-ref states would refer to an *(array, index)* pair instead of singleton reference, and aliasing analysis could be replaced by an induction-variable analysis. In our analyzer implementation used to obtain the results reported later, we restricted changes to the easier case of the modified *may-ref* state.

5.3 Transformation Algorithm

Input:

- Method code with numbered instructions.
- Live-variable information where *livevar(l)* is the set of all variables that could be used along some program path starting the node corresponding to label l.
- Abstract state information for each node n in the method flowgraph.

Output:

- Modified code

Algorithm: For each statement s that tests if an object reference R is null, where n is the node at the head of the *true* edge, and m is node at the head of the *false* edge, do the following:

1. If there exists a tuple $(r, l, v) \in \sigma_{must}(n)$ with $r = R$ and if $v \cap livevar(m) = \emptyset$, then:
   (a) For every line number in l, create a null-pointer exception *catch* block entry in the exception table:
      i. the `try`-block `start` and `end` labels are both set to l;
      ii. the destination is a new globally unique label attached to the following new code sequence:[3]

```
(catchlabel): pop
              goto label
```

   (b) For each tuple (r', l', v') in $\sigma_{may}(n)$ such that $r' \neq r$ and $l' \cap l \neq \emptyset$, do the following:
      i. set $check = l' \cap l$;
      ii. set `R` to the *ObjRef* corresponding to r;
      iii. for each *lnum* $\in$ *check*, modify the catch block created in the previous step for this line number such that it reads:

```
(catchlabel): R isnull? goto (newlabel)
              throw
(newlabel):   pop
              goto label
```

      where `(newlabel)` is some globally unique label generated for each *lnum* in *check*.

2. Delete statement s.

[3]The Java Virtual Machine specification [42] requires that "the only entry to an exception handler is through an exception. It is impossible to fall through or 'goto' the exception handler." We can

We have some observations regarding bytecode verification and control-flow through `finally` blocks:

- According to Sun's JVM specification [42], an `athrow` instruction's effect on the operand stack is to discard all items below the top item, *i.e.,* the exception object becomes the sole stack contents. For the transformation to be "verifier neutral" (*i.e.,* not a cause of verification failure), no stack item may be live at node m and at the node immediately preceding s. Here we consider a stack item to be "live" if it is ever popped from the stack. The condition is trivially true at any node where the stack is empty.
- If the transform is applied to bytecode, the semantics of code using a `finally` clause are preserved without extra work. For example, a line l could be nested within a `try` block with a `finally` clause. Dereferencing a null-pointer at l in such a case must take us out of the transformed loop *without* executing any of the `finally` code. The inserted handler is therefore safe as it transfers control directly to node m after catching the null-pointer exception. This assumes that handlers appear in proper order in the method's exception table.

5.4 Example

In our example analysis, we examine a method whose flowgraph corresponds to that in Figure 5.7. The loop contains a conditional expression whose `isnull` test may be eliminated if:

a. all paths from node corresponding to letter `I` (n of the algorithm) will dereference `p`, *i.e.,* a *null pointer exception* thrown by the virtual machine can transfer control out of the loop; and

b. all variable definitions occurring after the dereference of `p` **but within the same loop iteration** as the dereference *do not reach outside the loop, i.e.,* those defined variables are *dead* at node B (m in the algorithm).

---

interpret this to mean either (a) even when within an exception handler, code cannot use "goto" to jump to the beginning of another handler, nor fall through to an instruction which happens to be the start of another handler, or (b) once within a handler, code may jump into other handlers without restriction. The first interpretation has been used for this algorithm.

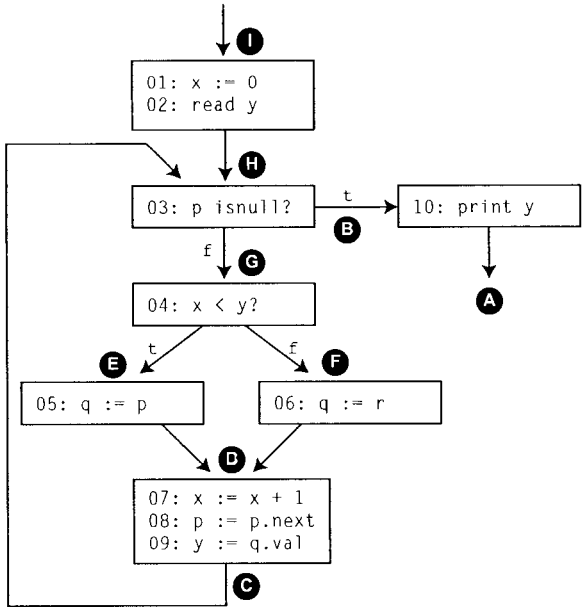


Figure 5.7. *Flowgraph with redundant loop-control expression*

If the test may be eliminated, then we must ensure that:

c. any *null pointer exceptions* other than those thrown by dereferences to `p` will not be consumed by the transformed code, *i.e.,* an extra check in the catch block will re-throw the exception if `p isnull?` evaluates to `false`.

We use *must-deref* information to determine (a) and (b), and *may-deref* information to generate the check code required by (c). The values of *must-ref* and *may-ref* states for each node are shown in Figure 5.8. Since the live-variable set for node B is equal to $\{y\}$, the `isnull` check at statement 3 may be eliminated, and the code in Figure 5.9b is the output from the transformation algorithm.

Our present analysis assumes that method calls are treated as side-effects, *i.e.,* only intra-procedural flows are considered. By adding inter-procedural analysis, we expect that more opportunities for this transformation may be found.

5.5 Code analysis results

A classfile analyzer was written using Purdue's `BLOAT` framework [73], and we selected a representative variety of Java packages for our tests. The analyzer reads the methods

point	*must-deref*	*may-deref*
A	$\emptyset$	$\emptyset$
B	$\emptyset$	$\emptyset$
C	$\emptyset$	$(p, \{8,9\}, \{p,x,q,y\}), (r, \{9\}, \{p,x,q,y\})$
D	$(q, \{9\}, \emptyset)$	$(p, \{8,9\}, \{p,x,q,y\}), (r, \{9\}, \{p,x,q,y\})$
E	$(q, \{9\}, \{p\}), (q, \{8\}, \emptyset)$	$(p, \{8,9\}, \{p,x,q,y\}), (r, \{9\}, \{p,x,q,y\})$
F	$(q, \{9\}, \{p,x\}), (p, \{8\}, \{x\})$	$(p, \{8,9\}, \{p,x,q,y\}), (r, \{9\}, \{p,x,q,y\})$
G	$(p, \{8,9\}, \{p,x,q\})$	$(p, \{8,9\}, \{p,x,q,y\}), (r, \{9\}, \{p,x,q,y\})$
H	$(r, \{9\}, \{p,x,q\}), (p, \{8\}, \{x,q\})$	$(p, \{8,9\}, \{p,x,q,y\}), (r, \{9\}, \{p,x,q,y\})$
I	$(p, \{8,9\}, \{p,x,q\})$	$(p, \{8,9\}, \{p,x,q,y\}), (r, \{9\}, \{p,x,q,y\})$
J	$\emptyset$	$(p, \{8,9\}, \{p,x,q,y\}), (r, \{9\}, \{p,x,q,y\})$
K	$\emptyset$	$(p, \{8,9\}, \{p,x,q,y\}), (r, \{9\}, \{p,x,q,y\})$

Figure 5.8. *States for Figure 5.7*

of a classfile and then reports the number of ref-chasing and array-indexing loops; the total such number is reported for each package under "all." The number which are transformable according to our analysis are reported under "transform." Results are shown in Figure 5.10.[4]

Loops that follow chains of references appear less often than array-indexing loops, and the number of transformable loops amongst the former is generally smaller than the latter. Ignoring the data for NINJA—a surprise, since numerical code would be expected to make heavy use of array indexing—the percentage of transformable loops ranged from 16% to 57% for array-indexing and 5% to 52% for reference chains.

We observe that our simple analysis rejects loops having a method call on a program path from the loop-condition expression to a suitable runtime-checked instruction. This is too conservative: many method calls have no side-effects that

[4]ArgoUML: Version 0.81a, UML-based CASE tool; CUP: Version 0.10j, LALR parser generator; GNU Classpath: Version 0.0.2, open-source implementation of essential Java class libraries; Java2 JRE: Linux version 1.2.2, from SDK; Jakarta: packages from Apache Java-based web server (ants, tools, watchdog, tomcat); HotJava Browser: version 3; NINJA: IBM's Numerically INtensive class library; Ozone: version 0.7, an open source OBDMS; SPECjbb2000: Version 1.01, Java Business Benchmark; SPECjvm98: Version 1.03.

```
    x := 0
    read y
03  p isnull? goto 10
    x < y? goto 05
    q := p
    goto 07
05  q := r
07  x := x + 1
08  p := p.next
09  y := q.val
    goto 03
```

(a) original

```
    x := 0
    read y
03
    x < y? goto 05
    q := p
    goto 07
05  q := r
07  x := x + 1
08  p := p.next
09  y := q.val
10  print y
    return
11  pop
    goto 10
12  p isnull? goto 13
    throw
13  pop
    goto 10
```

Try-start	Try-end	Dest	Exception
08	08	11	nullpointer
09	09	12	nullpointer

(b) transformed

Figure 5.9. *Original and transformed code for working example*

Package(s)	*ref-chasing loops*		*array-indexing loops*	
	all	transform	all	transform
`ArgoUML`	172	86	152	8
`CUP`	5	2	2	0
`GNU Classpath`	176	94	37	5
`Java2 JRE (rt.jar)`	362	190	365	91
`Jakarta`	621	342	248	35
`HotJava Browser`	155	74	87	22
`NINJA`	2	0	0	0
`Ozone`	287	165	330	42
`SPECjbb2000`	12	2	4	0
`SPECjvm98`	326	91	186	96

Figure 5.10. *Classfile loop analysis*

would invalidate *must-ref* states (*e.g.*, , an `append()` to a string, `clone()` on an `Object`). Beyond modified local variables in a loop, there are other cases involving some global variables (*i.e.*, instance and class), and these cases could be enumerated and integrated into the loop analysis at the cost of a little complexity.

Chapter 6

Improving EH performance: Dispatch out of Procedures

This chapter investigates several of the costs incurred when propagating exceptions out of methods. Several attempts at a solution are first presented, and then we introduce our concept of *farhandlers*—*i.e.,* program-wide exception-handling tables. The construction of such tables is discussed along with a proposed technique for using them at runtime within a Java Virtual Machine.

6.1 Motivation

When an exception is thrown, its handler must be found and control transferred to that handler. This process is called *dispatching an exception*, and if we wish to improve the performance of exception handling, the time taken to start and complete a dispatch operation must be reduced as much as possible. The previous chapter has described several techniques which accomplish this by transforming dispatch into a simple goto-statement. For this to be feasible, both throw site and handler must be in the same function or method; such techniques will work only in cases where this is already true or where the code is transformed to make it true (*i.e.,* method inlining). Method-inlining has been a favoured approach towards improving runtime performance of object-oriented programs [4] because of its elimination of the method-invocation overhead, but there is a space cost. In theory the overhead should be small, especially as object-oriented methods tend to be quite short. However, in practice and without the support of a just-in-time compiler (a JIT) with its access to instruction profiles, the resulting size of the code could grow exponentially.

If we are willing to pay a somewhat higher cost at runtime than that of inlined

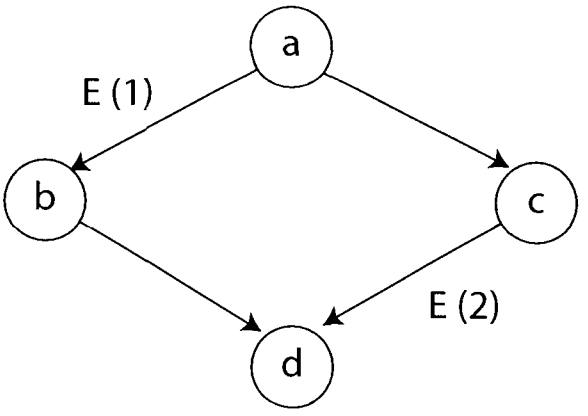


Figure 6.1. *Callgraph with exception-labelled edges: example 1*

techniques, then we could develop a technique for reducing dispatch times for those exceptions propagated outside a function; we must also avoid excessive inlining of recursive methods. Such a technique would also ensure that the space costs remain linear or quadratic in the number of handlers in the program. (We assume here that the dispatch of exceptions with local handlers can be easily converted to gotos.) Some compile-time and link-time support will be needed, but our goal is to devise a mechanism that remains within the domain of "user-pay" EH implementations.

6.1.1 First attempt at a solution

At first glance it appears that we could reduce EH overhead by using interprocedural control-flow analysis [21] [53] plus access to a little bit of runtime state. For example, Figure 6.1 displays a simple call graph involving four functions. Callgraphs as drawn in this chapter consist of nodes (representing functions), and directed edges (edge's tail at caller, edge's head at callee). Each edge corresponds to a single callsite within the calling function. Callsites within `try`-blocks are labelled with the exception types for which there are `catch`-blocks; the numbers in parentheses will be used to refer to specific exception-labelled edges. The topmost node is the single entry point to this program.

If an exception of type `E` is thrown within node `d`—and assuming that `d` does not contain a local handler for `E` — then the handler can be easily determined without any stack unwinding. If the return address stored in `d` indicates an address within

procedure b, then the exception is dispatched to the handler in a indicated by E(1); if the return address is a location in procedure c, then the exception is dispatched to the handler in c indicated by E(2). In the first case, transfer of control is effected by restoring just the state of a—with the state available on a runtime stack—which is the same as executing b's epilogue plus setting the program counter (PC) to the handler's starting location. A similar approach can be applied to the second case. There is no extra overhead for this scheme when procedures are entered or exited normally. Any information needed by procedure d to perform the "if-then"-based search above can be computed at compile time (for statically-linked code) or link-time (for dynamically-linked code). However, at least two issues are ignored in this first attempt: the impact of ad-hoc polymorphism and the possible existence of `finally` blocks.

6.1.2 Next attempt at a solution

Ad-hoc polymorphism exists when a *message send* has more than one possible destination or method. This is seen in object-oriented languages where *abstract classes* may be defined—these are classes which leave some methods undefined save for their signatures. Concrete subclasses of the abstract class are completed implementations of all methods in the abstract class. Code may be then be written assuming that instances of the abstract class exist, and the runtime system converts all uses of such instances to the concrete class instantiated at runtime (*message dispatch*). The canonical example of such an arrangement is a class `Shape` which declares but does not define a `draw()` method; subclasses `Circle`, `Square` and `Polygon` all define their `draw()` methods. A programmer may then construct a list which contains a mixtures of circles, squares and polygons. When iterating through the list to render the shapes, each list object is sent the `draw()` message. However, the data structure used by the programmer is a list of `Shapes`, and the runtime system determines whether or not the message to a particular node is to a `Circle`, `Square` or `Polygon`.

In reality, messages are implemented by methods, which themselves are code sequences. We may then still use the return address to determine the caller of a node. Example 2 in Figure 6.2 has nodes labelled `Y:b` and `Z:b`, where `Y` and `Z` represent classes, both descendants of some class `X`. Node `d` can distinguish amongst these two nodes using the return address. Even if there was no way to distinguish them, both

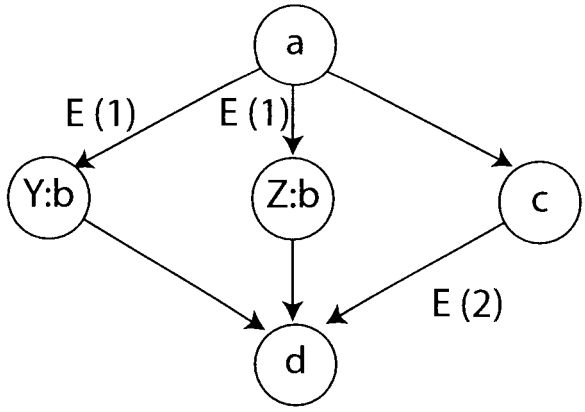


Figure 6.2. *Callgraph with exception-labelled edges: example 2*

`Y:b` and `Z:b` correspond to the very same call-site, in which case they will both have the same set of exception handlers and the issue is moot.

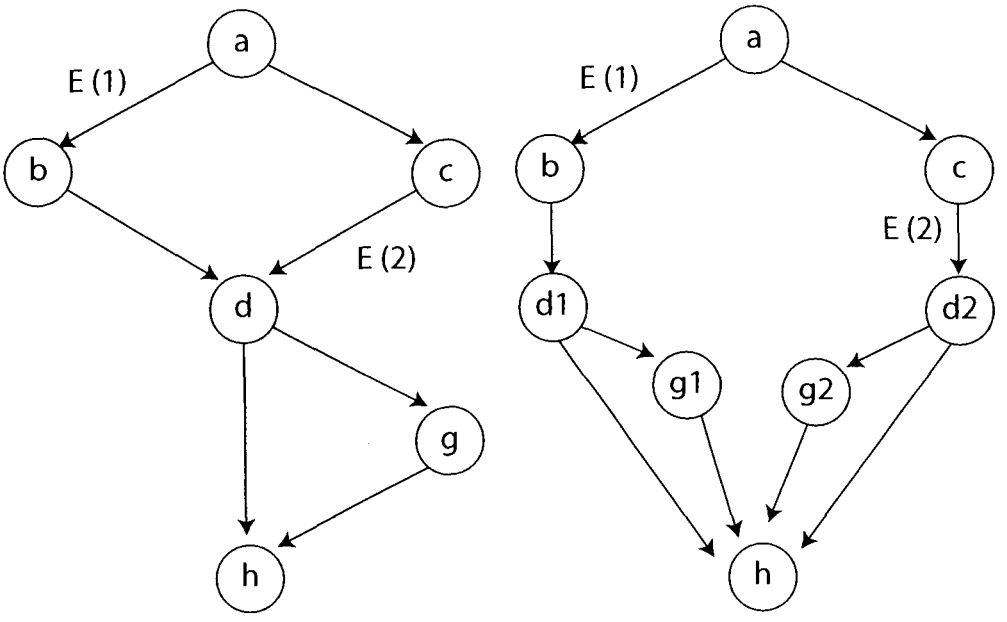
Another more serious issue is the necessity of executing `finally` blocks of the procedures that are no longer unwound as a result of the new dispatch technique. For example, if an instance of `E` thrown in node `d` results in a dispatch to handler `E(1)`, then there may exist at least two `finally` blocks.

a. The statement in `d` which raises the exception may be within a `try`-block having a `finally` clause.

b. One of `Y:b` or `Z:b` will have called `d`, and that callsite may be within a `try`-block having a `finally` clause.

Executing the code from a `finally`-block outside of that block's context is a bit more troublesome—in this example, code within node `b` executed while `d` is active—and requires knowledge of where variables are located in the runtime stack. These details will be available within a compiler and can also be made available to the linker via symbols. The construction of such *far finalizers* will be described in a later section of this chapter.

The callgraphs presented so far have been rather trivial as handlers were determined by looking up a single return address value. These scenarios are far too rosy as our next example shows.



(a) Original (b) With cloning

Figure 6.3. *Callgraph with exception-labelled edges: example 3*

6.1.3 Limits and costs of interprocedural analysis

The call-graph on the left side of Figure 6.3 is an example with a few nodes added to earlier examples. When an exception of type `E` is thrown in node `h`, which handler should the dispatcher choose? Examining the return address stored for `h` does not help as both `E(1)` and `E(2)` are candidates. Nor can we examine the frame-height of stack: heights of 4 and 5 possible, and neither height would give the dispatcher information it needs. We could attempt to perform more control-flow and data-flow analysis to discover if `g` is only reachable via program paths leading through `b` (*i.e.*, given such an analysis, a return address value in `h` for a location in `g` implies the handler is `E(1)`). However, a system supporting the dynamic linking of classes could require the above analyses to be performed each time a class is loaded into a running system.

Another approach would be to clone nodes such that the return address stored within a node always indicates the handler. One such cloning appears in the callgraph in the right side of Figure 6.3. Runtime information regarding the path from node `a`

to h is now encoded into the program such that the handler can found by examining just the return address. (There is no difference between the code found in d1 and d2, or that found in g1 and g2.) The cloning transformation can be very profitable when optimizing a program for some common path, *i.e.,* a path found either during compile-time analysis or the result of runtime profile information.

Here the transform is needed in order to satisfy the needs of our dispatch technique, but the resulting number of cloned procedures could become very large. Such cloning may result in an exponential space explosion [17] as procedure clones beget further procedure clones. This worst case will not, of course, always be the case; optimizations for EH could start by cloning nodes, but would abandon the approach if the transformed callgraph began to exceed some user-specified threshold for memory consumption. We will not pursue this technique further, however, as an additional complication is the need to clone not only for paths but also for exception types, *i.e.,* some paths to a node may indicate a handler for exception type E while other paths may indicate one for exception type F (with the path sets not necessarily disjoint).

Our approach instead will accept the necessity of some stack walking at runtime. Handler reachability information will be used to minimize the number of operations performed during stack walking—along with minimizing related stack examinations—needed when propagating an exception up a call-chain.

6.1.4 A running example

Figure 6.4 presents a callgraph that is used in this chapter's development of a dispatch data structure and dispatch algorithm. More information is added to that in the graph (*e.g.,* line numbers, handler locations, etc.) but is introduced into the discussion as needed. Here are a few initial observations about this example:

- From the point of view of node m there are now four possible handlers for exceptions of type E.
- Procedures h and k are mutually recursive.
- The nodes directly descending from d could represent the ad-hoc polymorphism mentioned earlier in this section. When we examine certain forms of dynamic class-loading, our example will involve the insertion of yet another node between d and h *i.e.,* the instance of a class not in existence when the callgraph was

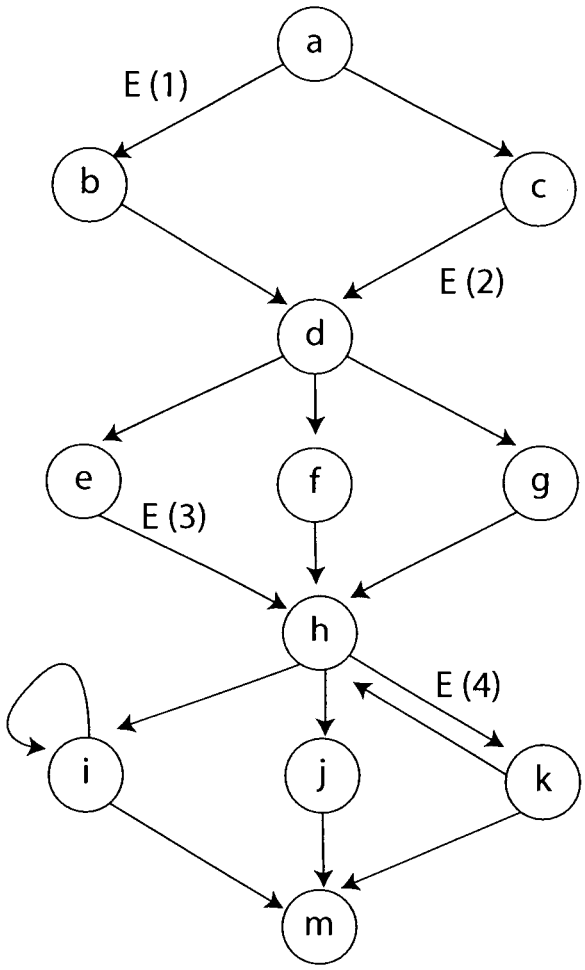


Figure 6.4. *Running example: Callgraph G*

constructed for the original program.

- We are not only interested in interprocedural handler information for node `m` but also for all other nodes in the graph. Therefore our dispatching scheme will construct per-program dispatch tables with handler lookups starting at the appropriate point in the table corresponding to the throwing node.

Section 6.2 lists the information that must be gathered for our proposed dispatch algorithm along with a simple algorithm for the construction of the dispatch table. This is followed in Section 6.3 with a description of the challenge presented by `finally` clauses. Section 6.4 presents details of the dispatch algorithm. The chapter ends with a few optimizations. Section 6.5 suggests techniques that can be used to reduce table size given a large number of exception types, and we end with some discussion of experimental approaches in Section 6.6.

6.2 Capturing program-wide handler information

The callgraph in Figure 6.4 represents a contrived but interesting program. What we wish to build is a table from which we can obtain handler information for a given exception thrown within a node. Note that we do not change the solution for exceptions caught within a handler local to the throwing procedure. Rather, we want to build a program-wide table such that failure to find a local handler results in lookup into this new table. To simplify the exposition, only a single exception type will be considered. Generalizing to multiple exceptions types will be dealt with in a later section.

A table for our example callgraph appears in Table 6.1 where there is a single row for each edge in the graph (plus a row for the distinguished entry node), and columns corresponding to node, return address, and handler information. Although return addresses would be actual memory addresses after linking, they are denoted here as symbols of the form `PC(caller.callee())+4` to indicate the "next" location. (Of course, the next location may not be 4 bytes after the call site; accurate information is available to the compiler.) The address of the handler is denoted by `PC(En)` where `n` is the instance of the exception `E` as it appears in the callgraph; for convenience of exposition, the node in which the handler is located is a prefix of the form `node:`. In the *Dispatch Address* column are two other sets of values. `Null` represents the

Node	Return Address	Dispatch Address	Handler Info
a	—	—	$\mathcal{H}$(a)
b	PC(a.b())+4	a:PC(E1)	—
c	PC(a.c())+4	—	$\mathcal{H}$(c)
d	PC(b.d())+4	a:PC(E1)	—
d	PC(c.d())+4	c:PC(E2)	—
e	PC(d.e())+4	—	$\mathcal{H}$(d)
f	PC(d.f())+4	—	$\mathcal{H}$(d)
g	PC(d.g())+4	—	$\mathcal{H}$(d)
h	PC(e.h())+4	e:PC(E3)	—
h	PC(f.h())+4	—	$\mathcal{H}$(d)
h	PC(g.h())+4	—	$\mathcal{H}$(d)
h	PC(k.h())+4	h:PC(E4)	—
i	PC(i.i())+4	—	$\mathcal{H}$(h)
i	PC(i.h())+4	—	$\mathcal{H}$(h)
j	PC(h.j())+4	—	$\mathcal{H}$(h)
k	PC(h.k())+4	h:PC(E4)	—
m	PC(i.m())+4	—	$\mathcal{H}$(h)
m	PC(j.m())+4	—	$\mathcal{H}$(h)
m	PC(k.m())+4	—	$\mathcal{H}$(k)

Table 6.1. *Program-wide handler table for Callgraph G*

information that no handlers for exception type `E` exist (other than those defined locally) for the callgraph edge represented by the row. The notation $\mathcal{H}(\mathtt{n})$ represents a form of indirection—if it appears in some row, the handler information for those rows are contained in rows for which the `Node` field has the value `n`.

The table is constructed by a three-pass procedure:

a. Values of *farhandler*() are computed for every node. Each set corresponds to a specific node and is a set of handlers, *i.e.,* all candidates to catch an exception of type E thrown in the node.

b. Values of *mergehandler*() are then computed for each node. When there is more than one control-flow path through a node (*i.e.,* more than one edge entering a node), precise information about reachable handlers *may* be lost. *mergehandler*() is used to help ensure that reachability information is as precise as possible.

c. Lastly the table itself is constructed row by row, one callgraph edge to a row, with the *Return* and *Dispatch address* fields filled in.

One short comment about this three-pass procedure is in order. The arrangement of nodes and edges in the callgraphs is reminiscent of basic-block diagrams. A temptation is to view the necessary computations as simply a flavour of *reaching definitions* analysis [2]. For instance, exception types could be modelled as global variables (one variable per exception), with calls from within `try`-blocks modelled as assignments to the global variables for which `catch`-blocks exist. Determining the handlers accessible from a node is "simply" computing the reaching definitions of the "exception-type" global variables at that node (along with computing `gen` and `kill` sets); the definitions of the exception global variable reaching the throw site denote the handlers. Alas and alack, it is not so easy as this. Edges correspond to call sites and hence are located anywhere within the corresponding procedure, *i.e.,* not necessarily the exit point. Actually the computation described in the next section is actually *simpler* than a reaching definitions analysis, but by themselves the *farhandler*() sets do not tell us everything we need to compute the far-table.

The notation for callgraphs here is that used by Muchnick [52]. Very briefly, this consists of:

- a program P consisting of procedures $p_1, \ldots, p_n$;

- a graph $G = \langle N, S, E, r \rangle$;
- the node set N is defined as the set of procedures $\{p_1, \ldots, p_n\}$;
- the set S of call-site labels which are addresses (or may be line numbers if this is appropriate);
- the set of labeled edges $E \subseteq N \times S \times N$; and
- the distinguished entry node $r \in N$ representing the main program.

Each edge $e = \langle p_i, s_k, p_j \rangle$ denotes a call from p_i to p_j that occurs at callsite s_k (*i.e.*, p_j may be called in several different places within p_i, and these calls must be kept separate).

6.2.1 Pass 1: *farhandler*()

Two support functions, *handlers* and *fallthrough*, are needed to define *farhandler*. The first returns the set of all handlers associated with a callgraph edge, and the second returns a set of nodes.

- *handlers*(e): This represents the set of tuples having form $\langle type, codepos \rangle$ where $e \in E$. If $e = \langle p_i, s_k, p_k \rangle$ and a `try`-block contains callsite s_k in procedure p_i, then the set contains the exceptions for which `catch`-blocks exist. Tuples denote the starting address of the catch-block plus the exception handled by that block. *Our initial algorithm handles the case of a single, program-wide exception type*, so *handlers*(e) should be either a singleton or the empty set.
- *fallthrough*(n): This is the set of all predecessor nodes m with an edge e to node n such that *handlers*(e) $= \emptyset$. Intuitively this describes paths through the callgraph where all of the handlers reachable from m are also reachable from n.

$$fallthrough(n) = \{m | \langle m, s, n \rangle \in E \wedge handlers(\langle m, s, n \rangle) = \emptyset\}$$

We can now describe the set of all non-local handlers reachable from a node n, denoted as *farhandlers*(n). For each node, the set consists of subsets of handlers:

a. Those attached to `try`-blocks enclosing the caller's end of an edge.

b. Those reachable along all other input edges incident on n.

In effect the first subset simultaneously acts as a *gen* and *kill* on all of the handlers reachable from nodes m calling n, while the second subset passes handler sets through

node	*farhandler(node)*
a	$\emptyset$
b	$\langle a, E(1), b\rangle$
c	$\emptyset$
d	$\langle a, E(1), b\rangle, \langle c, E(2), d\rangle$
e	$\langle a, E(1), b\rangle, \langle c, E(2), d\rangle$
f	$\langle a, E(1), b\rangle, \langle c, E(2), d\rangle$
g	$\langle a, E(1), b\rangle, \langle c, E(2), d\rangle$
h	$\langle a, E(1), b\rangle, \langle c, E(2), d\rangle, \langle e, E(3), h\rangle$
i	$\langle a, E(1), b\rangle, \langle c, E(2), d\rangle, \langle e, E(3), h\rangle$
j	$\langle a, E(1), b\rangle, \langle c, E(2), d\rangle, \langle e, E(3), h\rangle$
k	$\langle a, E(1), b\rangle, \langle c, E(2), d\rangle, \langle e, E(3), h\rangle, \langle h, E(4), k\rangle$
m	$\langle a, E(1), b\rangle, \langle c, E(2), d\rangle, \langle e, E(3), h\rangle, \langle h, E(4), k\rangle$

Table 6.2. *Value of* farhandlers() *for callgraph G*

unchanged. For each node n in the callgraph, the following flow equation must be solved:

$$\begin{aligned} farhandlers(n) \;=\; & \{h | \langle m, s, n\rangle \in E \wedge h \in handlers(\langle m, s, n\rangle)\} \\ & \cup \left(\bigcup_{m \in fallthrough(n)} farhandlers(m) \right) \end{aligned}$$

The initial value of *farhandlers*() for each node is the empty set.

The value of *farhandlers*() for callgraph G appears in Table 6.2.1.

6.2.2 Pass 2: *mergehandler*()

Although we now have a way of determining which handlers are reachable from any node, this information is not necessarily of use at run-time. For example, crunching through the equations will yield for callgraph node `m` a set containing *all* the handlers for `E`. However, if an exception `E` is thrown in node `m`, the only *acceptable* way to determine if `E(1)` is a handler is to examine the runtime stack. *Acceptable* here means that we may not add any instructions that will be executed at runtime when procedures are called even if this would help the dispatcher determine in constant time which path is taken from node `a` to `m`.

In our example, we could determine the handler if we knew the following four items, scanning the callgraph from the bottom up:

a. Was `m` called from `k`? If so, then `E4` is the handler.

b. Otherwise was `h` called from `e`? If so, then `E3` is the handler.

c. Otherwise was `d` called from `c`? If so, then `E2` is the handler.

d. Otherwise `E1` is the handler.

This example demonstrates that only certain parts of the callstack need to be closely examined to determine the handler. As we are attempting to reduce dispatch overhead as much as possible, any extra operations performed when stack-walking must be eliminated. We can achieve some of this by limiting the nodes at which we must perform operations such as searching a table for a handler.

The use of *mergehandler*() sets is one aid towards this. Nodes which have more than one input edge could reduce the precision of handler info for nodes "deeper" in the callgraph. For example, dispatching an exception thrown in node `e` cannot just use the return address to determine the correct handler; it must go further and examine the return address at node `d`. Three cases are presented to the function; in the description below, the *mergehandler* value is computed for node `dest`.

a. If one of the incoming edges to `dest` has `dest` as its source and $handlers() = \emptyset$, then this edge is not considered when computing *mergehandlers(dest)*.

b. If there exists one or more incoming edges to `dest` for which $handlers() \neq \emptyset$, then the value of *mergehandler()* for `dest` is set to `dest`.

c. If all incoming edges to `dest` have $handlers() = \emptyset$, and all source nodes have the same value for *mergehandler()*, than `dest`'s value is set to that of any one of the *source* nodes.

The value of *mergehandler*() does not depend on *farhandlers*(). The initial value of *mergehandler*() for each node is that of the node itself. Computation is by *procedure invocation order*. In the case of nodes such as `i` in callgraph G, recursive edges are not considered. Table 6.3 contains the values for callgraph G; each iteration of the computation is shown. (Iteration 1 and iteration 2 are the same; at this point the computation of *mergehandler* stops.)

node	*mergehandler*		
	initial	iteration 1	iteration 2
a	a	a	a
b	b	b	b
c	c	a	a
d	d	d	d
e	e	d	d
f	f	d	d
g	g	d	d
h	h	h	h
i	i	h	h
j	h	h	h
k	k	k	k
m	m	m	m

Table 6.3. *Callgraph G: mergevalues*

6.2.3 Pass 3: Table construction

Now that we have both *farhandlers* (mapping nodes to handler set) and *mergehandler* (containing names of individual nodes) we are almost ready to construct a table such as that in Table 6.1. We need two more support functions:

- *returnaddress*(e): With $e = \langle p_i, s_k, p_j \rangle$, the value of this function is the instruction after s_k in p_i, *i.e.,* where control-flow resumes after the call to p_j returns. As the exact layout of code in memory may not be known until link-time, the compile-time version of this function can emit symbols; these symbols can then be resolved into addresses at link-time.
- *sorttable*: Sorts the dispatch table first on the node field, then the return address, both in ascending order.

There is one row per callgraph edge, with a single extra row for the distinguished graph node edge r (*i.e.,* entry node). The algorithm is described below:

- Insert row $\langle r, null, r \rangle$ into table.
- For each edge $e \in E$
  a. Create a new table row.
  b. In the first field of the row, set the value of *node* to p_j where $e = \langle p_i, s_k, p_j \rangle$.
  c. In the second field of the row, set the value of *returnaddress*(e).

d. If $handlers(e) \neq \emptyset$, then $handlers(e) = \{h\}$ (a singleton set) where h is a tuple consisting of $\langle type, addr \rangle$ and $addr$ is the starting address of the handler. Set the row's *dispatch* field to $addr$ and the *handler* field to *null*. PROCESS NEXT EDGE.

e. If $|farhandlers(p_j)| = 1$, then $farhandlers(p_j) = \{\langle type, addr \rangle\}$ (a singleton set). Set the row's *dispatch* field to $addr$ and the *handler* field to *null*. PROCESS NEXT EDGE.

f. Otherwise set the row's *dispatch* field to *null* and the *handler* field to $\mathcal{H}(mergehandler(p_i))$. PROCESS NEXT EDGE.

- Sort the table using *sorttable*().

The table generated by this algorithm is that seen in Table 6.1.

6.2.4 Extending to more than one exception type

Most programs utilizing exceptions use more than one exception type, so for our scheme to be useful we will need to support more than just some type E. Fortunately very little needs to be changed.

- *farhandlers*() already consists of sets of handlers, and the exception type is stored in each handler tuple.
- *mergehandler*() is now parameterized by exception type and node. The various cases described in Section 6.2.2 must now also be specific to an exception type (*i.e.,* of handlers, *mergehandler* values in source nodes).
- The biggest change is that each exception type now corresponds to an additional column in the far-table (for *Dispatch address*).

This suggests that the space requirement for the far-table is of order $O(|E||\mathcal{E}|)$ where E is the number of callgraph edges and $\mathcal{E}$ is the set of all exception types, but in a later section we will investigate ways in which this can be improved.

6.3 The trouble with `finally`

Consider the code shown in Figure 6.5.

```
1   public class A {
        static int m = 10;

        int n = 0;
5
        public int select() {
            n = 5;
            return m;
        }
10
        public void proc1 () {
            try {
                if (some_condition) proc2() else proc3();
            }
15          catch (Exception e) {
                // Code here...
            }
        }

20      public void proc2 () {
            int p = 20;

            try {
                proc4();
25          }
            finally {
                p = 25;
                n = 10;
                m = 5;
30          }
        }

        public void proc3 () {
            int q = 30;
35          int r = 50;
            int s = 70;

            try {
                proc4();
40          }
            finally {
                q = this.select();
                m = 6;
            }
45      }

        public void proc4 () {
            int t = 20;
            throw new Exception();
50      }
    } // end of class
```

Figure 6.5. *Example of interaction between* `throw` *and* `finally`

The resulting callgraph will have four nodes. At compile time we know that `proc4` can be reached from `proc1`, and that when the exception is thrown in the latter, the former has a handler which will catch the exception.

More troublesome, however, are the `finally` clauses. When the exception is dispatched, flow-of-control exits the intervening try-block (either in `proc2` or `proc3`). This code must be executed regardless of whether the `try`-block is left as a result of an exception or left as a result of normal control flow. The issue here is not with the overhead of the `finally` block, but rather with the cost of restoring activation frames as the exception is propagated. But how can the `finally` code be executed otherwise? Another way of stating this is how we can access the variables of either proc2 or proc3 while we are in proc4?

We could make the following assumptions:

a. A local variable is either stored on the stack or available in a register. If it is stored on the stack, then its offset from the stack bottom *when the procedure is active* is known at compile time, and to access it from `proc4` we need only subtract `proc2`'s or `proc3`'s stack size from the current stack bottom (to adjust for the movement of "bottom" of the stack) and add the offset. If available in a register, then this is because caller-save and callee-save operations have not stored this variable to the stack and the register can be accessed. A complication is the possibility that the variable is stored on the stack but not in the portion corresponding to `proc2` or `proc3` but rather in callee-save locations of `proc4`. Somehow we must be able to determine this possibility at compile time and adjust for it. (The dynamic link might help here at run-time, but we would prefer to compute as much of the offset information as possible at compile time.) Although against our principle of "user-pay", we could require that the compiler ensures all variables referenced in a `finally` block and located in registers bbee saved into the runtime stack.

b. Instance variables may be manipulated in the same way that they are in `proc2` or `proc3`—that is, via accessing "this". We assume that "this" is a local variable in `proc2` or `proc3`, or is an implicit parameter to a procedure call, or is otherwise obtainable from the stack. The same applies to calling instance methods. This should be true even if it were the case the `proc4` was in a different class from

`proc2` or `proc3`.

c. Static variables may be manipulated in a similar manner to (b), *i.e.*, the way they are within `proc2` and `proc3`.

d. Global variables are trivially accessible in the finalizer.

In order to make our example more concrete, a simple Application Binary Interface (ABI) is suggested. The ABI arranges the stack as follows:

- byte 0: return address in caller
- byte 4: dynamic link to caller's stack bottom
- byte 8: "this" pointer
- byte 12 to (n*4+12): n parameters
- next m * 4 bytes: m callee-save registers
- next v * 4 bytes: v local variables

If we assume that our example has no callee save registers, then:

a. `proc2`: variable p is 12 bytes from its frame's bottom;

b. `proc3`: variable q is 12 bytes, variable r is 16 bytes, and variable s is 20 bytes from their frame's bottom.

This yields a snapshot of the stack as seen in Figure 6.6 taken just before the exception is thrown. The procedures `proc1`, `proc2` and `proc4` are active; addresses are started at 0 for convenience.

To access variable `p` within `proc4`, we take `proc4`'s stackbottom (028), and adjust it for `proc2`'s stackheight (less 16) to give us `proc3`'s stack bottom (012). We can then add the offset for `p` (12 in this case). To access instance variable `n`, we can make similar computations as for `p` in order to obtain "this", and then use this to update the value of the `n` field for the instance. To access static variable `m`, we obtain "this", use it to obtain a pointer to the class, and then update the class variable `m`.

Some detail has been spent on this example in order to motivate some of the state that must be maintained by the dispatch algorithm of the next section. Even though we can use the far-table to aid dispatching of the exception, we must still execute the finally clauses "on the way up", and preferably do this without restoring activation frames save that of the exception handler's context.

```
000: return address to proc1's caller
004: dynamic link to proc1 caller's stackbottom
008: pointer to instance of A receiving message "proc1"
---
012: return address to proc2's caller (proc1 in this case)
016: dynamic link to proc2 caller's stackbottom (000 in this case)
020: pointer to instance of A receiving message "proc2"
024: variable p
---
028: return address to proc4's caller (proc2 in this case)
032: dynamic link to proc4 caller's stackbottom (012 in this case)
036: pointer to instance of A receiving message "proc4"
040: variable t
```

Figure 6.6. *Snapshot of stack from* `finally` *example*

6.4 Dispatch algorithm

We now have nearly all of the information and functions needed to express the dispatching algorithm. In very general terms the dispatcher must perform the following three tasks, possibly interleaved with each other:

a. Find a handler for the thrown exception.

b. Execute `finally`-blocks on the way to handler.

c. Restore system state needed to begin executing the handler.

Finding the handler may involve several lookups in the far-table as the stack is walked, while *executing* `finally`*-blocks* must be performed for every exited `try`-block with a *finally*-block. The number of comparisons and computations required during handler dispatch should also be kept as low as possible.

A few extra support functions are needed by our algorithm. Some of these functions are used to extract properties of the code which are static, and hence may be stored in tables generated at compile-time.

- *finallycode*(*addr*): If there exists some edge $e = \langle p_i, s_k, p_j \rangle$ in the callgraph such that *addr* corresponds to s_k, then s_k may be in a `try`-block with a `finally` clause. This function returns the instructions in such a `finally` block with variable references transformed to offsets from the bottom of the activation frame (denoted in the code as `stackbottom`). Another way of stating this is that all variable references are now converted into accesses of some global array

Stack (the runtime stack). (Our algorithm will execute these code sequences only if the throwing exception is propagated out of the procedure. Therefore nested `try`-blocks around s_k can result in the corresponding `finally`-blocks being concatenated together and returned as a single code sequence.) If no `finally` clauses exist, then the function returns a code sequence with the single `nop` instruction.

- *epilogue*(n): With $n \in P$, this function returns the code that would be executed when an invocation of n returns to its caller. As we do not expect callee-saved registers, this should be a `nop` code sequence, but extensions to be proposed for *finallycode*() will deal with such registers and we therefore keep the more general definition for *epilogue*.
- *frameheight*(*addr*): If there exists an edge $e = \langle p_i, s_k, p_j \rangle$ such that *addr* corresponds to s_k, then the function returns the size of the activation frame at s_k in p_j. (The stack may have different heights in the same function as a result of nested blocks.) The result of the function is needed to update `stackbottom` as used in the dispatch algorithm.
- *runcode*(c, *stackbottom*): Given a code sequence c and a memory location for the bottom of the stack, execute the code sequence. It returns `true` if the code threw an exception, otherwise it returns `false`.
- *farlookup*(n, a, e), where $n \in N$ (*i.e.*, the callgraph), a is from the set of memory addresses, and $e \in \mathcal{E}$: This lookup returns a tuple of the form (*addr*, *handlerinfo*); *addr* is the address of the first instruction in a handler, while *handlerinfo* is the value of $\mathcal{H}(\mathtt{m})$. Either of these values can be null, but not both; if *addr* is null, then there is no handler attached to the edge. (If the table row refers to a recursive call of a method to itself where the edge has a handler, then both *addr* and *handlerinfo* will be other than null.) Parameters n and a together specify the row, and e specifies the column, in the far-table for the program.
- *node*(*addr*): Given an address *addr* in the program, return the node $n \in N$ corresponding to the address. This can be computed at link time, and as the addresses that will be passed to this function are a subset of all addresses in the program, we can improve the speed of the function.

- *getretaddr*$(a, Stack)$: Given an address corresponding to the bottom of a stack frame and the runtime stack, obtain the address to which the current activation will return upon procedure exit. (Values from this function will make up one part of the values used to lookup far-tables.)
- *defaultaddr*(e): With $e \in \mathcal{E}$, the function returns the address of the first instruction in the default handler for e. If there is no default handler, then the value returned is the address of code for terminating the program (*i.e.*, the entry point to a function like *exit*()).

The dispatch algorithm is presented in Figures 6.7 and 6.8. There are four states:

1: An exception has just been thrown and algorithm is about to begin locating a handler.

2: The dispatcher is at a callgraph node for which handler information is available earlier in the callpath.

3: The dispatcher has precise handler information—a handler exists and its location is known.

4: The dispatcher has precise handler information—a handler does not exist for the exception.

The loop beginning at line 1 is an infinite loop, and exiting can only occur from one of four places:

a. *line 7*: When executing the `finally`-block of some procedure activation, an exception was thrown. What should happen at this point depends on the language semantics, but for now this algorithm transfers control to the label `PANIC` at line 33.

b. *line 10:* No handler has been found for the exception. A default handler is found for the exception (which may just be a call to some `exit` function), and the address of this handler is returned by the algorithm.

c. *line 20:* A handler has been found for the exception, and control is about to be transferred to that context. However, an exception is thrown by the epilogue of the procedure that was called by the handler's procedure. Control is transferred to line 33.

- Input:
  - PC, the address of the instruction which has thrown the exception
  - $\mathtt{e} \in \mathcal{E}$, the type of the thrown exception
  - r, the distinguished entry node of the program's callgraph
- Output:
  - Address of the first instruction, with program state for handler's procedure context.
- Variables:
  - A far-table T used by *farlookup*() (there is one far-table per program)
  - *StackBottom* : *Addr*
  - *Stack* : *CallStack*
  - *da* : *Addr* (dispatch address)
  - *h* : *HandlerInfo* (handler entry)
  - a, ra : *Addr* (temporary addresses)
  - n, pn : N (temporary integers)
- Support functions:
  - *defaultaddr* : $\mathcal{E} \rightarrow$ *Addr*
  - *epilogue* : $N \rightarrow$ *CodeSeq*
  - *farlookup* : $N \times$ *Addr* $\times \mathcal{E} \rightarrow$ *Addr* $\times$ *HandlerInfo*
  - *finallycode* : *Addr* $\rightarrow$ *CodeSeq*
  - *frameheight* : *Addr* $\rightarrow$ *Integer*
  - *getretaddr* : *Addr* $\times$ *CallStack* $\rightarrow$ *Addr*
  - *node* : *Addr* $\rightarrow N$
  - *runcode* : *CodeSeq* $\times$ *Addr* $\rightarrow$ *Boolean*

Figure 6.7. *Algorithm:* DISPATCH 1 *(pars prima)*

d. *line 30:* A handler has been found for the exception, and the address of the first instruction in the handler is to be returned by the algorithm (presumably to be assigned to the PC).

The algorithm has been written such that there are a reduced number of operations performed when unwinding through activation frames between merge nodes and between a merge node and a handler.

6.5 Optimizations

Given that far-tables have a row for each exception type, there may be some concern about the space requirements of such tables. As this thesis advocates the increased

```
    loop
      a ← PC
      n ← node(a)
      ra ← getretaddr(a, Stack)
 5:   (da, h) ← farlookup(n, ra, e)
      if da is null and h is null then
        goto PANIC
      end if
      if n = r ∧ da is null then
10:     return defaulthandler(e)
        break
      end if
      if h is null then
        goto TryDispatch
15:   end if
      repeat
        c ← finallycode(a)
        th ← runcode(c, stackbottom)
        if th = true then
20:       goto PANIC
        end if
        stackbottom ← stackbottom − frameheight(a)
        pn ← n
        a ← ra
25:     ra ← getretaddr(a, stackbottom)
        n ← node(a)
      until h ≡ n
      TryDispatch:
      if da is not null then
30:     return da
      end if
    end loop
    PANIC:
```

Figure 6.8. *Algorithm:* DISPATCH 1 *(pars secunda)*

use of exceptions, one pleasant future outcome would be that the number of different exception types in any program could become larger and larger. Is there any way to reduce the table's memory footprint?

One of the intriguing aspects of exception dispatch is that it is like message dispatch, albeit in reverse. *Dynamic message dispatch* uses the run-time type of an object receiving a message to determine which implemented method is invoked for that message (*e.g.,* for polymorphic callsites). *Exception dispatch* uses the call-path implied by a stack of callframes to determine where a thrown exception is handled. Can techniques used to reduce the size of tables for message dispatch, as described by Vitek [70] and others, be used to reduce the size of our tables?

The answer appears to be "yes". If two columns have the same values for all rows, then the two columns can be merged, and the column labeled with the union of the exception types labelling the columns. Similarly if two rows have the same values for all columns, then those two rows can be combined together. Of course the degree of such *compaction* applied to the table must be paid for in a somewhat slower implementation of *farlookup*(). We could continue borrowing ideas from the object-oriented language designers, such as cacheing the results of lookups, in the anticipation that the paths of exception dispatches are repeated frequently enough to warrant the overhead of examining such caches (as suggested by the experimental results presented by [54]).

Another possible optimization involves code in `finally` blocks. At throw time we cannot know the exact sequence of such blocks even if the handler is known—there may exists many paths from the sole handler to the throw site. Assuming that we are willing to consume more space, however, we could store concatenations of `finally` blocks, and then apply optimizations such as *dead-code removal* on these instruction sequences. If we defer running the optimized sequences until the handler site is reached in the unwinding, then we can choose the correct sequence from a table indexed by some encoding of the path from throwsite to handler. The profitability of such a scheme naturally depends on the degree to which `finally` blocks are used.

6.6 Summary

This chapter has described a technique for improving the performance of exception dispatch when the handler is known to reside outside the throwing method. There are, however, even more techniques we can apply within a procedure to reduce exception handling overhead in Java. These are the topics of the next chapter. Once we have both sets of techniques—for handlers external to the throwing procedure and for those internal—we can then examine the improvements these provide to actual code.

Chapter 7

Improving EH performance: Reducing Exception Object Overhead

This chapter presents three new techniques for improving EH performance:

- *throw-site reduction*,
- *lazy stack-trace generation*, and
- *lazy exception-object creation*.

All three are an attempt to reduce the significant amount of effort wasted in most programs that use exceptions. This waste is described in the next section.

7.1 Motivation

The previous chapter focussed on reducing the cost of transferring control from throw site to handler. In languages such as Java, however, exceptions are themselves objects, and the runtime cost for constructing such objects is significant. This cost may be broken down by the several steps taken in standard JVMs when `new` is invoked for a subclass of `Throwable`:

- Acquire memory from heap for the object.
- Construct a stack trace and copy this into the object.
- Copy any user-supplied parameters into the new object instance.

All of these operations are performed by the JVM through native methods such as `fillInStackTrace()`, and therefore run at native machine speeds (*i.e.*, there is no bytecode interpretation overhead).

We can get a sense of the expense of such operations by obtaining timings from various operations performed by a JVM—with the VM usually written in a language such as C or C++—and then execute a contrived Java program that repeatedly throws an exception. Given a particular stack depth (*i.e.,* number of activation frames separating the throw site from its handler), our experiment measures the following quantities:

- Time taken to normally unwind a stack of depth n a total of m times *without exceptions* (`unwind`);
- Time taken to create an exception object a total of m times at a stack depth of n, *i.e.,* unwinding time not measured (`construct`);
- Time taken to construct the stack trace for an object at a stack depth of n a total of m times (`fill`); and
- Time taken to unwind a stack of depth n a total of m times using a newly constructed exception for each of the m iterations (`all`).

With these measurements we can estimate the percentage of time spent in each of these activities when handling an exception:

(a) Unwinding the stack, but without the time taken for in handler-table lookups: $100 \times$ `unwind/all`

(b) Build the exception object but without a stack trace: $100 \times$ (`construct` − `fill`)/`all`

(c) Construct a stack trace: $100 \times$ `fill/all`

(d) All other activities (*e.g.,* handler-table lookup, type equivalency tests, other JVM actions, etc.): $100 \times$ (`all` − `construct` − `unwind`)/`all`

Figure 7.1 illustrates the various costs and where they would appear in the midst of a program's execution.

In the case where we need to control stack-trace generation, the experiment creates a single instance of an exception per experiment; the `fillInStrackTrace()` method is then used when we need to measure such overheads. Each run of the experimental code uses a large number of iterations to produce significant timings. Tables 7.1, 7.2 and 7.3 contain results for stack depths n of 1, 10 and 20 along with different values of m at 500K, 1000K, 2000K and 5000K. In all tables, the units of times

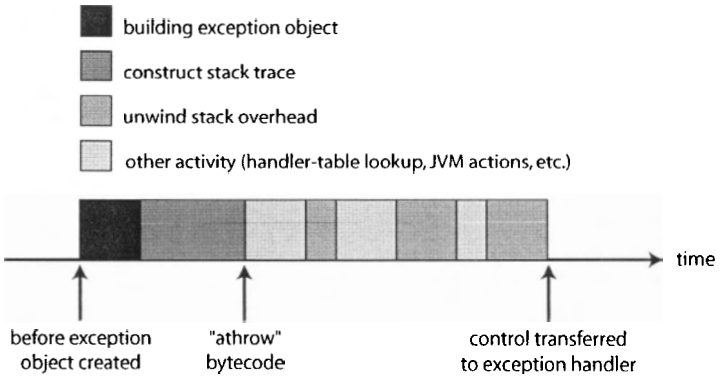


Figure 7.1. *Exception-handling activities*

#	millisecs				phase (out of 100)			
depth	`base`	`construct`	`fill`	`all`	a	b	c	d
500,000	28	2231	1929	2525	1	12	76	11
1,000,000	48	4444	3860	4967	1	12	78	10
2,000,000	92	8909	7715	9888	1	12	78	9
5,000,000	228	22184	19318	24691	1	12	78	9

Table 7.1. *Exception-handling overheads, stack depth 1*

are milliseconds as reported by the Java virtual machine (Pentium 4 at 1.3 GHz, 256MB RAM, Windows XP Professional, Java 1.4.2, no JIT, simple average of five runs). Table columns are labelled with the values given in the above two lists—"base" (or unwind-only cost), "construct", "fill", "all", and activities (a) through (d). The proportions are reported as out of 100 where 100 represents the time taken for `all`; a row might not add up to 100 due to rounding. Note that the numbers presented below are intended to be indicative of overall costs rather than precise measurements.

#	millisecs				phase (out of 100)			
depth	`base`	`construct`	`fill`	`all`	a	b	c	d
500,000	60	3102	2868	5303	1	4	54	40
1,000,000	121	6193	5730	10651	1	4	54	41
2,000,000	234	12382	11459	21245	1	4	54	41
5,000,000	589	30918	28669	52702	1	4	54	40

Table 7.2. *Exception-handling overheads, stack depth 10*

#	millisecs				phase (out of 100)			
iterations	`base`	`construct`	`fill`	`all`	a	b	c	d
500,000	128	4262	3805	6779	2	7	56	35
1,000,000	250	8520	7669	13529	2	6	57	35
2,000,000	499	17033	15228	27019	2	7	56	35
5,000,000	1238	42561	38057	67507	2	7	56	35

Table 7.3. *Exception-handling overheads, stack depth 20*

Keeping in mind that these figures have been gathered by using one specific implementation of a JVM, a few observations can be inferred from the tables.

- The effort required for unwinding the stack (columns marked "a")—outside of any exception activity—is swamped by the exception-specific effort to a factor of 50 to 100.
- For the smallest stack depth, the effort to create a stack trace (columns marked "c") comprises nearly three-quarters of the effort to throw the exception (columns marked "d"). Even as the stack depth increases (and assuming that a depth of 20 is rarely seen in production code) stack-trace generation still consumes over 50% of the total effort.
- Creating an exception object (columns marked "b") is a relatively small but significant part of the effort to throw an exception.

One surprise from these results is the relatively large magnitude of values in column "d", *i.e.,* everything left over after stack unwinding, object creation and stack-trace computing is done. The sample code used to generate these figures is very simple—only one method contains the handler, while everything other method invocation on the stack has no handlers. The "d" activity for the sample code should therefore be quite simple—there are no tables to search, no `finally` clauses to execute, no type tests to perform—yet the overhead is still relatively high. We suspect this is due to the decisions made by implementors of the HotSpot Java VM (which is used in SDK 1.4.2) that VM routines for exception handling not be optimized. This seems plausible given that:

- Using the HotSpot VM 1.4.2's JIT to generate results produces significantly *higher* values for the "`all`" columns. For example, at a stack depth n of 10

and a runlength m of 5000K, the JIT version of the code requires over 63000 milliseconds to complete, compared with the 52700 shown in Table 7.2.

- Other JVM implementations show a much different division of overall effort between stack-trace generation and other activities. For example, the same row from Table 7.2 referred to above has a value of 54% for stack-trace generation; results from a Pentium 200MMX using Windows NT 4.0 and the 1.2.2 version JVM produces a range of 14% to 21% for stack-trace generation.

We do not wish to criticize the implementors of the HotSpot JVM. Given the nature of most Java programs and benchmarks, these programmers have probably made appropriate implementation tradeoffs that (unfortunately) decrease EH performance.

Despite these surprises, we *can* conclude that reducing the effort in "d"—which was the focus of the previous chapter—will produce a benefit. An even more pleasant outcome of this brief study, however, is that the effort required for both "b" and "c" can quite often be eliminated entirely! Disassembling class files and reading through bytecode for exception handlers reveals that the first instruction in most handlers is a `pop` instruction. This is significant because Sun's JVM specification requires that the only object on a VM's operand stack at handler entry must be the exception object—and yet most handlers discard this object at the first opportunity. Of course, programmers do not themselves choose to discard these objects, but programmers using exceptions solely for control-flow transfer will not write handlers which use the exception objects. The compiler simply takes advantage of this by popping off the object, hence helping to reduce the size of the operand stack.

What the following sections describe are several new techniques for exploiting this *non-use* of exception objects:

- *Throw-site reduction* identifies throw sites—or regions of code which may throw implicit exceptions such as object dereferencings—which are guaranteed to have handlers that discard the exception object. Therefore these throw sites need neither generate a stack-trace nor create an exception object.
- *Lazy Stack-Trace Construction* defers stack-trace generation until the point in time at which the trace is needed. Such stack traces are most often needed during debugging, and in production code could provide useful data when *computational reflection* is used. In both cases there usually does not exist a need

for high performance, and therefore our technique transfers the cost to code that uses stack traces.

- *Lazy exception objects* is the name given for our technique that covers cases not amenable to throw-site reduction. This would occur where at least one handler for a throw-site uses the exception object even if all others do not. Here the exception object is created just before its first use—*i.e.,* messages sent to the exception object within a handler are actually messages to clone a static object, with the clone created at the point in time when the handler is entered.

The implementation of these analyses as used to produce the experimental results of the next chapter do require the whole program. If analysis is to be performed only upon class loading, then incremental versions of these analyses would only be made more complicated if classes were *not* loaded from the top-down (*i.e.,* from the `Main` class to those classes containing leaf methods). This complexity results from the lack of information available at a throw-site: some methods along some callpath to the throwsite might not yet have had their classes loaded.

7.2 Throw-site Reduction

We have observed that many uses of the Java exception idiom are solely for transfer of control—the first instruction in handlers written for such use will "pop" the exception object from the operand stack. This is the same as discarding the object, and hence such exception objects need not be created. *Throw-Site Reduction* (*TSR*) works by modifying the code around throw sites when it is safe to do so, *i.e.,* when all handlers reachable from the throw site are known to discard the exception object. If we eliminate the object's construction, the runtime system must use information other than the exception object when transferring control to a handler. To satisfy the needs of the handler's "pop" instruction, a null object may be pushed onto the stack in place of the exception object.

There are also a few other assumptions that must hold true for TSR to work. In the case of explicit `throw` statements, the throwsite identified by TSR for optimization must correspond in Java bytecode to an *athrow* instruction that follows shortly after the exception object's creation. For example, the four-bytecode fragment below is emitted by `javac` for the instruction `throw new NullPointerException()`:

```
new java/lang/NullPointerException
dup
invokespecial java/null/NullPointerException/<init>()V
athrow
```

The first line is the actual construction of the exception object. This is followed by `dup` and `invokespecial`, a `javac` idiom for invoking the constructor on new objects (in this case the constructor accepts no parameters and returns a `void` type, denoted by "()" and "V" respectively). The `athrow` statement then uses the object to trigger the JVM's exception handling mechanism. TSR must suppress the three instructions that appear before the `athrow`; if we are guaranteed that the instruction sequence shown is always generated, then standard peephole optimization can be used to support our optimization. Unfortunately this is a bit too optimistic as the following Java code illustrates:

```
Exception e;
if (condition) {
    e = new NullPointerException();
} else {
    e = new MyNullPointerException();
    e.setTimestamp (System.currentTimeMillis());
}
throw e;
```

This example illustrates code where separate paths reach a `throw` statement from different `new` statements. Even if analysis determines that the `throw` instruction will be caught by a handler discarding the exception object, there still remains the troublesome `setTimeStamp` method call. In effect, regardless of whether or not TSR determines that all handlers reachable from the `throw` statement discard the exception object, the exception object *must* be created in this case because of the use for `setTimestamp`. (If `condition` were determined to be always `true` by some other analysis, then we could come to a different conclusion here about exception object creation.) We therefore assume that TSR is only applied to throwsites where there exists no use between all `new Exception` statements that can reach a `throw` and the `throw Exception` instruction itself. Instructions that may throw an exception but which are themselves not `throws` (*e.g.,* a variable dereference, array access, type cast, etc.) are more easily handled by TSR (*i.e.,* all exception object construction occurs within the JVM and need not be analyzed as above).

Given this assumption, when is TSR safe for a throw site? We assume it is *unsafe* when at least one flow-of-control from the throw site to a handler uses the exception

object (*i.e., may* analysis).

- The simplest unsafe case is where no handler can be found in the code for the exception type. The runtime system will deliver the exception to either the user or to some black box (perhaps another system of classes). If the exception is known to be delivered outside a program (*i.e.,* causes system termination), then the only part of the exception object needed is its stacktrace; *lazy stack-trace generation* is discussed in the next section and helps deal with this case.
- Another unsafe case is the existence of at least one reachable handler using the object, whether that handler is within the procedure or is assumed to be in some procedure along the call chain.

Any analysis must therefore determine when the exception object *may* be used. We will present both *intraprocedural* and *interprocedural* algorithms.

7.2.1 Intraprocedural TSR

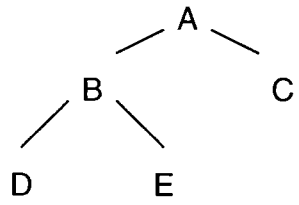
Figure 7.2 introduces our running example. On the left side of the figure is code for some method `s()` of some class `L`, with statements taking the form `s01`, `s02`, etc. Instructions which could raise an exception are commented with `PEI class` where `class` refers to the exception type. (Instructions which can throw different kinds of exceptions will have multiple comments.) All `throw` statements are also `PEI`s. Exception handlers which do not use the exception object begin with the instruction `pop`, otherwise handlers begin with `use`. On the right side of the figure is a simple single-inheritance exception hierarchy with exception types named `A` through `E` (`A` is the root class).

Our goal is to annotate instructions as suitable for TSR. We assume the intermediate representation used for the code associates each method-call operation with a single instruction, *e.g.,* `p() + q() - r()` would be represented by at least three instructions. We first define several helper functions for our analysis:

- *usesobject(c,m,b)* is a *boolean function* returning true if the code block starting at bytecode *b* in method `c.m()` uses the first item on the stack, and returns false otherwise. The only kind of code block we will pass to this function is a catch handler.

```
   L.s() {
 1   s01;          // PEI C
 2   s02;
     try {
 3     s03;        // PEI D
 4     s04;
 5     s05;        // PEI C
       try {
 6       s06;
 7       if (foo) {
 8         s07;  // PEI B
 9         t();  // PEI E
         } else {
10         s08;  // PEI D
         }
11       s09;      // PEI B
       }
       catch(B d) {
12       pop;
13       s11;
       }
14     s12;        // PEI E
15     s13;
     }
     catch (B b) {
16     use;
17     s14;
     }
18   s15;          // PEI C
19   s16;
   }
```

(a) code for L.s()



(b) Exception hierarchy

Figure 7.2. *Running Example (1)*

  - Given only a method's bytecode, a simple albeit conservative algorithm examines the first operation of the block, returning *false* if the first instruction is a `pop`, and *true* otherwise.
  - If more information is available from *live-variable* analysis, then the handler is treated as a code block in which the exception object is defined at an inserted pseudo instruction placed just before the handler's first instruction. *Definition-use chaining* information is computed for the block, and the exception object is indeed not used (*i.e.,* need not be created at the throw site) if there is no def-use chain starting at the inserted definition.

- *numinst(c,m)* is the *number* of instructions in method m of class c. If c is a subclass inheriting an implementation of m, the number refers to the superclass implementation; if it is a virtual class, then the function returns 0.
- *inst(c, m)* is a function that, for class c and method m, yields the array of instructions that make up m in some order, namely *inst(c,m)[1 ... numinsts(c,m)]*.
- *throwset(c, m, i)* is function from *instructions* to the *set of exception classes*, where the exceptions can be explicitly or implicitly thrown by the instruction or operation i. If the instruction is a method call, *throwset* returns the list of exceptions in the `throws` clause of the method signature; however, in this case the throw sites are located inside `c.m` (and we cannot apply TSR to a callsite). If the analysis is performed on method bytecode, the domain of *instructions* is the set of JVM instructions; if performed on some other intermediate-representation (IR) of the method, the domain is a set specific to that elements of that IR.
- *MUSTCATCH(c,m,lnum)* takes the instruction at position *lnum* in class `c.m()`, and returns a *list of tuples*. The tuples are of the form $\langle E, boolean \rangle$. If a tuple appears in the list with E as the first element of the tuple, then the instruction is statically enclosed by a catch clause having an exception parameter of type E. A tuple with the value $\langle E, true \rangle$ indicates that a handler exists and the exception object is used (*i.e.,* is not discarded); $\langle E, false \rangle$ indicates a handler exists that does not use the exception object. No two tuples in a list may have the same exception type. The value of *MUSTCATCH(c,m,lnum)* is also called a *handler environment*, denoted by H.

H is a list (*i.e.,* not a set) and represents the order in which `catch` clauses are examined when finding a handler for some instruction *lnum*. Handlers nested closest to the instruction will appear earliest in the list; handlers attached to the same `try`-block statement in source code will appear in the list in the same order. That is, a list H for instruction *lnum* having the form:

$$[\ldots, \langle E_i, b_i \rangle, \ldots, \langle E_j, b_j \rangle \ldots]$$

implies that one of two conditions *must* hold:

a. The `try`-block having a `catch`-block for E_j *statically encloses* the `try`-block with a `catch`-block for E_i.

b. The handlers for E_i and E_j are attached to same `try`-block, and the `catch`-block for E_i appears before the `catch`-block for E_j.

We further constrain the list such that E_j cannot have the same value (*i.e.,* type) as E_i. This matches the semantics of a language such as Java; if two or more handlers of type E enclose *lnum*, then first handler found for E is the one to which the exception of type E will be dispatched.
(*MUSTCATCH* can be computed by examinining the nesting of `try`-blocks in a method's text and by using the values produced by *usesobject*.)

- *MUSTDISCARD(c,m,i)* is used for notational convenience. It is a sublist of *MUSTCATCH*; each tuple in the list has the form $\langle E, \mathit{false} \rangle, E \in \mathcal{E}$, *i.e.,* all handlers which do not use the exception object. We assume that the $\in$ operation available to sets is also available to lists.

We also use notation based on Vitek *et al.* [71] to describe relationships amongst exception types. Single-inheritance (single subtyping) is assumed.

- The set of all types (classes) is denoted by $\mathcal{T}$.
- The set of all exception types (classes) is denoted by $\mathcal{E}$. It is a subset of $\mathcal{T}$.
- The relation "Exception Class `B` is derived from Exception Class `A`" is denoted $B <: A$.
- The relation "Exception Class `B` is derived from, or the same class, as Exception Class `A`" is denoted $B \leq: A$.

- The relation "Exception Class B is an immediate descendant of Exception Class A on the class hierarchy graph" is denoted by $B <:_d A$.
- $parent(x) \equiv \{y \in \mathcal{T} \mid x <:_d y\}$; this will be empty if x is at the root of the type hierarchy.
- $children(x) \equiv \{y \in \mathcal{T} \mid y <:_d x\}$.
- $ancestors(x) \equiv \{y \in \mathcal{T} \mid x <: y\}$.
- $descendants(x) \equiv \{y \in \mathcal{T} \mid y <: x\}$.

Our final helper function is *nearest(x, H)*. This function mimics the programming language's semantics for dispatching exceptions. It returns a set of tuples that correspond to `catch` clauses in handler environment H for an exception of type x, *i.e.*, x is the same type as, or a subclass of, the classes of exceptions handled by entries in *nearest(x, H)*. The returned set is either a singleton or is empty. We are focusing on Java; the language and JVM specifications do not specify the order in which catch clauses are to appear within Java source code bytecode. The specifications do state that control is transferred to the first matching catch clause (*i.e.*, matches the order of items in list H). We assume that catch-clause order is extracted by a compiler such as `javac`, or some other compiler/optimizer using `javac`'s order. The semantics implemented by `javac` are a bit stricter as shown by the following fragment.

```
try { ... }
catch (Exception e) { ... }
catch (NullPointerException e) { ... }
```

Sun's compiler reports an "`exception has already been caught`" error at the start of the second catch clause.

Function *nearest* is defined recursively as follows:

$$nearest(x, H) = \begin{cases} \emptyset & \textbf{if } H = \emptyset \\ \langle E, b \rangle & x \leq: E \\ nearest(x, H') & \textbf{otherwise} \end{cases}$$
$$\textbf{where } H = (\langle E, b \rangle : H') \textbf{ whenever } H \neq \emptyset$$

Deferring for now the details of computing *MUSTCATCH*, we present an algorithm for intraprocedural TSR in Figure 7.3. Here is a summary of the support functions just defined:

- *inst(c, m)*: returns an array of instructions belonging to method m in class c.

- Input:
  - Method m of class c.
- Output:
  - *tsr(c,m,i)*, a set of exception types, such that it contains t if the exception object of type t thrown by *inst(c,m,i)* is *not used* by the corresponding handler's catch-block.
- Support functions:
  - $inst : class \times method \rightarrow codeblock$
  - $numinst : class \times method \rightarrow integer$
  - $throwset : class \times method \times integer \rightarrow Set\ of\ exception\ types$
  - $MUSTDISCARD : class \times method \times integer \rightarrow Handler\ Tuple\ list$
  - $nearest : ExceptionType \times Handler\ Tuple\ list \rightarrow Handler\ Tuple\ Set$

Algorithm:

- For $i \leftarrow 1 \ldots numinst(c, m)$ do:
  a. $tsr(c, m, i) \leftarrow \emptyset$.
  b. $\mathcal{D} \leftarrow$ *MUSTDISCARD(c,m,i)*
  c. For each $t \in throwset(c, m, i)$ do:
     - $H' \leftarrow nearest(t, \mathcal{D})$
     - if $H' \neq \emptyset$ then $tsr(c, m, i) \leftarrow tsr(c, m, i) \cup t$

Figure 7.3. *Algorithm: Intraprocedural TSR*

- *throwset(c, m, i)*: returns the set of exception classes that can be thrown by instruction at bytecode position i in method m of class c.
- *MUSTCATCH(c, m, lnum)*: returns a list of exception types for which local handlers exist in method m in class c at bytecode *lnum*.
- *MUSTDISCARD(c, m, i)*: returns a subset of *MUSTCATCH* which are all exception types for which handlers do not use the exception object.
- *nearest(x, H)*: given the list of handlers H, returns a set of handlers (either a singleton or the empty set) which handle x, using the same semantics as that defined by the Java language specification.

If *tsr(c,m,i)* is empty, then regular code must be emitted for the instruction at i; if

not empty, then optimized code for each exception type ($t \in tsr(c, m, i)$) throwable at instruction i is emitted; this code will be executed when the exception of type t is thrown at that instruction. The optimized code for each individual exception type is common across all locations that may throw the exception. Therefore the program size increases in proportion to the number of *exception types*, not the number of *throw sites*.

Assuming we have some representation of the bytecode for method `c.m()`, our algorithm for computing *MUSTCATCH* requires:

- the method's bytecode in the form of an N-element array,
- the exception type hierarchy $\mathcal{E}$, and
- the (possibly empty) method's exception table of length M.

This exception table encodes the start and end of each `try`-block, along with each block's `catch`-clauses (exception type, bytecode location of handler); such line numbers belong to the domain *lnum*. One constraint on the format of the exception table entries is that `try`-blocks only nest within each other and do not overlap; there is *no* constraint on the order in which entries appear in the table [42]. The simple brute-force algorithm to construct *MUSTCATCH()* in Figure 7.4 (from which we derive *MUSTDISCARD*) loops through all method bytecodes and table entries, yielding a time complexity of $O(NM)$.

Should we worry about the time complexity of computing *MUSTCATCH*? Given that values of M are usually small for typical Java methods ($M < 3$), the usual answer would be "no". However, there are many methods with significantly larger exception tables ($M > 10$), and as the exception-handling programming idiom gains popularity, the average value of M can only increase. Another potential cause for higher values of M is classfile optimizations which, as the result of some transformation, change and re-arrange a method's bytecodes. If the optimized method has an exception table, the new code order may break up previously contiguous code blocks within a try-block, possibly resulting in an increased number of exception table entries.

We can reduce the size of *tsr()* by observing that TSR optimization information can be as easily attached to basic blocks in a control-flow graph as they can to individual statements. For example, the control-flow graph in Figure 7.5 is one such

- Input:
  - *etable* : *class* × *method* → *list of tuples of the form* $\langle lnum, lnum, lnum, \mathcal{E} \rangle$
  - *numinst* : *class* × *method* → *integer*
  - *usesobject* : *class* × *method* × *lnum* → *boolean*
- Output:
  - *MUSTCATCH* : *class* × *method* × *integer* → *Handler Tuple list*
  - *MUSTCATCH()* is defined for instructions in `c.m()`.
- Algorithm:
  a. $N \leftarrow numinst(c, m)$
  b. $M \leftarrow\mid etable(c, m) \mid$
  c. For each $i \leftarrow 1 \ldots N$ do:
     - *MUSTCATCH(c,m,i)* ← *emptylist*
     - For each $j \leftarrow 1 \ldots M$ do:
       * $\langle s, e, h, t \rangle = etable(c, m)[j]$
       * $p \leftarrow$ **not** *usesobject(c, m, h)*
       * if $(s \leq i < e)$ then append $\langle t, p \rangle$ to $MUSTCATCH(c, m, i)$

Figure 7.4. *Algorithm: Intraprocedural MUSTCATCH*

line #	MUSTCATCH	MUSTDISCARD	*tsr*
1	[]	[]	$\emptyset$
2	[]	[]	$\emptyset$
3	$[\langle B, true\rangle]$	[]	$\emptyset$
4	$[\langle B, true\rangle]$	[]	$\emptyset$
5	$[\langle B, true\rangle]$	[]	$\emptyset$
6	$[\langle B, false\rangle]$	$[\langle B, false\rangle]$	$\emptyset$
7	$[\langle B, false\rangle]$	$[\langle B, false\rangle]$	$\emptyset$
8	$[\langle B, false\rangle]$	$[\langle B, false\rangle]$	$\{B\}$
9	$[\langle B, false\rangle]$	$[\langle B, false\rangle]$	$\{E\}$
10	$[\langle B, false\rangle]$	$[\langle B, false\rangle]$	$\{D\}$
11	$[\langle B, false\rangle]$	$[\langle B, false\rangle]$	$\{B\}$
12	$[\langle B, true\rangle]$	[]	$\emptyset$
13	$[\langle B, true\rangle]$	[]	$\emptyset$
14	$[\langle B, true\rangle]$	[]	$\emptyset$
15	$[\langle B, true\rangle]$	[]	$\emptyset$
16	$[\langle B, true\rangle]$	[]	$\emptyset$
17	$[\langle B, true\rangle]$	[]	$\emptyset$
18	$[\langle B, true\rangle]$	[]	$\emptyset$
19	$[\langle B, true\rangle]$	[]	$\emptyset$

Table 7.4. MUSTCATCH, MUSTDISCARD *and* **tsr** *for Running Code example (1)*

representation of our running example. It is a *Factored Control Flow Graph* or *FCFG* representation of Figure 7.2. Choi *et al.* [15] designed FCFGs for use within the IBM Jalapeño optimizing compiler and JVM [9]. FCFGs have, amongst others, several important properties:

a. *Factored edges* connect basic blocks with exception handlers, specifically the handlers for a particular block. This minimizes the number of edges needed to represent the control-flow graph.

b. FCFG blocks resemble extended basic blocks (*i.e.,* single entry point, many exit points), and since there is only a factored edge from a block to the handler for a specific exception type, we need only maintain TSR information for each block. FCFGs have larger blocks than CFGs in the presence of exception handling, and therefore we reduce the cost of maintaining TSR information.

Table 7.4 contains the values of *MUSTCATCH*, *MUSTDISCARD* and *tsr* for the code in Figure 7.2; Table 7.5 contains the values for *tsr* assuming they are stored by the method's FCFG basic-blocks.

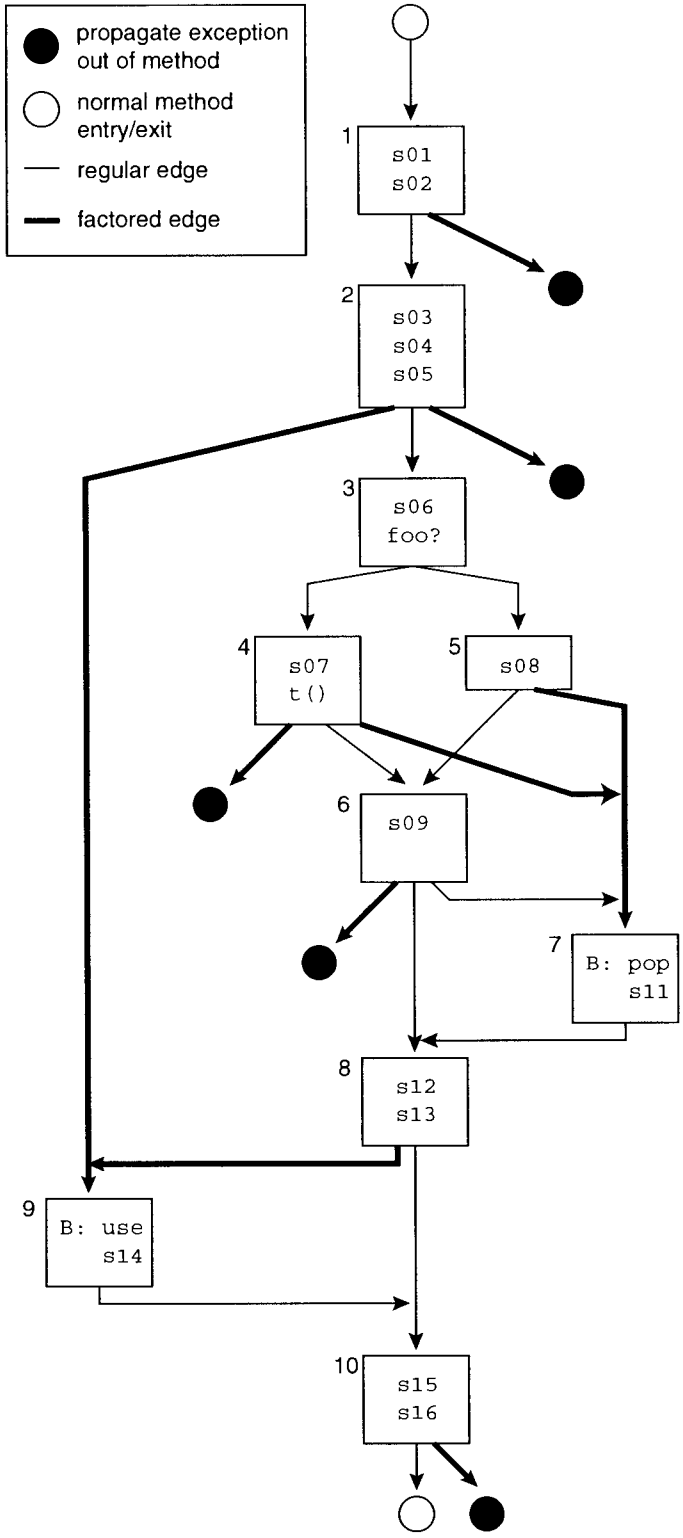


Figure 7.5. *Factored CFG for Running Example (1)*

block #	tsr
1	$\emptyset$
2	$\emptyset$
3	$\emptyset$
4	$\{B, E\}$
5	$\{D\}$
6	$\{B\}$
7	$\emptyset$
8	$\emptyset$
9	$\emptyset$
10	$\emptyset$

Table 7.5. tsr *stored by FCFG nodes for Running Code example (1)*

7.2.2 Interprocedural TSR

The measurements reported at the start of this chapter suggest that as call-depth distances increase between throw and handling sites, so do the benefits from eliminating exception-object creation. Therefore some attempt must be made to apply TSR interprocedurally. We need not perform interprocedural analysis from scratch, of course, as some analysis results will be available from that for *farhandlers* (see Section 6.2.1). For example, *farhandlers(n)* returns a list of a set of $\langle$*exception type*, *address*$\rangle$ tuples corresponding to handlers reachable from node n in the callgraph. We assume the existence of functions *node(c, m)* and *methodclass(n)* which convert between nodes and class/methods respectively. We also assume there exists a function `popanalysis(addr)` which determines if code starting from location *addr* uses the topmost object on the stack.

Determining the value of *tsr()* for a given method therefore involves two steps:

a. Local handlers are first examined, *i.e.,* the intraprocedural value of *tsr()* is computed.

b. If there are no local handlers for the given PEI, *i.e., tsr()* is $\emptyset$, then far-handlers are considered. The existence of at least one such handler using the exception object will eliminate the exception type from *tsr* for the instruction/FCFG block.

This suggests the algorithm which appears in Figure 7.6.

Figure 7.7 contains an example program for analysis along with its callgraph (edges

- Input:
  - Method m of class c.
  - Intraprocedural *tsr()* for method m of class c.
- Output:
  - *itsr(c,m,i)*, a set of exception types, such that it contains t if the exception object of type t thrown by *inst(c,m,i)* is *not used* by the corresponding handler's catch-block; the handler may or may not be local.
- Support functions:
  - $farhandler : N \rightarrow set\ of\ HandlerInfo$
  - $inst : class \times method \rightarrow codeblock$
  - $methodclass : class \times method \rightarrow \mathcal{N}$
  - $node : \mathcal{N} \rightarrow \langle class, method \rangle$
  - $numinst : class \times method \rightarrow integer$
  - $popanalysis : addr \rightarrow boolean$
  - $throwset : class \times method \times integer \rightarrow Set\ of\ exception\ types$

Algorithm:

- $fh \leftarrow farhandler(node(c, m))$
- For $i \leftarrow 1 \ldots numinst(c, m)$ do:
  a. $itsr(c, m, i) \leftarrow tsr(c, m, i)$
  b. if $itsr(c, m, i) \neq \emptyset$ then proceed to next loop iteration for i.
  c. For each $t \in throwset(c, m, i)$
     - $catch \leftarrow \{\langle E, addr \rangle | \langle E, addr \rangle \in fh \wedge (t = E \vee t <:_d E)\}$
     - $use \leftarrow \{\langle E, addr \rangle | \langle E, addr \rangle \in catch \wedge popanalysis(addr) = false\}$
     - $discard = catch - use$
     - if $use = \emptyset$ then $itsr(c, m, i) \leftarrow itsr(c, m, i) \cup t$

Figure 7.6. *Algorithm: Interprocedural TSR*

edge	*handlers(edge)*
⟨P, 01, Q⟩	$\{\langle E, 05\rangle, \langle F, 06\rangle\}$
⟨P, 02, R⟩	$\{\langle E, 05\rangle, \langle F, 03\rangle\}$
⟨P, 03, T⟩	$\{\langle E, 05\rangle, \langle F, 06\rangle\}$
⟨P, 04, T⟩	$\{\langle E, 05\rangle, \langle F, 06\rangle\}$
⟨Q, 07, S⟩	$\{\langle F, 08\rangle\}$
⟨R, 09, S⟩	$\emptyset$
⟨S, 15, T⟩	$\emptyset$
⟨S, 11, S⟩	$\emptyset$
⟨S, 11, 13⟩	$\emptyset$

Table 7.6. *Running Example 2: Handler information for edges*

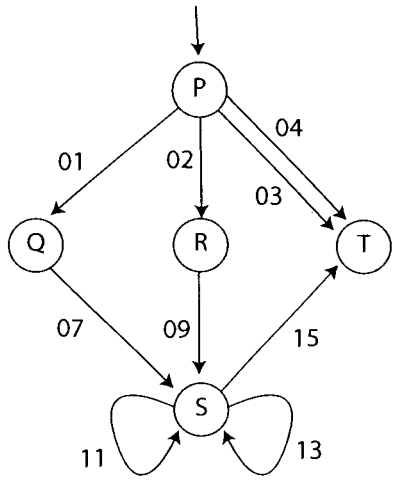
labelled by the line number in the code corresponding to the call). Table 7.6 lists the edges in the call graph along with values for *handlers(e)*; Table 7.7 lists values of *farhandler()* for each callgraph node; and Table 7.8 provides the values of *tsr* and *itsr* for the two throw sites in the example (lines 16 and 17). Exception types E and F are assumed to be related by only a common ancestor (*i.e.,* neither E nor F are derived from each other).

The precision of the interprocedural analysis for *farhandlers* (*e.g.,* flow-insensitive, flow-sensitive) should carry into the technique for computing *itsr()*; *farhandler* values from different nodes are not combined together. In our experimental framework, a flow-insensitive analysis is used to produce callgraphs.

However, the values of *farhandler* only indicate the existence of potential handlers at a node. There is still the possibility that a control-flow path may exist from the call-graph entry node to a throwsite, with nary a handler along the path. Therefore for throw-site reduction to work, we must provide a solution to the problem of exceptions *escaping* from the system, where the system is either a program or a module. If it escapes the program, then a stack trace must be printed; if it escapes a module, then an exception object must be created such that the module (for which we may not have access to source code) can handle the exception. The techniques described in the next two sections are possible solutions that can be used either for escaping exceptions or for code not yielding many opportunities for TSR.

```
   P() {
01   try { Q();
02         try { R(); }
03         catch (F f) { use; T(); }
04         T();
      }
05    catch(E e) { discard }
06    catch(F f) { discard }
   }

   Q() {
07   try { S(); }
08   catch (F f) { discard }
   }

   R() {
09   S();
   }

   S() {
10   switch (condition) {
11     case 1: try { S(); }
12             catch(F f) { use }
               break;
13     case 2: try { S(); }
14             catch(F f) { discard }
               break;
15     case 3: T();
               break;
16     default: throw new E;
     }
   }

   T() {
17   throw new F;
   }
```

(a) code



(b) call graph

Figure 7.7. *Running Example 2*

node	*farhandlers(node)*
P	$\emptyset$
Q	$\{\langle E, 05\rangle, \langle F, 06\rangle\}$
R	$\{\langle E, 05\rangle, \langle F, 03\rangle\}$
S	$\{\langle E, 05\rangle, \langle F, 03\rangle, \langle F, 08\rangle, \langle F, 12\rangle, \langle F, 14\rangle\}$
T	$\{\langle E, 05\rangle, \langle F, 03\rangle, \langle F, 08\rangle, \langle F, 12\rangle, \langle F, 14\rangle\}$

Table 7.7. *Running Example 2: farhandler() values*

line	*tsr* at line	*itsr* at line
16	$\emptyset$	E
17	$\emptyset$	$\emptyset$

Table 7.8. *Running Example 2:* tsr *and* itsr

7.3 Lazy Stack-Trace Construction

Section 7.1 contains a description of the efforts needed to throw an exception. A significant proportion of that effort goes towards computing a stack trace, yet most Java exception handlers do not use this trace. What we propose when static techniques fail is that stack-traces be created *lazily*, that is, only at the point at which they are needed by methods such as `printStackTrace()`.

A *Java stack-trace* is a set of strings with one string for each call-stack frame, starting from the handler site and ending at the exception's throw site. Save for the last string, all other strings describe a callsite; the last string indicates the instruction throwing the exception. As these traces are designed to be read by programmers, callsite and throwsite information is usually rendered as line numbers within Java source files. If no source is available, then only the class file is mentioned.

Figure 7.8 contains a stack trace that was generated by `Tomcat` (a reference implementation for Java Servlets and JavaServer pages). The stack trace itself was extracted unedited from a newsgroup posting. Its size is quite dramatic—the trace contains 61 different "`at`" lines indicating 60 method calls and 1 thrown exception. If the code which catches such an exception is expected to recover and continue, then it will need information about the exception itself. (A constructor for `VerifyError`

```
java.lang.VerifyError: Cannot inherit from final class
     at java.lang.ClassLoader.defineClass0(Native Method)
     at java.lang.ClassLoader.defineClass(ClassLoader.java:509)
     at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:123)
     at org.apache.catalina.loader.WebappClassLoader.findClassInternal
          (WebappClassLoader.java:1664)
     ...
     < 50 lines deleted >
     ...
     at org.apache.catalina.core.StandardHost.start(StandardHost.java:738)
     at org.apache.catalina.core.ContainerBase.start(ContainerBase.java:1188)
     at org.apache.catalina.core.StandardEngine.start(StandardEngine.java:347)
     at org.apache.catalina.core.StandardService.start(StandardService.java:497)
     at org.apache.catalina.core.StandardServer.start(StandardServer.java:2189)
     at org.apache.catalina.startup.Catalina.start(Catalina.java:512)
     at org.apache.catalina.startup.Catalina.execute(Catalina.java:400)
     at org.apache.catalina.startup.Catalina.process(Catalina.java:180)
     at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
     at sun.reflect.NativeMethodAccessorImpl.invoke
          (NativeMethodAccessorImpl.java:39)
     at sun.reflect.DelegatingMethodAccessorImpl.invoke
          (DelegatingMethodAccessorImpl.java:25)
     at java.lang.reflect.Method.invoke(Method.java:324)
     at org.apache.catalina.startup.Bootstrap.main(Bootstrap.java:203)
```

Figure 7.8. *Example of a stack trace generated by Tomcat*

takes a string as a parameter, and it is a string that the handler would need.)

In most Java programs, the only uses of stack traces are either within an exception handler or are generated by the JVM as a consequence of program termination. In either case, lazy generation can only occur once the stack has been unwound from the throw site. To facilitate laziness, we must keep some information about call frames as those same frames are unwound during exception propagation. As we cannot predict the size of a stack trace, our representation must grow dynamically. We also need to minimize (a) the amount of information stored per call frame and (b) the time taken to extract this information during unwinding. Our assumption is that it is fair to save time during exception propagation by increasing the effort needed for a `printStackTrace()` method call.

Each trace line refers to a method call within a specific `.java` file, along with the name of the class file containing that method *plus* the line number in the file from which the method's definition begins. To reconstruct this data at the handler, we need the value of the program counter in each stack frame along with an index into the VM's table of loaded classes.

Assuming that these pairs ⟨`line`, `classidx`⟩ are pushed onto a special stack named `lazytrace`, and pushed in the same order and at the same time as activation frames are unwound, the lines of the stack trace can be computed in the same order as entries appear from the top of `lazytrace`. The Program Counter (or `PC`) value of the handling activation frame *before* the exception reaches the frame is also needed (named `start`), along with the handler's class and method (named `hclass` and `hmethod`). The following functions are also needed:

- *CLASS(classidx)* : returns a reference (`class`) to the standardized classfile representation stored at `classidx` in the JVM's classloader.
- *CLASSNAME(class)* : given a reference to a classfile representation with the VM, returns the string corresponding to the full classname (including package).
- *INVOKEDMETHOD(class, method, linenum)* : given a class reference, method reference, and line number within the method, a constant-pool reference is returned if the bytecode instruction is either `invokespecial` or `invokevirtual` (otherwise it is null). This method may or may not correspond to one in `class`—its meaning can be determined only by knowing the class of the object that received the message *at message invocation time.* We can infer that object's class by examining our `lazytrace` stack.
- *METHODNAME(class, method)* : given a class reference and method reference, this function returns the string representation of the method name. The parameter `method` is actually an index into the constant pool for `class`.
- *FILENAME(class)*: filename containing the source code for `class`.
- *LINENUMBER(class, method, line)*: returns an integer of the line number *within the source file* corresponding to the bytecode at `line`.

Items on `lazytrace` can be represented as:

$$\langle \texttt{line}_1, \texttt{classidx}_1 \rangle, \ldots \langle \texttt{line}_i, \texttt{classidx}_i \rangle, \langle \texttt{line}_{i+1}, \texttt{classidx}_{i+1} \rangle, \ldots$$

To generate the initial stack-trace line, we create the following strings:

a. *CLASSNAME*(*CLASS*(`hclass`))

b. *METHODNAME*(`hclass`, `hmethod`)

c. *FILENAME*(`hclass`)

d. *LINENUMBER*(`hclass`, `hmethod`, `start`)

The strings are then concatenated appropriately to produce the stack-trace line.

The $i+1$ line can be computed by using stack item i to obtain the method, and stack item $i+1$ used for all other details. Therefore the method used when $i = 1$ is that of `INVOKEMETHOD`(`hclass`, `hmethod`, `start`).

a. `method` ← `INVOKEMETHOD`(`hclass`, `hmethod`, `start`)

b. while items on `lazytrace` do:

   (a) ⟨`line`, `classidx`⟩ ← `pop`(`lazytrace`)

   (b) $c \leftarrow$ *CLASS*(`classidx`)

   (c) $s_1 \leftarrow$ *CLASSNAME*(c)

   (d) $s_2 \leftarrow$ *METHODNAME*(c, `method`)

   (e) $s_3 \leftarrow$ *FILENAME*(c)

   (f) $s_4 \leftarrow$ *LINENUMBER*(c, `method`, `line`)

   (g) Use s_1, s_2, s_3 and s_4 to compute the stack trace line.

   (h) `method` ← *INVOKEMETHOD*(c, `method`, `line`)

The implementation of each support function clearly depends upon the underlying VM design and implementation.

7.4 Lazy Exception Objects

Even if we eliminate the effort involved in computing a stack trace, TSR still does not yet adequately account for exception objects which escape the system. One response is therefore to construct exception objects only when they are about to escape.

There is an intriguing further use of lazy exception-object construction, or *Lazy Exception Objects* (LEO), and it helps in the following case:

- TSR cannot be applied to a throwsite *th* because at least one potential handler uses the exception object, and
- there also exists at least one potential handler for *th* which discards the exception object.

The second part of the case is important—if no handler for *th* discards the object, then code may just as well construct the object at throw time as at the time it is

first used. LEO would just produce more complex code to obtain the same effect as unoptimized code. Yet when both parts of the case do hold, then a potentially significant benefit can be obtained from LEO. The greater the frequency at which the discarding handler executes for some run of the program, the greater the reduction in exception overhead.

For example, consider again the code appearing in Figure 7.7. Our analysis determined that the throwsite at line 17 could not be optimized with TSR because at least one potential handler for F uses the exception object. However, there do exist several handlers which discard instances of F (lines 6, 8, and 14). If `condition` at line 10 evaluates to 3 more often than 1 or 2, then LEO is a benefit—handlers at line 6 and 8 for F which are reachable from line 15 will discard the exception object.

One could respond by stating that a better interprocedural analysis—perhaps even abstract interpretation, or program profiling—could better predict paths taken from `P()` to `T()`. While this is undoubtedly true, we instead choose to "bet" that if an exception object *might* be discarded, then we can benefit by deferring exception construction to just before the object's first use.

- An implementation of LEO requires an analysis that determines when a throwsite would benefit. One line added to the end in Figure 7.6 would allow us to compute LEO information.

  $$\text{if } \mathit{uses} \neq \emptyset \wedge \mathit{discard} \neq \emptyset \text{ then } \mathit{leo}(c, m, i) \leftarrow \mathit{leo}(c, m, i) \cup t$$

  Figure 7.3 does not need to be modified (*i.e.,* if a local handler exists for a throw site, then a compiler can determine if the exception object is used or discarded).

- For each exception class in the program, an LEO-enabled JVM will keep one (static) copy. Each of these copies has a minimal—preferably empty—stack trace.

As with TSR, we assume that some analysis has been performed that guarantees LEO will be applied only in cases where exception-object use *does not occur* between the `new` statement which creates the object and the `throw`.

The `new` is now annotated to be a LEO-version of `new`—that is, rather than creating a new instance from heap, the arguments to the constructor are instead copied into the exception's static copy. This static copy is then pushed onto the stack

before the `throw`. Each handler for the candidate throw sites that *does not* discard the object now has an extra instruction inserted at the start. This instruction `clone`s the exception option at the top of the operand stack and then `pop`s off the static version.

7.5 Summary

This chapter has presented three new techniques for improving the performance of exception handling: *throw-reduction*, *lazy stack-trace elimination* and *lazy exception-object construction*. Combined together they eliminate a large amount of the effort often wasted when simply creating an exception object. The next chapter reports on the results of experiments in which proposals in this and the previous chapter are implemented, and their impact on run-time performance measured.

Chapter 8

Experimental Results

In this chapter we explore the effect of `farhandlers` and `throw-site reduction`. The timing results presented are from actual Java classfiles which are executed using the analyses presented in the previous chapters. A real Java virtual machine (JVM) has been modified to use the farhandler and throw-site reduction information; reported timings are those obtained from this system.

Before providing a brief preview of the chapter, the reader may wish to be reminded of the goal of this research. We have argued that the exception-handling idiom is flexible and powerful enough to express a richer set of control-flow structures than those induced by the needs of error handling. Our aim is to demonstrate that a programmer, without any knowledge of how exceptions are implemented by a particular JVM, can use these exceptions to structure their programs and reasonably expect the resulting extra run-time overhead to be small. This differs from present programming practice: programmers must either be very aware of where the runtime costs lie in using exceptions on their chosen JVM (*i.e.*, write their programs in such a way as to prevent these costs from occurring) or avoid exceptions entirely.

The next section describes our chosen compiler framework and JVM in more detail. Following this, our implementation is examined from three points of view:

- *Validation*: Components of the `SPECjvm98` benchmark are executed on the modified JVM and performance compared with an unmodified JVM.
- *Microbenchmarks*: Programs that we have written to examine specifically the several stages involved in exception handling are then executed, and their performance discussed.
- *Exception-Idiom Usage*: A typical programming problem is solved once using

exceptions thrown and caught within the same method, and then solved again where exceptions and their handlers are in different methods. Performance comparisons are made with similar solutions not using exceptions.

8.1 Experimental setup and methodology

In order to determine the effectiveness of our proposals, we chose an implementation-based approach instead of simulation. We began this work by using existing implementations of a bytecode-optimization framework and a virtual machine. To these two systems we added our analysis algorithms and dispatch mechanisms. Furthermore we restricted our investigations to two specific portions of our proposals, a decision informed by our examination of the `SPECjvm98` benchmarks.

- Farhandler tables and handler dispatch, without *finally*-block execution: We build interprocedural handler-dispatch (*i.e., farhandler*) tables, and then use these tables at run-time when propagating exceptions outside of the methods without a suitable local handler. Therefore regular bytecode execution is suspended from the point in time at which the exception is thrown until the time the correct handler is found. It is possible, however, that unwinding the stack from one method to another could require the execution of some *finally* block in a called method. The use of such blocks, however, is rare. In all of the benchmark programs provided in `SPECjvm98` there is only a single finally-block, and this is in `_213_javac` (in classfile `spec.benchmarks._213_javac.Javac.compile`); when executed using the benchmark workload, the finally block is never executed.[1]
- Throw-site reduction (TSR), without either lazy exception-objects or lazy stack-trace construction: TSR determines whether or not an exception object should

[1]No `finally` blocks appear in benchmarks for which Java source code is given. The largest benchmark programs, however, are only present as class-files as the program's contributor was probably from a commercial concern and did not want to release source code. Where only class-files exist, we looked for the existence of the `jsr` instruction in the classfile bytecodes; as Sun's reference `javac` compiler emits this instruction when compiling `finally`s, the presence of this instruction is a good indicator that a *finally* block existed in the original code.

be created. If all handlers reachable from the throwsite do *not* use the exception object, then it need not be created; if even just one of the handlers uses the object, then TSR cannot be applied to the throwsite. In the latter case, *lazy exception-object creation* (or LEO) could be used such that only handlers using the exception object incur the overhead of its creation. However, we decided against investigating the performance of LEO; again referring back to `SPECjvm98` benchmark programs and according to our own analysis, it appears that for each throwsite, either all reachable handlers use the exception object or all handlers do not use the exception object. While there may be some benefit from laziness, we cannot determine the degree of benefit using `SPECjvm98`.

We expect that exception usage will continue to evolve, especially as exception-handling mechanisms become faster and faster. This will result in new and more comprehensive benchmark suites especially designed to reproduce such new usage. The work of this chapter therefore concentrates the most profitable optimizations to exceptions, and this from the point of a view of a programmer who could be convinced to use them if they were not expensive, *i.e.,* that they do not incur too high a cost in comparison to code without exceptions.

8.1.1 Analyzer

Analysis takes two place in two phases: In the first, information from the classfiles is extracted; in the second, farhandler tables and throw-site reduction data is constructed using this information.

We use the `Soot` bytecode optimization framework developed by the Sable Compiler Research Group at McGill University in order to extract information on handlers, throwsites, callgraph edges and bytecode addresses [69]. This large framework provides a great number of easy-to-use and helpful tools, both for working on individual class files and also on whole programs (*e.g.,* dataflow analysis, intermediate formats for classfiles, classfile modification, annotations, classfile construction, callgraph construction, etc.). Some small modifications were made to the `Jasmin` system used by `Soot` to create classfiles.

This data is then used by a standalone Java program called `BuildTable`. The program outputs three sets of information for a Java program:

- A list of the exception objects mentioned in the program;
- The farhandler table as described in Chapter 6; and
- For every location in the program that constructs a new `Exception` object, a line indicating whether or not TSR may be applied to that locations `new` bytecode.

Eagle-eyed readers might notice that `Exception` was mentioned and not `Throwable`; the former is a subclass of the latter. We perform no optimizations or analyses on handlers for `Throwable`—when `Soot` transforms classfiles with methods using `finally`, it converts this `finally`-block into a catch-block for exceptions of type `Throwable`. As our farhandler tables and TSR lines contain no mention of `Throwable`, the modifed JVM (described in the next section) will use regular exception-handling propagation code for such exceptions. The result is that our analysis and VM will produce a system that works correctly in the presence of `finally`-blocks.

8.1.2 Virtual Machine

As a starting point for our work, we used the `SableVM` system developed at McGill University [23, 24]. This JVM is a result of the PhD thesis work of Etienne Gagnon. It has been carefully designed to support research, is clearly written, and can be compiled into several different configurations. Timing results presented in this chapter use the SableVM configured as below:

- switched threading (*i.e.*, pure interpretation);
- signals-for-exceptions disabled;
- copying garbage collector;
- traditional object layout;
- debugging features disabled.

One version of the VM has been kept unmodified (`sableorig`) and this is used for timings needed of a "default" VM. Both this default VM and our modified VM are compiled using `gcc` version 2.5 using optimization level 2 ("`-O2`").[2]

[2]The detail on optimization is important—when using switched threading, SableVM's interpreter loop is within a function named `_svmf_interpreter`. If level zero optimization is used when com-

The SableVM is packaged with an implementation of `classpath`, an open-source implementation of the Java core library. In order to keep comparisons between VMs fair, both the unmodified and modified VM use the same compiled copy of this library.

8.1.3 Equipment

The computer used to obtain timings is a Pentium 3 running at 750 MHz with 128 MB RAM. The operating system is Red Hat Linux 7.2. All timings are the average of five runs and are given in milliseconds. Additional runs (*i.e.,* usually one but sometime two) precede the timed runs in order to eliminate any overhead of class loading, code preparation, page faults, disk caching, etc. that occur during class loading; these additional runs are performed on both the unmodified and modified VMs. Therefore all timings specifically exclude JVM startup costs.

8.2 Validation

As with medical doctors who must administer treatment to patients, our modification of the VM should follow the rule of *primum non nocere*—"first of all, do no harm." Therefore our modified VM should not produce poorer performance for programs, regardless of whether or not they use exceptions. The standard benchmark programs for JVMs is that produced by the Standard Performance Evaluation Corporation (SPEC) and is named JVM98 [18]. The version used here is the maintenance release 1.04, but with two omissions:

- `_227_mtrt` (a ray-tracing program) raises a `ClassCastException` that causes program failure when run with either the unmodified VM or the modified VM. (The same error occurs when using the HotSpot VM from Sun.)
- `_213_javac` (Sun's Java compiler from JDK 1.0.2) causes `BuildTable` to fail from an `OutOfMemoryException`, and therefore no farhandler or TSR information can be generated.

The remaining benchmark programs are:

piling this function (*i.e.,* `-O0`), then placement of local variables can have a large effect on the final execution times.

- _200_check: sanity checks for JVM and Java features;
- _201_compress: an implementation of LZW [72];
- _202_jess: the Java Expert Shell System (an inference engine);
- _209_db: database benchmarking software written by IBM;
- _222_mpegaudio: decompresses MP3 files;
- _228_jack: a Java parser generator.

Of these, only _228_jack makes significant use of exceptions, and even then its programmers appear to have taken special care to eliminate a lot of exception-handling overhead (*i.e.,* the thrown exceptions are previously created objects, with the object creation cost amortized over the many throws which use it).

Each benchmark was run on four different VM configurations:

- original: This is the unmodified SableVM;
- fh: modified VM using only the farhandler table;
- tsr: modified VM using only the TSR information;
- fh+tsr: modified VM using both the farhandler table and the TSR information.

The timings are shown in Table 8.1; in Table 8.2 are the frequencies of locally-handled exceptions versus those handled outside the throwing method. The benchmarks run as fast—if not faster—under the modified VM.

We make two general observations about this data:

a. Only _228_jack throws a significant number of exceptions—all of them to handlers outside of the throwing method—and the benchmark's speed is significantly improved (about 1% on average, with 0.7% in the worst case and 1.2% in the best case). This gain is significant considering that much other computation is being performed by the benchmark program.

b. For all of the other benchmark programs, there is no observable difference (*i.e.,* "no harm").

In the next two sections, however, we show that for certain kinds of programs much better can be achieved.

Benchmark	original	fh	tsr	fh+tsr
_200_check	49	49	50	49
_201_compress	111635	111418	111475	111564
_202_jess	122402	122380	121380	122362
_209_db	212402	210672	209706	211114
_222_mpegaudio	505410	504821	504667	504856
_228_jack	68933	68131	68270	68432

Table 8.1. *SPECjvm98 benchmark timings (milliseconds)*

Benchmark	locally handled	handled outside throw method
_200_check	104	3
_201_compress	0	0
_202_jess	0	0
_209_db	0	0
_222_mpegaudio	0	0
_228_jack	0	241876

Table 8.2. *SPECjvm98 benchmark (exception-event frequencies)*

8.3 Microbenchmarks

In order to explore the effects of our analysis and modified VM, we need programs for which exceptions—and exception-related work—incur large costs in comparison to normal processing. We have developed two such benchmarks. In `ExceptionTest.java`, exceptions are thrown from the bottom of deeper and deeper callstacks; the benchmark is designed to expose the costs of the different phases in processing involved in throwing an exception. With `Thesis3.java`, the exception-dispatch algorithm is exercised (and therefore farhandler tables are used); exception-object creation is eliminated via TSR where possible. (The full text of both programs may be found in the appendices.)

8.3.1 ExceptionTest

This benchmark has four different modes, each distinguished by the degree to which they use exceptions and exception objects. Regardless of the mode in which the program is run, the depth of the callstack must be specified. Also specified is the number of times the benchmark is to be executed. The modes are as follows:

- `NO EXCEPTION` or `NE`: The callstack is created to the specified depth, and then the program returns back to the main method via the use of `return`.
- `JUST FILL` or `JF`: The callstack is created as before. Once control reaches that depth, an existing exception object is "filled" (*i.e.*, `fillInStackTrace` is repeatedly called). The program returns back to the main method via the use of `return`.
- `FILL AND CREATE` or `FC`: The call stack is again created to a specified depth; once at the greatest depth, an exception object is repeatedly created. (The `FILL` part may seem misleading; the mode name is trying to capture that exception object creation also "fills" the object.) Control is transferred back to the main method using `return`.
- `THROW` or `TH`: The callstack is created to the specified depth. Control is returned to the main method by creating a new instance of an exception and throwing this exception.

depth	original VM				modified VM		
	NE	JF	FC	TH	TH, fh	TH, tsr	TH, fh + tsr
1	29	2402	2671	6802	6769	64	50
5	74	2408	2674	15737	15796	141	123
10	129	2406	2669	26880	26851	232	211
20	239	2415	2685	49273	49000	409	385

Table 8.3. *ExceptionTest timings (milliseconds, 20,000 iterations)*

The modes were specifically chosen in order that the percentage of time taken by each of the activities indicated by the name could be broken out from the figures (as described in Section 7.1).

The timing results for this program as executed on the unmodified VM are shown in Table 8.3; four different stack depths are presented (1, 5, 10 and 20). This illustrates the cost of exceptions using `SableVM`'s architecture. Also shown in the table are the effects of running the mode `THROW` on our modified VM using just farhandler information (TH, fh), just TSR information (TH, tsr), and finally both combined together (TH, fh + tsr).

One curious thing emerges from these figures and that is the very large difference in timings between columns FC and TH. Given that the improvement introduced by using just farhandlers appears to be barely better than without it (and in one case, slightly worse), the extra time cannot be consumed in the unmodified SableVM by exception-dispatch code. For example, the garbage collector is never invoked for any of the timing runs used to generate column FC, but it is invoked eight times in column TH timing runs; both modes create exactly the same number of exception objects (and therefore theoretically the same amount of garbage).

Overall, however, our modified VM is providing a solid benefit. Comparing the NE and (TH, fh + tsr) columns reveals that farhandlers and throw-site reduction by themselves greatly reduce the cost of exceptions—that is, in these benchmarks the modified VM keeps the overhead well within 100% (as opposed to over 200,000% overhead without the modifications).

Path	node trace	original	fh	tsr	fh + tsr
1	*a* b d f h j m	7891	7826	73	63
2	a *c* d f h i i i m	9852	9781	92	82
3	a c d *e* h i i i m	9854	9763	83	74
4	a c d f h k *h* k m	9840	9750	69	62
5	*a* c d f h i i i m	9872	9779	104	87
6	a c *d* g h i i i m	9884	9813	93	83
7	a c d f h k *h* k m	9885	9798	81	68
8	a c d *e* h i i i m	9788	9791	96	78

Table 8.4. *Thesis3 timings (milliseconds, 10000 iterations)*

8.3.2 Thesis3

Our second microbenchmark explores the impact of farhandler tables on exception performance. `Thesis3` is based on the callgraphs described in Chapter 6, but with a few added twists. Figure 8.1 is a view of the core callgraph within `Thesis3` (for which code can be found in Appendix B). There are now three exception types used by the program; `N`, `A` and `E` represent `NullPointerException`, `ArrayIndexOutOf-BoundsException` and `Exception`, respectively.

We have described eight paths through the callgraph, each of which traverses a different set of exception handlers. These paths are described in Table 8.4; *node trace* represents the call stack as it would appear at the point the exception is thrown in `m` for the path. As mentioned in the discussion of `ExceptionTest`, the biggest gains in performance come from the use of TSR. Here, however, the additional gains from farhandlers are clearer; these range from an additional reduction in time from 10% (path 5) up to 19% (path 8), with the median reduction at 11%.

8.4 Exception-idiom Usage

If programmers wished to use exceptions as a way of structuring control flow in their program, what could they reasonably expect from a VM implemented according to our proposals? Our goal is to convince the programmer that EH overhead has

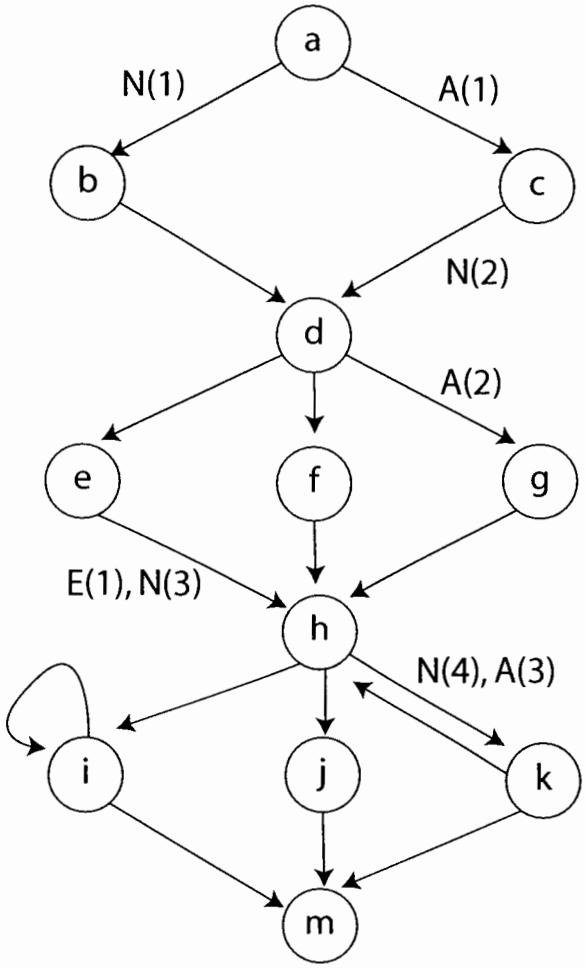


Figure 8.1. *Callgraph with exception-labelled edges: example 2*

been reduced to close to zero, and that this is accomplished without the programmer needing to specify `pragmas`, optimization levels, or caching created exception objects to reduce the impact of the construction on their program. One way to demonstrate this is to examine a typical programming problem.

We have chosen one based on the discussions of Chapter 4. An input file full of words must be examined, and a histogram of those words produced (as might be needed by a compression algorithm, for instance). When each word is input, a binary tree is searched. If the word is found, the corresponding tree node's `frequency` field is incremented. If the word is not found, then a new node must be created and linked into the existing tree.

Three versions of such a program have been written and can be found in Appendix C.

- `SearchLocal` uses exceptions to transfer control to node-creation code when a word is first encountered; all searching of the tree and node creation occurs within the same method.
- `SearchNonLocal` also uses exceptions as mentioned above, but now the tree is searched recursively. Control-transfer for new words now entails unwinding the stack.
- `SearchLocalX` and `SearchNonLocalX` do not use exceptions and perform searching local and via recursive calls, respectively. These should be the fastest versions of the programs.

Text files from the *Calgary Compression Corpus* provided the workload for various programs [73]. Timing results for `SearchLocal` appear in Table 8.5, and those for `SearchNonLocal` are in Table 8.6. The columns labeled "overhead (improvement)" are the percentage differences between the non-exception version of the program with the exception-using version of the program running on the modifed VM.

The modified VM is clearly a win—the overhead of using exceptions (*i.e.,* time difference between exception-free program and exception-rich one) is low, ranging from .3% to 1.1% for `SearchNonLocal`. A pleasant surprise from `SearchLocal` is that in some cases there is a *speedup* as in that for `book1` of about 0.6%.

file	*Using original VM*		*Using modified VM, fh + tsr*	
	SearchLocalX	SearchLocal	SearchLocal	overhead (improvement)
bib	2,699	3,243	2,700	0.0%
book1	23,389	25,932	23,233	(0.7%)
book2	16,955	18,793	16,870	(0.5%)
geo	835	945	834	(0.1%)
news	9,628	11,557	9,629	0.0%
obj1	237	309	238	0.4%
obj2	2,926	3,552	2,931	0.2%
paper1	1,331	1,657	1,333	0.2%
paper2	2,184	2,613	2,189	0.2%
paper3	1,146	1,476	1,152	0.5%
paper4	307	418	310	1.0%
paper5	299	399	298	(0.3%)
paper6	1,025	1,251	1,029	0.4%
pic	3,698	3,726	3,711	0.4%
progc	898	1,148	903	0.6%
progl	1,562	1,839	1,566	0.3%
progp	854	1,144	858	0.5%
trans	1,793	2,127	1,801	0.4%

Table 8.5. *SearchLocal timings (milliseconds, 10,000 iterations)*

file	*Using original VM*		*Using modified VM, fh + tsr*	
	SearchNonLocalX	SearchNonLocal	SearchNonLocal	overhead
bib	2,445	11,081	2,453	0.3%
book1	20,661	73,255	20,731	0.3%
book2	15,240	48,962	15,338	0.6%
geo	819	2,279	821	0.2%
news	8,645	42,262	8,725	0.9%
obj1	223	1,095	226	1.3%
obj2	2,737	14,949	2,762	0.9%
paper1	1,198	5,895	1,210	1.0%
paper2	1,941	8,269	1,958	0.9%
paper3	1,028	5,922	1,049	2.0%
paper4	276	1,596	279	1.1%
paper5	266	1,573	269	1.1%
paper6	916	4,167	921	0.5%
pic	3,705	3,886	3,715	0.3%
progc	823	4,160	828	0.6%
progl	1,426	5,281	1,434	0.6%
progp	805	3,644	809	0.5%
trans	1,647	6,713	1,655	0.5%

Table 8.6. *SearchNonLocal timings (milliseconds, 10,000 iterations)*

8.5 Summary

The experimental results of this chapter have been used to argue that our proposals are valid, reasonable and effective. Techniques presented here produce significant improvements for a variety of benchmarks, from an important 1% improvement in the exception-intensive `_228_jack` program in `SPECjvm98`, to the practically *complete* elimination of exception-handling overhead in other benchmarks, including a performance *increase* in the case of the optimized `SearchLocal` over `SearchLocalX`. Therefore this dissertation's proposals justify the improvements to code—in writability and readability—that can result from using an exception-handling style of programming.

Chapter 9

Conclusions

9.1 Summary of contributions

The work presented in this dissertation has one major focus—to argue and show that a programmer can choose exception handling from amongst a larger set of control mechanisms, and the resulting runtime overhead will not be much higher. This has been demonstrated. In the process of doing so, this thesis has made the following contributions:

- Exception-handling programming idioms have been identified and described.
- A methodology for analyzing the cost of exception-handling implementations has been presented, and the results used to guide further analysis and research.
- A program transformation has been introduced that can eliminate redundant computations—and therefore improve runtime performance—by using exceptions as a substitute for other control-flow mechanisms.
- Program-wide exception-handler tables have been introduced, along with an algorithm for making effective use of these tables.
- The throw-site reduction (TSR) optimization has been developed, along with lazy exception-objects and lazy stack-trace extraction.
- Program-wide handler tables and TSR have been implemented in a real Java virtual machine. The VM's performance has been analyzed by using both existing and new benchmarks.

9.2 Future work

The present Java bytecode analyzer and table construction algorithm assume that the whole program is available at once. However, JVMs usually load classes one at a time and as needed. As the differential in speed between processors and secondary store / networks increases, there are CPU cycles possibly available for analysis and optimization. One direction that future work can take is to investigate versions of the analysis algorithm which are *incremental*.

Our set of benchmark programs is small, but only because the typical workloads we have proposed do not yet really exist. Until they exist, it will be difficult to construct benchmarks. Over the past few years, the number of programmers using object-oriented languages such as Java or C# has exploded, and so a larger and larger body of work is becoming available every year. This body needs to be continuously analyzed, with the clever use of exceptions identified and disseminated.

One surprise during our experimental work was the huge increase in the time taken when exceptions were combined with stack unwinding. Although there are some interactions with the garbage-collection subsystem, this alone cannot explain the large decrease in overall performance when exception objects are created. How exceptions and memory models interact may yield some insights into the design of runtime organizations.

The experimental work in this thesis focussed on a single JVM implementation, but there are now many others. Another direction for future work is to investigate the impact of TSR and program-wide handlers on these others VMs (*i.e.*, Sun's HotJava and IBM's Jikes). Many VMs also use just-in-time compilers as a techniques for improving performance, and so we would wish to ensure that improvements in exception-handling performance also translate to JITs.

Appendix A

Code: ExceptionTest

```
1
    package cases;

    // Investigating the cost of building a stack trace when throwing an
    // exception.
    //

    public class ExceptionTest {

10      private long level = 10;
        private long times = 10000;
        private int test_case = 0;
        private NullPointerException npe;

        final static public int NO_EXCEPTIONS = 1;
        final static public int JUST_FILL = 2;
        final static public int JUST_CREATE_AND_FILL = 3;
        final static public int THROW = 4;

20

        public ExceptionTest (long times, long level, int test_case)
        {
            this.times = times;
            this.level = level;
            this.test_case = test_case;
            this.npe = new NullPointerException();
        }

        public void run (boolean game)
30      {
            if (game) {
                return;
            }

            switch (test_case) {
            case NO_EXCEPTIONS:
```

```
            for (long i = 0; i < times; i++ ) {
                run_no_exception (level);
            }
            break;
        case JUST_FILL:
            run_just_fill (level);
            break;
        case JUST_CREATE_AND_FILL:
            run_just_create_and_fill (level);
            break;
        default:
            for (long i = 0; i < times; i++ ) {
                try {
                    run_throw (level);
                }
                catch (NullPointerException e) {
                    // Just be there for me!
                }
            }
            break;
        }
    }

    private void run_no_exception (long level)
    {
        if (level == 0) {
            return;
        } else {
            run_no_exception (level - 1);
        }
    }

    private void run_throw (long level)
    {
        if (level == 0) {
            throw new NullPointerException();
        } else {
            run_throw (level - 1);
        }
    }

    private void run_just_fill (long level)
    {
        if (level == 0) {
            for (long i = 0; i < times; i++) {
                npe.fillInStackTrace();
            }
            return;
```

```
            } else {
                run_just_fill (level - 1);
            }
        }

        private void run_just_create_and_fill (long level)
        {
            if (level == 0) {
                for (long i = 0; i < times; i++) {
                    npe = new NullPointerException();
                }
                return;
            } else {
                run_just_create_and_fill (level - 1);
            }
        }

        public static void main(String[] argv)
        {
            ExceptionTest et;

            try {
                long l = Long.valueOf(argv[0]).intValue();
                long t = Long.valueOf(argv[1]).intValue();
                String cases = argv[2];
                if (argv[2].equals("one")) {
                    et = new ExceptionTest (t, l, NO_EXCEPTIONS);
                } else if (argv[2].equals("two")) {
                    et = new ExceptionTest (t, l, JUST_FILL);
                } else if (argv[2].equals("three")) {
                    et = new ExceptionTest (t, l, JUST_CREATE_AND_FILL);
                } else {
                    et = new ExceptionTest (t, l, THROW);
                }
                et.run (true); // To clear the pipes / compute code arrays
                long start = System.currentTimeMillis();
                et.run (false); // Now do some real work
                long end = System.currentTimeMillis();
                System.out.println (end - start);
            }
            catch(ArrayIndexOutOfBoundsException e) {
                System.err.println("usage: "
                    + "java Exception <times> <level> <case>");
                System.exit(1);
            }
            catch(NumberFormatException e) {
                System.err.println("usage: "
                    + "java Exception <times> <level> <case>");
                System.exit(1);
```

```
                }
                catch (StackOverflowError soe) {
                    System.err.println("Ouch!");
                    System.exit(1);
                }
140         }
        }
```

Appendix B

Code: Thesis3

B.1 Code listing

```
1   package cases;

    public class Thesis3
    {
        private boolean do_print = true;
        private int init_someval;
        private int someval;
        private NullPointerException npe;
        private ArrayIndexOutOfBoundsException ae;
10
        public Thesis3(int val, boolean do_print)
        {
            this.init_someval = val;
            this.do_print = do_print;
            npe = new NullPointerException();
            ae  = new ArrayIndexOutOfBoundsException();
        }

        public void method_a(int v)
20      {
            someval = init_someval;
            if (v == 1) {
                try {
                    method_b(v);
                }
                catch (NullPointerException e) {
                    if (do_print) {
                        System.out.println ("THESIS: E(1)");
                    }
30              }
            } else {
                try {
                    method_c(v);
```

```
            }
            catch (ArrayIndexOutOfBoundsException e) {
                if (do_print) {
                    System.out.println ("THESIS: F(1)");
                }
            }
        }
    }

    public void method_b(int v)
    {
        method_d(v);
    }

    public void method_c(int v)
    {
        try {
            method_d(v);
        }
        catch (NullPointerException e) {
            if (do_print) {
                System.out.println ("THESIS: E(2)");
            }
        }
    }

    public void method_d(int v)
    {
        if (v == 3 || v == 8) {
            method_e(v);
        } else if (v == 4){
            method_f(v);
        } else if (v == 6) {
            try {
                method_g(v);
            }
            catch (ArrayIndexOutOfBoundsException e) {
                if (do_print) {
                    System.out.println ("THESIS: F(2)");
                }
            }
        } else {
            method_f(v);
        }
    }

    public void method_e(int v)
    {
        try {
```

```
            method_h(v);
        }
        catch (NullPointerException e) {
            if (do_print) {
                System.out.println ("THESIS: E(3)");
            }
        }
90      catch (Exception e) {
            if (do_print) {
                System.out.println ("THESIS: X(1)");
            }
        }
    }

    public void method_f(int v)
    {
        method_h(v);
100 }

    public void method_g(int v)
    {
        method_h(v);
    }

    public void method_h(int v)
    {
        if (v == 4 || v == 7) {
110         try {
                method_k(v);
            }
            catch (NullPointerException e) {
                if (do_print) {
                    System.out.println ("THESIS: E(4)");
                }
            }
            catch (ArrayIndexOutOfBoundsException e) {
                if (do_print) {
120                 System.out.println ("THESIS: F(3)");
                }
            }
        } else if ( v == 1 ) {
            method_j(v);
        } else {
            method_i(v);
        }
    }

130 public void method_i(int v)
    {
```

```
        if (someval > 3) {
            someval--;
            method_i(v);
        } else {
            method_m(v);
        }
    }

140 public void method_j(int v)
    {
        method_m(v);
    }

    public void method_k(int v)
    {
        if (someval > 4) {
            someval--;
            method_h(v);
150     } else {
            method_m(v);
        }
    }

    public void method_m(int v)
    {
        someval = 0;
        if (1 <= v && v <= 4) {
            throw new NullPointerException();
160     } else if (5 <= v && v <= 8) {
            throw new ArrayIndexOutOfBoundsException();
        }
    }

    static public void main(String args[])
    {
        NullPointerException ee = new NullPointerException();
        int path = 1;
        int loop_length = 1;
170     boolean print = true;
        try {
            path = Integer.parseInt(args[0]);
            loop_length = Integer.parseInt(args[1]);
            if (args[2].equals("false")) {
                print = false;
            }
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println (
180             "defaulted to path 1, loop length 1"
```

```
                );
            }
            catch (NumberFormatException e) {
                System.out.println (
                    "defaulted to path 1, loop length 1"
                );
            }
            Thesis3 t = new Thesis3(5, print);
            t.method_a(path);
190         long throwaway = System.currentTimeMillis();
            long start = System.currentTimeMillis();
            for (int i = 0; i < loop_length; i++) {
                if (1 <= path && path <= 8) {
                    t.method_a(path);
                } else {
                    System.out.println (
                        "Try a number between 1 and 8."
                    );
                    System.exit(1);
200             }
            }
            long end = System.currentTimeMillis();
            System.out.println (end - start);
        }
    }
```

Appendix C

Code: SearchLocal

C.1 Listing for SearchLocal

```
1   package cases;

    import java.io.*;
    import java.util.StringTokenizer;

    public class SearchLocal
    {
        private BufferedReader br;
        private WordNode word_tree;
10
        public SearchLocal (String filename)
            throws     FileNotFoundException, IOException
        {
            br = new BufferedReader (new FileReader (filename));
        }


        public void build_frequencies() throws IOException
        {
20          String line = null;
            word_tree = new WordNode ("");
            while ((line = br.readLine()) != null) {
                StringTokenizer st = new StringTokenizer(line);
                while (st.hasMoreTokens()) {
                    String tok = st.nextToken();
                    WordNode temp_word = word_tree;
                    try {
                        for (;;) {
                            if (temp_word.value.compareTo(tok) == 0) {
30                              temp_word.frequency++;
                                break;
                            } else if (temp_word.value.compareTo(tok) > 0) {
                                if (temp_word.left != null) {
```

```
                                temp_word = temp_word.left;
                            } else {
                                throw new NewLeftLeaf();
                            }
                        } else if (temp_word.value.compareTo(tok) < 0) {
                            if (temp_word.right != null) {
                                temp_word = temp_word.right;
                            } else {
                                throw new NewRightLeaf();
                            }
                        }
                    }
                    catch (NewLeftLeaf le) {
                        WordNode new_word = new WordNode(tok);
                        temp_word.left = new_word;
                    }
                    catch (NewRightLeaf re) {
                        WordNode new_word = new WordNode(tok);
                        temp_word.right = new_word;
                    }
                }
            }
        }

        public void dump_frequencies(WordNode w)
        {
            if (w == null) {
                return;
            }
            dump_frequencies (w.left);
            System.out.println (w.frequency + " : " + w.value);
            dump_frequencies (w.right);
        }

        public void dump_frequencies()
        {
            dump_frequencies (word_tree);
        }


        class WordNode
        {
            public String  value;
            public int     frequency;
            public WordNode left;
            public WordNode right;

            public WordNode (String word)
```

```
        {
            this.value = word;
            this.frequency = 1;
            this.left = this.right = null;
        }
    }

    class TraversalException extends Exception { }
    class NewLeftLeaf extends TraversalException { }
    class NewRightLeaf extends TraversalException { }


    public static void main (String[] args)
    {
        String fn = "";

        try {
            fn = args[0];
            SearchLocal s;

            s = new SearchLocal (fn);
            s.build_frequencies();
            // s.dump_frequencies();

            for (int i = 0; i < 5; i++) {
                long start = System.currentTimeMillis();
                s = new SearchLocal (fn);
                s.build_frequencies();
                long stop = System.currentTimeMillis();
                System.out.println (stop - start);
            }
        }
        catch (ArrayIndexOutOfBoundsException ae)
        {
            System.err.println ("usage: java SearchLocal <input file>");
            System.exit(1);
        }
        catch (FileNotFoundException fe)
        {
            System.err.println ("error: cannot find file " + fn);
            System.exit(2);
        }
        catch (IOException fe)
        {
            System.err.println ("error: Houston, we have "
                + "a problem opening "
                + fn);
            System.exit(3);
        }
```

```
    }
}
```

C.2 Listing for SearchNonLocal

```
1   package cases;

    import java.io.*;
    import java.util.StringTokenizer;

    public class SearchNonLocal
    {
        private BufferedReader br;
        private WordNode word_tree;
10      public  WordNode curr_node;

        public SearchNonLocal (String filename)
            throws     FileNotFoundException, IOException
        {
            br = new BufferedReader (new FileReader (filename));
        }


        public void build_frequencies() throws IOException
20      {
            String line = null;
            word_tree = new WordNode ("");
            while ((line = br.readLine()) != null) {
                StringTokenizer st = new StringTokenizer(line);
                while (st.hasMoreTokens()) {
                    String tok = st.nextToken();
                    try {
                        traverse (word_tree, tok);
                    }
30                  catch (NewLeftLeaf le) {
                        WordNode new_word = new WordNode(tok);
                        curr_node.left = new_word;
                    }
                    catch (NewRightLeaf re) {
                        WordNode new_word = new WordNode(tok);
                        curr_node.right = new_word;
                    }
                    catch (TraversalException e) { }
                }
40          }
        }

        private void traverse (WordNode wn, String curr_tok)
```

```
        throws TraversalException
    {
        if (wn.value.compareTo(curr_tok) == 0) {
            wn.frequency++;
            return;
        } else if (wn.value.compareTo(curr_tok) > 0) {
            if (wn.left != null) {
                traverse (wn.left, curr_tok);
            } else {
                curr_node = wn;
                throw new NewLeftLeaf();
            }
        } else {
            if (wn.right != null) {
                traverse (wn.right, curr_tok);
            } else {
                curr_node = wn;
                throw new NewRightLeaf();
            }
        }
    }

    public void dump_frequencies(WordNode w)
    {
        if (w == null) {
            return;
        }
        dump_frequencies (w.left);
        System.out.println (w.frequency + " : " + w.value);
        dump_frequencies (w.right);
    }

    public void dump_frequencies()
    {
        dump_frequencies (word_tree);
    }


    class WordNode
    {
        public String  value;
        public int     frequency;
        public WordNode left;
        public WordNode right;

        public WordNode (String word)
        {
            this.value = word;
            this.frequency = 1;
```

```
            this.left = this.right = null;
        }
    }

    class TraversalException extends Exception { }
    class NewLeftLeaf extends TraversalException { }
    class NewRightLeaf extends TraversalException { }
100

    public static void main (String[] args)
    {
        String fn = "";

        try {
            fn = args[0];
            SearchNonLocal s;

110         s = new SearchNonLocal (fn);
            s.build_frequencies();

            for (int i = 0; i < 5; i++) {
                long start = System.currentTimeMillis();
                s = new SearchNonLocal (fn);
                s.build_frequencies();
                long stop = System.currentTimeMillis();
                System.out.println (stop - start);
            }
120     }
        catch (ArrayIndexOutOfBoundsException ae)
        {
            System.err.println ("usage: java SearchNonLocal <input file>");
            System.exit(1);
        }
        catch (FileNotFoundException fe)
        {
            System.err.println ("error: cannot find file " + fn);
            System.exit(2);
130     }
        catch (IOException fe)
        {
            System.err.println ("error: Houston, we have "
               + "a problem opening " + fn);
            System.exit(3);
        }
    }
}
```

C.3 Listing for SearchLocalX

This solves the same problem as SearchLocal but without exceptions.

```
1   package cases;

    import java.io.*;
    import java.util.StringTokenizer;

    public class SearchLocalX
    {
        private BufferedReader br;
        private WordNode word_tree;
10
        public SearchLocalX (String filename)
            throws     FileNotFoundException, IOException
        {
            br = new BufferedReader (new FileReader (filename));
        }


        public void build_frequencies() throws IOException
        {
20          String line = null;
            word_tree = new WordNode ("");
            while ((line = br.readLine()) != null) {
                StringTokenizer st = new StringTokenizer(line);
                while (st.hasMoreTokens()) {
                    String tok = st.nextToken();
                    WordNode temp_word = word_tree;
                    for (;;) {
                        if (temp_word.value.compareTo(tok) == 0) {
                            temp_word.frequency++;
30                          break;
                        } else if (temp_word.value.compareTo(tok) > 0) {
                            if (temp_word.left != null) {
                                temp_word = temp_word.left;
                            } else {
                                WordNode new_word = new WordNode(tok);
                                temp_word.left = new_word;
                                break;
                            }
                        } else if (temp_word.value.compareTo(tok) < 0) {
40                          if (temp_word.right != null) {
                                temp_word = temp_word.right;
                            } else {
                                WordNode new_word = new WordNode(tok);
                                temp_word.right = new_word;
                                break;
                            }
```

```
                    }
                }
            }
        }
    }

    public void dump_frequencies(WordNode w)
    {
        if (w == null) {
            return;
        }
        dump_frequencies (w.left);
        System.out.println (w.frequency + " : " + w.value);
        dump_frequencies (w.right);
    }

    public void dump_frequencies()
    {
        dump_frequencies (word_tree);
    }


    class WordNode
    {
        public String  value;
        public int     frequency;
        public WordNode left;
        public WordNode right;

        public WordNode (String word)
        {
            this.value = word;
            this.frequency = 1;
            this.left = this.right = null;
        }
    }


    public static void main (String[] args)
    {
        String fn = "";

        try {
            fn = args[0];
            SearchLocalX s;

            s = new SearchLocalX (fn);
            s.build_frequencies();
            // s.dump_frequencies();
```

```
                for (int i = 0; i < 5; i++) {
                    long start = System.currentTimeMillis();
                    s = new SearchLocalX (fn);
100                 s.build_frequencies();
                    long stop = System.currentTimeMillis();
                    System.out.println (stop - start);
                }
            }
            catch (ArrayIndexOutOfBoundsException ae)
            {
                System.err.println ("usage: java SearchLocalX <input file>");
                System.exit(1);
            }
110         catch (FileNotFoundException fe)
            {
                System.err.println ("error: cannot find file " + fn);
                System.exit(2);
            }
            catch (IOException fe)
            {
                System.err.println ("error: Houston, we have a "
                    + "problem opening "
                    + fn);
120             System.exit(3);
            }
        }
    }
```

Appendix D

Glossary

activation record : Memory set aside on the runtime stack for a call of a subroutine. This is usually used to store values of local variables, the return address of the caller.

aliasing : Describes a situation at run-time where two variables refer to the same location in memory. While they can improve the writeability of programs, the result can be code that is difficult to understand and analyze, especially if the programmer did not intend for the alias to be created.

basic block : A sequence of instructions (usually assembly language) for which the only entry point is the first instruction and the only exist is the last instruction. Basic blocks are the nodes in flowgraphs, where edges of these graphs correspond to possible flow of control from one basic block to another.

bus error : One of a set of errors that can be detected by hardware and which will then be reported back to the program generating the error. Typical errors are invalid address alignment and accessing a physical address that does not exist. (The "bus" here corresponds to either the address or data bus.)

callstack traversal : Assuming that the activation frames corresponding to a program's execution are arranged in a stack, traversing the callstack is the same as examining each stack element from bottom to top.

computational reflection : Describes computing about computing; programs written in this style are able to examine their own state and environment, and from this are able to exploit new functionality or respond to the absence of expected functionality. This is also the same as "metaprogramming".

destructors : In an object-oriented language, this is the code that must be executed

when an object's lifetime ends (either by explicitly releasing the object's memory or via garbage collection).

dynamically enclosing scope : Refers to the fact that some facts about the program at a certain point cannot be known until run-time, *i.e.,* the program path taken to reach that point determines such things as the activation frames on the callstack. This is as distinct from "lexically-enclosing scope" describing what can be known about the program point by reading only the program text around that point (*i.e., not* simulating execution).

escaping : Control leaving a module boundary.

executable specification : A description of a program's semantics ("intended meaning") which also has the feature that it may be executed in such a way that its violation has consequences at run-time.

may-ref : "May reference," *i.e.,* "it is the case that something *might* be used at this point."

must-ref : "Must reference," *i.e.,* "it is the case that something is *definitely* used at this point."

optimize away : Applied to operations that are deemed redundant as the result of some optimization analysis, such that these operations do not appear in the executable version of the program.

policy vs. mechanism : Classic description of the separation of *what* should be done (policy) from the set of mechanisms that describe *how* it can be done (mechanism).

reachable : An adjective usually applied to a the action of assigned a value to a variable (a "definition"). Such an action is said to be *reachable* at some program point if there is a control-flow path from the assignment to the program point.

segmentation fault : In UNIX systems, corresponds to a program attempting access to memory not allocated to the program.

status variables : A set of variables used to record specific parts of a program's state (such as the occurrence of an error after calling a library routine).

Bibliography

[1] Samson Abramsky and Chris Hankin. An introduction to abstract interpretation. In *Abstract Interpretation of Declarative Languages*, Computers and their Applications, pages 9–31. Ellis Horwood Limited, 1987.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, March 1988.

[3] M. Stella Atkins. *The Role of Exceptions Mechanisms in Software Systems Design*. PhD thesis, University of British Columbia, 1985.

[4] John Aycock. A Brief History of Just-In-Time. *ACM Computing Surveys*, 35(2):97-113, June 2003.

[5] Thomas J. Bergin and Richard G. Gibson, editors. *A History of CLU*, 1996.

[6] Andrew P. Black. *Exception Handling: The Case Against*. PhD thesis, Balliol College, 1982.

[7] Egon Boerger and Wolfram Schulte. A Practical Method for Specification and Analysis of Exception Handling—A Java/JVM Case Study. *IEEE Transactions on Software Engineering*, 26(9):872–887, September 2000.

[8] Peter A. Buhr and W.Y. Russell Mok. Advanced Exception Handling Mechanisms. *IEEE Transactions on Software Engineering*, 26(9):820–836, September 2000.

[9] 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 Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, pages 129–141, June 1999.

[10] Don Cameron, Paul Faust, Dmitry Lenkov, and Michey Mehta. Portable Implemention of C++ Exception Handling. In *USENIX C++ Technical Conference*, pages 225–243. USENIX Association, 1992.

[11] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kaslow, and Greg Nelson. Modula-3 report (revised). Technical Report Technical Report 52, Digital Systems Research Center, 1989.

[12] Tom Cargill. Exception Handling: A False Sense of Security. *C++ Report*, 6(9), Nov/Dec 1994.

[13] David Chase. Implementation of exception handling, Part I. *The Journal of C Language Translation*, 5(4):229–240, June 1994.

[14] David R. Cheriton. Making exceptions simplify the rule (and justify their handling). In *Proc. of IFIP Congress 1986*, pages 27–33, 1986.

[15] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and Precise Modelling of Exceptions for the Analysis of Java Programs. In *Proceedings of the SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '99)*, pages 21–31, September 1999.

[16] Morten M. Christensen. Methods for Handling Exceptions in Object-Oriented Languages. Master's thesis, Odense University, Denmark, January 1995.

[17] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure Cloning. In *Proceedings of the 1992 IEEE Internation Conference on Computer Languages*, pages 96–106, April 1992.

[18] Standard Performance Evaluation Corporation. SPEC Releases SPECjvm98, First Industry-Standard Benchmark for Measuring Java Virtual Machine Performance. http://www.specbench.org/jvm98/press.html, August 1998.

[19] Flaviu Cristian. Exception Handling. In T. Andersen, editor, *Dependability of Resilient Computers*, pages 68–97. BSP Professional Books, Blackwell Scientific Publications, 1989.

[20] Qian Cui and John Gannon. Data-Oriented Exception Handling. *IEEE Transactions on Software Engineering*, 18(5):393–401, May 1992.

[21] Saumya K. Debray and Todd A. Proebsting. Interprocedural control flow analysis of first-order programs with tail-call optimization. *ACM Transactions on Programming Languages and Systems*, 19(4):568–585, July 1997.

[22] S. Drew, K. John Gough, and J. Ledermann. Implementing Zero Overhead Exception Handling. Technical Report FIT Technical Report 95-12, Queensland University of Technology, 1995.

[23] Etienne Gagnon. *A Portable Research Framework for the Execution of Java Bytecode.* PhD thesis, School of Computer Science, McGill University, 2002.

[24] Etienne Gagnon and Laurie Hendren. SableVM: A Research Framework for the Efficient Execution of Java Bytecode. In *Proceedings of Java Virtual Machine Research and Technology Symposium (JVM '01)*, April 2001.

[25] N. H. Gehani. Exceptional C or C with Exceptions. *Software: Practice and Experience*, 22(10):827–848, October 1992.

[26] Charles M. Geschke, James H. Morris Jr., and Edwin H. Satterthwaite. Early Experience with Mesa. *Communications of the ACM*, 20(8):540–553, August 1977.

[27] John B. Goodenough. Exception Handling: Issues and a Proposed Notation. *Communications of the ACM*, 18(12):683–696, December 1975.

[28] Rajiv Gupta. Optimizing Array Bound Checks using Flow Analysis. *ACM Letters on Programming Languages and Systems*, 2(1–4):135–150, 1993.

[29] John Hennessy. Program Optimization and Exception Handling. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 200–206, Williamsburg, Virginia, January 26–28, 1981. ACM SIGACT-SIGPLAN, ACM Press.

[30] Charles Anthony Richard Hoare. The Emperor's Old Clothes: 1980 Turing Award Lecture. *Communications of the ACM*, 24(2):75–83, February 1981.

[31] M. Hof, H.-P. Mössenböck, and P. Pirkelbauer. Zero-Overhead Exception Handling using Metaprogramming. In *Proc. of 24th Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*, pages 423–431. Lecture Notes in Computer Science, Springer-Verlag, March 1997.

[32] Simon Peyton Jones, Tony Hoare, Alastair Reid, and Simon Marlow. A semantics for imprecise exceptions. In *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, pages 25–36. ACM Press, New York, NY, 1999.

[33] Brian Kernighan and Dennis Ritchie. *The C Programming Language (Second Edition)*. Software Series. Prentice-Hall, 1988.

[34] Jorgen Lindskov Knudsen. Exception Handling—A Static Approach. *Software: Practice and Experience*, 5(14):429–449, May 1984.

[35] Donald E. Knuth. Structured Programming with go to Statements. *ACM Computing Surveys*, 6(4):261–301, December 1974.

[36] Andrew Koenig and Bjarne Stroustrup. Exception handling for C++. *Journal of Objected-Oriented Programming*, pages 16–33, July/August 1990.

[37] Andreas Krall and Mark Probst. Monitors and Exceptions: How to implement Java efficiently. *Concurrency–Practice and Experience*, pages 837–850, 1998.

[38] Jun Lang and David B. Stewart. A Study of the Applicability of Existing Exception-Handling Techniques to Component-Based Real-Time Software Technology. *ACM Trans. Programming Languages and Systems*, 20(2):274–301, March 1998.

[39] Seugnll Lee, Byung-Sun Yang, Suhyun Kim, Seongbae Park, Soo-Mook Moon, Kemal Ebcioğlu, and Eric Altman. Efficient Java exception handling in just-in-time compilation. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 1–8, June 2000.

[40] Seungll Lee, Byung-Sun Yang, Suhyun Kim, Seongbae Park, Soo-Mook Moon,

Kemal Ebcioğlu, and Eric Altman. On-Demand Translation of Java Exception Handlers in the LaTTe JVM Just-In-Time Compiler. In *Proceedings of the Workshop on Binary Translation*, October 1999.

[41] Roy Levin. *Program Structures for Exceptional Condition Handling.* PhD thesis, Carnegie-Mellon University, 1977.

[42] Tim Lindholm and Frank Yellin. *The Java (tm) Virtual Machine, Second Edition.* Addison Wesley, 1999.

[43] Barbara H. Liskov and Alan Snyder. Exception Handling in CLU. *IEEE Transactions on Software Engineering*, SE-5(6):546–558, November 1979.

[44] David C. Luckham and W. Polak. Ada exception handling: an axiomatic approach. *ACM Transactions on Programming Languages and Systems*, 2(2):225–233, April 1980.

[45] International Business Machines. *VisualAge PL/I Language Reference, Version 2 Release 2.1.* IBM, September 2000.

[46] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous Exceptions in Haskell. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming language design and implementation*, pages 274–285. ACM Press, New York, NY, 2001.

[47] Bertrand Meyer. *Eiffel: The Language.* Prentice-Hall International, 1992.

[48] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Professional Technical Reference, 1997.

[49] S.P. Midkiff, J.E. Moreira, and M. Snir. Optimizing array reference checking in Java programs. *IBM Systems Journal*, 37(3), 1998.

[50] James G. Mitchell, William Maybury, and Richard Sweet. Mesa Language Manual. Technical Report CSL-79-3, Xerox Palo Alto Research Centers, April 1979.

[51] Mitre Corporation. *Ada Reference Manual*, 2000. ISO/IEC 8652:1995(E).

[52] Steven S. Muchnick. *Advanced Compiler Design.* Morgan Kaufmann, 1997.

[53] Flemming Nielson and Hanne Riss Nielson. Interprocedural control flow analysis. In *Proceedings of the European Symposium on Programming (LNCS 1567)*, pages 20–39. Springer-Verlag, 1999.

[54] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. A study of exception handling and its dynamic optimization in Java. In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems, Languages and Applications*, pages 83 – 95. ACM Press, New York, NY, October 2001.

[55] David Orchard. Better Performance with Exceptions in Java. *BYTE Magazine*, pages 53–54, March 1998.

[56] Elliot Irviny Organick. *The Multics system; an examination of its structure.* MIT Press, 1972.

[57] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1996.

[58] George Radin. The Early History and Characteristics of PL/I. In Richard L. Wexelblat, editor, *Proceedings from the ACM SIGPLAN History of Programming Languages Conference*, pages 551–575, June 1978.

[59] Dennis M. Ritchie. The Evolution of the Unix Time-Sharing System. *AT&T Bell Laboratories Technical Journal*, 63(6):1577–1593, October 1984.

[60] Barbara G. Ryder and Mary Lou Soffa. Influences on the Design of Exception Handling. *AGM SIGSOFT Software Engineering Notes*, 28(4):29–35, July 2003.

[61] Carl F. Schaefer and Gary N. Bundy. Static Analysis of Exception Handling in Ada. *Software: Practice and Experience*, 23(10):1157–1174, October 1993.

[62] Jack Shirazi. *Java Performance Tuning.* O'Reilly and Associates, Inc., 2000.

[63] David A. Solomon and Mark Russinovich. *Inside Microsoft Windows 2000, 3rd Edition.* Microsoft Press, 2000.

[64] Robert F. Steinhart and Seymour V. Pollack. *Programming the IBM System/360.* Holt, Rinehart and Winston, 1970.

[65] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison Wesley, 1994.

[66] Bjarne Stroustrup. *C++ Programming Language: Third Edition.* Addison Wesley, 1997.

[67] M. D. Tiemann. An exception handling implementation for C++. In *Proceedings of the USENIX C++ Conference*, pages 215–232, Berkeley, CA, USA, 1990. USENIX Association.

[68] Tatyana Valkvych and Sophia Drossopoulou. Formalizing Java Exceptions (position paper). In *Workshop on Exception Handling in Object-Oriented Systems (EHOOS), ECOOP 2000 (Unpublished position paper)*, June 2000.

[69] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot – A Java Optimization Framework. In *CASCON 1999*, pages 125–135, September 1999.

[70] Jan Vitek and R. Nigel Horspool. Compact Dispatch Tables for Dynamically Typed Programming Languages. In *Proceedings of the International Conference on Compiler Construction (CC 1996)*, pages 309–325, 1996.

[71] Jan Vitek, R. Nigel Horspool, and Andreas Krall. Efficient Type Inclusion Tests. In *ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 1997)*, pages 142–157, October 1997.